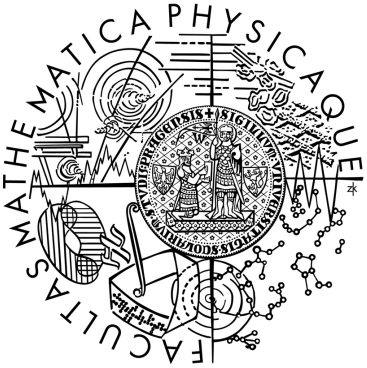
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Ondřej Kudláček

SOFA 2 graphical tools improvements

Katedra distribuovaných a spolehlivých systémů (32-KDSS)

Supervisor of the bachelor thesis:
RNDr. Michal Malohlava

Study programme: Computer Science

Specialization: Programming

Prague year 2011

At first I would like to thank my family. They have been a great support for me. I know, that there were times when I was not easy to cope with, and I am grateful that they did not let me down. I am also thankful for the help my supervisor provided me. Without his backing I would not have been able to complete this work. And finally thanks to my colleagues that they let me use their computer for development, while I was not able to use mine.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In…..... date..... signature

Název práce: Vylepšení grafických nástrojů komponentového systému SOFA 2

Autor: Ondřej Kudláček

Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů (32-KDSS)

Vedoucí bakalářské práce: RNDr. Michal Malohlava

Abstrakt: Tato práce se zabývá vylepšením komponentového systému SOFA 2 a je zaměřena na grafické řídící rozhranní. SOFA 2 je založena na modelovacím nástroji Eclipse Modeling Framework. Pomocí něj je vykonstruován hierarchický model tohoto systému. Rozšíření vyvinutá v rámci tohoto projektu poskytují skrz grafické rozhraní možnost upravovat tzv. Deployment plány aplikací vyvinutých v systému SOFA 2 a přenášet jejich komponenty mezi repozitáři. Grafické rozhraní je provedeno jako samostatný program nebo jako rozšiřující součástka (plug-in) do platformy Eclipse. Vylepšení systému je zahrnuto v součástce MConsole, pomocí které se dají vyvíjené aplikace v systému SOFA 2 spravovat. Celý program je implementován v jazyce Java. Grafická rozhranní jsou zkonstruována pomocí knihovny JFace, která umožňuje práci s modely založenými na EMF. Při úpravách modelu se používá framework EMF.Edit a jeho podknihovna Command.

Klíčová slova: komponentový systém, grafický nástroj, Eclipse, Java

Title: SOFA 2 graphical tools improvements

Author: Ondřej Kudláček

Department / Institute: Department of Distributed and Dependable Systems (32-KDSS)

Supervisor of the bachelor thesis: RNDr. Michal Malohlava

Abstract: This enhancement of component system SOFA 2 is focused on the graphical interface. SOFA 2 is based on Eclipse Modeling Framework, through which is constructed a hierarchical model of the system. The extension of SOFA 2 developed in this project provides – via the graphical interface tools for the so called Deployment Plan – an editing of applications developed by SOFA 2 and copying of their components between repositories. The graphical interface is created as a stand-alone program or as a plug-in for the Eclipse platform. The enhancement is added to the management console called MConsole, which allows editing models and applications developed in SOFA 2. The whole library is programmed in the Java programming language. The graphical environment is build on JFace library for EMF-based models editors. EMF.Edit framework and its library Command are used for the editation of the models.

Keywords: component system, graphical tool, Eclipse, Java

Contents

Preface

Over the past years the component-based development became one of the good ways to build software systems. The SOFA 2 system is an environment providing tools for creating those applications. These applications consist of models made of components. By this system they can be created and managed. SOFA 2 consists of a few independent parts. One of them is MConsole, a tool designed for administering the model's components and SOFA's runtime environment. It is a stand-alone application based on Eclipse Rich Client Platform or it is also available as a plug-in for the Eclipse environment. Enhancements created in this project are part of MConsole. They add new features to the system to provide more functionality and comfort during the work with MConsole and SOFA 2.

Enhancement called In-memory ADL editation implements recommended techniques of Eclipse Modeling Framework. It takes the settings of a developed application in SOFA 2 stored in ADL to the memory of MConsole.

Merging and cloning feature adds a new functions to MConsole. The repository of the system – another independent part of SOFA 2 – contains components of a modeled application. The SOFA 2 system may contain not only one repository. This feature adds an user interface to MConsole and provides merging and cloning of components between SOFA's repositories.

1. Introduction

Creation of applications and software in general is quite a new domain. Programming techniques like Object-oriented programming appeared in 1960s. Developers try to grasp the idea of what could a computer program do through these techniques. For example using the Object-oriented technique programmer divides the application into objects. Divisions of the program describe real or abstract objects that may have no meaning without the rest of the program. Another approach is called Component-based engineering. Its idea was first published at NATO conference in Germany, 1968 and IBM used it to create System Object Model in early 1990's. [8],[9] This technique takes the other side of programmer's view. Applications created by it are put together smaller components, packages containing related data and functions.

Components can be reused in contrary to the objects and to achieve it they implement interfaces for communication. In a typical implementation components do not have access to inner structure of other components. Moreover, components are substitutable, if user component required services are satisfied with substituted one's. Applications constructed by these independent packages are called component models. As components use their interfaces for communication, the model can be distributed through computer network.

The model can be of two types, flat or hierarchical. The flat component models are older and more advanced. And despite being not able to apply multiple techniques for one action, they are widely used. The hierarchical component models may be composed of other components, which makes them easier to make use of, e.g. more than one communication style between elements of a model. Because of that, they support more advanced concepts and features. But hierarchical models do not often appear outside an academical environment. These usually provide only a very limited platform without any repository or container or any basic service for components. These advanced features are very often applied in the flat models, that is why hierarchical models are used lesser. [2]

SOFA 2 is a hierarchical component system and includes support for tools, runtime environment, repository for storing components. It serves as a component-based application modelling environment and a complete framework [1]. Tools

contained in the SOFA 2 system are Cushion – an application development and repository manipulation tool – SOFA IDE – a graphical tool and an Eclipse environment plug-in – and MConsole – monitoring, SOFA 2 runtime environment maintenance tool and a stand-alone or Eclipse plug-in application. By these parts of the SOFA 2 system components can be created and models assembled. Settings of components and their communication can be done in MConsole. Because the parts of applications modelled in the system can be distributed through network, these settings may not be available locally but on a remote computer. But SOFA 2 does handle these settings directly.

1.1. Goals

The main purpose of this project is to enhance graphical tools of the SOFA 2 system. This includes a feature for the MConsole tool and an improvement of the tools-api library. These enhancements are directly related to the user interface of the SOFA 2 system and addition of them would increase its usability.

The SOFA 2 system contains except the repository model a meta-model or ADL. This meta-model is used for defining components and their relations, the settings of components. Developers who use the stand-alone or Eclipse plug-in versions of SOFA 2 have a copy of ADL stored in a local files. MConsole directly accesses and modifies them. This mostly concerns a deployment plan files containing execute instructions for the SOFA 2 runtime. To provide indirect accessing of component settings during editation process an enhancement called *In-memory editation* should be added to the SOFA 2 system.

Components are stored in repository of the system. SOFA 2 allows to operate not only with one repository. Its repositories are divided into development and stable and allow stored components to be cloned between them. MConsole as a tool for environment maintaining should provide a feature to support migration of components between repositories. This should be allowed through clone – copying a component from stable repository – and merge – copying a component from development repository – operations.

1.2. Structure of the text

Preface and the Section 1 introduces to the topic of component modelling and describes goals of this project.

The Section 2 provides basic description of the SOFA 2 EMF-based model and the meta-model processed by In-memory editation enhancement.

Section 3 contains construction steps that led to the solution of In-memory editation.

Section 4 tells about technical content. In the first subsection are described libraries and approaches used in the In-memory editation. The second subsection contains technical description of the *In-memory editation* part. And in the third technically describes the *Clone and merge tool*.

Section 5 shows how to use both enhancements. In the first subsection is described which parts of MConsole and SOFA 2 environment use the In-memory editation. In the second is shown how to clone or merge model components through Clone and merge tool in MConsole.

The last Section Conclusion sums the whole project up and tells about stability of the new enhancements.

2. Introduction of SOFA 2 model

2.1. Introduction to EMF

Modelling tools provide various functions for application development. There are tools for data modelling, applications structure languages for modelling or even object modelling tools. Since SOFA is based on the Eclipse platform and is also a plug-in, it uses its tools for modelling. The graphics of SOFA and mainly MConsole (which is described in the Used techniques Section 4.1 of Program documentation) come from Graphical Modeling Project of Eclipse foundation. It provides libraries for graphical editors based on Graphical Modeling Framework (GMF) and Eclipse Modeling Framework (EMF).

The EMF is a library provided by the Eclipse Foundation. It allows to access facilities of Eclipse platform. The main point of the EMF is that it creates a connection between Java programming language, Extensible Markup Language, and Unified Modeling language via generating code. It allows to describe a model in any of the mentioned languages and generate the others. For example using EMF after creating a XML Schema file Java implementation classes can be generated.

With the EMF come other tools, libraries, and frameworks. One of them is applied in In-memory editation part of this project. It is EMF.Edit framework which provides classes for building editors for EMF-based models. With its Command subsection it is possible to set, add or remove attributes of objects of a model. Moreover these modifications can be undone in the contrary with classic programmatic change to an attribute of a class. For more description and application of this framework see Sections 4.1 an 4.2.

2.2. Basics of the SOFA 2 model

Since the In-memory Editation part of this project works with the structures of the SOFA 2 model and meta-model, it is useful to mention it. This Section is taken from [2]'s Section Overview of the SOFA 2.0 component model.

"*In the SOFA 2 system, components interact through provided and required Interfaces with other components which can be either black-box or grey-box. The black-box does not provide any view of its internals, while the grey-box provides view of its inner structure. The black-box is represented by the component Frame.*

The grey-box is Architecture with implementation of Frame. The Architecture can be an implementation of a component or it can be a collection of other components (Subcomponents)." More detailed description can be found in the cited text.

In a programmatic look at the model, there are common classes as *NamedEntity*, *VersionedEntity*, and *Version*, which are used throughout the model. These provide to all its implementations a name and a version for versioning system of SOFA 2. *Frame* is a base element for component representation of the black-box type. It has references via provided and required lists to Interface to provide communication. The Interface is of a type defined by *InterfaceType*. Other elements of *Frame* are *Annotation* and *Property*. Annotation can be used to mark a *Frame* as top-level, which represents an entry component of the model. Property may define properties, as expected. For more details see Section 3 of [3].

The SOFA 2 system contains also a meta-model which is used to define components and capture relations between them. Through tools of repository the meta-model can be also used for generating components. Its structure does not differ from the model much. It contains also *Frame* or *Architecture* but these structures do not express the same as information as model structures do. The meta-model components contain references to components of the model and by these references the structure of an application developed in the SOFA 2 system is described. Structure of the meta-model shows the Figure 2.2.1.

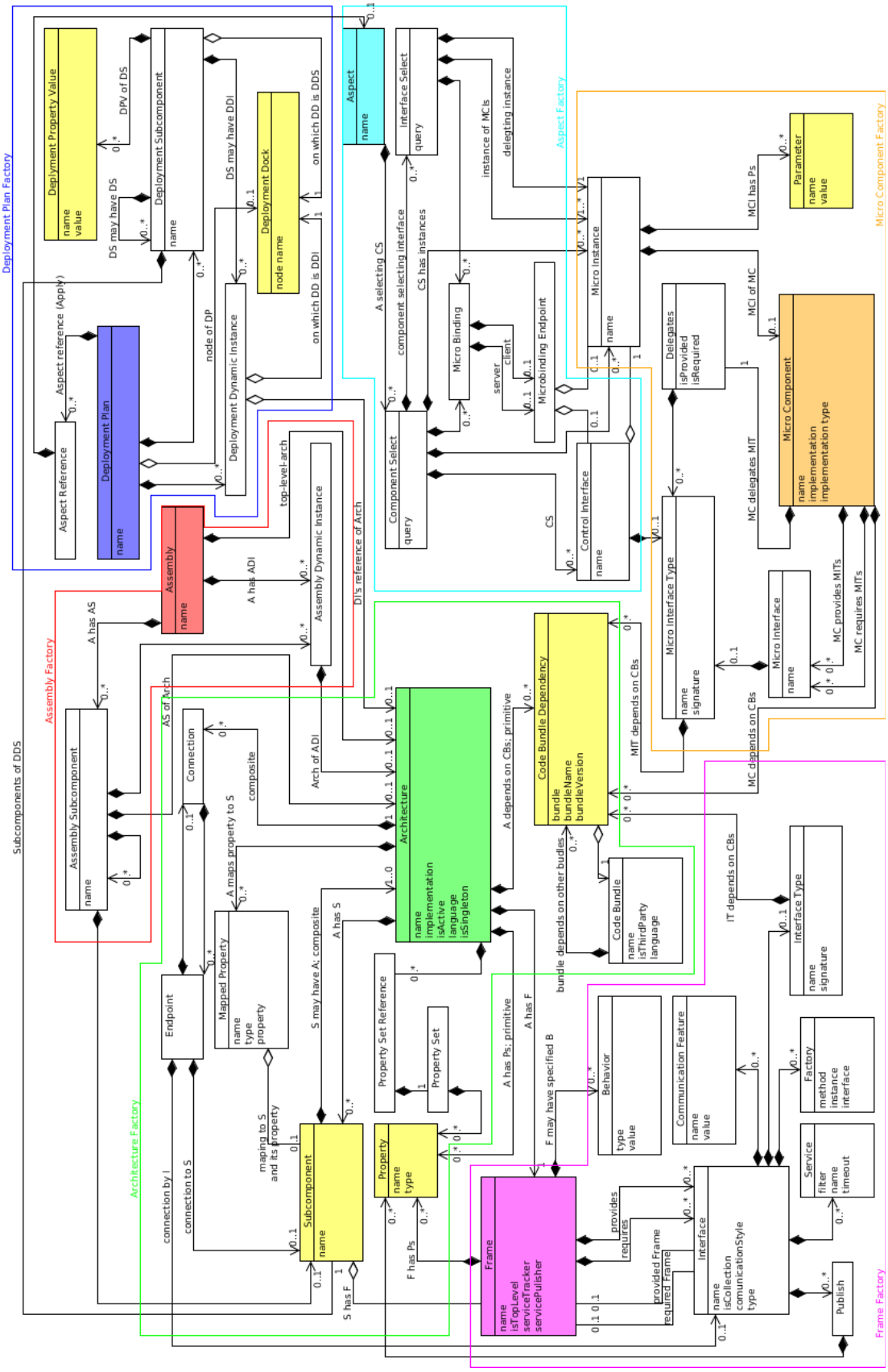


Figure 2.2.1: Structure of the ADL meta-model and objects division between factories

3. Analysis of the solution

The original version of SOFA 2 handles ADL meta-model by JDOM tools. The best interest is to simulate operations done with files containing ADL without any difference between the previous and the new versions of the manipulation routines. Fetching and saving of objects from and into repository is needed. The previous version of the modified routines already used repository access tools from sofa-repository project. This way are entities loaded from the repository to be processed and afterwards saved. The only thing required to provide editing of ADL in memory is to replace the implementation of JDOM tools.

MConsole and other parts of SOFA 2 use a library for meta-model manipulation operations, it is called tools-api (located in *org.objectweb.dsrg.sofa.tools* package). Classes located in the sub-package api contain tools for generating, exporting or creating the ADL files. These Actions need to be modified to use the In-memory-adl library (added to the tools-api library), which substitutes functions provided by JDOM. Two types of functions were done by the JDOM tools, creation and editation of ADL is the first one. The second is serializing and saving informations in ADL to a file in a computer file system. In-memory-adl is intended to provide only in-memory editation of ADL files, but since all the needed operations can be done through EMF, the added library provides construction and save procedures of the ADL structures too.

Starting with the creation and editation routines of an ADL model object a library is needed (the In-memory-adl library). To be able to set up every attribute there is in the model a pack of static functions should be provided. Approach of usual use of the set and get routines is possible. But since objects of ADL model do not contain same attributes an abstraction for setting up process is required. The ADL model is based on EMF, that means abstract set and get routines are provided. Using these routines to set up objects would be better, because of the abstraction they provide. But EMF models also have a description structure of components and their attributes or features. This contains information about every attribute there is in the model. The framework EMF.Edit – mentioned in Sections 2 and 4.1 – uses these feature descriptions and allows to set up attributes of model's objects. The framework does the process by creating a command for every modification. This command can be executed to perform the change and afterwards undone if needed. Moreover the

use of the Edit framework is recommended for EMF model editation. For more information see [7].

Routines for setting, adding, removing and clearing attributes are required to imitate the processes done with ADL files by the JDOM tools. *AdlCreatingAction* class responsible for generating ADL uses the JDOM routines the most from the classes. Feature identifying numerical constants – provided by the ADL model – are used to specify operations e.g. adding a dependency to an architecture. Every constant (in *SOFA2ADLPackage* class located in *org.objectweb.dsrg.sofa.adl* package) uniquely describes a certain feature of an object in the model. This method of determining a feature allows to use the In-memory library without any need of including any other libraries in the user class. On the other hand creating a *Command* to edit a feature by the Edit framework requires a literal of that feature (*EStructuralFeature* class). This feature class also uniquely describes any attribute and is also accessible in the ADL packages. The technique using numeral identifiers requires to write a unique setting function for every feature. Each function would than represent a unique feature literal class that would be used inside it to create a set up command. Also to be able to use these functions a tool would be needed for deciding which numeral identification goes to which unique function.

As it turns out the comfort of no need to use other libraries comes for a high price of unsuitably static pack of functions. It would require a suitable modification every time the ADL model is changed. More dynamic solution is needed to support future changes of the SOFA 2 ADL model. This leads to second solution of object modification routines. Using feature descriptions (*EStructuralFeature* classes) allows to provide same functionality as using provided constants, but in a dynamic way. Feature describing objects contains all required data, e.g. name of the feature or type of the value contained in the feature. Using Reflection API of Java language even type checking can be done.

The other thing that the previous JDOM approach provided is saving ADL structures. JDOM handles ADL in XML structured files. Those files are stored in a file system, where the current part of the SOFA 2 system is running, e.g. MConsole. Storing of the files is done by the *java.io* library. But the EMF tools can be used to provide local file system independent ADL storing. To use the serialization provided by EMF and in SOFA 2 implemented XML factories the In-memory editation library

contains save, load and remove routines required by the Action classes of tools-api. EMF provides structural containing of objects. As is described in the Sections 4.1 and 4.2. *Editing domain* contains *Command* classes. It also may contain the edited objects under a structure of resources. *Editing domain* contains *Resource set*, this contains *Resource* objects which are bound to a specific location described by a uniform resource identifier or in short URI. EMF provides also load, save and delete routines which require the identification. To allow access to these routines to Action classes of tools-api, the library simply provides them to imitate the previous JDOM version.

The in memory editation is achieved by the EMF tools and its XML Java transformation tools. By adding an object of the ADL model to a *Resource* in an *Editing domain* and setting its URI makes it ready to be transformed and saved to a specified location. When objects are serialized from their memory form, they can be deserialized back to the memory when needed.

Using this approach of EMF model storing allows to perform similar action while handling the ADL files as if the JDOM tools were still used. It provides access to edited modelled object in memory while it can be stored in a serialized form in file system available for other parts of the SOFA 2 system.

4. Program documentation

4.1. Used techniques

It is quite hard to compare this enhancement to other tools or libraries since it is made for a specific system. The only way to confirm that the In-memory editing is done in the right manner is comparing it with other EMF model editors and recommended approaches of EMF-based models editation. All entities of the SOFA 2 model are based on Eclipse Modeling Framework. It was found useful that EMF has its own library for modifying model's objects. According to the Java language it is located in the *org.eclipse.emf.edit* package. That is provided mainly by the sub-package command. For every EMF object the command toolkit allows to create certain modification commands and add it to the *EditingDomain* object. If needed, a command can be created by specifying the *EditingDomain*, the owner of a modified feature, the feature itself, and a new value. Thereby the concrete object of model can be modified.

As described above, this toolkit takes care of editing. The loading process of the ADL data is done by XML-based techniques, since EMF works with XML schema. This simplifies work with ADL files. Basic routines processing the ADL files use libraries of JDOM project. That provides for SOFA necessary abstraction for reading and writing ADL files.

EMF.Edit and EMF is suited for JFace *viewers*. In SOFA's MConsole, these viewers are used for visualization of components and model in a repository. The model object is accessible via *ContentProvider* adapter. It provides a mechanism for a JFace *viewer* to get required object's attributes or sub-objects. This comes very handy in case of the hierarchical model. As the *TreeViewer* is in MConsole used quite often, its description is in place. This JFace object provides a structural view of a object for the user interface. Through its *TreeContentProvider* class, which inherits the *ContentProvider*, it accesses the model by getting children and parent items of the observed object. Figure 4.1.1 describes this situation.

The viewer actually does not handle the received components of the model as objects. The *TreeViewer* works with generic items which are obtained through *AdapterFactoryContentProvider* class, by EMF adapters knowing how to process the structure of components for JFace type *viewer*. [7]

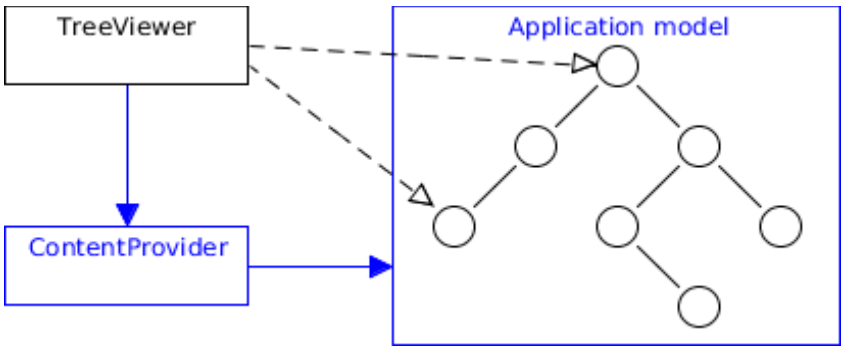


Figure 4.1.1: Access of *TreeViewer* to the model [7]

4.2. In-memory editation

The SOFA system is implemented in the Java programming language and so is this enhancement. The documentation to the source codes, generated by the Javadoc tool, can be found in the attachments of this thesis.

Structure of the enhancement is shown on the Figure 4.2.1. According to the Java packaging, its location is *org.objectweb.dsrg.sofa.tools.adl.edit.memory*. The main entry point class of the library is located in the sub-package *impl*. Its name is *InMemoryAdl*. It extends the *InMemoryModelFactoryImpl* class to provide a getter of resources and a Universal Resource Identificator (URI) creator according to the usual SOFA resource identification. The extended class actually provides all the abstract routines for working with the model's objects. This could be also used as an entry point of the library. However to keep the SOFA class structure and use common (not only Java) programming technique, the *InMemoryAdl* class was added to represent the top of the library. It also adds loading and saving routines for model's object storage.

According to the Analysis Section, two approaches are possible a dynamic and a static. For observation both of them are available in the In-memory editation library. To this point of the technical description both are the same. Numeral identification approach is described first.

Lower level factory classes in the library represent handlers of concrete operations like creating an *Architecture* object of the model and call static functions

in the *Helper* class. The main factory has abstract routines for creating (*createModelObject*) and modifying (*updateValue*) the model's objects. These routines decide which sub-factory will take care of the request. The decision depends on the type of the modified object while updating a value or on a number of the type from *SOFA2ADLPackage* from *org.objectweb.dsrg.sofa.adl* package. When the request is forwarded to a lower level factory class, its routine calls one or more static functions from *Helper* class, which represents the lowest level of the library. The low level factories do not do any processing but they provide a certain level of abstraction of the Helper class, like adding a collection of objects to a feature. Since the *EditingDomain* object is necessary for performing the modifications, they also contain a pointer to it. Thus every call to modify the object does not need it as a parameter. This means that the domain object has to be the same in the instance of the factories – the instance of the main *InMemoryModelFactory* contains pointers to the lower level factories which makes them contain the same domain object. If the domain was changed in one of the factories, the objects returned by every modification call would not be consistent.

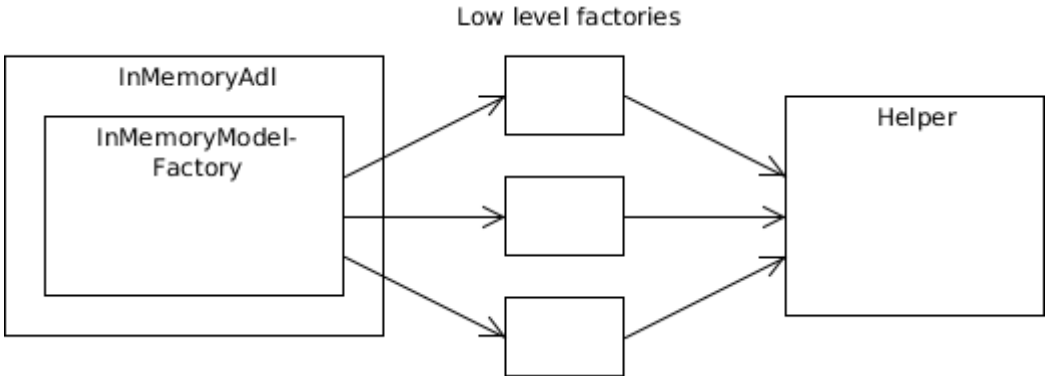


Figure 4.2.1: Structure of the library

The lowest level of processing the modifications is the *Helper* class. It contains only Java language static functions, which means the function works only with the parameters and nothing else. Beacause the count of the functions is high, making them not static could be quite confusing. The creator functions work only with the *SOFA2ADLFactory* (package *org.objectweb.dsrg.sofa.adl*). They do not add the created object to the *EditingDomain* because on the low level the library does not have any information about where to add the new object. The adding of it is therefore

left for the user of the library. Edit operations on the other hand add new information to the domain.When the modification is specified via choosing the right function in Helper through factories, the library has all the necessary information to perform the action.

Using the EMF.Edit framework, the particular function creates a command to edit an object of the model. Creating a command requires *EditingDomain*, which is specified by a low level factory, an edited object, a feature of the object – specifying which attribute is to be changed – and the new value. When a new command is created, it is performed using the execute routine of *CommandStack* (*org.eclipse.emf.common.command* package), which is accessible in *EditingDomain*. This processes the command according to the EMF.Edit approaches. It performes a test whether the command is executable, it cleares all undone commands (as described in the 4.1 Section, EMF.Edit provides an undo-redo interface), and executes the new one. After the command is performed, that means the modification is in place, the modified object is returned as a result of the operation. The returning might seem useless, since one of the parameters is the model's object, but in some cases it could be used e.g. to campare the original and the modified objects.

Now the secondary approach is described. Using the same entry classes to the library, the dynamic editation does not require the low level factories. *InMemoryAdl* class provides also the same pack of set, add, remove and clear routines as the static approach does. But these ones require *EStructuralFeature* parameter and directly use three functions in the *Helper* class, which are *setValue*, *addValue* and *removeValue*. Because the ADL model implements EMF, the *SOFA2ADLFactory* can create new instances of classes by only having an *EClass* object. The edit routines refer directly to the features owned by the classes of ADL model. In class *SOFA2ADLPackage.Literals* is listed each of those features. These are used in calls of the edit routines from Action classes modified in tools-api – Figure 4.2.2 describes access into the library. Moreover implementation of EMF in ADL provides in feature literals classes contained in features of the ADL objects. This in combination with Java Reflection allows to dynamically check types of inputed values and throw exceptions if needed.

As described in the Analysis Section, the second approach is implemented in the tools-api project because of its dynamism. Routines of the first approach are

marked as deprecated to ensure further users of the In-memory editation library that the dynamic routines should be preferred.

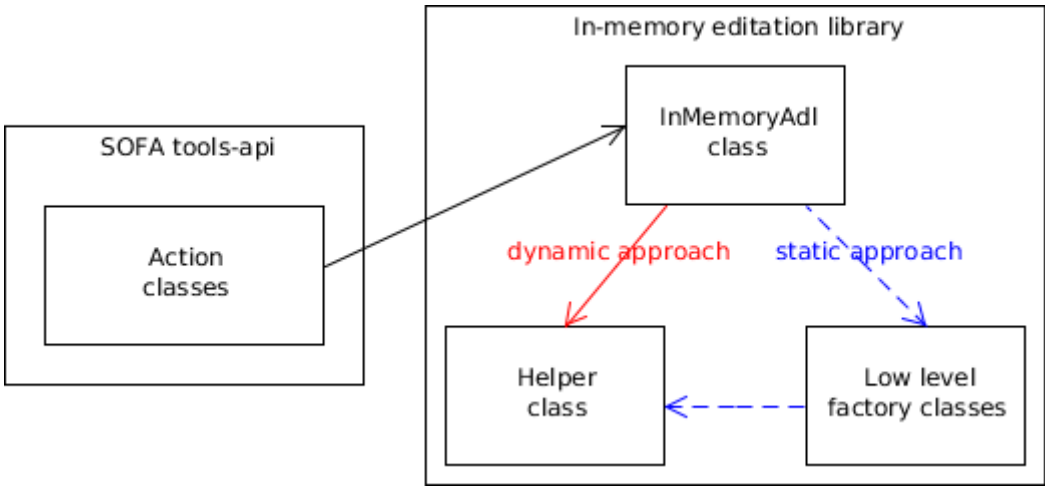


Figure 4.2.2: Two approaches of the ADL objects editation

To get insight of the ADL model and how objects are divided between factories see Figure 2.2.1. The Unified Modeling Language (UML) diagram shows relations and all components of this model.

Modificatins done in the SOFA 2 system to implement this enhancement were done mainly in the tools-api project. In the package *org.objectweb.dsrg.sofa.tools.api* in Action classes. Mostly modified class is the *AdlCreatingAction* which takes care generating ADL objects out of a model objects. JDOM *Element* classes were replaced with *EObject* classes of EMF. Classes for checking out resources in repository, exporting, deploying and committing created components – operations accessible in SOFA 2 repository view – are modified. Also actions for preparation of deployment plans and assemblies are changed to implement approach of In-memory editation library.

The editing domains mentioned previously are in the modified sections of SOFA 2 used separately for each operation to divide logical actions of constructing, loading and saving an ADL object. For example, performing the *Checkout* action loads first domain to seek for an existing files of an ADL object that is about to be generated from a model's object. Than a second domain is used to construct an ADL object using the *ADLCreateAction* class. Finally by a third domain a serialization and a save operation are performed. This behavior can be changed in the In-memory

editation library by forbidding construction of *ADLCreateAction* an Editing domain object for separate operations.

4.3. Clone and merge tool

Since MConsole is an Eclipse plug-in or a stand-alone application, it is defined by *plugin.xml*, *MANIFEST.MF* and source code files. The first two describe how and in which scenarios is the programmatic representation used. The Clone and Merge operations are accessible from a context menu selecting one or more entities in the MConsole Navigator, as described in the User documentation Section 5.2. The access point of the copy procedures are classes *MconsoleCloneAction* and *MconsoleMergeAction* in the package *org.objectweb.dsrg.sofa.mconsole.ui.actions*. To follow the same feature style, these Action classes just construct and launch the wizards as Figure 4.3.1 displays. Extending the *MconsoleAction* class and overriding run method of *org.eclipse.ui.IActionDelegate* represent an Eclipse plug-in. The call of the whole MConsole UI to launch this action starts a new thread, as typical user interfaces do.

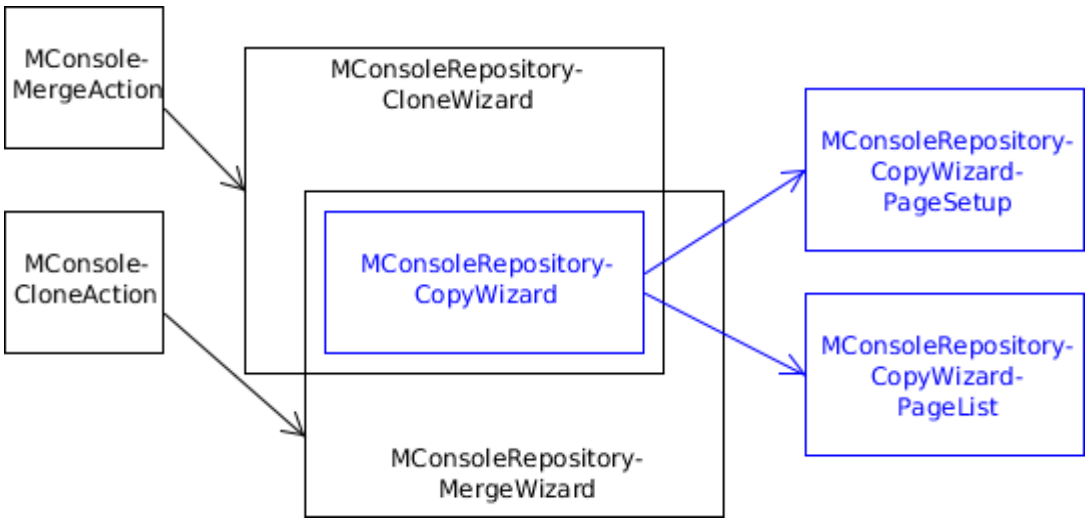


Figure 4.3.1: Class structure of the Clone and merge tool

The Action prepares selected resources for *org.eclipse.jface.wizard.WizardDialog* and launches it using the open routine. The wizards of merge and clone operations are represented by the *MConsoleRepositoryCloneWizard* and *MconsoleRepositoryMergeWizard* classes.

Both of them contain an unique identification of wizard. These two classes extend *MconsoleRepositoryCopyWizard*. Since both wizards would be the same – expect wizard's labels and action called by Finish button – an abstraction was made. The copy wizard itself extends *Wizard* class of *org.eclipse.jface* package, overriding *init*, *addPage* and *performFinish* routines, and *INewWizard*, an interface for creating wizards for Eclipse plug-ins (package *org.eclipse.ui*).

The initialization method transforms selected objects from input (MConsole Navigator menu) to a list of *IMConsoleRepositoryResource* objects, which are used through the operation of copying. The most important is the *performFinish* routine. It is called by the plug-in when the Finish button is pressed. It loads input information, initializes the cloning or merging process, and launches it. As is described in the user documentation Section 5.2, this is the point where a dialog listing touched entities may appear.

The wizard consists of two pages. Both of them extend *org.eclipse.jface.wizard.WizardPage* and represent the UI of the pages. The only technical things in these classes are buttons for checking connection to repositories. Listener technique is used here. It uses an inner private class with another thread. The connection is checked by a try to create a *RepositoryAgent* (package *org.objectweb.dsrg.sofa.repository*) class with a specified URL. The agent represents a handler of a repository in the SOFA system. Result of the try is displayed through a label next to the Check button. If the repository is found, its content will be loaded to the viewer on the next page of the wizard.

Since connecting to the repository may take a while, launch of the wizard is not instant. The same issue is when switching to the second page of the wizard, when the repository data is loaded to the tree viewer.

The processes of cloning and merging itself does not involve any new features added in this project. This feature focuses only on setting up and handing over the received information to the *RepositoryCloner* class in *org.objectweb.dsrg.sofa.repository* package. This class takes care of copying the components between specified repositories.

Libraries mentioned in the previous Sections were used according to the recommended approaches in the Clayberg, Rubel: Eclipse Plug-ins (3rd Edition) [5].

The structure of *Action*, *Wizard* and *WizardPage* was taken from other wizards of SOFA 2 to keep the same flow of computation. There are other possibilities of the wizard, e.g. creating the wizards with more pages. But to keep the UI as simple as possible, this turned out, after few programming cycles, to be the right way. There was also a problem with running the merge and clone process in a separated thread. It would be very useful to indicate still running operation by changing text of the Finish button but logically that would corrupt the idea of it being a finish button. Showing another dialog for the operation would also defeat the simple working UI. That is why any extra thread was not added.

One thing may appear not finished in this feature. While launching the wizard, the repository connection check may take a while so a status dialog may have been used. The same thing happens when switching this wizard to the second page. The repository URL is loaded and its contents are visualised in the tree. But while the repository is being contacted and its contents loaded, there is no status indication, thus through a visualisation the progress would appear to be stopped. So adding this would be useless.

5. User documentation

5.1. In-memory editation

Tools of SOFA can be accessed as a plug-in for the Eclipse development environment or as a stand-alone application. In both cases, the user interface (UI) is the same. The main points of the SOFA 2 system are the environment and the management console MConsole. The environment provides tools for creating and modifying elements of the model. MConsole allows to modify communication and relations between elements of the whole model. It also provides access to repository of the model's elements. The enhancement described in this Section can be accessed through MConsole and used by tools of SOFA 2 environment.

The user interface and routines done during the use of MConsole's edit wizards have not changed compared to the previous version. The new feature effects only the inner mechanics and behavior of the program. The modified tools are in the edit wizard for *DeploymentPlan*. The wizard is accessible in MConsole Navigator view via Edit element in context menu of a Deployment plan item in Deployment plans folder in list of components in repository.

On its left side, the first page of the wizard shows a tree structure of the edited deployment plan and its sub-components. Selecting elements from the tree changes the right side of the page. For deployment plan it shows options such as those shown on the Figure 5.1.1. This allows to set name and the name of the node of the deployment plan. Selecting a sub-component in the tree (if available) shows a setup of the *DeploymentSubcomponent*, allowing to change its name, node, and properties. The tree also lists elements of the *DeploymentDynamicInstance* type if there is one owned by the *DeploymentPlan* or *DeploymentSubcomponent* elements. After the selection, fields with dynamic instance's node and architecture name and version are displayed. Editing any of the mentioned attributes runs a component change process through the new feature's library. The next page of the Deployment Plan edit wizard (Figure 5.1.2) provides a view of *Aspect* elements connected to the edited deployment plan (the feature of *DeploymentPlan* component is called *Apply*). It allows to add, remove, or set aspects. It even provides buttons to move aspects in

the list of the edited deployment plan. This comes in handy in case that the deployment plan applies to interdependent components.

Setting or adding attributes through the wizard requires those attribute to exist in the SOFA repository. For example adding a property value to a deployment sub-component requires that property value to be already present in the repository. If an invalid configuration is selected an error dialog will appear after confirming the edit wizard. After confirmation of the dialog the wizard starts up again repeatedly with the previously selected configuration. The repetition continues until there is no error or the wizard is canceled.

Tools of the In-memory editation can be also accessed through SOFA 2 environment. Operations done during creation of new components or checking out of objects in repository use routines of In-memory editation, but do not change any of its functionalities.

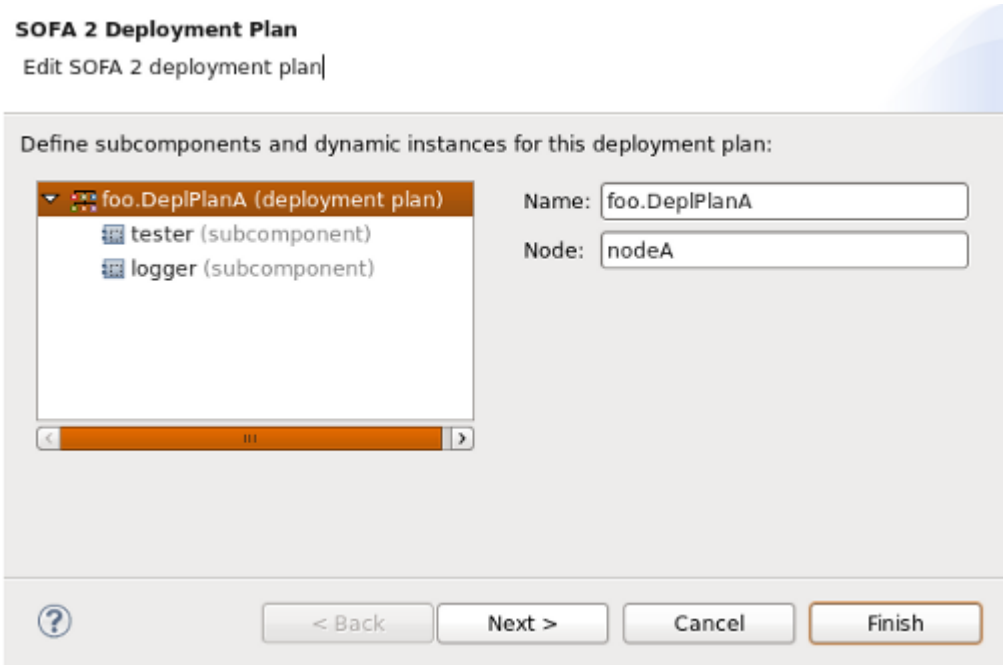


Figure 5.1.1: Deployment plan edit wizard main page

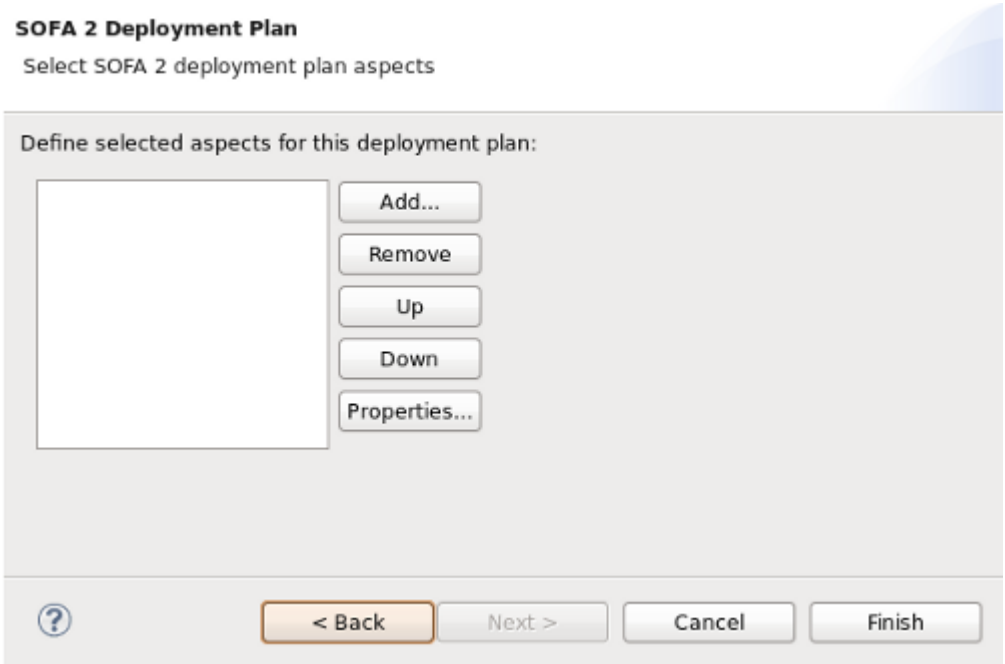


Figure 5.1.2: Deployment plan edit wizard aspect setting page

5.2. Merge and clone tool

This feature makes cloning and merging processes easier without any need to use command line. It is accessible in the MConsole's user interface (UI). The merge wizard is launched by selecting one or more components in a repository of a *SOFAnode* in Navigator menu and selecting Merge in context menu. The clone wizard is run likewise.

The wizards are nearly the same for both the actions. They have two pages. The first one allows to select source and destination repositories by entering their Uniform Resource Locator (URL) addresses. Accessibility and correct type of repositories can be checked by pressing the Check connection button. Dry run a Non recursive run check-boxes provide more setting for the operation. Checking the Dry run box processes only a simulated operation without copying any components. After everything is done, a list of components effected by the procedure is displayed. The Non recursive run box results that only the selected components are processed during the operation, all dependencies being ignored. Figure 5.2.1 shows the interface.

On the next page a list of components available in the source (if the repository is accessible) is displayed. Checking component's boxes makes it part of the operation specified on the previous page. The view of repository behaves as expected. By checking for example element Frames, all Frame components are selected. The same happens if the components are selected one by one. Figure 5.2.2 shows an example of this page.

The most important thing of the wizard is, of course, the Finish button, which launches the specified operation. After the operation is done a dialog appears with list of the touched components. Moreover, if Dry run is selected and the dialog is skipped by Cancel button, the UI will return to the wizard. This allows to launch normal clone or merge operation after simulating the process, without any need to select required components and set up source and destination repositories all over again.

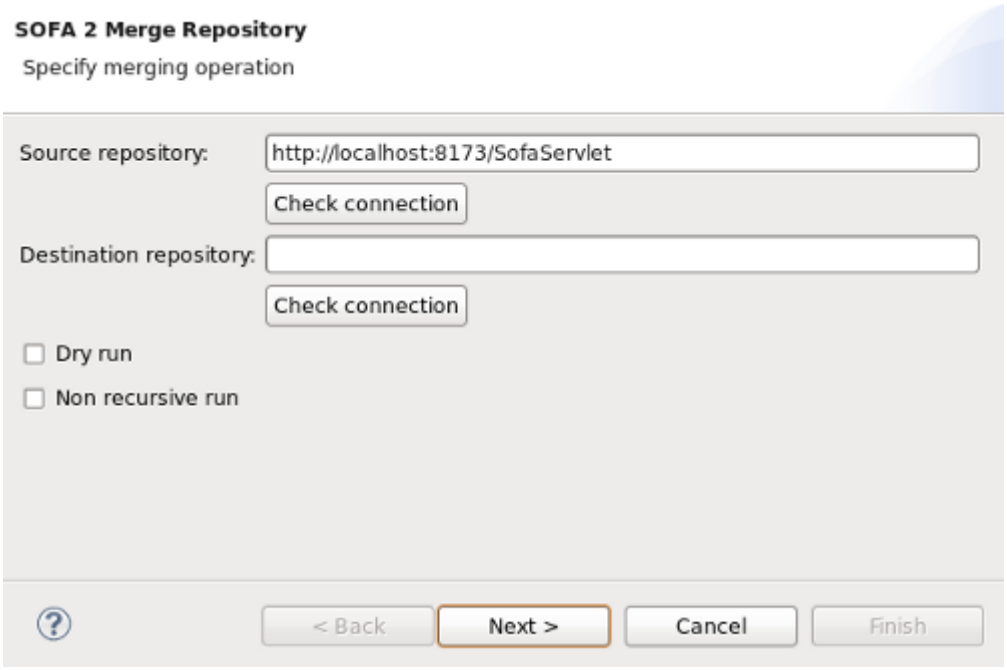


Figure 5.2.1: The first page of the merge wizard

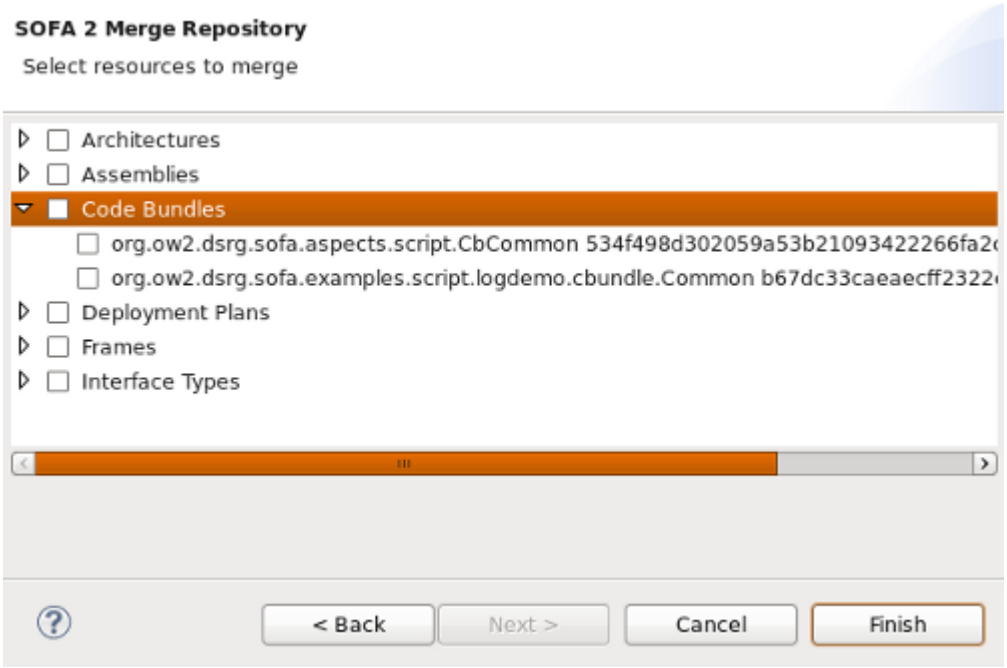


Figure 5.2.2: An example of a component list in the merge wizard

Conclusion

The SOFA 2 system is a well constructed base for developing component-based applications by providing advanced features and hierarchical model editors. The In-memory editor of ADL files makes the environment of SOFA 2 more usable. Which environments used for developing large scale software systems typically are, so that numerous programmers can work simultaneously on a project. For administrators of developed applications, merge and clone features provide more comfort than the command line scripts that were part of the previous version of the SOFA 2 system. Moving of developed components between repositories is a common action, like committing a written code to a version system server. Development and stable repositories are supposed to be part of development environment of applications developed in SOFA 2. These new features bring work with repositories closer to the user and the MConsole's interface.

Stability of the added library was tested via its implementation in the *sofa.mconsole.ui*, *sofa.adl.presentation.wizards* and *sofa.tools* packages. During the testing of the new features and the changes, no crashes of the program were observed.

Bibliography

[1] SOFA 2 component system website [online]. 2006 [citation 2011-11-30]. Accessible at WWW: <http://sofa.ow2.org/>.

[2] Bures, T., Hnetynka, P., Plasil, F., Klesnil, J., Kmoch, O., Kohan, T., Kotrc, P.: Runtime Support for Advanced Component Concepts, Proceedings of SERA 2007,
Busan,Korea, IEEE CS, ISBN 0-7695-2867-8, pp. 337-345, Aug 2007.

[3] Bures, T., Hnetynka, P., Plasil, F.: SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model, Proceedings of SERA 2006, Seattle, USA, IEEE CS, ISBN 0-7695-2656-X, pp.40-48, Aug 2006.

[4] Steinberg, D., Budinsky, F.,Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework (2nd Edition),
Addison-Wesley Professional, December 26, 2008.

[5] Clayberg, E., Rubel, D.: Eclipse Plug-ins (3rd Edition), Addison-Wesley Professional, December 21, 2008.

[6] Cerny, O., Hosek, P., Papez, M., Remes, V.: SOFA 2 Component System User's Guide, November 9, 2009

[7] WebSphere Application Server [online]. June 1, 2004 [citation 2011-11-30]. The EMF.Edit Framework Overview. Accessible at WWW: <http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=%2Forg.eclipse.emf.doc%2Freferences%2Foverview%2FEMF.Edit.html>.

[8] Rainer Niekamp: Software Component Architecture in Institute for Scientific Computing, TU Braunschweig. July 29, 2011

[9] Report on a conference sponsored by the NATO Science Committee. *Software Engineering*. October 7th to 11th, 1968, Garmisch, Germany

List of Abbreviations

ADL – Architecture Description Language; a language used to describe and represent software architectures

EMF – Eclipse Modeling Framework; a library provided by Eclipse foundation to model and generate code for development of applications based on component data model

GMF – Graphical Modeling Framework; a library provided by Eclipse foundation to create editors of EMF-based models with UI

UI – User Interface

UML – Unified Modeling Language; a standardized language with graphic notation techniques for creating models of software systems

URI – Uniform Resource Identifier; a character sequence for identifying a resource on network; more abstract than URL

URL – Uniform Resource Locator; a character sequence with a defined structure for specifying exact locations of resources

XML – Extensible Markup Language; a type of encoding documents in computer-readable form

List of Figgures

Attachments

Content of the enclosed CD ROM

With this thesis comes a CD ROM containing source code of the implementation and binaries for installation to the Eclipse development platform. The CD ROM contains:

/bin/
: Files for installation of the environment to the Eclipse platform.

/doc/
: Electronic version of this thesis and a PDF file containing a scalable meta-model visualisation.

/src/
: Source code of the modified and added classes with documentation in Javadoc.