COMPONENT DESIGN TOOL FOR EMBEDDED SYSTEM COMPONENTS

by

Tien Le Quang
qle06001@student.mdh.se

Supervisor: Mikael Åkerholm
mikael.akerholm@mdh.se

Examinator: Kristian Sandström
kristian.sandstrom@mdh.se

Master of Computer Science
Department of Computer Science and Electronics
Mälardalen University
Västerås, Sweden
- 2008 -
ACKNOWLEDGEMENT

I would like to take this opportunity to express my deep sense of gratitude and profound feeling of admiration to my thesis supervisor, Mikael Åkerholm, who has been with me at the beginning and has stayed to the end. I’d also like to thank CC-Systems AB Company for a wonderful working environment with joy and fun.

Finally, I’d like to thank to my parents, my sisters, and all of my friends for their encouragement.
ABSTRACT

One of the interesting principles of component-based software engineering is reusability of components. However, it is often a big challenge for embedded software domain to achieve because of the complexity of embedded software components. Hence, embedded software developers often attempt to apply partial reuse their components for new projects. In particular, a suitable component will be reused if its requirements match the new ones of the new project. In this thesis, we take advantage of the notion of metadata to build a tool which is called a design tool so that the developers can integrate the metadata into their components. The metadata is used to store all of the internal information of the components in order for the developers to get knowledge about the components and make a faster reuse decision. On the other hand, this tool also supports for component modification due to information defined in metadata and detection of unplanned side-effects during regression testing of modified component.
# TABLE OF CONTENTS

1.0 INTRODUCTION........................................................................................................ 1

2.0 PROBLEM DEFINITION .......................................................................................... 4

2.1 SOFTWARE ENGINEERING INDUSTRIES:................................................ 4

2.2 SUBCONTRACTORS ........................................................................................ 6

3.0 RESEARCH METHODOLOGIES ............................................................................ 8

4.0 BACKGROUND ........................................................................................................ 12

4.1 SOFTWARE COMPONENT OVERVIEW ................................................... 12

4.2 REUSABILITY ISSUES OF SOFTWARE COMPONENT ................................. 16

4.2.1 Black-box and white-box component reuse.............................................. 17

4.2.2 IEC 61131-3 standard – A CBSE success ................................................. 19

4.3 COMPONENT VARIANTS AND METADATA .............................................. 21

4.3.1 Component variants.................................................................................... 21

4.3.2 Metadata Unit.............................................................................................. 23

4.3.3 Metadata Management Model................................................................... 29

4.4 COMPONENT DEVELOPMENT PROCESS ............................................... 31

4.5 TEAM FOUNDATION SERVER OVERVIEW ............................................ 34

5.0 SUGGESTED SOLUTION ....................................................................................... 40

5.1 USE CASE.......................................................................................................... 43
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>CLASS DIAGRAM FOR METADATA MANAGER (DLL)</td>
<td>44</td>
</tr>
<tr>
<td>6.0</td>
<td>DEMONSTRATION</td>
<td>48</td>
</tr>
<tr>
<td>6.1</td>
<td>DESCRIPTION</td>
<td>48</td>
</tr>
<tr>
<td>6.2</td>
<td>CREATE METADATA FOR COMPONENTS</td>
<td>51</td>
</tr>
<tr>
<td>6.3</td>
<td>NEW REQUIREMENTS</td>
<td>54</td>
</tr>
<tr>
<td>7.0</td>
<td>THESIS EVALUATION</td>
<td>60</td>
</tr>
<tr>
<td>7.1</td>
<td>REALISTIC APPLICATION</td>
<td>60</td>
</tr>
<tr>
<td>7.2</td>
<td>FEEDBACK</td>
<td>61</td>
</tr>
<tr>
<td>8.0</td>
<td>CONCLUSION</td>
<td>64</td>
</tr>
<tr>
<td>A.1</td>
<td>ESTABLISH CONNECTION TO TEAM FOUNDATION SEVER</td>
<td>65</td>
</tr>
<tr>
<td>A.2</td>
<td>SELECT A WORKSPACE</td>
<td>66</td>
</tr>
<tr>
<td>A.3</td>
<td>CREATE A NEW WORKSPACE</td>
<td>67</td>
</tr>
<tr>
<td>A.4</td>
<td>CHECK-OUT</td>
<td>69</td>
</tr>
<tr>
<td>A.5</td>
<td>CHECK-IN</td>
<td>70</td>
</tr>
<tr>
<td>A.6</td>
<td>BRANCHING/CREATING A NEW VARIANT</td>
<td>71</td>
</tr>
<tr>
<td>REFERENCES</td>
<td></td>
<td>72</td>
</tr>
</tbody>
</table>
1.0 INTRODUCTION

Software technology evolves rapidly. Because of competition, many common elements in software are shared by many applications in order to reduce the development cycle. Component-based software development is a process in which pre-defined components are assembled into an application in order to get satisfactory functionality. Today, components are applied from small to large systems; they are no longer limited in one system or one company boundary; they are shared in different applications and even different companies. Therefore, reusability provides a good concept for software components to be included in several applications. Apart from reusability, integrating software from assemblies of components brings to us some important features such as extensibility and maintainability. Moreover, it also has some possibilities to improve business issues such as decreased time to market and long-term maintenance costs.

Embedded systems are tiny computer systems. They are present everywhere in the world around us. They exist in cell-phones, cars, or in complicated systems such as airplane. Essentially, an embedded system is a complex system because its non-functional requirements are, in fact, as important as functional requirements [1]. Therefore, software developers usually deal with many challenges during the development. In particular, some non-functional requirements, such as small size,
platform independency, low cost, low resource consumption, and safety require the developers to make more effort and consideration from the initial steps of the development. Furthermore, the design process of a system is much more complicated with respect to the complex requirements. Therefore, components in embedded systems are considered as specialized components in component-based software engineering.

For subcontractor companies, applying component-based software engineering (CBSE) is beneficial, and it saves a lot of work effort because it reduces redundant work during the development. And reusability of components also enables to improve reliability of the software because the built components were tested and verified in the previous developments [11]. Unfortunately, because of the complexity of requirements of an embedded system product, the demands of customers – usually OEM (Original Equipment Manufacturing) companies on the subcontractors, and other factors such as application specific, the ability to approach this application is by far difficult.

By taking advantage of reusability principle of CBSE, components provided by subcontractors should be built and verified once and reused many times. However in CBSE, reuse has been usually attempted at implementation phase, while in complex embedded systems, reuse could be more effective by starting at the early phase of the development cycle. The implementation effort is just the minor part of the processes, and the cost of reusing code is inconsiderable compared to the cost of re-validating reused code [2]. Therefore, widening and reusing specifications, design documents, tests, analyses, test results, and analysis results in a structured and controlled way could greatly increase the impact of component reuse for subcontractors.
In this thesis, we focus on an approach to support reuse for embedded system components. It is the notion of component variants and the use of metadata to manage the variants. It is worth noting that component variants are not the same as component versions. A variant of a component is a realization of a specialized set of requirements, whereas a version corresponds to the same set of requirements. Metadata is used to manage the variant evolution of the components. It contains all of component’s information such as specification, design, test, and so on. The aim of this approach is to achieve the followings:

- Ability of evaluation of the design efforts and the test efforts needed when the modification of the component is made. For example, changes of contexts such as hardware or software; changes of functional and non-functional requirements.

- Support to rapidly create specialized variants of components when the requirements, the design, and the verification are reused or replaced.

- Detection of unplanned side-effects that are caused by changes of a component. This way could improve the knowledge about the component in order to avoid the same problems when the component is used next times.
2.0 PROBLEM DEFINITION

Original Equipment Manufacturing (OEM), Commercial-Off-The-Shelf, and Subcontractor are three software engineering industries today. Each of them more or less has been applying CBSE principles in their project development in order to attempt achieving success. In this section we distinguish their characteristics and how CBSE principles are applied in their projects. On the other hand, we address the main problems with which the subcontractors are dealing.

2.1 SOFTWARE ENGINEERING INDUSTRIES:

Original Equipment Manufacturing (OEM) companies are companies which own the products, lead the projects and work as system integrators by selecting different Commercial-Off-The-Shelf (COTS) solutions and specifying other needed functions. OEM applies very well CBSE principles in their software development. In particular, their software products are often integrated by different components from different third-party vendors such as COTS or Subcontractor companies. This manner has been considered as beneficial for their software development and business success. For instance, it reduces the development processes, development effort and, at the same time, decreases the time to market. However, due to the presence of third-party components and the lack of components’ information, it also causes lots of difficulties.
with the use of components such as “locating the code”, “hidden dependences among the components”, “difficulties in program understanding” [4], and so on.

Commercial-Off-The-Shelf (COTS) software companies develop their own solutions and supply them to the other parties such as OEM companies. There are some dominant vendors in COTS industry including Microsoft, Bosch, Adobe, etc. COTS components are usually developed independently by those companies, therefore COTS component users often have little knowledge about them. Some examples of COTS components are operating system, processor, and so on. The characteristics of COTS components, which differ from other kinds of components, can be seen as follows: [8]

- COTS components are black-box components. They are built in a black-box manner, so component users will not have full access to their underlying implementations when using them.
- COTS components are not built and maintained by component users. As a result, component users can not control or have a minimal control over the evolution of components used in their system. Thus, any update to a component’s functionality may result in unexpected interaction with other components.
- COTS components do not always have correct and complete specifications. Although component vendor provides functional descriptions, they do not always satisfy component users who may need to understand the details of behavioral specifications of the components.
- COTS components are built as a standalone application. Therefore, the possibilities of either mismatch or interactive problems among COTS
components may not be anticipated. They are often found at the end of development process.

Subcontractors are specialized organizations which develop customized solutions for Original Equipment Manufacturing (OEM) in certain engineering disciplines for example control system software, environmental resistive electronics, or mechanical construction. More detailed information about the subcontractors will be discussed in following section.

2.2 SUBCONTRACTORS

Subcontractors provide customized embedded software solutions for different types of engineering disciplines. Those solutions are often ordered by OEM companies. Unlike COTS companies which have independent projects, subcontractor’s projects are usually tailored based on customers’ needs. In other words, the project’s requirements are usually changed according to customers’ needs, and each customer has different requirements. This causes difficulties for subcontractors during their project development. In particular, once the requirements have changed, developers have to make more efforts to identify the changes and relating parts with those changes, and this prolongs the development time, increases the cost, and affects business issues.

It is believed that applying CBSE principles to build reusable software components is necessary for subcontractors. Unfortunately, this application is facing some challenges which are often attributed to some constrains, such as generality, adaptability, environments, and so on. On the other hand, the requirements do not depend only on the customers; they are also changed by each type of products, as
illustrated in Figure 2-1. Therefore, applying CBSE to expand component reuse from coding phase to other earlier phases in a structured and controlled way could reduce the impact of overall development by requirement changes.

**Figure 2-1: Different requirements from different situations**
The overall thesis will comprise five different steps:

- Problem Analysis: problems will be analyzed and defined at the early phase of the thesis. During this phase, there are communications between thesis.
maker and other different sources such as supervisor, as well as CC-Systems’s personnel to identify the problems.

- **Research Setting:** some routines are set up with respect to the defined problems. Thesis maker will start to conduct the research routines by studying related information from different resources. At the same time, he or she will also specify methods to be implemented in the Implementation phase.

- **Implementation:** thesis project implementation.

- **Verification:** experiment the project with samples to verify its correctness in conjunction with the Research Setting.

- **Real Project Deployment:** the project will be deployed in the real environment in the future.
Research Question 1:

How do subcontractor and OEM approach component-based software engineering (CBSE) principles to enable component-based software development?

OEM usually builds their projects by integrating different components from other parties such as COTS and Subcontractors. Therefore, this situation makes it easier to approach CBSE principles. However, OEM also gets some major difficulties during their component integration because of the presence of different third-party components.

Component-based software development has been considered in subcontractor projects’ development. However because of complex requirements of embedded products, and demands from customers, the ability to completely approach CBSE principles for subcontractors is a quite big challenge. For instance, the ability to reuse a component is applied but not completely applied at subcontractors’ site. In particular, an existing component can be partially reused in the new project when some of its functionalities match the new requirements. As a result, it creates one specialized variant for changes.

Research question 2:

Can component metadata support the specialization of components?
A component metadata is created together with a component in order for developers to get the knowledge about the component. Due to metadata, developers can manage their components in an effective way. At the same time, it supports to evaluate the work effort before its specialized variant is created. The evaluation includes identifying the relations between the component’s entities, setting up the work-order/work-effort such as reused entities, affected entities, removed entities, and so on with the help of their dependencies. On the other hand, the work-order will help to detect unplanned effects which are undetected in the specialized variant.

**Research question 2.1:**

*Can we achieve component reusability due to the support of component metadata within embedded system components?*

Component metadata is not a solution for component reuse; it only supports reusing process during the development of an embedded component. In particular, because the metadata of the component contains the entities information and the relations between them, it will help the subcontractor’ developers answer some questions during the development: should we start building a component from scratch or reuse one of the existing components according to the new requirements? Which parts of the given component should be reused? Which parts of the component should be removed or modified? Which parts of the component will be affected? And so on.
4.0 BACKGROUND

4.1 SOFTWARE COMPONENT OVERVIEW

Today, there are various definitions about software component because it is understood by different aspects. For instance, the software component has different views between academy and industry; the academy views component as a black-box and well-defined unit; whereas the industry views it as a large piece of software which has a complex internal structure [19].

According to Szypersky:

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties”
And according to Heineman’s and Councill:

“A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.”

There are some common perspectives about software component in both definitions. First of all, a software component should be independent during integration and deployment phases. Typically when updating the component, it can be updated at run-time without any negative effect of the whole system, and it also requires no modification of itself. Second of all, the component specifications should be well defined in order to support composition standard. It is worth noting that a component is closely related to an object. Therefore component-based development is also an extension of object-oriented development in software development paradigms. However, the difference between them is that in object-oriented programming, the main unit is objects, and the main unit in component-based programming is components.

A software component often consists of a set of internal objects. The objects in a component can communicate or interact with each other through their implementations. However they should not directly interact with the objects of other components. In other words, the components can only communicate with each other through their interfaces. The interface is the fundamental element of component. Typically in COM component model, every COM component must have at least one IUnknown interface. The IUnknown is the special interface because every COM
component has to implement it, and it is present in every COM component. The interface offers only a collection of named operations, operation descriptions and protocols so that other components or clients can understand its functionality and the utilization of component. Thus a component has to interact with some other components or its environment through its interfaces. The interfaces must be obviously defined and visible to the users. At the same time, the ways of how the component is implemented or how it behaves are encapsulated or hidden.

“The component specification can be achieved through contracts.” [19]. In reality, particularly in business, a contract is an agreement between 2 sides: client and supplier. Each side has to do some required tasks (obligations) for the other to gain a benefit. Similarly in software components, a client component requires a supply component to provide services, and the supply component also requires the client component to call its operations. The client component has to provide the correct input (pre-condition) to the supply component’s operation in order to get an expected output (post-condition), Figure 4-1.

Pre-condition: “are assertions that the component assumes to be fulfilled before an operation is invoked.” [19]

Post-condition: “are assertions that the component guarantees will hold just after an operation has been invoked, provided the operation’s pre-conditions were true when it was invoked.” [19]
There is also still no official definition of the framework. A framework is a place where provides services so as the components can be dynamically inserted. When a component is placed on a framework, the framework will require it to perform certain operations controlled by the framework. It is worth noting that there is distinction between component model and component framework. “A component model defines a set of standards and conventions used by the component developer, whereas a component framework is a support infrastructure for the component model.” [19].

Figure 4-1: Contract between components C and D [20].
Software components today are no longer limited in one system or one company boundary; they are shared in different applications and even different companies. Therefore, they are also one of the key strategies for business in software companies today. The main reasons for reusing components are saving time, effort and costs of development and maintenance. In other words, developers will not have to develop their new solution for the old problem from scratch. They can use the existing components to solve it instead.

A component can be assessed in many manners. Some people believed that the quality of reusability of a component more or less affects the quality of component itself. This assessment is performed based on object-oriented design concepts: cohesion\(^1\) and coupling\(^2\) [5]. A looser coupling will result in better performance of the component. Also if the sub component consists of unrelated methods of the classes, it is

\[^1\] Cohesion is based on methods similarities; a class which has much similarity of methods is considered highly cohesive. Similarity of methods is assessed by their usage of class variables; two methods are considered similar if the collection of class variables used by both of them, is a big portion of the class variables.

\[^2\] Coupling is defined as when a method in one object makes use of methods or variables of another object; this shows the coupling between their classes. A design with a great coupling degree means more complexity, compared to those with less coupling degree.
considered to have a poor functionality. Therefore, the better the design of the subcomponents is, the higher will be the cohesion level of them.

4.2.1 Black-box and white-box component reuse

Visibility of a component’s implementation refers to black-box and white-box abstractions. Black-box abstraction encapsulates component’s implementations; it only reveals the component’s interface and specification to clients, whereas in white-box abstraction, component’s implementation is available for manipulating. Black-box reuse is to apply an existing component into the new system regardless of understanding its internal implementations. In other words, we may reuse the existing components by using the interface and specification provided by this component and its implementations are hidden or encapsulated. In contrast, white-box reuse requires modification of a component before (re)using it. For instance, while reusing a component into the new system, it may be either modified or extended so that it will be able to fulfill the new requirements of the new system.

Some people believe that black-box reuse is more beneficial than white-box reuse because of the fundamental characteristics of component-based software development. For instance, component-based software often consists of a set of components which offer only functionalities and services through their interfaces, and the internal implementation is encapsulated and hidden to clients. However, the fact is that the success in applying black-box reuse is a challenge to achieve especially for subcontractors because building a reusable component is much more difficult and time consuming than building usual one.
Although white-box reuse technique is less beneficial than black-box in cost and time, because of the complexity of requirements of embedded system components, it is currently an option for subcontractors in their component-based oriented development. In particular, an existing component will be reused, and its internal structure will be modified if its functionalities fit new requirements.

Subcontractors would, however, successfully achieve component reuse if they took the following considerations:

![Diagram showing system development and component development separation](image)

**Figure 4-2: System development and component development separation**

- The development process at subcontractor should be clarified by separating between system development and component development, as shown in Figure 4-2. More detailed about component development is discussed in Section 4.4.
- There should be a common component model to be applied by both OEMs’ side for integration from subcontractor contributions and subcontractors’
internal development. This model should define what component abstraction should be, and how components interact with other components.

- The component development process should be modeled in an abstract level [7] so that the components can obtain enough their generality to be used or reused in different applications.

### 4.2.2 IEC 61131-3 standard – A CBSE success

IEC 61131-3 is a standard of programming language which is globally used in automation domain. The fact is that it is not only used within programming languages, but this standard can facilitate different solutions such as distributed, integrated, or centralized solutions [9]. In particular, the defined libraries of reusable components help increase hardware compatibility, enhance reusability of application software, and scale up the application. Due to the support of this standard, different program elements are created by different people with different skills during the development phases. In order to acquire such advantages, the definition of Function Blocks has been taken into account. It is worth imagining that the Function Blocks are similar to Integrated Circuits (ICs) [10] which are made to perform the certain specialized functionalities.

In the case of CBSE, the Function Blocks are also considered as the black-box components because each Function Block has “well-defined interface” and “hidden internals” [10]. Thus, the defined Function Block can be (re)used either within the same program, different programs or even different projects. Figure 4-3 represents one of the typical Function Block interfaces for a motion of a motor axis.
Each Function Block has the input variables (on the left side) and output variables (on the right side); the two top variables – Axis – refer to the Through Variables which are the special variables and shared with the other connected Function Blocks – Figure 4-4.

Figure 4-4 shows how defined Function Blocks are connected. These Function Blocks can be selected and placed in the development of a new system as long as they match the interconnection between the events and variables.
In software industry, reusability of software component has been attempted in many ways by software developers in order to achieve benefits. For a complex system, the fact is that one of the ways to perform component reuse is reusing the component’s functionalities once they match requirements of a new project. In this way, a new variant of an existing component will be created and modified to fit the new usage. In this section, we have a brief introduction about the repository layout through which we can get a general view of how components and their variants are stored. At the same time, we suggest a metadata structure which will be integrated in each component in order to hold any relevant information about the component. On the other hand, this component information will help software developers get the knowledge about the component so that they can evaluate the design and test effort when the component is used to adapt the new usage. In addition, we also propose a metadata management model which could be a suitable deployment for the metadata structure.

### 4.3.1 Component variants

A repository contains all existing components and their variants and versions as depicted by Figure 4-5. It is worth noting that component variants are not the same as component versions. A variant of a component is a realisation of a specialized set of requirements, whereas a version corresponds to the same set of requirements. In this thesis, we employ the “Repository Layout” recommended by [6].
The root level of the repository stores all based components from $C_1$ to $C_n$ in a horizontal arrangement. One component can have many variants $C_1 = (C_{11}, C_{12}, ..., C_{1n})$. Each of which has different specialized functionalities or requirements, and once again they are created horizontally and independently. In other words, they have no dependencies with each other. This way of storing can avoid inheritance problem in object-oriented approach, and it aims to separate the variants into independent units for ease of use and maintenance [6]. Each variant can exist in various versions $C_{11} = (C_{111}, C_{112}, ..., C_{11n})$ which represent the same set of requirements. The versions are arranged vertically in conjunction with the variant from which they are created. The later a version is created, the higher version number it has.

From the above repository layout description, we can see that there are two forms of arrangement from which the layout uses to store components, variants and versions. They include vertical arrangement and horizontal arrangement. Each arrangement refers to a guarding condition as shown in Figure 4-5. The horizontal
arrangement refers to Commonality Guard condition. The Commonality Guard condition implies that all of the variants, versions of a certain component must have at least one common requirement. If this guarding condition is not fulfilled, the variants and versions cannot be stored under the same component [6]. The Compatibility Guard condition implies that the new version of a variant should fulfill at least the same requirements as previous version. In other words, it should be backward compatible with the previous version. “If this guard is not fulfilled, the component may be qualified as a new variant; otherwise a new component should be created” [6].

4.3.2 Metadata Unit

Metadata of a component, in general, is data used to describe and manage component information. Using metadata together with component is not a new technique [4]; some component technologies such as JavaBeans, .NET, COM/DCOM have used metadata to describe their component aspects [4]. For instance, BeanInfo object is used together with JavaBeans component to support additional kinds of metadata about component, including component name, textual description of its functionality, textual description of its properties, and so on [4]; in .NET, metadata is used together with an assembly to describe: types of assembly (class, struct, interface, etc.), containing namespace, methods, method’s parameters, and so on. The metadata which is used by such technologies is to serve some specific tasks, and they are lacking generality of the metadata. According to [4], “any software engineering artifact can be a metadatum for a given component”, thus the metadata can be either component models or even component documentation. In particular, they can be embedded in component
at every phase of component development. In our case, metadata should be used to
describe and manage the key information of components and their variants. On the
other hand, they should present the relations between the component’s internals such as
requirements, designs, and test cases so as software developers can get the knowledge
and have a specific view about given components. In addition, the information about
component must be updated to metadata at every phase of component development
cycle.

The suggested structure of metadata as shown in Figure 4-6 comprises
Description, Variant Management, Construction, Dependency, and Work-order.
Figure 4-6: Metadata Structure
**Description** contains high-level information of component. It includes a set of Name, Abstract, and Keywords of the component. This makes it easy to identify the component and its variants, moreover it also facilitates for browsing and searching components in repository.

**Variant management** contains central measurements for components, and the relationships between variants. Central measurement is an entry that can hold any measures of relevance to the users. However, in our application we are interested in keeping track of the followings:

**Variant Relationships contains:**

- Parent: contains an identity of a higher-level component of a current variant. A variant has only one parent. If a variant has no parent, it will be a root component.
- Children: contains identities of the lower-level components of either a current root component or a current variant. A component or a variant can have more than one child.
- Number of variants: keeps track of the number of variants that a component has. Due to this information, we can measure the degree of a component commonality. Typically, the greater number of variants, the more common a component has got.
Construction is the specifications of all components and their variants, and it is the main core element of metadata unit. It includes requirements, designs, and verifications. The requirements of a variant have to fulfill the requirements of its parent component which is recorded in Variant Management part. The design entities can be, for instance, functions, objects, data structures, models, analysis, and analysis results. The verification entities contain test-cases, together with expected results, and actual results from the tests.

Dependency refers to relations among the entities which exist in the construction part of the metadata unit. The role of these relations is important because it supports for estimating the amount of work, and creating specialized variants. Each entity contains links to the requirements. For instance, a certain function can have links back to a functional requirement and a certain timing requirement. The links from design entities to requirements are called “causal relationships” [6], because they exist based on certain requirements. Each test-case has links, called “verify relationships” [6], back to requirements and different elements of the design. Links to the requirements can be seen as black-box tests and such verifications can typically be established before the design begins, while links back to the design can be seen as white-box tests and those are typically created during the design.

Work-order is simply used to estimate the work effort for each component; however, it plays an important role in selecting appropriate component for the new usage, and guiding the work during the development. In the initial process, the work-orders of targeted components enable the developers to select an appropriate component to be adapted by comparing them together. The component will be the
candidate for creating a new variant if it requires least effort to be made in the work-order. In the later process, the development of new variant is performed based on the work-order’s guidance. The work-order displays specific designs and test cases to be reused, modified, and removed in the new created variant, and it also helps to detect unimplemented requirements. This makes it easy for the developers to perform the work more effectively.

Another important role of the work-order is the association with verification process to detect unplanned side effects of the changes. In order to do that, regression testing is applied based on information in the work-order. The results of reused test cases of the changes are allowed if they were defined in the work-order. On the contrary, the other unexpected test results must be investigated again. These unexpected results are usually caused by two possibilities [6]:

- There could be unexpected or unnecessary parts which were changed during the development of the new variant.
- There could be some hidden or undocumented dependencies between the entities. They should be updated and added to achieve a continuous improvement of the decision supporting relations. It may also be useful to store statistics when undocumented dependencies are discovered, to estimate a precision for work-orders.
In this model, we present a metadata management system which facilitates the usage of metadata for the overall software component development. The metadata management model presented in Figure 4-7 consists of Administration tool, Metadata Manager, a set of components called Converter or Plug-ins, a component repository such as CVS, Source Safe or Team Foundation Source Control which will be discussed in Section 4.5, and External tools which include modelling tools and testing tools.

- Metadata Manager is the central element that provides an APIs to which other tools can connect. It can gain access to the metadata associated with components in repository. The followings are its basic functionalities:
  - Searching metadata by keywords.
  - Comparing metadata from different variants.
• Editing the metadata
• Tracing links to find dependencies between design entities, verification entities and requirements.

- The administration tool is a stand-alone application tool that provides possibilities to search, browse, and update component’s information based on the metadata.

- The converters or the plug-ins are components that provide connectivity from external tools to the metadata manager. For example, parsers that read the format of the external tools and use the API provided by the metadata manager to add the information to the metadata.
4.4 COMPONENT DEVELOPMENT PROCESS

Similar to other developments, V-model is currently employed in subcontractor’s system development. However, it has some necessary changes in some processes to suit subcontractor’s situations. Typically, as shown in Figure 4-8, the noticeable change in this model is that the implementation phase has been replaced by “Select and Adapt” phase. Additionally, the development process is separated into system development and component development. The red rectangle in Figure 4-8 refers to the component development process, and reuse activities are applied in this process.

![Figure 4-8: V-model applied in subcontractor development](image-url)
In this thesis work, we focus on the internal component development, and especially we apply metadata, as introduced in above section, to support “Select and Adapt” process, as shown in Figure 4-9. This makes it easy to select a most feasible component to be adapted in new scenario, and this selection process is based on the help of the work-order. In particular, we take advantage of the work-order to estimate the work effort for each appropriate component which is supposed to be suitable for new requirements. And the most feasible component will be selected based on the comparison of the work-orders. If the work-order of a certain component shows least effort to adapt, it will be selected. And again, the adaptation process of selected component will be guided by the work-order. Typically, developers will take a look at the work-order to know which designs and test cases are affected or which designs and test cases will be removed, and so on, so that they can get the full picture of what and how to do. As we mentioned earlier, the work-order also helps to detect unplanned side effects via regression testing process. The component will be (re)tested after being adapted. The unplanned side effects will be detected if the result is not the one that we expected in the work-order; thus, the developers have to find the “hidden” problem and correct it. Figure 4-9 shows the inner steps of “Select and Adapt” process that is currently applied in the development at subcontractors.
Figure 4-9: Component Adaptation Process
4.5 TEAM FOUNDATION SERVER OVERVIEW

Team Foundation Server (TFS) is a “set of tools and technologies” [12] developed by Microsoft. It is a central point to effectively support collaboration for all team members working together during the whole project development. Team members mentioned here include not only the developers, but also the architects, testers, project managers, and so on. Therefore, it plays an important role in the project development. Its key features are summarized as follows [12] [16]:

- **Project Administration** is a tool to support project management for administrators. The administrator can create the new team project, configure role/permission for team members, manage project milestones, and so on.

- **Project portal** is a central point which holds any information about the team project. Team members can have access to view or download project documents, project reports, as well as the announcements about the team project.

- **Version control system** in TFS is similar to other version control systems such as Subversion [13] or Concurrent Version System (CVS) [14]. It is used to store any files or source code and keep track of changes of them. Its tasks include Branching, Merging, Shelving, and Policies.

- **Work Item Management** consists of various related work to be finished in a project. It is used to track the requirements, assign the tasks to each member, manage product bugs, and so on.
• **Builds** are known as Team Foundation Build. Team members can create and manage product builds by following steps: retrieving and compiling codes, running tests, releasing the build to the server, publishing the build report, and updating Work Items.

• More features and information about TFS can be found in [12].

In this thesis, we use a part of Microsoft Team Foundation Server as a repository to store components. It is Version control system. Team Foundation Version Control (TFVC) System is one of the core elements of TFS. In the case of software components, it is viewed as a repository to store components and manipulate the operations over the components. The components are stored as the directories under their project, and the implementations or source codes of a component are stored as the files under the directory. TFVC provides some operations to work with the components as follows:

• Check-in and check-out: when a team member wants to modify a file or a source code, he has to check out this file to his local computer. When he has done the changes on this file, he checks it in the repository. Checking out action is like making a copy of a file or files in a directory/component from server to local computer, as shown in Figure 4-10. TFVC handles check-in and check-out in two models. Exclusive model which is also known as lock-modify-unlock model [16]. In this model, only one person can perform check-out the file at a given time. The other one cannot perform check out this file until it is checked in back to the repository. The second model is shared model which is also known as copy-modify-merge model [16], and it is the default model in TFVC. In this model,
multiple people can perform check out the same file at the same time. The system will check the latest status of the file when it is checked in. Typically, when a team member check in this file, the system will check whether or not the status of the file was latest before he performed check out. If it was not latest, the system will merge his changes with the latest one.

**Figure 4-10: Check-in and Check-out a source code file**

- **Branching** is used to work with a file in an area where is separated from the old file’s area. Its purpose is to create an area to fix the bug of the certain code. However, it is also used to create a new variant from an existing component during the “Select and Adapt” process which was mentioned in above section.

- **Merging** is used to merge changes between two branches or two versions (as mentioned in check-in and check-out shared model). In particular, this action can move changes from a branch to a folder from which the branch came or vice versa.
• **Shelving** enables a team member to temporarily save the changes of file(s) or source code to the server without any influence with the original one. For example, a team member is adding the new functionalities for a component. His colleague calls and asks him to send those new functionalities to check. He cannot check in an incomplete and untested implementation to the repository. Instead, he “shelves” the source code to the server, and his colleague can “unshelve” it. Besides, there are some other reasons to use shelving such as backup the incomplete work and bug fix. The shelving and unshelving operations are depicted in Figure 4-11.

![Figure 4-11: Shelving and Unshelving](image)

The features mentioned above are the main features of TFS which will be involved in this thesis project. Since TFS infrastructure is client-server, the thesis project needs a way to connect to the server/repository to work with the
components. Fortunately, Microsoft provides a full set of APIs in order for programmers to interact with. It is called “Microsoft Team Foundation Sever SDK” [17]. First things first, we need to know some fundamental notions as well as steps that we should bear in mind when working with TFS SDK (in this thesis project we use a suite “Microsoft Visual Studio 2005” with C# programming language):

- The first step to interact with APIs is to import necessary DLL files to the project. These files can be found in TFS SDK installed folder. They include:
  - Microsoft.TeamFoundation.Client.dll
  
  And declare their namespace on the top of the class:

```csharp
using Microsoft.TeamFoundation.Client;
using Microsoft.TeamFoundation.VersionControl.Client;
```

- In order to have access to the repository, each team member has to have permission to connect to a valid server which consists of a server name, a protocol, and a port [15]. The code to establish connection to repository is stated in Appendix A.1

- In order to perform the necessary operations on TFS, each team member should have at least one workspace. Workspace is just a mapping between a local folder
at client’s computer and a folder of a working project in the repository. When we check out a component’s file, due to the workspace, TFVC knows where it should store that file in the local computer. A team member can have more than one workspace if he or she has either more than one project or more than one local computer. However, there are some rules for creating workspace [16]: We cannot map two local folders in a workstation of computer to the same working project’s folder in the repository because TFVC will not know which local folder to store the files. Also, we cannot map one local folder to two working project’s folders. Appendix A.2 and A.3 shows how we select a workspace or how we create a new workspace.

- Those are the necessary steps to start working with TFVC. From here we can perform operations such as check-in/check-out, or create new variant for the component. Please note that the check-in operation will be made if the checking file/folder has a change set. This change set, in TFVC, is called “Pending Change”. These operations can be found in Appendix A.4, A.5, and A.6.
5.0  SUGGESTED SOLUTION

In this thesis project, we are going to build a tool set to support component development processes for subcontractor projects. According to the metadata management model, which was suggested in Section 4.3.3, elements to be implemented are Administration Tool, Metadata Manager, and Converters or Plug-ins. The tools do not change or interrupt the processes of component development. Rather, they are integrated into every development phase to facilitate building the component. As illustrated in Figure 5-1, there are two important steps that are taken during the development cycle. We call them “Entrance” and “Exit” steps. The development model in Figure 5-1 refers to the “Select and Adapt” process – as we mentioned earlier in Section 4.4. In this model, the Entrance step refers to the “Select” process where the software designers will analyze new requirements, select appropriate existing components, setup work-order, and so on; whereas the Exit step will be taken after the adaptation process. In this step, as shown in Figure 5-2, the developers will be able to set the new dependencies, evaluation the tests, and so on.
The **Administration Tool** in a metadata management model will be mostly used in both Entrance and Exit steps because it is created to support administrative work for components such as searching or browsing components and variants, comparing the component specifications, setting up the work-order, evaluating the tests and so on. On the other hand, it is also used to support other processes such as requirement management, designing or testing process. Typically, it can add new requirements, designs, or verifications, modifying entity’s information, and delete the
entities. The aim of this feature is to prevent the problem caused at the plug-ins. For example, the plug-in at design process cannot add a new design entity to the metadata.

The role of the converters/plug-ins is depicted in Figure 5-2. Each plug-in is seen as a component which is integrated in the development tools such as design tool like UML, and coding tool like Microsoft Visual Studio, and so on to hold any information of the components. The plug-ins will save this information in each component metadata.

The Metadata Manager element will be built as an independent component. It is a DLL file, and it offers built-in interfaces through which the administration tool and the plug-ins communicate.

![Figure 5-2: A specific view of component development with metadata management tool support](image-url)

---

42
Because of the limitation of time and the lack of integration support from the development tools, we are not going to build the plug-ins for this tool set. Instead, we are going to build the Metadata Manager (DLL) component, and the Administration Tool as a stand-alone application. On the other hand, the Metadata Manager component will be designed to support the plug-in integration in the future development.

5.1 USE CASE

![Diagram of Use Case for Entrance, Exit, and Plug-in Actors]

*Figure 5-3: Use Case for Entrance, Exit, and Plug-in Actors*
Figure 5-4: Classes Relationships
<table>
<thead>
<tr>
<th>Metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Keyword: string</td>
</tr>
<tr>
<td>+Abstract: string</td>
</tr>
<tr>
<td>+Name: string</td>
</tr>
<tr>
<td>+ID: string</td>
</tr>
</tbody>
</table>

+AddEntity(RequirementObj: Requirement)
+AddEntity(DesignObj: DesignEntity)
+AddEntity(VerificationObj: Verification)
+AddEntity(MeasurementObj: Measurement)
+RemoveEntity(RequirementObj: Requirement): void
+RemoveEntity(DesignObj: DesignEntity): void
+RemoveEntity(VerificationObj: Verification): void
+UpdateEntity(OldRequirementObj: Requirement, NewRequirementObj: Requirement)
+UpdateEntity(OldDesignObj: DesignEntity, NewDesignObj: DesignEntity)
+UpdateEntity(OldVerificationObj: Verification, NewVerificationObj: Verification)
+UpdateEntity(OldMeasurement: Measurement, NewMeasurement: Measurement)
+GetComponentIDNumber(): int
+GetDependencies(): ArrayList
+GetDependencies(RequirementObj: Requirement): ArrayList
+GetDependencies(DesignObject: DesignEntity): ArrayList
+GetDependencies(VerificationObject: Verification): ArrayList
+GetDependency(From: string, To: string): Dependency
+GetDependencyIDNumber(): int
+GetDesignEntities(): ArrayList
+GetDesignIDNumber(): int
+GetMeasurement(): Measurement
+GetMeasurementIDNumber(): int
+GetRequirements(): ArrayList
+GetVerifications(): ArrayList
+GetWorkOrder(): WorkOrder
+GetWorkOrder(): WorkOrder
+RemoveDependency(From: string, To: string): void
+RemoveWorkOrder(): void
+SetDependency(DependencyObj: Dependency): bool
+SetWorkOrder(WorkOrderObject: WorkOrder): bool
+UpdateDependency(OldDependencyObj: Dependency, NewDependencyObj: Dependency)
+UpdateWorkOrder(WorkOrderObject: WorkOrder)

Figure 5-4-1: Detailed Metadata class
<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Connect_to_Repository(ServerName: string): void</td>
</tr>
<tr>
<td>+Branch(SourceDir: string, TargetDir: string): void</td>
</tr>
<tr>
<td>+GetDefaultWorkspace(userWorkstation: Workstation): Workspace</td>
</tr>
<tr>
<td>+Get_Currently_Working_Workspace(MappedPath: string, serverPath: string)</td>
</tr>
<tr>
<td>+Checkout(FileName: string): void</td>
</tr>
<tr>
<td>+CheckIn(FileorFolderPath: string, Comment: string)</td>
</tr>
<tr>
<td>+AddNewMetadataFile(FileName: string, NewMetadataInfo: Metadata)</td>
</tr>
<tr>
<td>+Search(Keywords: string): ArrayList</td>
</tr>
<tr>
<td>+Create_Workspace(WorkspaceName: string, ServerName: string, LocalPath: string, Comment: string): void</td>
</tr>
<tr>
<td>+Get_Selected_Workspace(workspaceName: string): void</td>
</tr>
<tr>
<td>+GetChildBranches(workingDirectory: string): List&lt;item&gt;</td>
</tr>
<tr>
<td>+GetComponentInfo(ComponentFolder: string): Metadata</td>
</tr>
<tr>
<td>+GetHighestComponentIDNumber(): int</td>
</tr>
<tr>
<td>+GetParentBranches(workingDirectory: string): List&lt;item&gt;</td>
</tr>
<tr>
<td>+UpdateComponentInfo(OldComponent: Metadata, NewComponent: Metadata, xmlFilepath: string): void</td>
</tr>
</tbody>
</table>

**Figure 5-4-2: Detailed TFS class**
Figure 5-4-3: Detailed Requirement, DesignEntity, Verification, Measurement, Dependency, and WorkOrder classes
6.0 DEMONSTRATION

In this demonstration part, we borrow a scenario from [18] to experiment this project on the real context. This scenario is to build a software system for a library.

6.1 DESCRIPTION

In this system, the librarian can add a new book, reserve a book, loan a book, return a book, and search a book. All of these operations must require login. They are depicted in following use case diagram:

![Use case diagram for library software system]

Figure 6-1: Use case for library software system
There are six requirements in this system. They are summarized in the below table.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>Log in</td>
<td>The librarian has to login before perform any actions.</td>
</tr>
<tr>
<td>R2</td>
<td>Add New Book</td>
<td>The librarian will be able to add new books to the system.</td>
</tr>
<tr>
<td>R3</td>
<td>Reserve Book</td>
<td>The librarian will be able to make a reservation of book(s) for student.</td>
</tr>
<tr>
<td>R4</td>
<td>Loan Book</td>
<td>The librarian will be able to make a loan of book(s) for student.</td>
</tr>
<tr>
<td>R5</td>
<td>Return Book</td>
<td>The librarian will be able to return book(s) when the book(s) have been returned.</td>
</tr>
<tr>
<td>R6</td>
<td>Search Book</td>
<td>The librarian will be able to search book(s).</td>
</tr>
</tbody>
</table>

The system is modeled into two components:

- One component is for the login
- Another one is for the book operations: add, reserve, loan, return, and search.

On the other hand, the system includes three design entities with respect to 2 components. One entity is for login component. The other two entities are for Books Operations component. They are described by following table:
<table>
<thead>
<tr>
<th>Component ID</th>
<th>Requirement ID</th>
<th>Design ID</th>
<th>Design Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>R1</td>
<td>D1</td>
<td>Design for login</td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td>D2</td>
<td>Design for book operations: Add, Reserve, Loan, and Return.</td>
</tr>
<tr>
<td>C2</td>
<td>R3</td>
<td>D2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R4</td>
<td>D2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R5</td>
<td>D2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R6</td>
<td>D3</td>
<td>Design for Searching</td>
</tr>
</tbody>
</table>

In verification process, we assume that three white-box test cases are used to test corresponding three design entities:

<table>
<thead>
<tr>
<th>Component ID</th>
<th>Requirement ID</th>
<th>Design ID</th>
<th>Verification ID</th>
<th>Verification Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>R1</td>
<td>D1</td>
<td>V1</td>
<td>Testing login</td>
</tr>
<tr>
<td></td>
<td>R2</td>
<td>D2</td>
<td>V2</td>
<td>Testing book operations</td>
</tr>
<tr>
<td>C2</td>
<td>R3</td>
<td>D2</td>
<td>V2</td>
<td>Testing book operations</td>
</tr>
<tr>
<td></td>
<td>R4</td>
<td>D2</td>
<td>V2</td>
<td>Testing book operations</td>
</tr>
<tr>
<td></td>
<td>R5</td>
<td>D2</td>
<td>V2</td>
<td>Testing book operations</td>
</tr>
<tr>
<td></td>
<td>R6</td>
<td>D3</td>
<td>V3</td>
<td>Testing searching book</td>
</tr>
</tbody>
</table>
6.2 CREATE METADATA FOR COMPONENTS

Initially, we assume that the components were already built, and they are currently storing in the repository. However, they have not had any metadata to manage their information. The task in this scenario is creating metadata for the two components so that the developers will be able to get knowledge about them in the later projects.

We use the Administration Tool which was mentioned in Section 6 and 9. On the left pane of the tool - Figure 6-2, it shows the working project and its components. In this case, the components are Book Operations and Login.

![Figure 6-2: Administration Tool shows components in the repository](image)
We add all of component information to metadata. See Figure 6-3.

Figure 6-3: Entities information for Book Operations Component
Setting up dependencies among entities.

Figure 6-4: Entities Relationships for Book Operations Component
6.3 NEW REQUIREMENTS

Those are components’ artifacts for a library software system. Now we would like to expand the system with new and improved functionalities. In this expanded system, the librarian will be able to print the report with detailed information of loan and return. On the other hand, we want to improve Search Book functionality.

After analyze the existing components in the repository together with these new requirements, we found that we can reuse login component; we will make some changes in Book Operations component; and we will add a new requirement to the new component variant.

Figure 6-5: Identify affected entities by graphical dependency
Requirements from R1 to R4 do not get involved in new development, so we can reuse them. From dependency in Figure 6-5, we also reuse D1 and V1. On the other hand, R5 will be change. This causes D2 and V2 changed. We set up a work-order for this candidate. It is worth noting that if we have located appropriate several components, we have to set up the work-orders for all of them so that we can compare the work effort based on their work-orders. The most appropriate component will be selected if it has least work effort shown in the work-order. In this case, we just have one appropriate component. Therefore we can skip the comparison step. Rather, we will create a variant from this component and go to the adaptation process (adding new functionality and making some changes in the new variant).
Figure 6-6: Work-order for candidate
Figure 6-7 shows that a new variant named “Book Operation Variant” has been created. We modified its requirement R5, and this modified requirement has been stored as a new requirement R6. According to the dependency graph or the work-order, we see that once we modify R5, D2 and V2 will be affected. Therefore, we modified D2 and V2, and they are stored as D3 and V3, see Figure 6-6.

On the other hand, we also added a new requirement R7, and of course we have to add one new Design (D4) and one new Test Case (V4) for R7.
Now the new dependency would look like this:

![Figure 6-7: New dependency](image)

The next step is setting up the work-order for the new variant. This work-order plays a role as work guidance. It will be used throughout the variant development processes. In Figure 6-8, we see that we will reuse D1 and V1; we will change/modify D2 and V2 (now they are D3 and V3). Uncovered requirement means that it is the new added requirement, and its design and test case will be built from scratch.
Figure 6-8: Work-order for new variant component

This work-order also says that there is only V3 test result which is allowed to be changed during testing phase. If the test result of V1, for example, is changed, there could be some unexpected changes or hidden dependency during the development of the new variant.
7.0 THESIS EVALUATION

In this part, we evaluate this thesis project by experimenting it on a real component at CC-Systems AB – a subcontractor company. This experiment aims to prove the necessary contribution of the tool at the early phase of project development.

7.1 REALISTIC APPLICATION

- We build a software component, called BECU, for a car’s brake system. This component is used to read pedal sensors’ signal; it receives messages from TECU for two axles’ brake load, and dynamic brake capability. And finally, it calculates and sends actual mechanical brake load to the actuator. At the same time, we use our tool to store all of the information of this BECU including the requirements, designs, test cases, test cases’ results, and their dependencies.

- We analyze new BECU requirements of a construction vehicle’s brake system. Due to the information provided by the tool, we know that some parts of previous component can be reused in the new project. We create a new variant of the old BECU; we identify reused parts of previous component, affected parts when we modify some old requirements to fit the new BECU’s requirements. We create new dependencies for them, and finally we set up the work-order.
Finally, we apply regression testing of the new BECU based on the work-order.

7.2 FEEDBACK

We set up a meeting with CC-Systems staffs for this thesis demonstration. They included 2 managers, 4 developers, and 1 sales director. During the meeting, we had an open-ended discussion for collecting the feedback:

“This can be used to browse our repository to see what components we have, and what basic support they have.”

Yes, the tool with built-in TFS plug-in will easily support browsing and selecting existing components in the repository.

“This tool can be used to estimate the complexity of a planned change. It is good that it covers the whole process from requirements to test.”

Indeed, metadata holds all of the necessary information (XML format) from the beginning of component development, and the Administration Tool shows that technical information into readable format. Hence it makes it easy to estimate complexity of planned change, and unplanned side-effects which are detected during regression testing process.

“The dependency graph can be used as illustration of the coverage of our test cases. Such graphs are extremely valuable, when impact analysis of a planned change of an old software component is about to be done.”
The dependency graph and the work-order are two major elements of this tool. The internal construction of a component shown in the graph will completely support modification/change management. Whereby, we cover the test cases which will be applied for regression testing after a modification has been made.

“There is a need for integration of many of the tools that we use in the engineering; a concrete example is Starteam and Caliber. The metadata could do it.”

Because of the nature concept of metadata, we can use it with any other tools. In the future of next releases, there would be a function called “import” and “export” metadata so that we can integrate metadata itself to the other tools which support the same format.

“This is good for impact analysis of change requests.”

“Very valuable from sales since it might finally mean that we can get some precision in estimating the amount of work changes.”

Developers can analyze and estimate the impact of change by look at the dependency graph and especially the data shown in the work-order of component.

“You ideally should work with as few tools as possible when you work, requirements, design, test cases. This tool might only be another tool that you have to use.”

If this tool has a real deployment, it will be present and employed in every phase of component development. However, it will not prolong and interrupt the development; rather it can save a lot of time because internal information and developing
Information of components are shown, hence it helps the developers fasten their work.

“It definitely leads to better documentation, but that of course will come to an effort.”

At first, users will do pretty much manual work to document their components. However for the new components, such manual work will be significantly reduced because every component information will be automatically updated at every phase of its development by plug-ins. Unfortunately, the plug-ins are not built in this thesis project because of the complexity of development tools such as design tools, coding tools, and testing tools in which the plug-ins will be integrated.

“The dependency graph can be very complex to build.”

It could be complicated for big components. We have tested the graph with a real small component, and it worked very nicely.

“When you find a bug, the tool gives traceability between generations so that you can see where the bug also exists.”

For the time being, the bug could be traced back to earlier generations by looking at the variant tree shown in the Administration Tool (right click on a component and choose properties). In this tree, we can see the parent component(s) of a component on which we have found the bug.
Using the metadata suggested above could boost the component development in embedded domain because it is not only the documentation of the components but also the work guidance for the developers to keep track of the work during the development. At the same time, the tool set suggested above will not change or interrupt the component development processes. Rather, they are integrated into every development process to facilitate building the component. In particular, the Administration Tool will help the designers review and analyze the existing components with respect to the new requirements. On the other hand, it is also seen as a decision support tool in which the work-order will support the designers to decide which component will be the most appropriate candidate to be adapted in the new usage. However, one of the difficulties that we have seen is building the converter or the plug-in because it has to be integrated in the developing tools such as modeling tool or testing tool in order to update the component information in each development process. These tools should be open enough in order for the plug-in to be imported.
Establish connection to Team Foundation Sever

private static TeamFoundationServer m_TFSServer = null;
private static VersionControlServer m_TFSVersionControl = null;
private static Workspace m_TFSWorkspace = null;
private static bool isValidated = false;
private static string m_serverName = "";

public void Connect_to_Repository(string protocol, string tfsserverName, int port)
{
    try
    {
        m_serverName = String.Format("{0}://{1}:{2}" , protocol, tfsserverName, port);
        m_TFSServer = TeamFoundationServerFactory.GetServer(m_serverName);
        m_TFSServer.Authenticate();
        m_TFSVersionControl = (VersionControlServer)m_TFSServer.GetService(typeof(VersionControlServer));
        isValidated = true;
    }
    catch
    {
        throw new TFSException("Cannot connect to " + protocol + "://" + tfsserverName + ":" + port);
    }
}
A.1 Select a workspace

```csharp
public static void Get_Selected_Workspace(string workspaceName)
{
    // Get local computer information
    Workstation myWStation = Workstation.Current;
    // Get all existing workspaces
    WorkspaceInfo[] WorkSpaces = myWStation.GetAllLocalWorkspaceInfo();
    foreach (WorkspaceInfo aWSP in WorkSpaces)
    {
        if (aWSP.Name == workspaceName)
        {
            m_TFSWorkspace = aWSP.GetWorkspace(m_TFSServer);
        }
    }
}
```
A.2   Create a new workspace

    public static void Create_Workspace(string WorkspaceName, string ProjectName, string LocalPath, string Comment)
    {
        try
        {
            m_TFSWorkspace =
            m_TFSVersionControl.CreateWorkspace(WorkspaceName, m_TFSServer.AuthenticatedUserName, Comment);

            //Map current project to local path
            m_TFSWorkspace.Map(ProjectName, LocalPath);
            m_TFSWorkspace.Get();
        }
        catch (Exception ex)
        {
            throw new TFSException(ex.Message);
        }
    }

Note: “ProjectName” parameter is the name of the working project. It also includes the path of this project’s folder on the server/repository. The format of the project name always starts with “$” sign. For instance, “$/SampleProject”.

To get all of project names existing on the server, the code would look like this:

```csharp
VersionControlServer VCServerObj = new VersionControlServer();
TeamProject[] projectcollection = VCServerObj.GetAllTeamProjects(true);
```

If we have a local path, we can get the project for this path. The following code is for getting a specified project from, for example, “C:\LocalSampleProject” path.

```csharp
TeamProject aProject = m_TFSWorkspace.GetTeamProjectForLocalPath("C:\LocalSampleProject");
```
A.3 Check-out

```csharp
public void Checkout(string WorkingFile)
{
try
{
    //Perform checking out
    m_TFSWorkspace.PendEdit(WorkingFile);
}
catch
{
    throw new FileNotFoundException("Metadata File cannot be found in this folder");
}
}
```
**A.4 Check-in**

```csharp
public void CheckIn(string FileorFolderPath, string Comment)
{
    PendingChange[] PC;
    try
    {
        //Get change set for checking file or folder
        PC = m_TFSWorkspace.GetPendingChanges(FileorFolderPath,
                                                RecursionType.Full);
    }
    catch
    {
        throw new FileNotFoundException("Metadata File cannot
                                           be found in this folder");
    }
    if (PC.Length != 0)
    {
        //Perform checking in the change set with comment
        m_TFSWorkspace.CheckIn(PC, Comment);
    }
    else
    {
        throw new ApplicationException(string.Format("No pending
                                                         changeset found"));
    }
}
```
public void Branch(string SourceDir, string TargetDir)
{
    // SourceDir or TargetDir should be a server path, for example 
    // "$/Simtech_CMMI/..."

    try
    {
        m_Workspace.PendBranch(SourceDir, TargetDir, VersionSpec.Latest);

        CheckIn(TargetDir, "");
    }
    catch (Exception ex)
    {
        throw new TFSException(ex.Message);
    }
}
REFERENCES


[3] Ivica Crnkovic, Magnus Larsson – A Case Study: Demands on Component-based Development – ACM 23000 1-58113-206-9/00/06.


[9] PLCopen Standardization in Industrial Control Programming: Creating Reusable, Hardware Independent Motion Control Applications via IEC 61131-3 and PLCopen Function Blocks for Motion Control – Eelco van der Wal, PLCopen, Zaltbommel, The Netherlands, evdwal@plcopen.org.


[12] Team Foundation Overview


[15] Team System
http://msdn.microsoft.com/msdnmag/issues/07/01/TeamSystem/default.aspx


[17] Microsoft Team Foundation Sever SDK

