Model-based and component-based development of embedded systems

Master Thesis, D-Level

Author:
Rumen Vladimirov Kyusakov

Mälardalen University

School of Innovation, Design and Engineering

Supervisor:
Tomas Bures
tomas.bures@mdh.se

Examiner:
Ivica Crnkovic
ivica.crnkovic@mdh.se

June 19, 2008

Abstract

Although the component-based software engineering is proven to be very successful in enterprise and desktop applications, it encounters some difficulties when applied to development of embedded systems. Specific requirements like execution time, memory footprints, predictability etc., also known as extra-functional system properties, makes it difficult to use available component models for real-time and safety-critical applications. That arouses a need of domain specific, software-component approach for developing embedded systems. Defining methods and tools for utilizing this approach is one of the main goals of the PROGRESS research centre.

This thesis is focused on building a repository for reusable components as part of ProSave Integrated Development Environment (IDE) – a framework containing developed within PROGRESS tools. The primary goal of this repository is to advance components' reuse by providing features like storage, versioning and support for multiple clients and concurrent connections. Different technologies can be selected for actual implementation of the repository and even different architectures within the same technology. Each of these scenarios leads to particular constraints and limitation on the system more or less concerning fulfillment of its required functionality. After evaluation of these design scenarios regarding our repository concept, a version control system was chosen for data storage; in particular Subversion and for a persistent layer implementation a Java library SvnClientAdapter was used.

Table of Contents

1. Introduction

Traditional practice in software development denotes the compiling of a complex, coupled source code with built-in dependencies. As a result, the process of changing program logic, or adding new capabilities, becomes factual trouble. Developers have to modify the primary source code, go over testing again to ensure correctness of changes and recompile application. Drawbacks of this approach are best described in one prominent example [2] – imagine if the car axis had to be modified and because of that some other parts of the chassis, too, just because the tires need changing with high-performance ones. That would cost a lot of money and efforts in conjunction with unpredictable behavior. However the car tires, as well as all other machine parts, are standardized. This allows a specific tire model to be used in a wide variety of vehicles and vice versa – a specific vehicle can use different tire models without the need of modifications. Like their mechanic analogues, software components reduce the complexity of the software systems and provide means for reusing the existing source code. Contrary to the traditional approach, component-based development separates development of components from development of systems. This way a software system is being built using pre-existing components. The same as the machine parts example, component-based development requires all components in the system to be well-specified and to comply with a common standard – that is a formal set of rules defining interactions and composition principles [3] also known as component model. For different software domains, different component models must be applied. Having a suitable component model and a set of well specified components on hand is in fact not sufficient to facilitate component-based development process. Looking back on the machine parts analogy, imagine that a bolt must be removed; a specific tool is certainly needed i.e. a wrench, and both of them the bolt and the wrench have to comply with common standard for example metric or "English standard." So the moral of this comparison is that in order to take advantage of component-based software development there is a need of a right tool or better the right set of tools. This set should include tools for assisting the component creation like text editors, graphical designers, testing tools etc. as well as tools for assembling the components into a system like architecture viewers, model viewers and editors, and analysis tools. Another important tool is a component repository which is storage area for components providing sharing, addition and browsing of the saved components. It promotes components reusability and facilitates the development process.

In order to successfully apply the component-based software engineering to a particular domain e.g. desktop, enterprise, web applications, embedded systems, a specific

component technology should be used. It includes a component model and a set of tools which have to fulfill the requirements derived from the software domain. Since the embedded systems are very important for Swedish industrial sectors, a strategic research centre PROGRESS [13] has been established by Mälardalen Real-Time Research Centre. The key objective of POGRESS is to apply a software-component approach to engineering and re-engineering of embedded software systems by providing theories, methods and tools. As part of the PROGRESS research, ProSave Integrated Development Environment is being developed. It is a programming environment containing variety of tools for development of real-time software systems. One such tool is a component repository – a means for storing and sharing components.

1.1. Purpose

Promoting reusability of the real-time components is the cornerstone of today's efforts in embedded software engineering. In order to address this issue, common standards and a dedicated component model are created within the PROGRESS project. Another area where the reusability can be enhanced is in the component-based development process which ProSave IDE is designed to facilitate. Besides tools for assisting components creation, ProSave IDE is supposed to provide means for storing and sharing them. This functionality is presented by a component repository which creation is the aim of this thesis. When designing and implementing the repository, the thesis investigation should take into consideration the following aspects:

- The repository must be available for remote and concurrent access
- The components can have many versions which need to be stored
- Integration with the ProSave IDE should be provided

1.2. Conventions and thesis organization

This section covers the various conventions used throughout this paper.

Typographic conventions:

`Constant width font` is used for source code.
Italic fond is used for terms, when they are defined.
Italic bold font is used for diagrams' captions.

Acronyms:

PROGRESS – strategic research centre funded by the Swedish Foundation for Strategic Research

IDE – Integrated development environment
API – Application programming interface
VCS – Version control system
OSGi – Open Services Gateway Initiative
OS – Operating system

The thesis is organized in the following sections:

1. *Introduction* – Points the basic advantages of using component-based software engineering and the need of tools during the development process. Motivation of the thesis investigation is also provided.

2. *Background* – Provides concise review of the needed theories and technologies behind the thesis work.

3. *Software requirement specification of the component repository. Problem formulation* – Describes the expected functionality of the system and the constraints concerning its implementation.

4. *Repository versioning policy* – Explains how the prospective system will handle the component versioning process.

5. *System architecture* – Covers the first steps in the system implementation – identifies the system components and their interactions.

6. *Implementation process* - Describes the most important steps in the system implementation along with the problems emerged, and the investigation of different methods to solve them.

7. *Overview of the implemented system* – Describes in details the implemented system.

8. *Related work* – Presents a brief survey of the work done in the area and how this thesis fits in it.

9. *Conclusion and future work* – Summarizes the work done in this thesis and provides hints for possible improvements.

Appendix A: Use cases – Describes in details the system functionality using UML activity diagrams.

Appendix B: User manual – Provides instructions for using the system.

2. Background

As the scope of PROGRESS research is a vehicular, telecom and automation domain, specialized component model has been designed – ProSave. It shaped the component concept and defines how components can be combined to create a system [4]. The thesis work includes a study of the component model, because the relationships and dependencies between ProSave components are of prime importance for correct implementation of the repository operations on components.
Based on the ProSave component model, ProSave Integrated Development Environment is currently being developed. Like his predecessor SAVE IDE, ProSave IDE uses Eclipse platform which is specially designed for building Integrated Development Environments (IDEs) and arbitrary tools. Since the component repository must be part of the ProSave IDE, the thesis investigation examines the question of Eclipse platform contributions.

The next sections provide brief descriptions of the ProSave component model, Eclipse platform and basic concepts of the version control systems as they are essential for the subsequent investigation.

2.1. ProSave component model

ProSave is a simple component model designed for low-level component-based development of vehicular systems. It has been developed from the very beginning to facilitate analysis and synthesis [6] hence providing a way to define a system's extra-functional properties in design time. ProSave component model is based on a pipes-and-filters architecture and it strictly separates data transfer and control flow.
The target system is considered as a set of subsystems which communicate to each other asynchronously using messages. A subsystem consists of ProSave components, clocks, sensors and actuators. It has its own threads of execution and uses its input and output message ports to send and receive data from other subsystems.
A ProSave component is an encapsulated, reusable piece of functionality. It has input and output ports used to connect the component to other components or subsystem elements. A port can be used either for data transfer or for control flow, thus it is a data port or a trigger port, respectively. Each port is part of exactly one interface and each interface is part of exactly one service. Interfaces and services present a component's external view in more comprehensible way.
Another important aspect of the ProSave components is their attributes. They are used to store information about component properties, for example, execution time or resource consumption.
There are two types of ProSave components: primitive components which provide

desired functionality using source code and composite components which consist of subcomponents, connections and connectors. A connection is a directed link between two ports – either input data port to output data port or input trigger port to output trigger port. Connectors on the other hand are used to adjust data flow and control flow. Some examples of connectors are “Data fork” which splits a data connection to several others. “Selection” forwards the incoming trigger to a particular path depending on a specific condition etc.

2.2. Eclipse platform

Eclipse is a Java-based, extensible open source development platform. One of the main features of the Eclipse platform is its mechanism for discovering, integrating, and running modules called plug-ins, which are in turn represented as bundles based on the OSGi [http://osgi.org] specification [5]. Except for a small kernel known as the Platform Runtime, all of the Eclipse Platform's functionality is located in plug-ins.

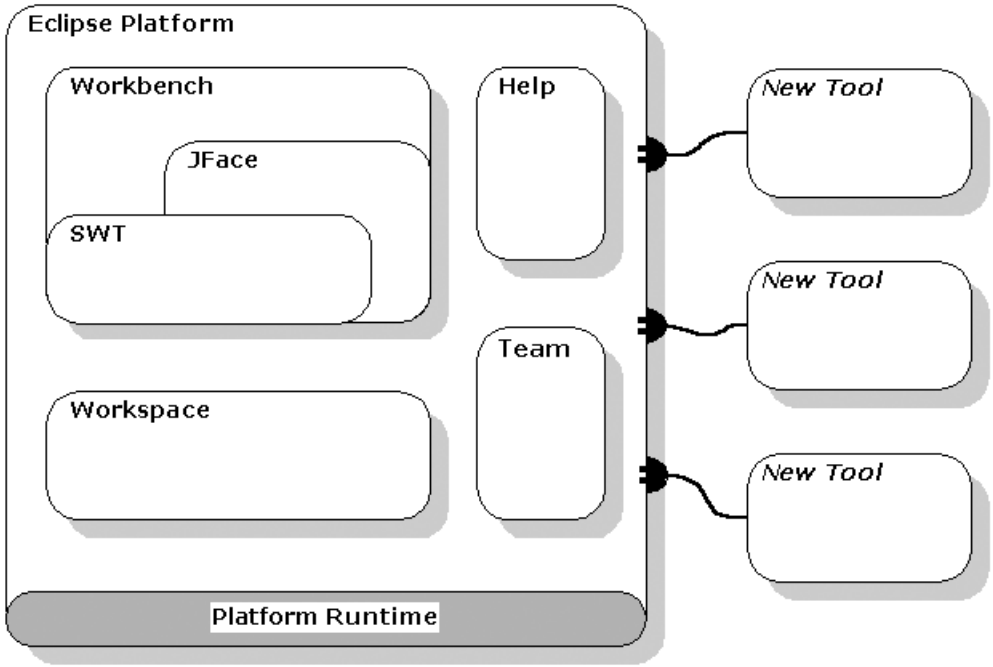


Figure 1 - Eclipse platform

The above picture shows the major components of the Eclipse Platform. Each supplementary tool which adds functionality to the platform is written as single or several plug-ins. Plug-ins are coded in Java. A typical plug-in consists of Java code in a JAR library, some read-only files, and other resources such as images, web templates, message catalogs, native code libraries, etc [7]. The Eclipse platform provides generic

functionality which can be easily extended as in the ProSave IDE. However not all of its components are really needed when working in the ProSave development environment. For example Ant tool used for automatic Java builds or Java debugger can confuse prospective ProSave users. High quality software should include only this functionality which complies with its intended use. Fortunately, Eclipse is flexible enough and it offers a mechanism for discarding unused plug-ins. Applications which make use of this mechanism is called Rich Client Application and the minimal set of plug-ins needed to build them is collectively known as the Rich Client Platform or RCP. The ProSave integrated development environment is intended to use RCP in order to provide simple and intuitive user interface.

2.3. General version control concepts

Version control is a system which records changes to arbitrary source files and resources, manages releases, and controls access to shared files [9]. It is a mandatory part of every big software project where a large number of software developers work on the same data. Version tracking means that the version control system makes it possible to retrieve any historical version of any stored file or even a state of the files at some moment in time. Most VCS rely on a central storage called *repository* that saves all the information about file changes, including the time and users making these changes. The repository stores this information in the form of a file system tree — a typical hierarchy of files and directories. Any numbers of clients connect to the repository, and then read or write to these files. By writing data, a client makes the information available to others; by reading data, the client receives information from others [10].
In order to start working with the data in the repository a user needs to create a *working copy* on his local machine. This is a directory where the local modifications of the data take place. The operation of downloading files and directories from the repository to a working copy is called *check out*. In reverse, operation of publishing user's changes to repository is called *check in* or *commit*. Different VCS use different strategies to achieve collaborative editing and sharing of data. The main problems when many people work with a same set of data are collisions in their changes. There are two ways of addressing this – the Lock-Modify-Unlock solution and the Copy-Modify-Merge solution. In the first model, the repository allows only one person to change a file at a time. This is usually done by creating a lock every time a user starts modifying the file. Other users must wait for the editor to release the lock before they can edit the file. On the other hand, the Copy-Modify-Merge model allows many users to work simultaneously and independently on a same file. Finally, the private copies are merged together into a new, final version. If two or more private copies modify the same part of the file in different ways a conflict occur. In this situation, the users are responsible for correct file merging. This mechanism proves to be very useful in practice and it somehow increases a team's

productivity. This is especially true when working with source code files and other line-based text files. However, files in a binary format like pictures and sound files are almost impossible for merging and in this case using locks for ensuring serialized access is preferred.

3. Software requirement specification of the component repository. Problem formulation

Component-based development of an embedded system is a process which includes many activities; starting with the system architecture which draws the big picture of the prospective software system and divides it into smaller parts or subsystems; the designing phase where all necessary components are identified regarding requested functionality and execution environment. Then, the components are being developed, either from scratch or reusing existing ones, and assembled together during the implementation phase which also includes testing. The development of the system can take place in one project or it might be spread across several projects with different teams work on them. Against the background of this working environment, the ProSave IDE repository must serve as common sharing facility for all concerned with components elaboration within the organization. There can be several systems being developed and even teams only responsible for development of components. Each team's collaborative work is kept in a regular version control system where the frequent modifications evolve the line of development of the ProSave components. When a particular component is complete the relevant team can share it with other teams by exporting it to the ProSave IDE repository. In this way, the developers from other projects can import and reuse the component or they can contribute to its development by modifying it and create a new version of it. The next example tries to present the repository system from the user perspective in the context of using ProSave IDE for developing an embedded system:

The work on the system starts when the users create a project, which holds all development data. The project meta-data should include parameters of the connection with the repository like URL of the server. When the architecture of the system is complete and needed components are identified, the users can check for already developed components which fit the desired specification by browsing the component repository. The components which exactly fulfill the requirements can be imported and used out of the box. Some of the components in the repository may need improvements in order to be included in the system. The users can import them, make the needed

modifications –for example implement a better algorithm, and export the components back into the repository as a new version. In the case when a specific component from the repository can be reused in the system but it needs modifications which change its specification – for example adding a new port or changing internal logic, the users can create a new branch of this component; that is import the component, implement the required changes and instead of exporting it to the repository as a new version, create a new component in the repository as derived from the first one. Then the users may create a component from scratch using an architecture editor, code generator and other assisting tools. While their work on the component is in progress, they can use a version control system to keep the data and ensure collaborative modifications. Once the component is complete the users can share it by making export to the component repository.

The next sections summarize the requirements and constraints on the repository system which are divided into two categories – functional category and implementation specific category, as this represents two different views on the system. On one hand the system users are only concerned with its functionality while on the other hand prospective developers of ProSave IDE are interested in its implementation details.

3.1. Functional specification

The repository has to be available on the network or on a local machine for potentially many users working concurrently. It is required that every modification on components is safe and doesn't affect the work of other users of the repository. The system has to support the following operations on components:

- Browse available components in the repository together with their older versions
- List all subcomponents of a particular component
- Export a component made from scratch to the repository
- Import a component of a specific version from the repository to a project in ProSave IDE
- Export a newer version of existing component
- Export a modified existing component as a different component and save its ancestor's history
- Delete(hide when browsing) a component

The diagram below shows a wider look of all ProSave IDE functionality related to repository system.

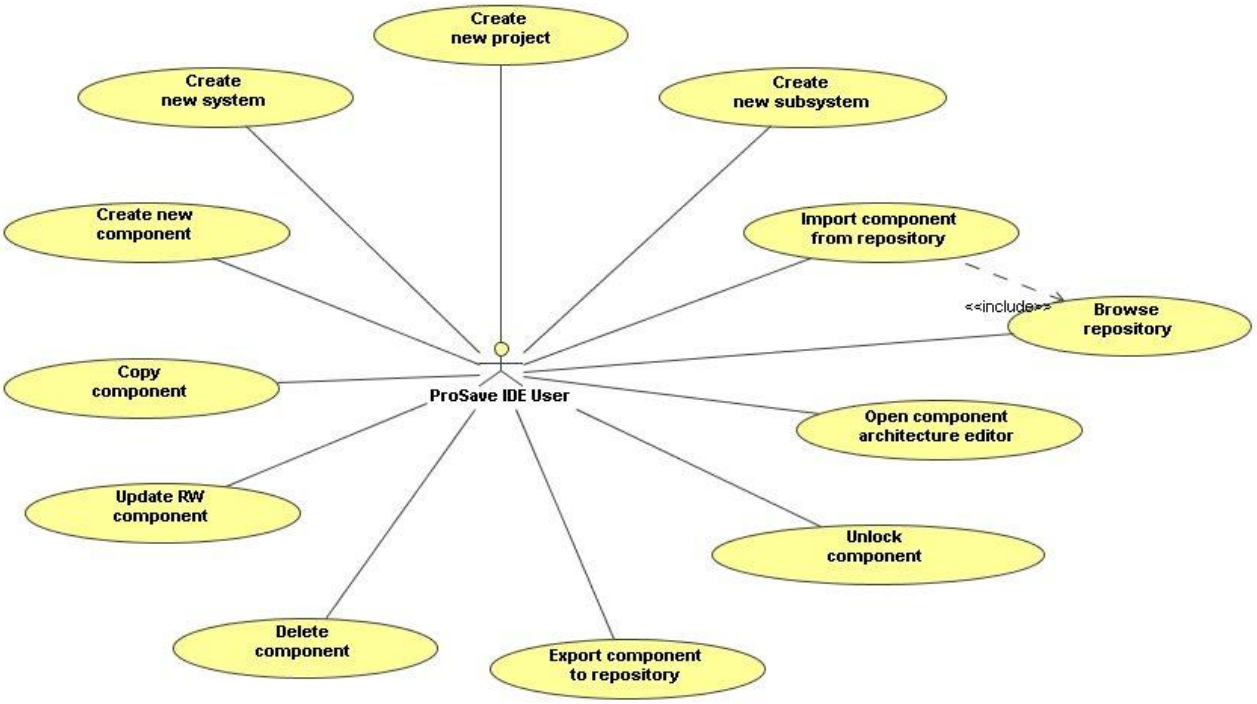


Figure 2: Use case diagram

Each one of these use cases is described in details in **Appendix A** using activity diagrams. For better understanding of what is expected behavior of the repository system one is advised to refer to them.

3.2. Implementation requirements and constraints

First of all, as it is part of the ProSave IDE, all system APIs have to be available for Eclipse plug-ins hence implemented as Java interfaces. Then, the components' data that has to be stored in the repository is represented and edited as files. It includes source code, documentation, references and binary files. In a file system it corresponds to hierarchy folder structure and a collection of files in each folder. For example subdirectory "src" which contains source code files, subdirectory "doc" which contains component documentation and so on. The description of the parent-child relationships is included in the components' meta-data although the concrete format is not yet known. Besides that, a ProSave component can have many versions, and each of its referenced subcomponents is also in a specific version. Considering these conditions, the thesis investigation should use a version control system for implementation of the server part of the system.

The repository's APIs have to be independent from actual server implementation. That

means that if the version control system is changed with a file server or other storage system then all interfaces remain the same.
And last but not least the repository must be reliable, scalable and extensible.

4.Repository versioning policy

The presence of parent-child relationships between the ProSave components in addition to their number of different versions, sets the pattern for complex hierarchy associations. This creates an open problem when it is considered that the ProSave IDE users can modify the components stored in the repository. And here is the question: What will happen when one modifies a component already used in other composite component(s)? Can the system propagate the changes to all concerned composite components?
The answer is “Depending on the changes.” If the modification is safe – documentation update, adding code comments etc. then the changes can be propagated, otherwise this can lead to errors as in the case of adding/deleting a port, changing internal logic or attributes. However, the boundaries of this set of safe changes are not yet known and defining methods to determine if a particular modification belongs to this set seems to require extended investigation. That is why the thesis examines simple solution to this problem. Each modification creates new version of the component. All subcomponents are referred using their particular version and the changes are not propagated. In this way the users are responsible for upgrading the version of the subcomponents. The next example illustrates this matter:

Components ***A***, ***B***, ***C*** and ***D*** are in the repository. ***A*** is subcomponent of ***C*** and ***D***. ***B*** is subcomponent of ***C***, and ***C*** is subcomponent of ***D*** as shown in ***Figure 3***.

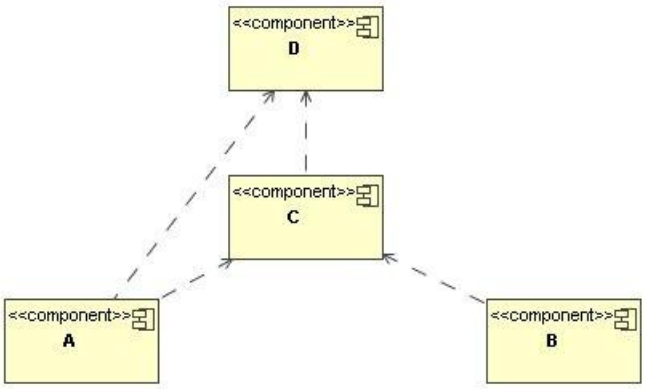


Figure 3 - Components associations

The users find out that in order to improve some parameters in component ***D*** they have to modify its subcomponent ***A***. If the modification is not safe and the system propagates the changes to ***C***, then both of the components ***C*** and ***D*** will not work correctly because ***D*** depends on ***C***. When the fact, that the hierarchy associations can be much more complex, is taken into consideration, it is evident that the effect of this practice is completely unacceptable. Because of that, the repository creates a new version of ***A*** regardless of the modification type. ***C*** will contain the old version of ***A***, and ***D*** will contain both versions of ***A*** – the newly created and the version of ***A*** from ***C***. Then the users have to decide whether the changes to ***A*** can be propagated. In the case when the characteristics of ***C*** will be improved if it uses the new version of ***A***, the users can create a new version of ***C*** which uses the new version of ***A***. The same procedure has to be applied recursively for all dependent components.

5. System architecture

This section divides the system into modules and describes their structure. At first glance there are three distinctive tiers which are common to most applications containing server part. That is data, application and presentation tier. Each of them encapsulates specific functionality and communicates with other tier(s) via public interfaces.

5.1. Data tier

This is the place where all components' data has to reside. Also, it is required to be available for many clients working concurrently and accessing it remotely. As already mentioned in system constraints section, this thesis investigation is limited to using version control software as a data storage facility. Thus, the capabilities of VCS to provide storage and versioning will be reused which will make possible the fulfillment of required functionality of the repository system in the frame of this thesis. There are many open-source and proprietary products in use these days but they all share similar functionality common to all revision systems. They aim at tracking files and directories changes over time. This allows clients to examine the history of how their data changed, who change it and when. Usually a version control system consists of server side or repository, where the data is stored, and many clients situated on the network. They work with the same set of data and interact with the server to save their changes, restore particular information to previous state, examine the history, resolve any conflicts etc.

Different revision systems use various ways to support these operations. The most appropriate for our data tier implementation is however defined by the need of integration with ProSave IDE. There is no doubt that open-source's best-known revision systems are Concurrent Versions System (CVS) and Subversion (SVN.) Both of them have their functionality available as Java interfaces and are used in Eclipse plug-ins. Designed to be a successor of CVS, the Subversion lacks most of CVS's noticeable flaws (for example lack of directory versioning) and it is chosen by ProSave IDE developers for default project revision system. That's why Subversion is selected to be the core of data tier implementation in our component repository.

5.2. Application tier

This part of the system is where components' operations are realized using basic SVN commands and interactions with the data tier. Since Subversion is a collection of C libraries there is a need of so called adapter which will allow using these libraries in Java. There are three low level Java libraries that provide access to SVN's API – JavaHL, SVNKit and SVN command line client wrapper. JavaHL is a subversion library which binds SVN's binary executables to Java interfaces using Java Native Interface (JNI) and is provided by Subversion's creator CollabNet. SVNKit is pure Java implementation of the SVN client and does not use the original C libraries. The last one wraps calls to SVN's command line client. Each one of them can be used as a connection between Subversion functionality and our application. However there is another adapter built on top of these three which is easier to use. That is a SvnClientAdapter library used in the Subclipse project. It can use any of these three low-level SVN API implementations to achieve smooth transition between the native C functions and high-level Java interfaces.

So far there are SVN server situated in the data tier and well-defined Java interfaces which allow us to execute remote SVN commands. The other indispensable thing in the application tier is implementation of specific logic module which will transform a component operation to composition of basic SVN commands. For example:

Suppose the ProSave IDE user made a component from scratch and now he wants to export it to the repository so the other developers can use it. He will probably use the IDE's graphical user interface to achieve this which is built in presentation tier. Once received the user request, presentation tier will delegate actual export to application tier where implemented logic has to translate the operation to several SVN commands e.g. "svn add," "svn propset," "svn commit" and execute them upon SVN server using the adapter library.

In our work, we have named the module where this logic will be implemented "Repository access."

5.3. Presentation tier

This tier establishes the actual connection between the ProSave IDE and the repository itself. It consists of one or more Eclipse plug-ins which extends the workspace with additional GUI and exposes the repository functionality to the end user.

5.4. Summary

Up to here, three separate tiers constructing our system were outlined, but how do they relate to each other in terms of network nodes, communication protocols and deployment environment? As shown in ***Figure 4*** the server side of the system consists of our data tier implementation.

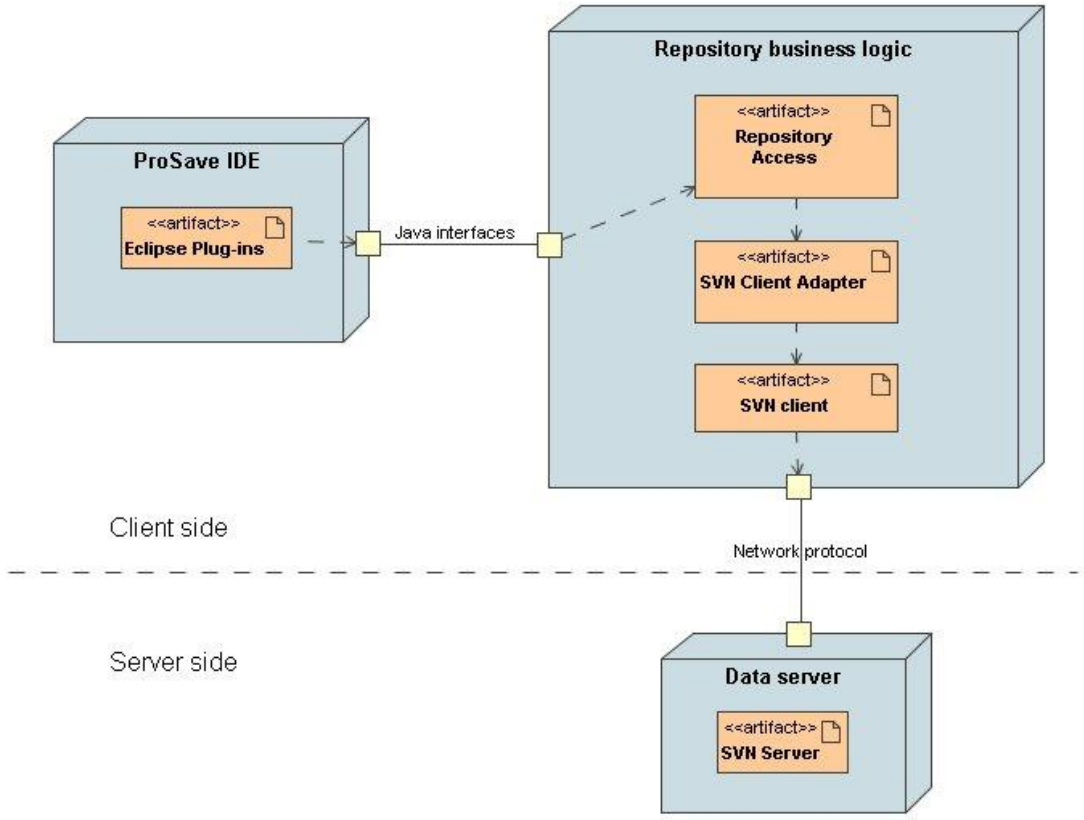


Figure 4: System architecture – the big picture

On the other hand, application and presentation tiers are both located on the client side. The reason why the three tier architecture is not used is because the application tier is relatively simple and placing it on an application server will unreasonably increase complicity of development, deployment and maintenance of the system. Moreover,

Subversion initial design is based on client-server model and thus it does not support integration of application server.

The next step is to take closer look on the big picture and our system tiers. The data tier is nothing but SVN server with some requirements on data it stores. That means that it can be installed on remote or local machine. Depending on its configuration it supports five different access methods:

- Direct repository access (only available for local disk repositories)
- Access via WebDAV protocol `http://`
- Access via WebDAV protocol with SSL encryption `https://`
- Custom SVN protocol
- Custom SVN protocol through an SSH tunnel

All of them represent different sets of tradeoffs concerning security, user accounts management, error logging and complicity of maintenance. From our system's point of view the actually chosen access method does not affect repository's functionality in any aspects. It just provides additional flexibility available to ProSave IDE administrators and we are not going to examine the details of each protocol.
Using SVN client, SVN Client Adapter library provides access to SVN server in the form of Java interfaces. By their means the Repository access module implements the components' operations already defined in system requirements section. This is assumed as an application tier of our system.
The last part is devoted to integration with ProSave environment. It is implemented as Eclipse plug-ins, which extends the graphical user interface of the IDE with repository operations.

6. Implementation process

This section describes the most important steps in the system implementation along with the problems emerged, and the investigation of different methods to solve them.

6.1. System design

As already expressed in the above section our system consists of different parts. Some of them have to be implemented and the others can be used out of the box. The SVN server will be utilized as it is, i.e. a regular installation is needed or even an existing one can be

reused. The next step is to allocate a tangible directory for our repository on the server's virtual file system. In this directory the components will be stored in separate subdirectories. In our work, we have named this directory `/prosave_repo/components`.

As the only way to uniquely identify a component in the SVN repository is to guarantee that its root directory is the only one of its kind, the name convention is needed. An assumption is made that as part of the component meta-data there are a unique identifier and a human readable label. Thereby a component's root folder has the following format: <*short_name*>_<*id*>.

Now that our server part of the system is configured, the survey can continue with our application tier.
Since the SvnClientAdapter library uses JavaHL or SVNKit or SVN command line client wrapper, at least one of the following is needed deployed on the client side: JavaHL's native libraries and `svnjavahl.jar` in our project's classpath, `svnkit.jar` and `ganymed.jar` in our project's classpath or regular SVN client installation. These libraries ensure the connection with SVN server and expose its functionality to our Java project. Now it is possible to take advantage of SvnClientAdapter's high level APIs by including `svnClientAdapter.jar` in our classpath.

When designing our application tier, it is preferred to provide high level repository interfaces to the presentation tier and in the same time to enable replacement of SVN repository with another storage type without affecting these interfaces. This ensures loose coupling between system "Storage" and "IDE integration" components as illustrated in ***Figure 5***.

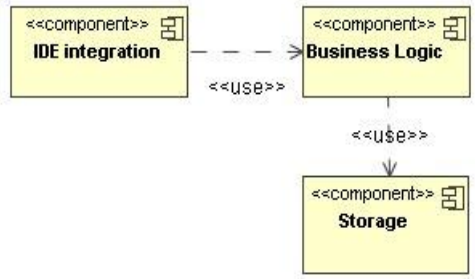


Figure 5 - The system components

Benefits of this design are easy to understand when one looks in the following example:

It is assumed that our system is fully implemented and integrated in the ProSave IDE. After a comprehensive test process users might notice lack of a very important repository feature or even an inappropriate for their work system behavior. This may brings the

need of using another version control system or even different type of repository – file server, database server etc. If the "IDE integration" component is much coupled with the current SVN repository implementation this will enforce the rebuilding of the whole system.

In order to address this issue, the system objects need to be defined. If the repository is seen as an autonomous part of our system, then there is an object *repository* which can be of different type – SVN, CVS, file server, DB storage. Another entity in our system is a ProSave *component*. Independently from the type of the repository, there is a need of unified interface for utilizing its functionality. We can name this interface *IRepository* and attach some operations to it. They can be derived directly from our system requirement section (paragraph 3.1.).

```
/**
 * Common interface for accessing the repository
 */
public interface IRepository {
	/**
	 * List all components available in the repository.
	 */
	public IComponent[] getAllComponents;

	/**
	 * List all older versions of particular component
	 */
	public IComponent[] getOldVersions(IComponent comp);

	/**
	 * Delete a component
	 */
	public void deleteComponent(IComponent comp);

	/**
	 * Import a component from repository to a particular project.
	 */
	public void importComponent(IComponent comp, File project_root)

	/**
	 * Export a component from a project to the repository
	 */
	public File[] exportComponent(File project_root, IComponent comp)

	…
	…
	…
}
```

Then if the application tier can ensure that every type of repository complies with this interface, it will be very simple to use different types of repositories even at the same time. This is a common problem that the OO designers face and it has a well known

solution. *The Abstract Factory* design pattern provides a way to make our "IDE integration" component completely independent of actual type of the repository.

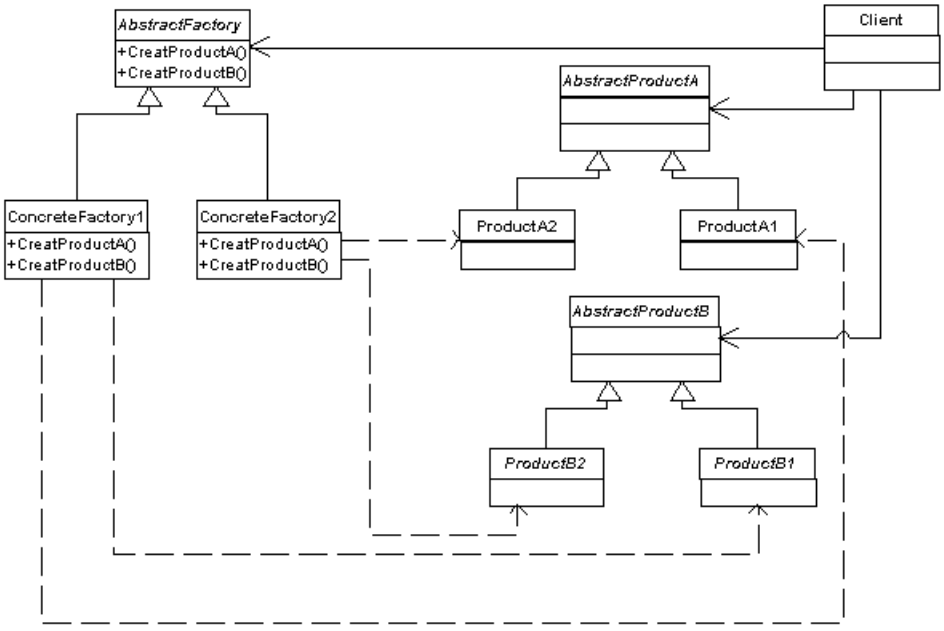


Figure 6 - Abstract factory design pattern

As one can see from the UML diagram above, the "Client" does not know which concrete objects it receives from each of these internal factories, since it uses only the generic interfaces. Objects of a concrete type are indeed created by the factory, but the client code accesses such objects only through their abstract interface [11]. This makes the "Client" independent from the concrete product implementation.

Now it is possible to apply this pattern to our system design. The first thing needed is a mapping between our system's entities and the pattern's participants. The "Client" is actually our "IDE integration" component, which will use an *abstract factory* to get the generic interface *IRepository* or so called *the abstract product*. *The Concrete products* are the different types of repositories in our case there is just one such product - SVN server, but later on other types can be easily added. An illustrative example of how this pattern directs our design can be seen in ***Figure 7.***

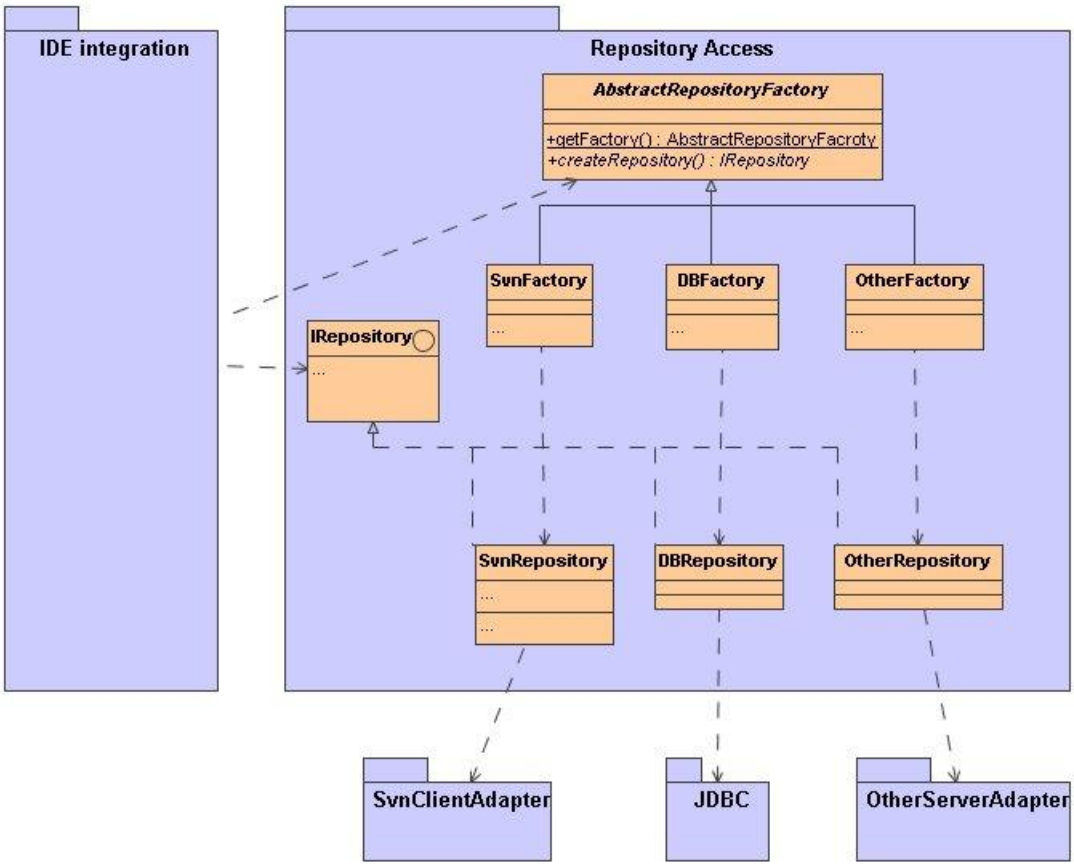


Figure 7 - System class diagram - general view

The "IDE Integration" component does not know what type of repository is actually using because it works with the common interface IRepository. Our implementation will include only the classes *AbstractRepositoryFactory, SvnFactory and SvnRepository*. The most important is however *SvnRepository*. It provides implementation of the repository operations and communicates with the SVN server using the SvnClientAdapter library. Despite the fact that in a certain ProSave project more than one SVN repositories can be used, our system needs only one instance of the SVN client in order to perform component operations with all of them. That's why it is suitable to create the SvnRepository class as singleton. This way no more than one object of this class can be created which will ensure simple and safety usage. Now, thorough description of our design can be easily derived.

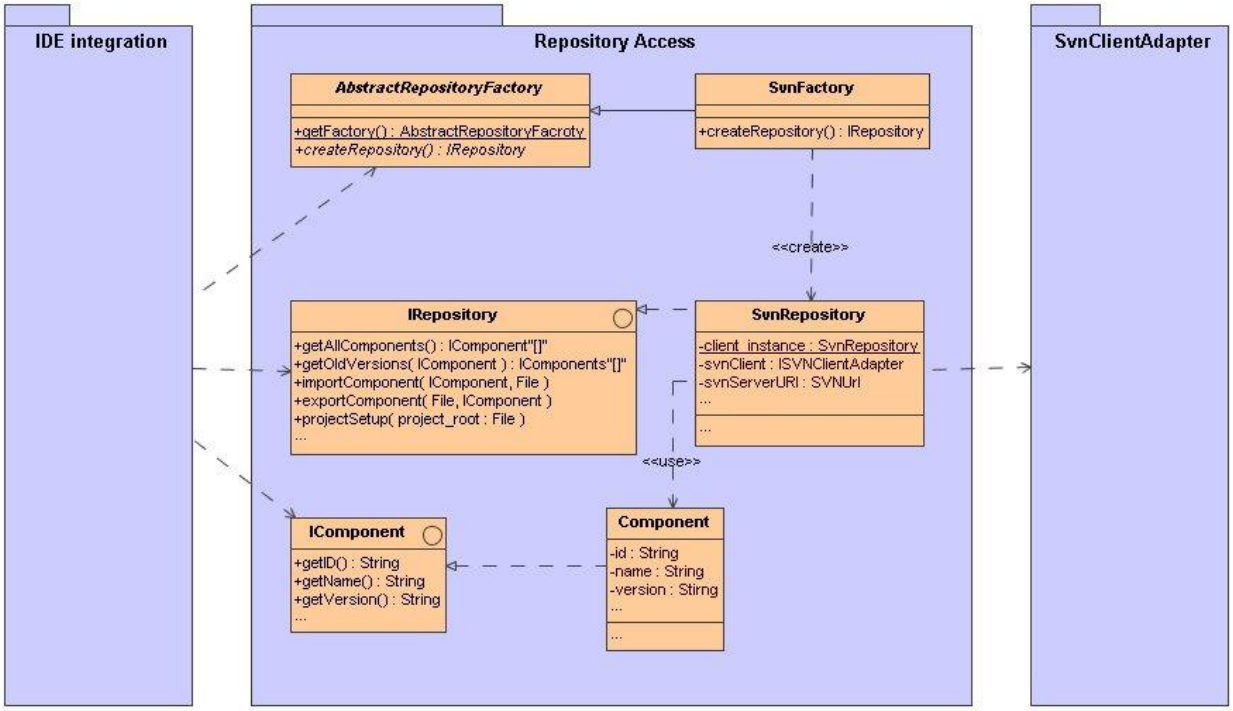


Figure 8 - System class diagram - detail view

The *Component* class represents the repository's view of a ProSave component, i.e. it contains only repository specific meta-data like an identification number of the component, its current version etc.

6.2. Project structure prerequisites and conventions

This section describes the system's procedures and accepted conventions on the file structure of a ProSave project. For that purpose a definition of "a ProSave project" is needed. Technically speaking it is a hierarchy of folders and files; some of them contain system information and some of them are created by the ProSave IDE users. They store all the information about components, subsystems and systems currently being developed including: architecture views, attributes, models, relations and connections between them. In order to import and export components to and from the project their position is fixed in a separate folder beneath the project root folder namely `\components`. The entire component's data is bundled as a subfolder of the `\components` folder and it name follows the structure already defined in the above section – *<short_name>_<id>*.

A typical repository operation includes interacting with local, temporary folder which in the case of a SVN repository will be our *working copy* folder. All primitive SVN commands will be executed upon this folder. Its position is set as a subfolder of the project root and it is named `rep_working_folder`. For example importing a component from the repository to the project will be a sequence of the following operations:

- Check out the specified component from the SVN server to the working copy.
- Copy the component's data from the working copy to the project location.

Since a single component can be presented in multiple versions in a particular project, a way to distinguish the location of each of these versions is needed. Thereby the final format of the component's root directory is:
<*short_name*>_<*id*>[#<*version*>], where

- <*short_name*> is a human readable component identification,
- <*id*> is a component identification number,
- <*version*> is a component version number. The root folder has a version attached only if the particular component is in the repository and it is not currently being modified.

The next diagram depicts a simple example of a ProSave project which contains two components.

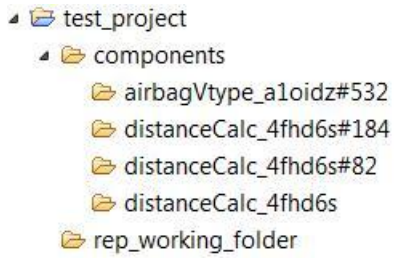


Figure 9 - Sample ProSave IDE project structure

As one can see, the *distanceCalc* component is presented in two different versions, namely 184 and 82, and it is also being modified by the users. The modifications take place in the directory `components\distanceCalc_4fhd6s` and that is why it has not attached version.

Another important assumption is presence of a configuration file for the repository in the project root folder. It has a standard *properties file* format and contains the type and the URL address of the repository which is currently being used in the project. At

presence, only the SVN type is supported. The switching between different repositories is accomplished by changing the URL property in the configuration file.

When the system executes a repository operation, it needs specific information about the components which are part of this operation. For example in the case of an import action all subcomponents have to be identified so that all dependencies are presented consistently in the project. All this repository-specific meta-data is stored as *properties file* and attached to each component.

6.3. Repository business logic. Mapping between operations on components and SVN commands

Using Subversion for component repository is feasible because there exists a way to transform a component operation to a set of SVN commands and a set of file system commands. Once having this equivalence, it is just a matter of coding to implement the component repository. Firstly, a brief description of the needed SVN commands will be provided.

- **svn list** – List directory entries in the SVN server virtual file system. It has the following syntax:
  svn list <target>,
  where <target> is the URL of the directory being examined.

- **svn log** – Show the history of the changes which are done on a particular directory. The syntax is:
  svn log <target>

- **svn propget** – Get the value of a property attached to a SVN file or directory. The syntax is:
  svn propget <prop_name> <target>,
  where <prop_name> is the name of the property which is inquired.

- **svn propset** – Set the value of a property. The syntax is:
  svn propset <prop_name> <value> <path>,
  where <path> is an OS path pointing to the working copy.

- **svn checkout** – Download a directory tree from a SVN server to the local working copy. Syntax:
  svn checkout <target> <path>

- **svn add** – Schedule files or directories in the working copy for addition to the repository. Syntax:
  svn add <path>

- **svn delete** – Delete an item from a working copy or the repository. Syntax:
  svn delete <target> or
  svn delete <path>

- **svn copy** – Copy a file or directory in a working copy or in the repository. Syntax:
  svn copy <source> <destination> ,
  where <source> and <destination> can each be either a working copy path or server URL. After the command is executed the <destination> is a mirror copy of the <source> and moreover the history of changes is also copied.

- **svn update** – Bring the changes from the repository into the working copy. If local modifications are made to the working copy, Subversion will try to merge them with the changes from the server. If it fails a conflict occur. Syntax:
  svn update <path>

- **svn commit** – Send changes from the working copy to the repository. Syntax:
  svn commit <path>

In order to describe this mapping in a clear way, this paragraph will use the diagram on ***Figure 9*** as an example of a ProSave project. The URL of the component repository is set to `http://sample_server/prosave_repo/components`. For simplicity, the examples below refer to it as `/components`. For each of the listed component operations, a sequence of SVN and OS commands is given.

- **Browse available components in the repository**
  List components' latest version:
  ```
  svn list /components
  ```
  Extract the meta-data from each entry:
  ```
  svn propget Name /components/distanceCalc_4fhd6s
  svn propget Root /components/distanceCalc_4fhd6s
  …
  ```
  Obtain the old versions of a pointed component:
  ```
  svn log /components/distanceCalc_4fhd6s
  ```
  Extract the meta-data from each version:
  …

- **List all subcomponents of a particular component**
  Get a list of names and versions of the root directories of the subcomponents:

```
svn propget Subcomponents /components/distanceCalc_4fhd6s
```
Extract the meta-data from each subcomponent:
```
svn propget Name /components/distCalsSubcomponent1
```
…

- **Export a component made from scratch to the repository**
  OS copy the component folder tree from the project to the working copy:
  ```
  copy /test_project/components/distanceCalc_4fhd6s
  /test_project/rep_working_copy/distanceCalc_4fhd6s
  ```
  Schedule the component's working copy folder for addition:
  ```
  svn add /test_project/rep_working_copy/distanceCalc_4fhd6s
  ```
  Read the meta-data of the component from the properties file and attach it to the working copy:
  ```
  svn propset Name "Distance to obstacle"
  /test_project/rep_working_copy/distanceCalc_4fhd6s
  ```
  …
  Commit the new component to the repository:
  ```
  svn commit /rep_working_copy/distanceCalc_4fhd6s
  ```
  Read the revision number after the commit operation and set the version of the new component in its properties file and attach it to the root folder.

- **Import a component of a specific version from the repository to a project in ProSave IDE**
  Get a list of names and versions of the root directories of the subcomponents:
  ```
  svn propget Subcomponents /components/distanceCalc_4fhd6s
  ```
  For each subcomponent, if it's version is not in the project yet:
  Check out the correct version to the working copy:
  ```
  svn checkout /components/distCalsSubcomponent1
  /rep_working_copy/distCalsSubcomponent1
  ```
  OS copy its folder to the project path:
  ```
  copy /rep_working_copy/distCalsSubcomponent1
  /test_project/components/distCalsSubcomponent1
  ```
  Create a properties file with the meta-data in the component's project folder and fill it in using:
  ```
  svn propget Name /components/distCalsSubcomponent1
  ```
  …
  After all subcomponents are imported to the project, repeat the same procedure for their ancestor.

- **Export a newer version of existing component**

Read the number of the pristine version from the component's properties file and check out that version from the repository to the working copy:

```
svn checkout /components/distanceCalc_4fhd6s
/rep_working_copy/distanceCalc_4fhd6s
```

Synchronize the component's working copy and project folders. Detailed description of this procedure is provided in the end of the section.

Bring the latest changes made to this component from the repository:

```
svn update /rep_working_copy/distanceCalc_4fhd6s
```

If a conflict occurs, the export stops here and the ProSave IDE users are responsible for the merging of the files which are in conflict state. After they are merged the procedure can continue.

Read the meta-data of the component from the properties file and attach it to the working copy:

```
svn propset Name "Distance to obstacle"
/test_project/rep_working_copy/distanceCalc_4fhd6s
```

…

Commit the new version of the component to the repository:

```
svn commit /rep_working_copy/distanceCalc_4fhd6s
```

Read the revision number after the commit operation and set the version of the new component in its properties file and attach it to the root folder.

- **Export a modified existing component as a different component and save its ancestor's history**

  OS copy the existing component from its project location to the project again but under different name and assign it a new id:

  ```
  copy /test_project/components/distanceCalc_4fhd6s
  /test_project/components/distanceCalcBranch_zqr1oe
  ```

  Check out the right version of the existing component from the repository to the working copy:

  ```
  svn checkout /components/distanceCalc_4fhd6s
  /rep_working_copy/distanceCalc_4fhd6s
  ```

  Create SVN copy of the component in the working copy and set its new meta-data from the properties file:

  ```
  svn copy /rep_working_copy/distanceCalc_4fhd6s
  /rep_working_copy/distanceCalcBranch_zqr1oe
  svn propset Name "Branch of distance to obstacle"
  /rep_working_copy/distanceCalcBranch_zqr1oe
  ```

  …

  Commit the new copy of the component to repository:

  ```
  svn commit /rep_working_copy/distanceCalcBranch_zqr1oe
  ```

Read the revision number after the commit operation and set the version of the new component in its properties file and attach it to the root folder.
After this procedure is complete, the users can start modifying the newly created copy of the component and use the regular export operation to save their changes to the repository.

- **Delete(hide when browsing) a component**
  Hide the component by creating a new revision in the SVN server in which the component is removed:
  `svn delete /components/distanceCalc_4fhd6s`
  That operation does not delete the component from the ProSave project if it exists there.

As already stated above, the process of exporting a new version of existing component requires synchronization between the component's working copy and project folders. The purpose is to bring the changes made in the project folder tree to the working copy in a way that will allow presenting them obtainable for the Subversion. The next bullet provides a recursive algorithm for synchronizing the two folder trees:

- **Synchronize(** `/test_project/components/distanceCalc_4fhd6s`**,** `/test_project/rep_working_copy/distanceCalc_4fhd6s` **)**

  For every subfolder ***X*** of the working copy folder `/rep_working_copy/distanceCalc_4fhd6s` which is not subfolder of the component project folder execute:
  `svn delete` ***X***

  For every subfolder ***Y*** of the project folder `/test_project/components/distanceCalc_4fhd6s`
  If ***Y*** is not subfolder of the working copy folder execute:
  OS copy `/test_project/components/distanceCalc_4fhd6s/`***Y***
  `/rep_working_copy/distanceCalc_4fhd6s/`***Y***
  `svn add /rep_working_copy/distanceCalc_4fhd6s/`***Y***

  Else, there is a folder `/rep_working_copy/distanceCalc_4fhd6s/`***Y***:
  Run recursively:
  Synchronize(***Y,*** `/rep_working_copy/distanceCalc_4fhd6s/`***Y***)

  For every file ***F*** in the `/rep_working_copy/distanceCalc_4fhd6s/` which is not in the project folder run:

`svn delete` ***F***

For every file ***P*** in the project folder
`/test_project/components/distanceCalc_4fhd6s`
If ***P*** does not exist in the working copy folder execute:
OS copy `/test_project/components/distanceCalc_4fhd6s/`***P***
`/rep_working_copy/distanceCalc_4fhd6s/`***P***
`svn add /rep_working_copy/distanceCalc_4fhd6s/`***P***

Else, there is a file `/rep_working_copy/distanceCalc_4fhd6s/`***P***:
OS replace `/rep_working_copy/distanceCalc_4fhd6s/`***P*** with the file
`/test_project/components/distanceCalc_4fhd6s/`***P***

7. Overview of the implemented system

The system covers the entire functionality described in the requirement section (paragraph 3) that is browse the repository, export, import, and delete components, create branches of a component. Additionally, it provides mechanisms to attach a working copy to a project, change the SVN server dynamically and handle conflicts. The import operation is consistent which means that a composite component is imported together with all dependant components if they are not already existed in the project. Also, the system is responsible for assigning versions – both in the component meta-data and project directory structure. The repository adopts the Copy-Modify-Merge model in order to provide concurrent access. This means it allows simultaneous modifications of a component by different users. When the component is exported in the repository, the system uses the built-in mechanism of Subversion for merging the changes. If conflicts in the modified data occur, the export operation stops and the users have to handle the collisions by themselves; then the system completes the export and commits the new version of the component. This model is considered as more efficient for the collaboration and it provides more flexibility to the repository system.

The source of the created system is spread out in three projects: a java project containing the application tier implementation and two Eclipse plug-in development projects. The java project contains a package *repository_access* where the designed in section 6 (paragraph 6.1) classes and interfaces are coded together with two subsidiary classes: *FileSystem*, which implements several basic OS operations on files and folders, and *RepositoryException* for handling the exceptions. All the source code is commented

using *JavaDoc* tags and conventions. This project also includes an automatic *Ant build script* which is structured as two files – build.properties and build.xml. It compiles the source code and creates a Java archive library – RepositoryAccess.jar. Then a distribution folder is created which contains RepositoryAccess.jar and all dependant libraries namely svnClientAdapter.jar, svnjavahl.jar, svnkit.jar, ganymed.jar together with the native JavaHL libraries and compiled *JavaDoc* API documentation. That way, all the needed deployment units are packed together including the low level libraries for accessing the SVN server: SVNKit and SVN command line client wrapper. They are only used if JavaHL is not available which may be caused by interference with other native libraries or deployment on unsupported operating system. The usage of these two libraries however brings some limitation on the repository system. For example, SVN command line client wrapper does not implement some of the methods for handling the conflicts and when used in the repository system the users are not notified about occurring conflicts; Using SVNKit on the other hand brings some errors during export operations because of the different behavior of some of the svnClientAdapter methods: for example `getList` returns an empty collection when the SVN directory is empty and when the JavaHL or the client wrapper are used, but when SVNKit is used on the same directory it returns a collection with one dummy element. This behavior is very confusing especially when one looks at the svnClientAdapter official web page: "*Besides the simpler use, it provides unified adapter to the low-level APIs enabling seamless interchange of the underlying library. (e.g. in case JavaHL is not available, command line wrapper can be used without any impact to existing code)*" [quote from http://subclipse.tigris.org/svnClientAdapter.html]. So far this statement is misleading and not true, but the svnClientAdapter project is still in progress and the next versions could fix these flaws.

The second project packs all libraries from the distribution folder into a single Eclipse plug-in. That way our application tier APIs are made available for GUI contributions to the Eclipse workbench which the ProSave IDE extends.

The third project is an Eclipse plug-in which depends on the plug-in created in the second project. It makes use of the Common Navigator Framework (CNF) [14] and exposes the repository operations to the ProSave IDE user interface. The CNF is designed to facilitate content integration of arbitrary resources into an all-purpose navigator view. The framework is available in Eclipse Platform 3.2 and above. It provides mechanisms for programmers to create their own viewer based on the CNF as well as means to extend the Project Explorer which is a navigator presenting the resources in the Workbench in a hierarchical view. The Project Explorer implements all of the needed operations on projects, folders and files like open and close projects; copy, move and delete files and folders and so on. It supports addition of other resource types and operations through CNF extension points. By their means our plug-in contributes two

new resources – "*Repository*" and "*Component*" and the following operations available as a pop-up menu items:

Resource type	Operation name	Description
Project	Repository configuration	Creates a repository working folder and binds a repository resource to the project
Component root folder	Export component	Exports the pointed component from the project to the repository
Component root folder	Make Branch	Creates branch of the pointed component in the project tree and in the repository
Component root folder	Resolve and commit	Resolves the conflict state during the export and commits the changes to the repository
Component	Import component	Imports the pointed component from the repository to the project
Component	Delete component	Deletes the pointed component

Where "Resource type" is the type of the Project Explorer entry on which the operation is available. In other words, when the user right-click on the pointed entry the described operations are listed as a pop-up menus.

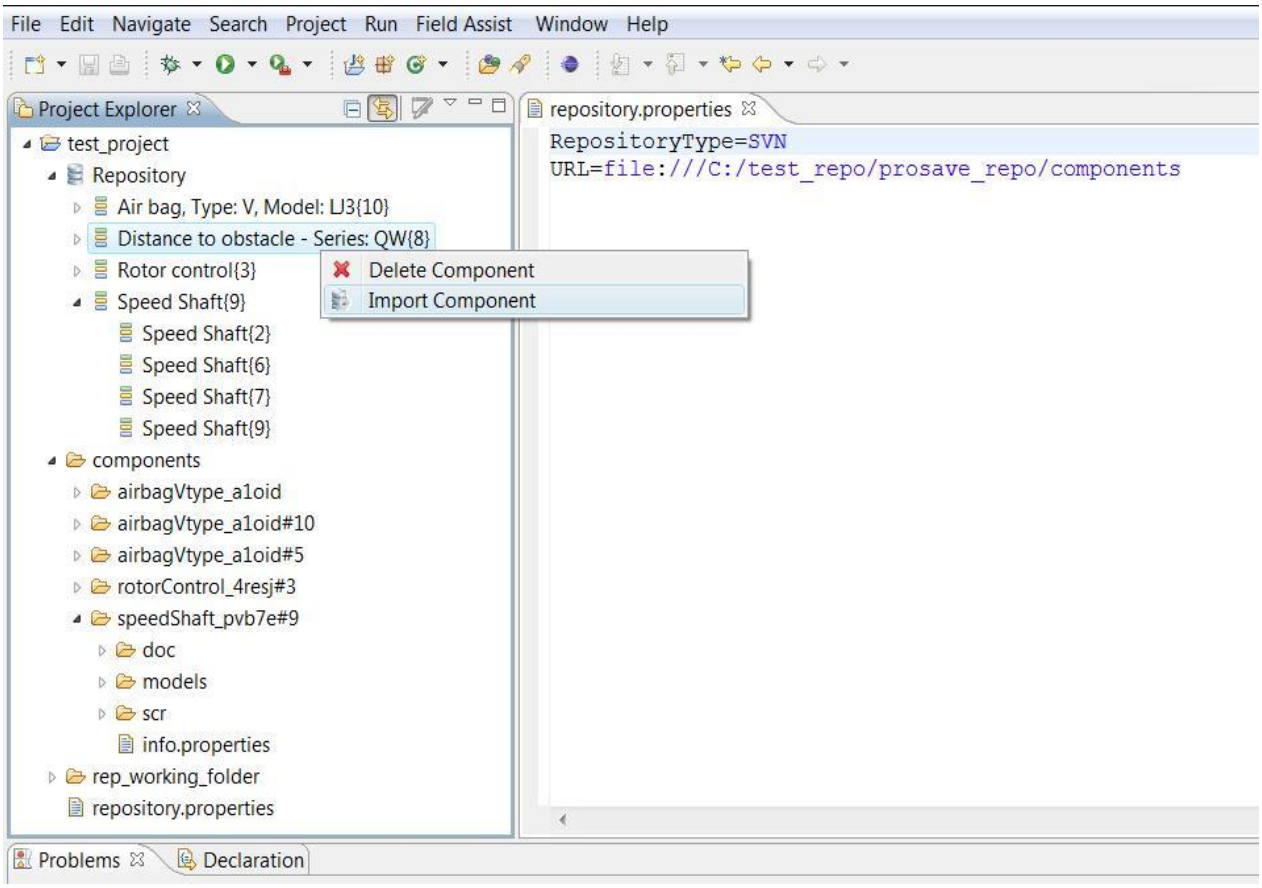


Figure 10 - Sample project screenshot

The plug-in also contributes a hierarchy presentation of the new resources. The "*Repository*" is attached to the project and it can be unfolded which shows the available components in the repository. The Project Explorer displays the components' name and their version in curly brackets as shown on ***Figure 4***. In the appendix B a short user manual for the system is provided.

8. Related work

This section presents a brief survey of the work done in the area and how this thesis fits in it. For that purpose, two examples of component repositories will be examined – one from academia and one commercial. Rather than describing them in details, a comparison between their features and architectures will be provided.

The investigation will start with the repository for SAVE components, part of the SAVE IDE. It is described in the master thesis "*Building of a component development process in an Integrated Development Environment"* [15]. The required functionality for the SAVE repository is very similar to our system because the ProSave IDE is successor of the SAVE IDE and both of them share nearly the same specification. However the work in the above master thesis is concentrated on the presentation tier of the system which defines the differences between the two repositories. It uses a file server for implementation of the repository operations and a simple overwrite mechanism for saving the data. The same as our system, a component is stored as a directory tree consisting of all relevant to the component files. The SAVE repository provides a remote and concurrent access, but lacks any versioning of the components. On the other hand, it includes a repository browser which shows the properties and the architecture of the available components in a dialog window.

The other example is Microsoft Windows XP Embedded component database [16]. It is used to create images of the Windows XP Embedded by assembling a desired set of components. In this way, creating a software system for a particular device (web camera, printer, router etc.) is accomplished by first develop an application, then wrap it as a component, calculate all dependant components using the component database and in the end make an image of the Windows XP Embedded which includes a minimal set of components needed to run the application. The usage of this database is oriented to development of systems which differs to the intended use of the ProSave repository – to facilitate both the systems development and the components development. When developing a component for Windows XP Embedded, a certain set of tools is used (MS

visual studio) but when developing the system by assembling the available components the other set is used (Target Designer, Component Designer and Component Database Manager). The MS component database uses relational database for storing the data i.e. Microsoft SQL Server. Instead of keeping all the files and resources in the database, it stores only their definitions in the form of SLD (Source Level Definition) files which is a XML based format. The actual location of the component data can be anywhere on the network – the SLD files contain just a reference to it. The database provides remote and concurrent access, versioning of the components, and sophisticated search and filter features which allow developers to browse through the component database and search by category, driver type, design template, footprint estimation and so on. Having in mind the number of components in the database (over 10000) it is clear that these are indispensable features which make it possible to select the most appropriate components for the system being developed. The database uses Lock-Modify-Unlock model which allows only one user to modify a component at a time. Another difference regarding the ProSave repository is that there is no need of presence of the components on a developer's local machine – each development tool work directly with the database without downloading the component data.
As it was stated in paragraph 6.1, the ProSave repository relays on existence of unique identifier in the component meta-data, while the MS component database automatically assign two Globally Unique Identifiers (GUID) to each component. The first one is Version Independent GUID which allows the system to uniquely identify the component and the second one is Version Specific GUID which is used for handling the different versions.

9.Conclusion and future work

The thesis investigated the problem of implementing a component repository as part of the ProSave IDE. The purpose was to enhance reusability of the ProSave components by providing a way to store and share them during the component-based development process. The study was limited to using version control system as a basis in order to endow the component repository with features like remote access and component versioning.

Considering the component dependencies the thesis work applied simple versioning policy when a component in the repository is being modified. By this means the changes in one component do not affect its dependent components. Future work in this area may provide a way for the users to decide whether to propagate the changes or not. When the

final format of the component data is set, it is also possible to implement an algorithm which automatically decides if the modifications are safe and if they can be propagated.

The thesis proposed flexible architecture of the system which allows using different types of data servers without a need to rebuild the whole system. The current implementation includes SVN server and uses SvnClientAdapter library as client side SVN service provider. It should be mentioned that the usage of SvnClientAdapter library was accompanied by many problems stem from the very poor documentation provided. The future work may consider removing this library by using only the pure java implementation of the SVN client – SVNKit. In addition, other server types can be provided – different version control servers, database server etc.
The repository system does not guarantee the atomicity of export operation in the case of network connection breaks or machine crashes and prospective improvements of the system could fix this omission.

Although the implemented system is integrated in the ProSave IDE through the Eclipse plug-ins "*Repository Access Plug-in*" and "*Repository ProSave IDE Integration Plug-in*", provided GUI for interacting with the repository is rather limited. The reason is the scope of the thesis investigation – it is not possible to implement complete, fully-functional set of menus, filter, viewers and other GUIs in the frame of this thesis. Moreover the format of the component data is not completely defined yet. Further investigations in this area may include implementation of a component viewer which shows the component data when the users browse the repository. Also, as the number of expected components in the ProSave repository is relatively large, it would be very hard for the users to find the most appropriate components for their system just by looking over all available ones. There is a need of searching mechanism or at least presence of filters which allows users to sort out the components regarding some characteristics. For example the system should be able to provide results for the queries like this one: "Show all components which have 3 data ports of type *Integer* and have worst execution time less than 0.001 ms, or their attribute *Use* has value *Head-lights*."

References

[1] Crnkovic I., Larsson M., (2002), Building Reliable Component-Based Software Systems, Artech House, Boston

[2] Robinson S., Krassel A., August 8, 1997, Components – COM General Technical Articles

[3] Crnkovic I., January 28, 2007, Introduction to Component-Based Software Engineering

[4] Hakansson J., Akerholm M., Carlson J., Fredriksson J., Hansson H., Nolin M., Nolte T., Pettersson P., The SaveCCM Language Reference Manual

[5] Beaton W., Rivieres J., April 19, 2006, Eclipse Platform Technical Overview

[6] Bures T., Carlson J., Crnkovic I., Sentilles S., Vulgarakis A., January 29, 2008, Prosave reference manual - version 0.5

[7] Gallardo D., November 1, 2002, Getting started with the Eclipse Platform

[8] Object Technology International, Inc., February 19, 2003, Eclipse Platform Technical Overview

[9] Purdy G., August 2003, CVS Pocket Reference, Second Edition, O'Reilly

[10] Sussman B., Fitzpatrick B., Pilato C., 2007, Version Control with Subversion: For Subversion 1.4, O'Reilly

[11] Wikipedia, Abstract factory pattern, http://en.wikipedia.org/wiki/Abstract_factory_pattern

[12] Bolour A., July 3, 2003, Notes on the Eclipse Plug-in Architecture

[13] About PROGRESS, http://www.mrtc.mdh.se/progress/index.php?choice=about

[14] Elder M., May 20, 2006, Building a Common Navigator based viewer

[15] Vu-Huy H., April 10, 2008, Building of a component development process in an Integrated Development Environment – Master thesis

[16] Windows XP Embedded, http://msdn.microsoft.com/en-us/library/ms950428.aspx

[17] SVNKit, http://svnkit.com/

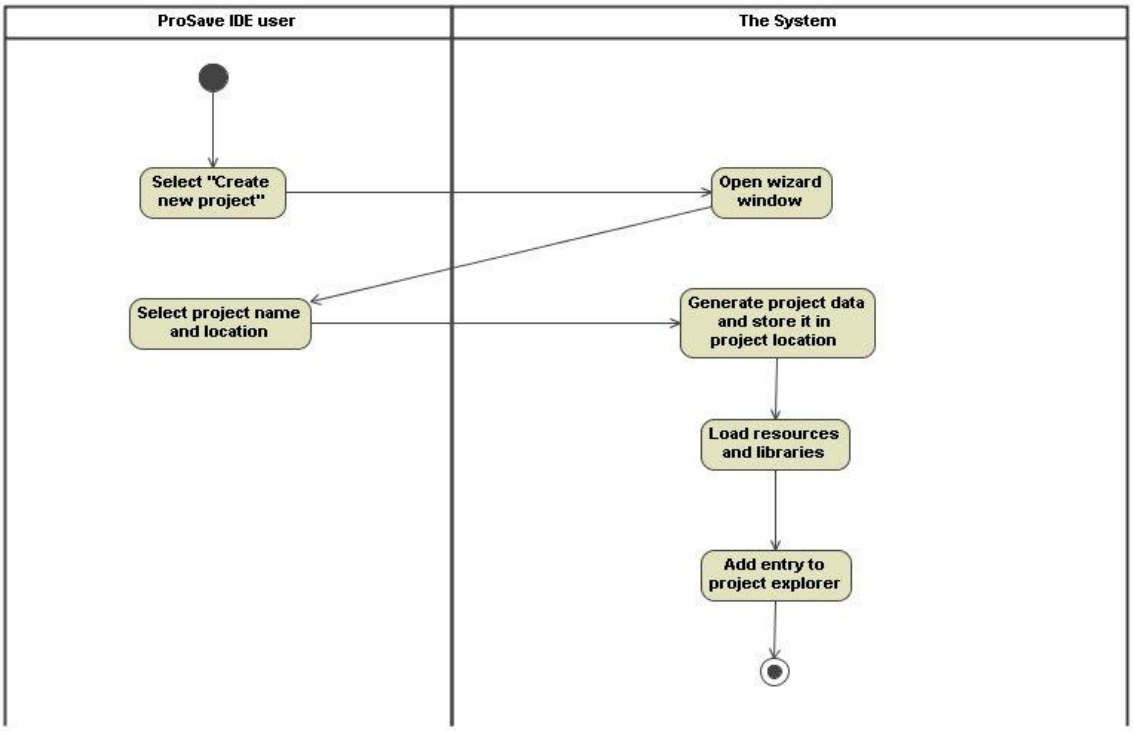
[18] SvnClientAdapter, http://subclipse.tigris.org/svnClientAdapter.html

[19] Clayberg E., Rubel D., March 22, 2006, Building Commercial-Quality Plug-ins, Second Edition, Addison Wesley

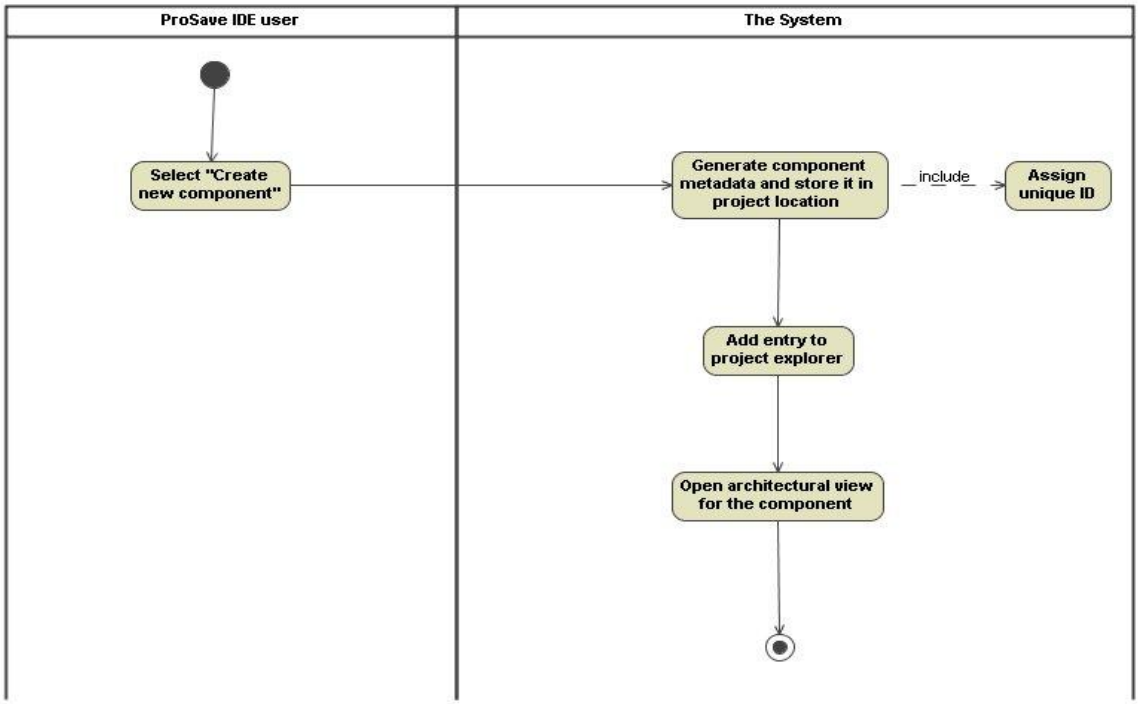
Appendix A: Use cases

This appendix is devoted to detailing the description of system functionality using UML activity diagrams. Some of the use-cases are not directly connected to the repository but they are included here because of the implicit dependencies and for illustrating the place of our system in the ProSave IDE environment.

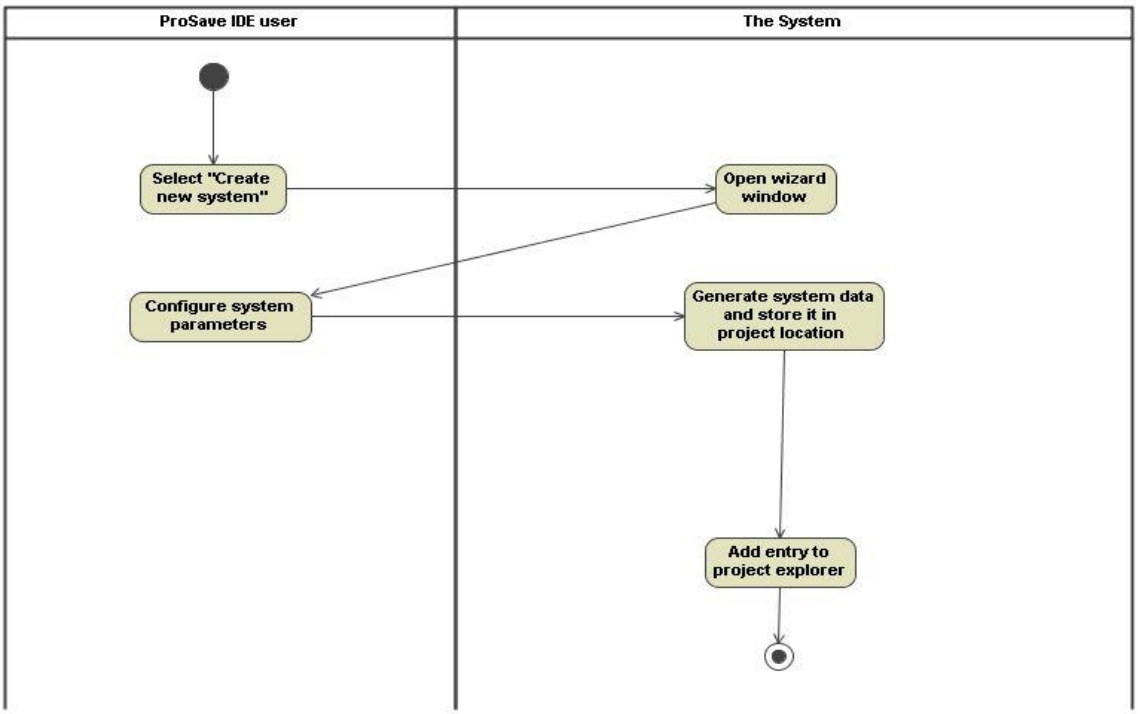
1. Create New Project



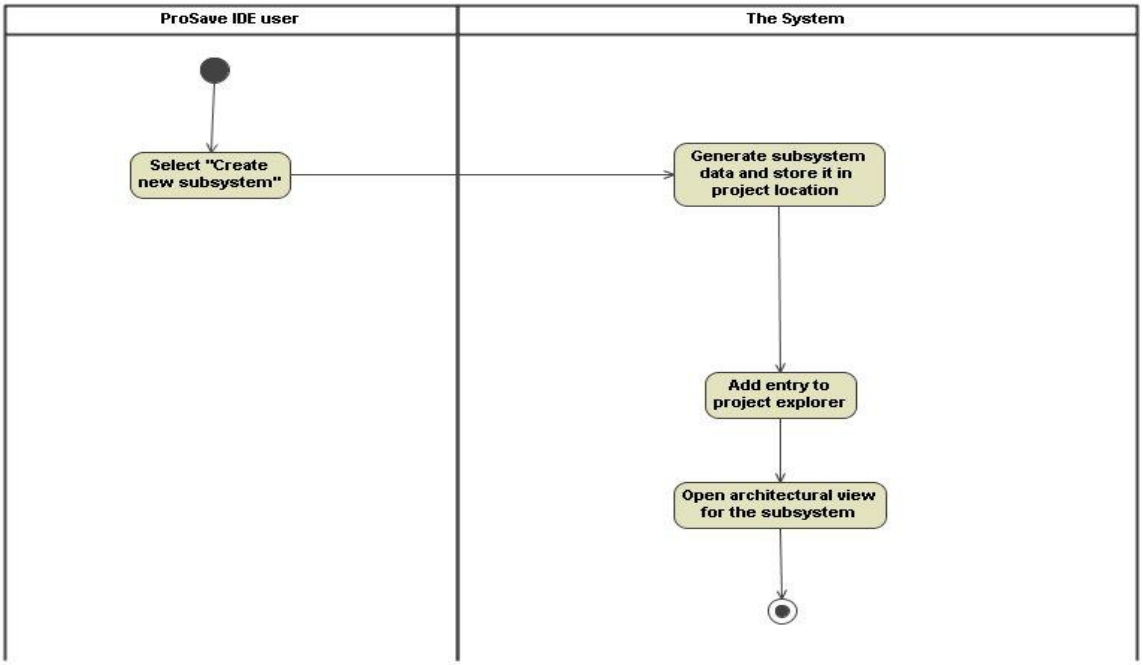
2. Create new component



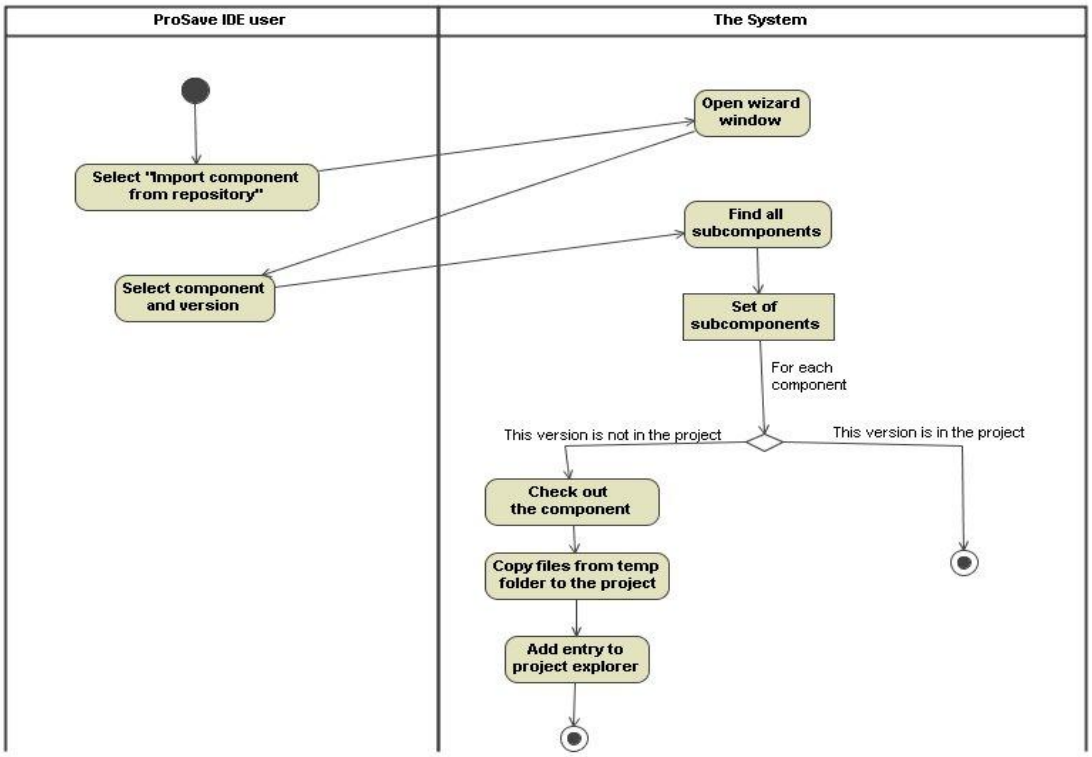
3. Create new system



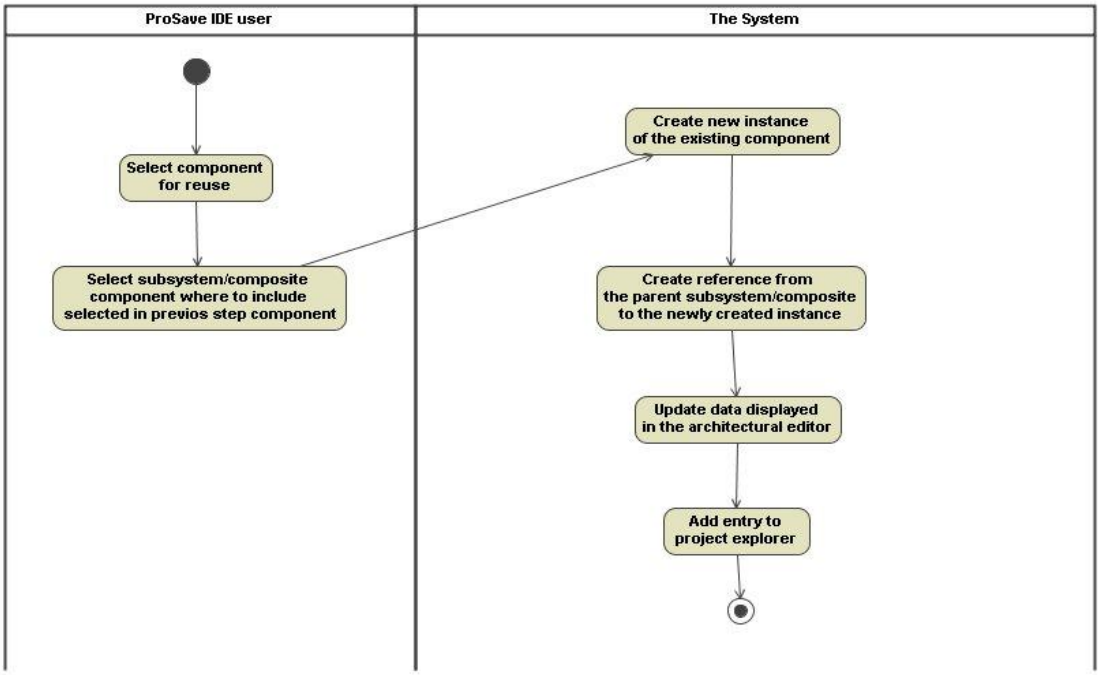
4. Create new subsystem



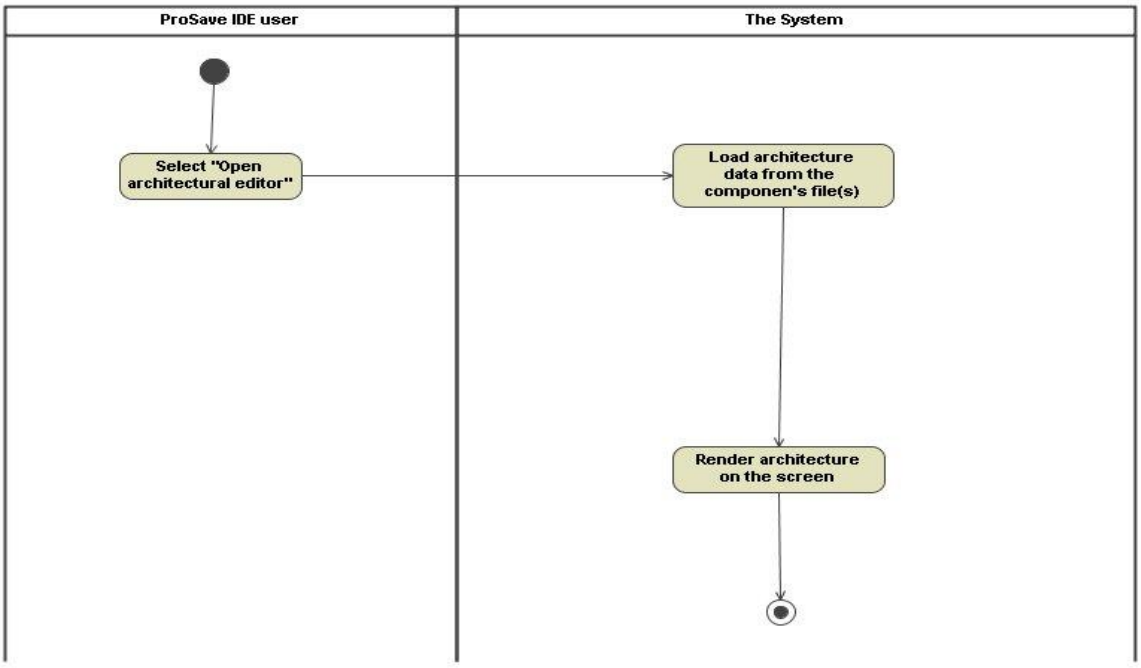
5. Import component from repository



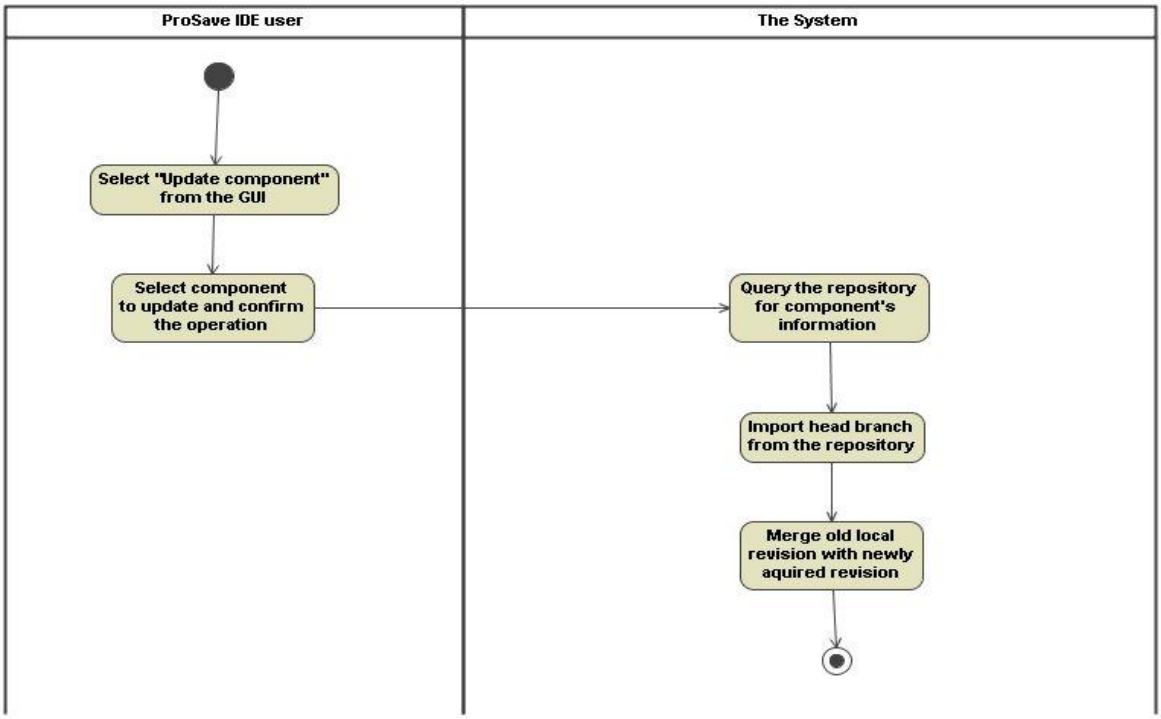
6. Copy component



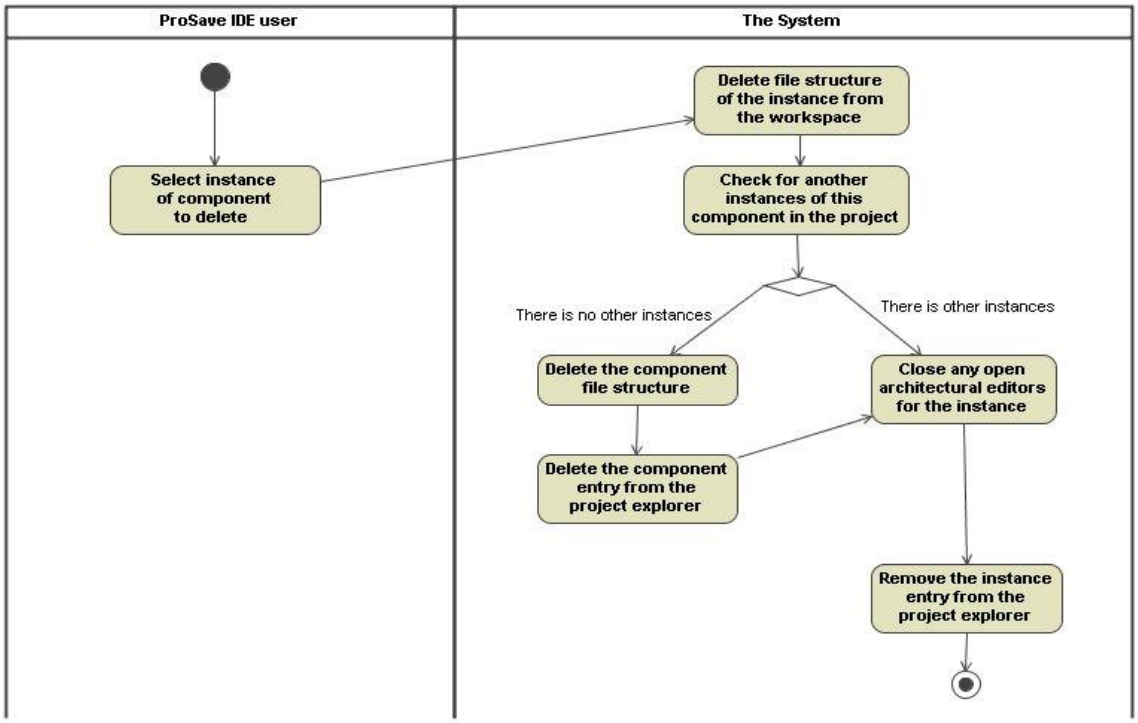
7. Open component architectural editor



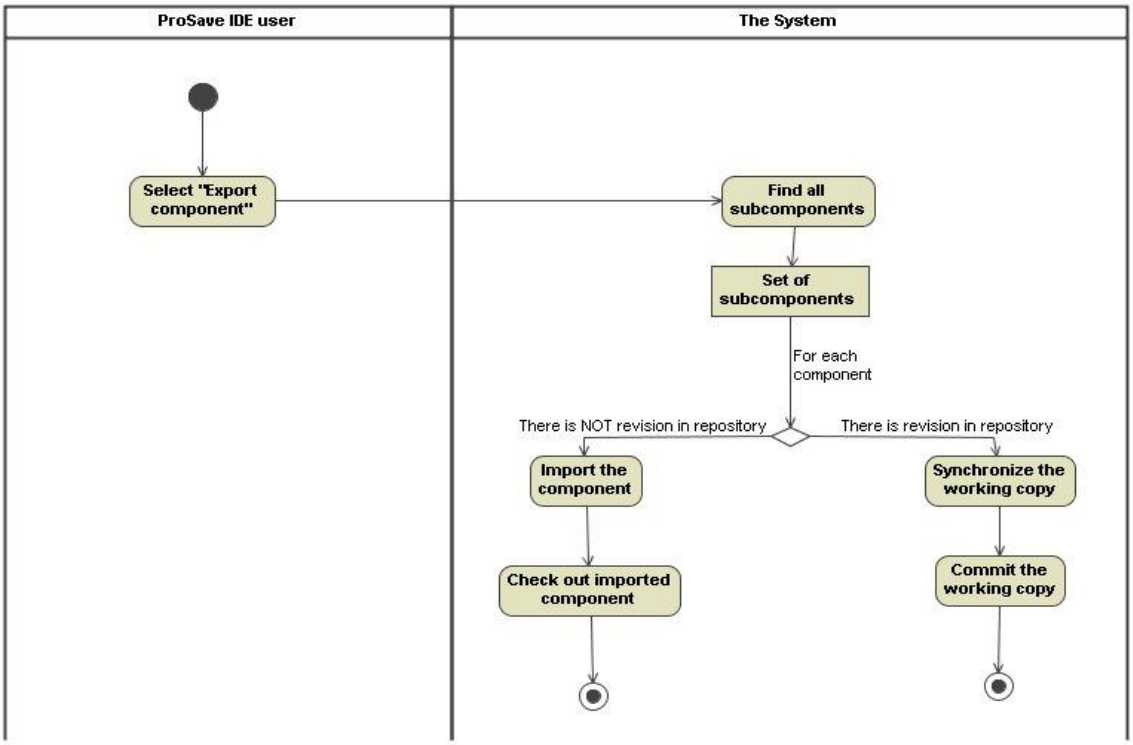
8. Update RW component



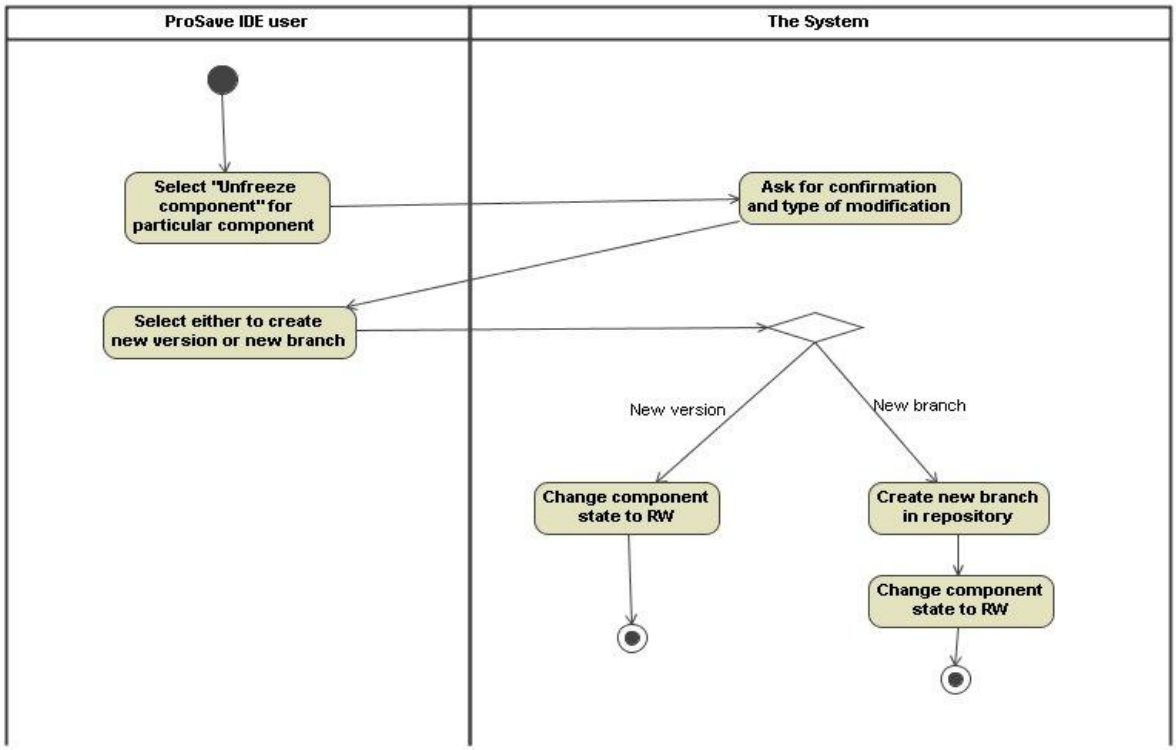
9. Delete component



10. Export component to repository



11. Unlock component



Appendix B: User manual

This section presumes that the system is executed in standard Eclipse environment as the work on the ProSave IDE is still in progress.

1. Prerequisites

Installation of Eclipse platform version 3.2 or above is needed. More information and downloads are provided on http://www.eclipse.org/downloads/. Then the two plug-ins "*Repository Access Plug-in*" and "*Repository ProSave IDE Integration Plug-in*" must be placed in */plugins* folder located beneath the Eclipse installation folder.
In order to connect to a SVN repository a reachable SVN server should be available along with a dedicated folder on its virtual file system. More information on how to set up a SVN server can be found here http://svnbook.red-bean.com/ , downloads and release information are available on http://subversion.tigris.org/ .

2. Working with the system

After the Eclipse is started, the Project Explorer view can be shown using the menus *Window* > *Show View* > *Other…* > *General* > *Project Explorer*. The general use information for the Project Explorer can be found here http://help.eclipse.org/stable/index.jsp?topic=/org.eclipse.platform.doc.user/reference/ref-27.htm . Before the work with the repository to begin, a properly created project is needed. The structure of the project must conform to the rules stated in paragraph 6.2. That is the project root must contain */components* folder, where all of the components are placed as a subfolders, and a file named *repository.properties*. The file has the following format:

```
RepositoryType=SVN
URL=<repository_url>
```

Where `<repository_url>` is the URL address of the dedicated folder on server's virtual files system. After this, the project must be configured by right-clicking on it and select "*Repository config*" command from the pop-up menu. Once the command is executed a new resource "*Repository*" is shown in the project. The resource can be unfolded which displays the available components in the repository regarding their latest

version – ***Figure 10***. All versions of a particular component can be seen by expanding the component tree. A component can be imported in the project by right-clicking on it and select "*Import Component*". The command "*Delete component*" hides the component when the repository is being browsed.

In order to export a component from the project to the repository, its root folder must be placed beneath the */components* folder and it must contain *info.properties* file which holds the repository-specific meta-data. The file has the following format:

```
Name=<component_name>
Id=<component_id>
VersionDesc=<version_description>
Children=<children_list>
Version=<version_number>
Subcomponents=<subcomponents_list>
Root=<root_folder>
```

Where:
`<component_name>` is the component full name,
`<component_id>` is the unique identifier,
`<version_description>` is short description of the changes in this particular version,
`<children_list>` is list of all children components in the following format
`<child01_root>;<child01_version>,<child02_root>;`
`<child02_version>,<child03_root>;<child03_version>, …`

`<version_number>` is automatically set by the system,
`<subcomponents_list>` is list of all dependant components in the same format as the `<children_list>`,
`<root_folder>` is the name of the component's root folder.

When a component already existing in the repository is exported from the project, its state as being modified must be represented in its root folder. For example the component with root folder */airbagVtype_a10id#20* is not in modification state, because it has attached version and hence it cannot be exported. The state of the component can be changed by renaming the root folder.
During the export operation it is possible that the conflicts occur between the local modifications and the changes in the repository. The repository system shows dialog window with a list of files which are in conflict state – ***Figure 11***. The users have to merge the files by themselves and then to right-click on the same component and select the command "*Resolve and commit*" which concludes the begun operation.

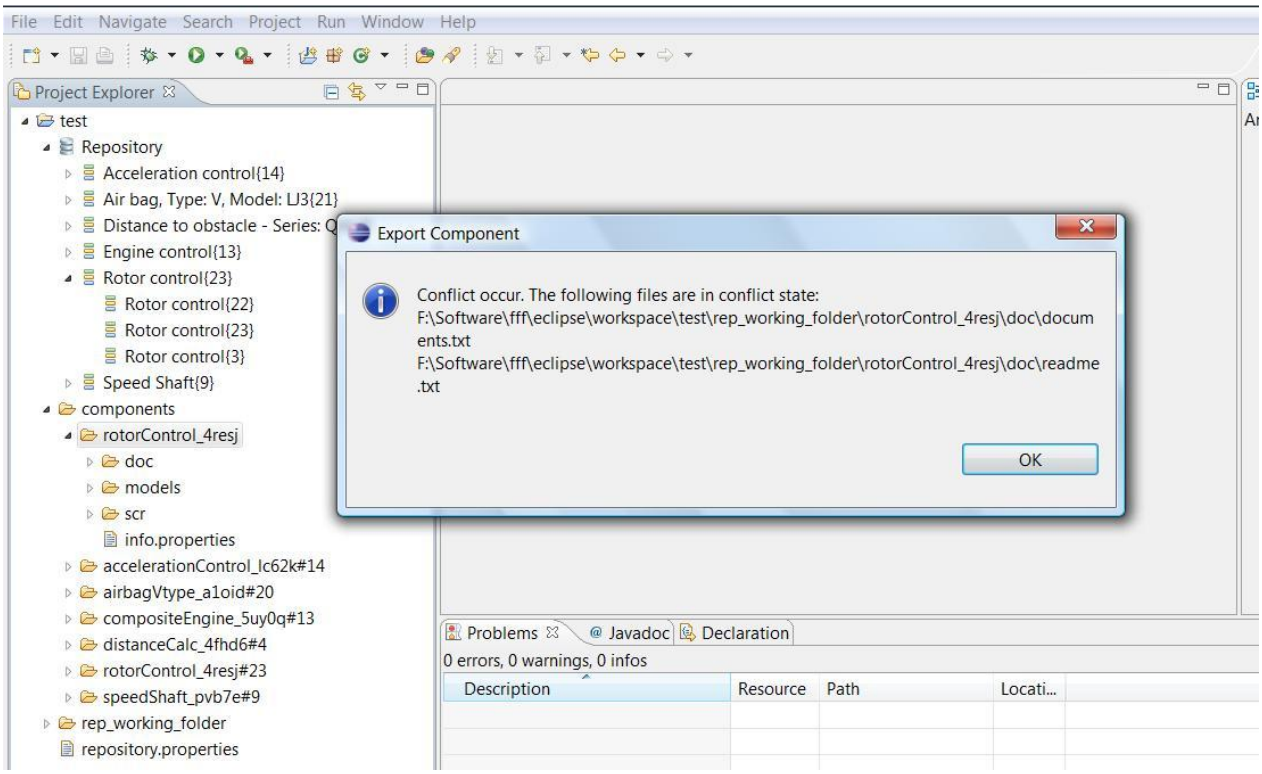


Figure 11 - Conflicts during export operation

The command "*Make Branch*" can be executed only on components which are already in the repository and are not in modification state.