SOFA 2 Component System

Developer's Guide

Ondřej Černý
Petr Hošek
Michal Papež
Václav Remeš

SOFA 2 Component System: Developer's Guide

by Ondřej Černý, Petr Hošek, Michal Papež, and Václav Remeš

Table of Contents

List of Figures

List of Examples

Introduction

The Programmer's documentation of the *SOFA 2* project contains the documentation needed by the programmer who wants to extend the functionality of *SOFA 2* or the tools provided with it - *SOFA Tools API*, *Cushion*, *SOFA IDE* or the *MConsole*.

In this text you will find the description of the inner functionality and representation.

Before reading this text you should be familiar with the User's documentation of *SOFA 2*. Also note that this text focuses on basic principles of how our the project was developed and not on implementation details which can be found in the javadoc or directly inside the source code of the appropriate parts.

Part I. SOFA 2

The sections in Part I present SOFA 2 component system.

Chapter 1. SOFA overview

The next few sections give an overview about the main parts of the *SOFA 2* framework. Each part is then discussed in more detail in a separate chapter.

1.1. The structure of the SOFA 2

Although *SOFA 2* is a quite complex framework it can be divided into separate parts with clear borders. This division is useful because it allows to distribute only those parts which are necessary at the moment and thus contributes to better lucidity. The *SOFA 2* framework has been divided into central Repository, Deployment Dock, Deployment Dock Registry and Microarchitecture parts. Central Repository is used to store and share all data and metadata among other subsystems of *SOFA 2*. Deployment Dock is in role of component container, which instantiates and controls all components. Because in most cases there will be more than one Deployment Dock running, there has to exist also a global registry which would register all these docks. Through this registry, Deployment Docks can get a reference to other Deployment Docks and communicate with them. This registry is called Deployment Dock Registry. Each component also has an internal infrastructure. This infrastructure is called Microarchitecture. It involves all internal parts of components except their business code.

1.2. Packages

The following paragraphs describe the individual projects and their packages. This gives better orientation in each project.

`org.objectweb.dsrg.sofa` - interfaces, exceptions and the class loader used in microarchitecture

`org.objectweb.dsrg.sofa.bootstrap` - bootstrap aspects implementation

`org.objectweb.dsrg.sofa.deployment` - deployment implementation

`org.objectweb.dsrg.sofa.deployment.launcher` - implementation of the Launcher class which takes care of the client side of autoconfiguration

`org.objectweb.dsrg.sofa.deployment.launcher.gui` - classes implementing graphical user interface of the Launcher

`org.objectweb.dsrg.sofa.deployment.util` - utilities used to run the *SOFA 2* runtime.

`org.objectweb.dsrg.sofa.deployment.launcher.zeroconf` - the Zeroconf server and related classes take care of holding and exposing the configuration of the *SOFA 2* node.

`org.objectweb.dsrg.sofa.dockregistry` - dock registry implementation

`org.objectweb.dsrg.sofa.microarchitecture` - microarchitecture implementation

`org.objectweb.dsrg.sofa.repository` - classes which are used to work with repository (initialization, querying, downloading, uploading, ...)

`org.objectweb.dsrg.sofa.repository.core` - core classes which are used during the work with repository

`org.objectweb.dsrg.sofa.repository.model` - repository model interfaces (generated)

`org.objectweb.dsrg.sofa.repository.model.impl` - repository model implementation (generated)

`org.objectweb.dsrg.sofa.repository.model.util` - repository model utilities (generated) `org.objectweb.dsrg.sofa.repository.renamer` - classes which are in charge of class renaming

`org.objectweb.dsrg.sofa.repository.server` - the servlet container

`org.objectweb.dsrg.sofa.repository.server.webapp` - http servlet classes

`org.objectweb.dsrg.sofa.repository.server.webapp.jar` - jar inspector (extracts parts of jar files)

`org.objectweb.dsrg.sofa.repository.server.webapp.repositorydata` - classes used to generate the RepositoryData object which holds all content of repository

`org.objectweb.dsrg.sofa.util` - useful utilities that are used throughout the code

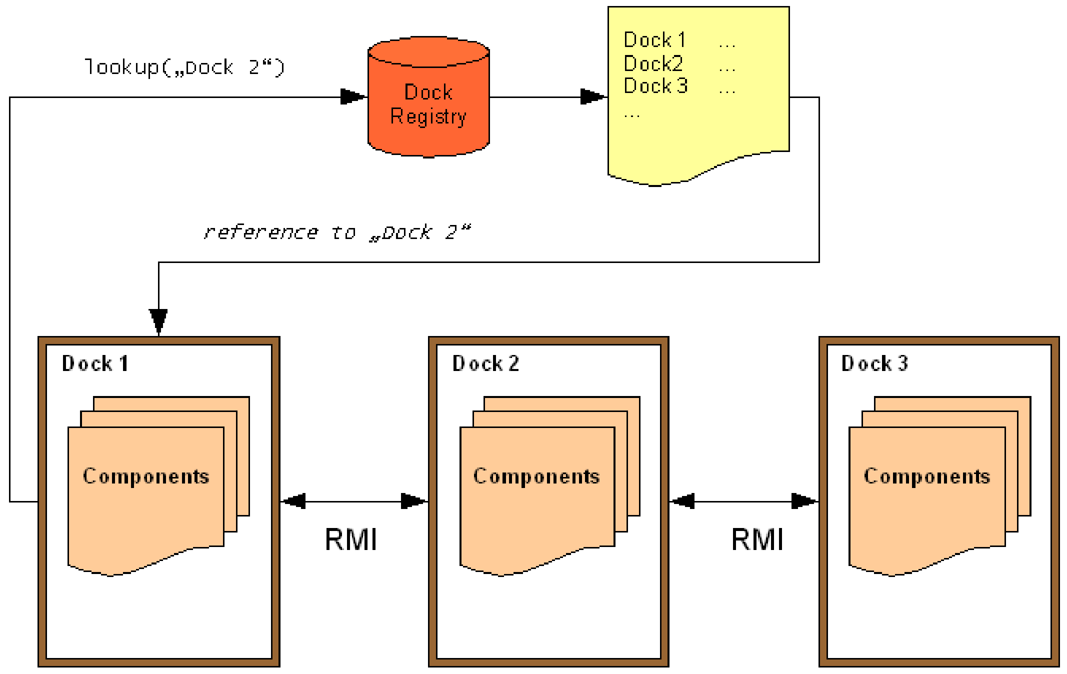
`org.objectweb.dsrg.sofa.util.checker` - behavior checking utilities

Chapter 2. Deployment Dock Registry

2.1. Overview

The Deployment Dock Registry (DDR) is a simple RMI registry which keeps track of all the Deployment Docks running in a SOFAnode. Whenever a dock needs to communicate with another dock, it queries the DDR for its reference. After the reference is returned, the dock can call the other dock's methods via RMI. This mechanism is used particularly during the application deployment - the deployment process follows the hierarchy of the given application and each dock instantiating a component with some subcomponents must contact the docks where these subcomponents are supposed to run.

Figure 2.1. Obtaining reference using the deployment dock registry



The Figure 2.1, “Obtaining reference using the deployment dock registry” shows how the deployment docks communicate. Whenever a dock (*Dock 1* in this case) needs to call a method on another dock (here, *Dock 2*) - usually during the deployment process - it calls the `lookup` method on the DDR which returns the appropriate RMI reference. After that, *Dock 1* can directly call *Dock 2*.

2.2. Implementation

There are four methods in the `DeploymentDockRegistry` interface which are implemented by the `DeploymentDockRegistryImpl` object:

- `boolean register( DeploymentDockRegistryClient deplDock, String name )`

  The `register` method registers a dock in the given DDR. Due to project dependency issues it takes a `DeploymentDockRegistryClient` reference as the first argument, instead of a `Deployment-Dock` reference. The second parameter is the name of the new dock. The dock name must be unique in the whole SOFAnode.

Note

When a dock (let's call it *dock1*) attempts to register with the same name (e.g. "*dockA*") as another already registered dock (e.g. *dock2*), the registration for *dock1* fails. However, a check is performed during this process whether the "older" dock (in this case *dock2*) is still available. If it is not (e.g. due to a network failure), its reference is deleted and the new dock (in this case *dock1*) is registered instead.

- `boolean unregister( String name )`

  The `unregister` method removes the reference to the dock from the DDR.

- `DeploymentDockRegistryClient lookup( String name )`

  The `lookup` method searches the DDR for a dock with the name passed in the parameter. If it is present, it returns the reference to it. Otherwise, it throws a `DockNotFoundException`.

- `String[] getAllNames()`

  The `getAllNames` method returns an array of the names of all docks currently registered in the DDR.

2.3. Controlling

There is a simple command-line interface for launching the dock registry. It is implemented by the `RegistryCLI` class in the `dockregistry` project. You can launch the dock registry using this command, as shown in Example 2.1, “Launching the Deployment Dock Registry”

Example 2.1. Launching the Deployment Dock Registry

```
java org.objectweb.dsrg.sofa.dockregistry.RegistryCLI new
```

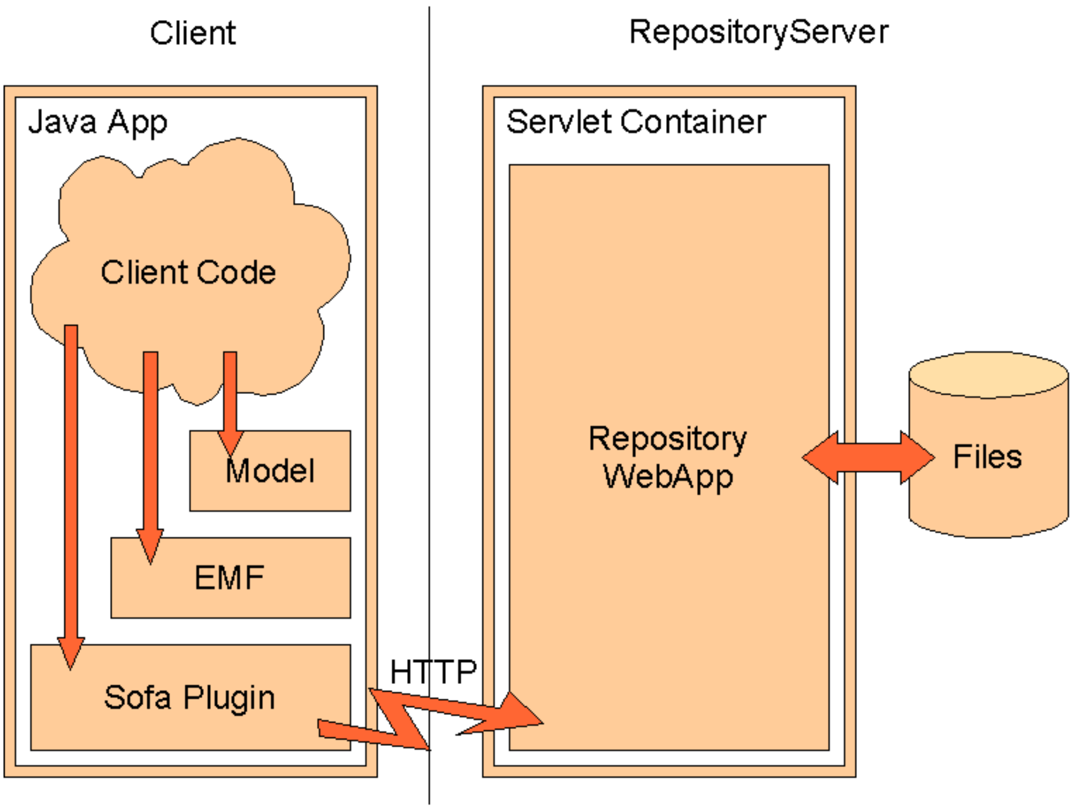
You can also specify the port on which the registry will run. It is set in the `sofa.registry.port` Java property. The default port number is 1099. This property can be set in the main configuration file (*run.properties*).

Chapter 3. Repository

3.1. Overview

Repository is a client/server application. Communication between the client and the server is done using HTTP protocol. HTTP protocol is enough for these purposes and is a standard way how to communicate in a client/server environment. Repository clients are Java classes, which are used by client programs. On the client side EMF and entire model facilities are situated. Through HTTP only the raw XML data are sent and on the client side EMF deserializes this XML into model objects. These objects are then returned to caller class and the client can read necessary informations from this object.

Figure 3.1. Repository architecture



3.2. Eclipse Modeling Framework

We have used Eclipse Modeling Framework [http://www.eclipse.org/emf/] (EMF) for generating model classes from EMF *Ecore* metamodel. Eclipse Modeling Framework is a powerful Eclipse modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model.

The model is situated in the `sofa2.ecore` file. Apart from this, the `sofa2.genmodel` file is also necessary. This file is a derivate of the ecore model but it contains additional options for the generator. This is used to generate the Java classes, interfaces and some utility classes. The entire model generated by EMF is situated in the Repository JAR in the package `org.objectweb.dsrg.sofa.repository.model`.

3.3. Repository Client

In fact, the Repository client consists of EMF libraries, HTTP client libraries and model classes. All model facilities sit run the client side. This implementation is simple but it is enough for Repository. Client/Server feature is added to EMF generated classes by an additional `URIConvertor` class, which is called `SofaURIConverterImpl`. This class is responsible for opening the data streams, when EMF objects are serialized. SofaURIConverterImpl opens these streams to the Repository server for downloading or uploading data. Thus Repository clients are for the most of the time offline. Only when user needs some data, the client connects to the server.

Addressing Repository is done by SofaURIs and these URIs are Repository instance independent. They do not contain any connection parameters. During request on any Repository object, the client has to convert SofaURI into HTTP request URL for a specific server instance. This conversion is very simple and can be done only by hand, for example for debugging. This means that user is able to convert SofaURI into URL and then use it in web browser to test, if the RepositoryServer is running well.

A HTTP client is a part of Java Standard Edition, but it is too simple for our purposes. For example, it does not support POST method at all. Therefore the Apache HTTPClient is used on the client side to access the Repository server. For downloading, the standard GET method is used and for uploading the POST method is used. The POST method is very well supported in the HTTPClient and it is simple to use it. Each RepositoryAgent keeps an instance of the HTTPClient and it is recommended to use it by other *SOFA 2* facilities, which need to communicate by HTTP to Repository or may be other servers in the future.

3.4. Repository Server

The chosen HTTP protocol is in the Java world very well supported by servlets and containers. Repository server is a HTTP servlet container. An open source Servlet container called Jetty [http://www.mortbay.org/jetty/] was used. This servlet container is very small, it is only a single JAR file. The Repository does not need the entire J2EE infrastructure, which is usually offered by other servers like for example Tomcat. The second part of RepositoryServer is the SOFA 2 web application, which runs in this container and replies to the clients requests. All data is stored as files on the server. The Server only performs serving these files. For performance issues the support for JAR files introspection was added. Therefore, the classloader in a Deployment Dock can download specific classes from a CodeBundle without the need to download the entire JAR.

The simplicity of Repository server is a big advantage for debugging the Repository, because the user is able to test the requests only by using a web browser.

3.4.1. Addressing in Repository

Repository represents a storage for information in a SOFAnode and there should be a way to find specific information by some form of address. In fact, there are several ways to do this. Because of the fact that the Repository meta content is generated by EMF and the Uniform Resource Identifier (or URI) standard is implemented in EMF, every EMF generated model uses URIs for addressing parts of this model. Therefore, it is used in the Repository as well. A big advantage of this approach is that the addressing of the resources is done in a standardized way.

The Repository uses only a subset of the entire URI set which is called *SofaURI*. The terminology is inherited from the EMF. By using the SofaURI, the users of the Repository are able to specify any versioned entity, data bundle or internal meta information stored in the Repository. For better understanding how it can look like the following examples present this concept more accurately.

```
sofarep://meta/cz/cuni/ComponentArchitecture1?version=1.0#/
```

The preceding example shows a string representation of an instance of meta SofaURI. It is called a *meta* SofaURI because it always addresses a metamodel entity in the Repository. It is recommended to use the Java naming conventions for the entity names, especially dot-separation. The names following this rule are then converted into slash-separated URI by the Repository internal classes. The URI from the previous example contains a "meta" authority. An authority is the EMF term for the first part of the hierarchical path in the Repository. This authority is significant for meta SofaURI. A meta SofaURI addresses metamodel parts and these parts are versioned entities, therefore a mere name is not enough for its specification. There needs to be also a version information. It is specified by the "version" parameter at the end of the URI after the question mark. The URI ends with a mandatory EMF specific suffix.

```
sofarep://data/cz/cuni/CodeBundle1?version=1.0&
          data.stream=original#/
```

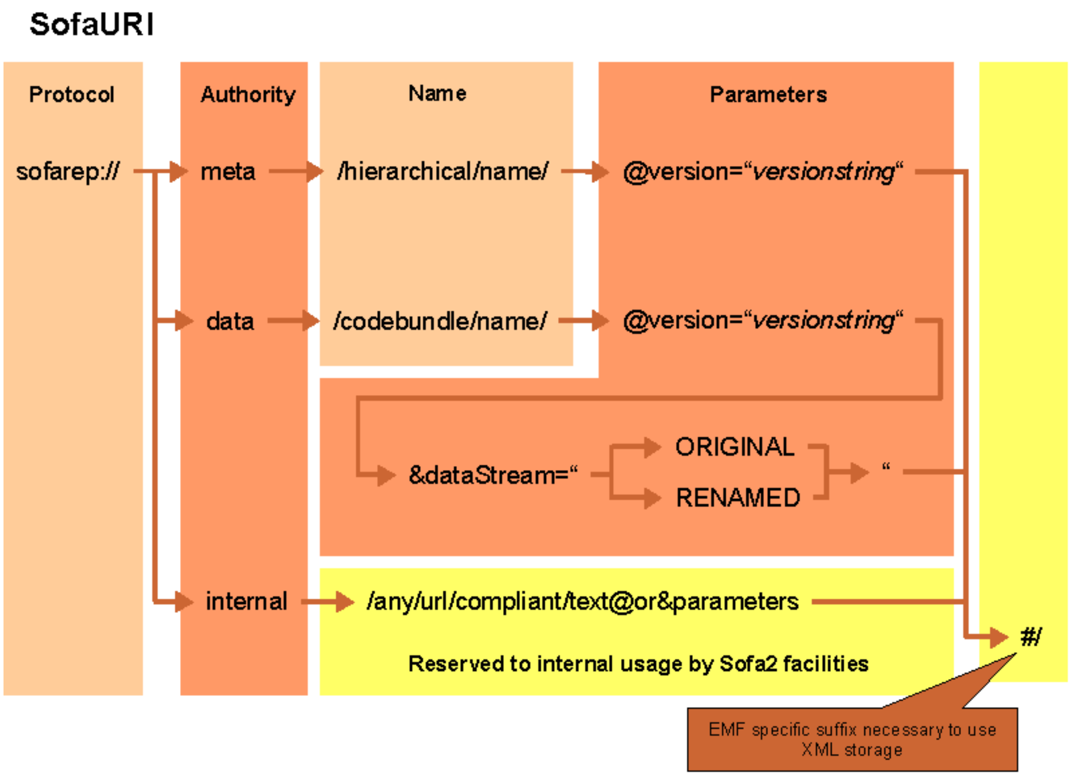
The second example is an instance of data SofaURI. This URI addresses a data bundle stored in the Repository. Each data bundle has to also have a metadata object which consists at least of the data name and version. This metadata entity is called CodeBundle and it is also the only entity in Repository, which has a code part. This code part is addressed by a data SofaURI. For simplicity, the name and version in the meta URI of a CodeBundle is the same as the name and version in the data URI of its code part. Every CodeBundle is can contain two streams of data: a so called `ORIGINAL` one and a `RENAMED` one. The renamed data is used during the runtime of *SOFA 2* to support loading different versions of the same class into a single Java Virtual Machine. The data SofaURI has a different authority than the meta SofaURI: it is the "data" authority. The name and version are used in the same way as in meta SofaURI. A new parameter is used to specify which data stream the user demands. The renamed stream should only be used by the internal parts of the *SOFA 2* system. Again the data SofaURI contains the EMF specific suffix.

```
sofarep://internal/repositoryData#/
```

The last example of a SofaURI demonstrates a special kind of URIs used by the Repository itself. These URIs have an "internal" authority and the rest is not specified. Generally it should be anything, but it should follow the Java naming conventions. These URIs can address for example non-persistent data in the Repository. It is demonstrated by this example, which points to a `RepositoryData` object that is generated dynamically by server from the Repository content.

The Figure 3.2, “SofaURI Specification” shows the entire structure of the URI subset used by the *SOFA 2* Repository to address its content. The SofaURI always contains the "sofarep" protocol. The next part is the authority, which can have one of these three values: "meta", "data" and "internal". By the authority we recognize three categories of SofaURIs: meta SofaURi, data SofaURI and internal SofaURI. The naming conventions for meta and data SofaURI are the same. The parameters part for the meta and data SofaURI always contains the version parameter. The data URI has an additional parameter which specifies the data stream being addressed. The version format is specified neither in SofaURI nor in the metamodel. It is only an extended identifier of the entity name. It is up to user, how to specify the versions and what structure the version string should have. More details about versioning can be found in the in the User's Guide, Part I, Section 5.3, "Versioning in Repository". The internal SofaURI has only the authority specified. The rest of it is not specified and can be used freely by *SOFA 2* facilities. Generally, the internal SofaURI specifies metadata and therefore it could be part of the meta SofaURI. It is separated because of the interference of namespaces. The user of the Repository is not limited in the metamodel entities naming.

Figure 3.2. SofaURI Specification



The SofaURI is mainly used by the Repository internal code to keep the stored model linked together. The user can use the URI to get entities from Repository, but it is not the only way to get them. There is also a root object in the Repository called `RepositoryData`. For details how to load objects from the Repository, please refer to the Developer's Guide.

SofaURI instances are created by the `SofaURIFactory` class, which has static methods to create *ALL* the possible styles of SofaURI. SofaURIs should not be created in other way than by the `SofaURIFactory`. These instances can be freely stored for example by developer tools for future usage in a string representation.

3.4.2. Serving entities

Getting entities from the repository is executed using the HTTP GET request. The request consists of the address of the repository and URI of the component to be downloaded passed using the GET method.

3.4.3. Uploading

Uploading the entities into the repository is performed using the HTTP PUT request. It is possible to upload more files at once.

3.4.4. Deleting

Deleting entities from the repository is done using the HTTP DELETE request which is handled using the `doDelete` method. The request consists of the address of the repository and URIs of the components to be deleted passed using the GET method, separated by '&'.

The fact that we allow the client to delete the entities by directly passing which files to delete means that while deleting a non-trivial entity, the repository can become inconsistent and it is only up to the client to assure that the operation is executed and finished correctly. To ensure that the deleting operation leaves the repository in consistent state, several classes were added to the repository client part.

For more information about classes concerning deleting entities from the client point of view, see Section 3.5.3, “Deleting entities”.

3.5. Using the Repository

In the previous sections, the repository concepts and theories were described. In this part, information about the repository usage follows. There is limited access to the repository from the commandline; developers are encouraged to use the repository directly from Java programs by calling the methods of the Repository objects. There is a simple introduction to the Repository usage in this section and for complete description of the Repository interfaces, please refer to the generated Java documentation, which is part of *SOFA 2* distribution too.

Any user requiring an access to the Repository, either for reading from it or writing any data to it, should use the `RepositoryAgent` class.This is an entry point to the Repository and during its lifetime it holds the information how to connect to the specified Repository server.

The `RepositoryAgent` object is usually initialized by a static `init` method. This method tries to load the `NodeConfiguration`. The `NodeConfiguration` can be given to the `RepositoryAgent` in form of an object or described by properties. If it is successful to load the configuration, it can access the repository. It is *important*, that RepositoryAgent does not explicitly during its initialization try to detect, whether the supplied connection parameters from the NodeConfiguration are valid. The RepositoryAgent offers the `NodeConfiguration` object by the `getNodeConfiguration` method, which should be used by client applications, for example deployment docks, for self configuration. This way, it is possible to centrally configure the entire SOFAnode only by supporting different properties or editing this file.

When initialized, a `RepositoryQuery` object can be used to obtain objects from the Repository. SofaURIs are used for addressing the objects and this URI is passed to the `getEObject` method. Then the specified object is loaded from the Repository. Entities can be also obtained from the Repository by using the `RepositoryData` object. This is a special *singleton* object. It is dynamically generated by the Repository and it cannot be modified. Simply put, it is the Repository "table of contents". It knows about all `VersionedEntities` in Repository. So all these entities can be obtained by listing this inventory of all the objects stored in Repository.

There is another special object in the Repository. It is called `RepositoryInfo`. It is again a *singleton* object. It can be changed, but only by internal features and processes, because this object contains information about the data state and cloning. As stated before, some Repositories can be clones of another ones. Any user can read this information using that object. It knows how to connect to parent Repository too.

3.5.1. Loading metadata

Metamodel objects, can be persistent in the Repository. This section describes, how to load them.

The loading functions of the meta data objects in the Repository interface are centralized into the `RepositoryQuery` class and its instance is obtained from the `RepositoryAgent` by calling the `getQuery` method. The RepositoryQuery object is responsible for loading all metadata from the Repository. Model objects contain metadata. These objects can be obtained by the RepositoryQuery in two ways. It is possible to directly obtain any `VersionedEntity` by its name and version. The objects which are not VersionedEntities are always owned by some VersionedEntity which must be known. A SofaURI for the specific name and version must be created and then used in the `getEObject` method. If everything goes well, the required object will be returned. This method does not explicitly throw any exception, but that should be changed in the future.

Example 3.1. Loading meta objects from Repository

```
RepositoryAgent ra = RepositoryAgent.init();
URI u = SofaURIFactory.createURI("cz.cuni.ComponentArchitecture1",
                                 "bd34d46cd589a26edc134e30c455a99be00a967a");
Architecture a = (Architecture) ra.getQuery().getEObject(u);
System.out.println(a);
System.out.println(a.getImplementation());
```

Example 3.1, “Loading meta objects from Repository” presents the usage of the RepositoryQuery object. Firstly, the URI specifying the demanded meta object is created. It points to an object called `cz.cuni.ComponentArchitecture1` and to its version `bd34d46cd589a26edc134e30c455a99be00a967a`. Then the initialized RepositoryAgent is used for obtaining a RepositoryQuery object and then its `getEObject` method is used for loading the object specified by the URI parameter. The example ends with simple demonstration of printing the object information.

A different way of reading Repository objects involves obtaining the `RepositoryData` object, which is like a "table of contents" of the Repository. It keeps lists of different types of VersionedEntities stored in the Repository. It is possible to iterate through the chosen list and find the required object.

Note

It is important to be aware of the fact that iterating over the lists in the `RepositoryData` object causes loading all the meta objects from Repository step by step and therefore it can be quite slow. It is not intended to be used by the infrastructure for finding a specific object.

For better understanding, a code example follows, which shows how to load meta data objects by the `RepositoryData` object.

Example 3.2. Using RepositoryData object

```
RepositoryAgent ra = RepositoryAgent.init();
RepositoryData rd = ra.getQuery().getRepositoryDataObject();
for (Object o : rd.getAspect()) {
  EObject x = (EObject) o;
  System.out.println(x);
  }
}
```

In the Example 3.2, “Using RepositoryData object”, the `RepositoryAgent` object is initialized first. Then the `RepositoryData` object is obtained, using the `RepositoryQuery` object. Then example then presents listing of all the micro aspects installed in the Repository. For each aspect it prints its string representation to standard output.

3.5.2. Repository Facade

The `RepositoryFacade` class follows, as its name suggests, the *Facade* design pattern. It is the common interface for programmers to access the *Repository*. It exposes interface for almost any action which can be committed on the *Repository* and as such should be exclusively used when modifying the *Repository* contents.

The `RepositoryFacade` should be used as high-level means of manipulating the entities within the *Repository* which can effectively hide the usage of the relatively low-level classes like `RepositoryQuery`, `RepositoryAgent`, `RepositoryPersister` and `RepositoryDeleter`.

The functions of the RepositoryFacade can be divided into several groups:

- **lookup**

  The entities can be sought from the *Repository* by the combined values of their names, versions and tags.

- **create**

  The `create` functions are used both to create new entities and to introduce new versions of existing ones.

- **delete**

  The `delete` functions are implemented as a front-end for the otherwise similarly working `RepositoryDeleter` class to complete the `RepositoryFacade` functionality as the only object needed when accessing the *Repository* from outside.

  The `delete` methods should always be used with the `deleteDependencies` parameter set to `true` unless you really know what you are doing.

  There is also a possibility to "dry run" the deleting logic and only get the list of entities to be deleted.

- **update**

  The `update` functions make the entity up-to-date according to the given model.

- **clone**

  The `clone` functions need another *Repository* to be running and clone the given entity/entities with dependencies into it. They also return all the entities which were cloned.

- **merge**

  The `merge` functions of the `RepositoryFacade` are used to merge entities from a development *Repository* into the stable one along with all the dependencies needed by the entities.

3.5.3. Deleting entities

Classes handling the deletion process on the client side of the repository are these:

- **DeleteDependencyCalculator**

  The DeleteDependencyCalculator class is responsible for calculating all the dependencies which should be deleted with a given entity. Those are the entities which have a transitional reference to the entity about to be deleted.

  The deleting dependencies are resolved when the `DeleteDependencyCalculator` object is instantiated. That is done by creating an oriented graph structure where each node holds a reference to the entity it represents and sets of entities it depends on and which are dependent on it. There are two types of edges in the graph - edges representing the "parent-child" relation and "container-containment" relation, so the graph is actually a modified multigraph. When a parent is deleted, all its children need to be deleted also. When a container is deleted, all the contained entities need to be deleted and vice versa, when a contained entity is deleted, all its containers need to be deleted with it for the repository to remain consistent.

Apart from storing the entities in the graph structure there is also a `HashMap` in the `DeleteDependencyCalculator` called `nodesCache` which helps with finding the starting node of the graph from which to traverse the graph for dependencies.

The nodes of the graph are represented using the `DeletingNode` class which also has the `addEntityDependenciesToSet` function which is used to recursively get all the dependencies of the node.

To get the dependencies which should be deleted alongside with an entity, there are the `getDeletingDependencies` functions which return a collection of the dependent entities.

- **RepositoryDeleter**

  The `RepositoryDeleter` class is a cousin of the `RepositoryQuery` and `RepositoryPersister` classes. It is used from the `RepositoryFacade` and takes care of transforming the entities it gets from the `DeleteDependencyCalculator` into their URIs and sending both the URIs of their metadata and data files to the `SofaRemoteFileDeleter` which then sends them as a DELETE request to the Repository server.

  The main method of this class is the `delete` function which is present in two variants for either deleting a single entity or deleting a set of entities. The second of the parameter, the `deleteDependencies` boolean determines whether the method should count the dependencies of the given entity/entities and delete them also or whether the method should delete only the given entity/entities.

  ## Warning

  Deleting entities without deleting their dependencies is highly unrecommended and may lead to making the whole repository inconsistent without an easy possibility of recovery!

  This functionality should be used only by advanced programmers who know what they are doing.

  Apart from the delete methods the `RepositoryDeleter` class has also the getDeletingDependencies function which however only forward the request to the `DeleteDependencyCalculator` class.

  Note that when manipulating several entities at once it is always more efficient to use the methods/function which allow passing a collection of entities instead of calling the single entity function more times. This is caused by the nature of the `DeleteDependencyCalculator` which caches the whole repository any time it is instantiated (which is needed for always having the current state of the repository).

- **SofaRemoteFileDeleter**

  The `SofaRemoteFileDeleter` class is the most "low-level" of all the classes which handle deleting entities from the repository. It follows the *Singleton* pattern implemented by static methods on the class and a private constructor (any more complex implementation of the *Singleton* pattern would be superfluous).

  The *Singleton* pattern was chosen to make the deleting process as atomic as possible (this class was implemented far before the locks on the repository were even thought of). Before starting to delete entities in the repository, the calling side (usually it's just the `RepositoryDeleter` class) must call the `beginDelete` function, which gets the `RepositoryConfiguration` object as a parameter to know which repository to delete in. The `beginDelete` function returns false if the deleting process isn't available, otherwise true.

  After successfully calling `beginDelete`, the `deleteFile` function can be used. It takes a URI of the file to delete (that means that it doesn't know how to delete entities on general - instead the entity URIs must be passed) and stores it in a `List` of entities to be deleted - it doesn't delete them, that is the responsibility of the `endDelete` function.

When all the files which are to be deleted are stored in memory by the `deleteFile` function, the `endDelete` function is executed to create the DELETE request on the repository and pass it the files. The approach of using the `endDelete` function to delete all the processed files was chosen also as a little network optimization - the requests for file deletion are usually carrying several file paths which reduces network traffic nontrivially.

Even though the `RepositoryDeleter` class can be used quite easily to delete entities from the repository (the other two classes shouldn't be used if the programmer doesn't really know what they are doing), there is also an interface for deleting in the `RepositoryFacade`, see Section 3.5.2, “Repository Facade”.

3.5.4. Cloning and Merging entities

The cloning and merging functionality is performed by the `RepositoryCloner` class. Its main business methods are `clone` and `merge`. Both `clone` and `merge` support recursive operation on dependencies, which are calculated using the `DependencyCalculator` class.

The `InitialClone` is a utility class, that creates a new Repository content with the duplicates of all selected versioned entities from the original repository. It's only purpose is implemented in it's `main` method.

The cloning and merging functionality is also integrated into `RepositoryFacade` by `cloneEntity`, `mergeEntity`, `cloneEntities` and `mergeEntities` methods. The facade only delegates the calls to the `RepositoryCloner` object. It's up to the developers which approach will they choose, however using the facade will more likely offer simpler code.

Chapter 4. Component code management

In *SOFA 2*, code for components, microcomponents and connectors is stored in the central Repository. Because this code is deployed on the Deployment Docks the remote class loading and code versions have to be supported. The following sections discuss these topics in detail.

4.1. Concurrent code versions

SOFA 2 supports running different versions of the same component in parallel. However, Java Virtual Machine does not permit loading of two different classes with the same name within one context. This limitation can be overcome in two ways. Firstly, every component can be loaded with a unique classloader object which creates the component's private context. This solution is difficult to implement and is not usable in low profile Java environments. Secondly, the names of the classes can be altered to be unique for every component-version combination. This alteration can be done prior to code use and does not need major runtime support.

4.1.1. Class name augmentation solution

The class name alteration was chosen for the *SOFA 2* framework. It is needed to make the mangling (encoding the class name and version to the new class name) process transparent to component developers. Therefore, it is performed when the code bundle JAR file is uploaded into the central Repository by the `uploadFile` method of the `CodeBundleHelper` class which resides in the `org.objectweb.dsrg.sofa.repository.renaming` package. This method calls the `uploadRenamedFile` method of the `CodeBundleHelper` class where the names of all classes contained in the JAR file are collected and new names are created using the `getNewClassName` method of the `RenamerHelper` class. A new instance of the `Renamer` class is created to map the original names to the mangled names and all the classes from the dependency code bundles are also added to the mapping. A new empty JAR file is created for the renamed classes.

The mangled form of the class name is obtained from the original name and a string representation of the version of the code bundle. These two components are joined by the "_v" string. There are some characters replaced in the version string to ensure that the new class name will conform to Java specification and that the class will be placed in the same package as the original class. Currently, dots, question marks and exclamation marks are replaced by ".", "_Q_" and "_E_" strings, respectively. For example, the mangled name for a `org.foo.HelloWorld` class stored in a code bundle with version "1.0" will be "org.foo.HelloWorld_v1_0".

After the new names are created, the actual bytecode transformation is performed. *SOFA 2* uses the ASM framework [http://asm.objectweb.org] for parsing the classes and manipulating the bytecode. All references to the original names are replaced with the mangled ones. The *ASM* framework uses the visitor design pattern. For our purposes, the custom `RenameVisitor` and `RenameMethodVisitor` classes are provided. The classes are read in sequence from the JAR file, modified by these visitors and then stored in the new JAR file. Whenever the visitor finds a class name in the Java bytecode it asks the `Renamer` instance for the new name of that class. If the `Renamer` has no mapping for the class name the original name is preserved. The new JAR file is stored in central Repository alongside the original JAR file.

4.2. Loading classes from central Repository

In Java, code is loaded into a virtual machine by objects called classloaders. Java runtime provides a couple of different implementations of these objects suitable for specific situations. Unfortunately none of them is suitable for loading code from the central Repository. Therefore, *SOFA 2* comes with a custom classloader called `SOFAClassLoader`. This classloader overrides the standard `loadClass(String)` method and provides some additional ones.

Figure 4.1. SOFAClassLoader control and data flow

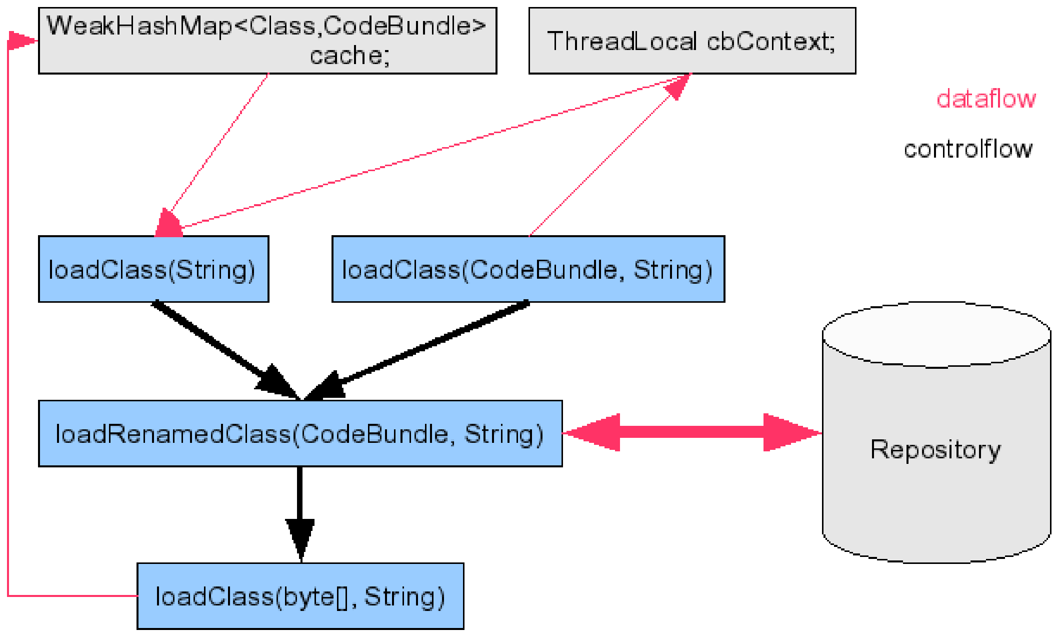

WeakHashMap<Class,CodeBundle> cache;
ThreadLocal cbContext;
dataflow
controlflow
loadClass(String)
loadClass(CodeBundle, String)
loadRenamedClass(CodeBundle, String)
Repository
loadClass(byte[], String)

Figure 4.1, "SOFAClassLoader control and data flow" shows the interaction between the methods and fields of the `SOFAClassLoader` class. The `cache` stores the code bundle the class was downloaded from. The mapping is represented by a weak hash map so when the class is unloaded from the virtual machine (i.e. by garbage collector) the cache is kept up-to-date. The `cbContext` field is set when the class is being loaded into the JVM. When the class is loaded it can provoke loading of other classes which, in principle, have to be stored in the same code bundle or its dependencies, otherwise the class will not refer to them. This `cbContext` is per thread variable to avoid collisions between two parallel threads. Now a description of the methods follows:

- `loadClass(CodeBundle, String)` variant derives the mangled class name from the given name, stores the given `CodeBundle` into the `cbContext` variable and calls the `loadRenamed-Class` method. If the given code bundle parameter is *null* the `loadClass(String)` method is called instead.
- `loadClass(String)` method looks for already loaded class of the given name. If the class is not loaded yet, it searches in standard system locations (i.e. CLASSPATH system variable). If it is not found, it tries to load class from the code bundle. First, the code bundle set in the thread variable `cbContext` is looked up. Then the call stack is inspected and the class is searched in code bundles containing the classes from the call trace. At last, the `cache` is used to search in all active code bundles that have some classes loaded into the JVM. The method `loadRenameClass` is used for loading the requested class from the code bundles found.
- `loadRenamedClass(CodeBundle, String)` downloads the class of the given name from the given `CodeBundle` and calls the `loadClass(byte[], String)` method for the downloaded bytecode.
- `loadClass(byte[], String)` uses the standard `defineClass` method to load the class stored in a byte array into the JVM. The mapping for the loaded class is stored in the `cache`.

4.3. RMI class loading

When using connectors based on the Java *RMI* middleware, a transfer of classes across address spaces could possibly happen. To support class-loading from the central Repository, we override the standard

RMI classloader by implementing the `RMIClassLoaderSpi` interface. This implementation is called `RMIClassLoaderServiceProvider`. The `java.rmi.server.RMIClassLoaderSpi` Java property is used to override the standard service provider for the *RMI* classloader.

When instantiated, the *SOFA 2* service provider initializes a connection to the *SOFA 2* repository. Then at run-time, the classes loaded by the `SOFAClassLoader` that are going to be transferred over to another address space are annotated by the `getClassAnnotation` method with the URI of the code bundle that stores them. In the target address space, the `loadClass` method is called and the URI from the class annotation is used for retrieving the `CodeBundle` model object from the central Repository. This `CodeBundle` is used to load the requested class with the `SOFAClassLoader`.

In Java 5.0, the *RMI* uses `Proxy` objects for the *RMI* connections. They are serialized on the calling side and sent to the called side of the *RMI* connection. Although these `Proxy` objects can be loaded by the standard system classloader, they may implement some interfaces that need to be loaded from the central Repository. We use Java reflection to extend the *RMI* annotation of the `Proxy` object with the annotations of the implemented interfaces. These interfaces are loaded by the `SOFAClassLoader` before the serialized `Proxy` object is being loaded on the called side.

Chapter 5. Deployment Dock

The Deployment Dock is the container for all *SOFA 2* components running on it. It handles component instantiation and controls the component through its life-time. It also handles proper connector instantiation and binding. The individual tasks are described in more detail in the following sections.

5.1. Startup

Upon startup, the deployment dock performs three important actions:

- It attempts to register itself in the Deployment Dock Registry with the name specified through the `DockCLI` command which started this dock.
- It sets the `SOFAClassLoader` as the default classloader for its thread.
- It creates the `DockConnectorManager`, which is used for creating the connector units for the individual components. It also gets the reference to the `GlobalConnectorManager`, which is unique per SOFAnode; this manager is used for rebinding the individual connector units and thus creating the whole connector.

5.2. Deployment of an application

5.2.1. Application launch

The whole deployment process starts when the `DockCLI` command is executed with the *launch <deployment_plan_name> <deployment_plan_version>* parameters. The `launch` method of the `DockCLI` class is called which connects to the Deployment Dock Registry (DDR), gets the list of all Deployment Docks registered in the current SOFAnode and sends a launch request to the first of them by calling the `launchApplication( String, String )` method of the `DeploymentDockImpl`. The `DeploymentPlan` meta-model object, specified by its name and version, is downloaded from the central Repository. Then the connector architectures (needed for proper connector instantiation and specified in the deployment plan) are added to the `GlobalConnectorManager`. Next, the DDR is queried once more for the top-level dock reference (the reference of the dock where the top-level component will run) and the deployment itself is started on that dock. The description of individual steps of the deployment process follows.

5.2.2. Connector preparation

The connectors used in an application are described using the `ConnectorUnitDeploymentDescription` (CUDD) objects in the Deployment Plan representing that application. Each CUDD describes one connector unit bound to a specific interface on the given component which may run on any deployment dock in the current SOFAnode. The method `void addConnectorUnitITs( String dpName, String dpVersion, String componentInstanceName )`, called at the beginning of the deployment process registers all the connector units belonging to the given component (specified by the *componentInstanceName* parameter which describes the component's position in the application hierarchy) in the current `DockConnectorManager`. The method then recursively calls itself on the other docks where subcomponents of the given component will run. This way, starting from the top-level component, all connector units needed for the deployment are registered before the component instantiation starts.

5.2.3. Component instantiation

The component instantiation starts by calling the `ComponentHandle instantiateComponent( String dpName, String dpVersion, String componentName, String componentInstanceName, ComponentHandle parent, LinkedList<ConnectorUnitBindable>`

`cuList`) method. It runs recursively (once for every component in the application hierarchy) and besides the actual creation of the component it also handles the bindings between the instantiated components. However, for the proper identification of a component in the application hierarchy, it needs to be uniquely named. Since the application model can contain two or more components implemented by the same `Architecture` object, the architecture name is useless for this purpose. Therefore, each component is identified by its name concatenated with its version.

During the instantiation of a component Java Reflection API is used to check the interfaces which the component implements. If it implements the `SOFAParametrized` interface, properties along with assigned values defined in the `DeploymentPlan` are passed using the `setProperties` function. If the component implements the `SOFASelfShutting` interface, the `SOFAAppShuttingContext` is passed to it through the `setShuttingContext` method. The `SOFAAppShuttingContext` is actually an instance of the `SOFAAppShutter` class which remembers the component id of the top-level instance that is used to shut down the application.

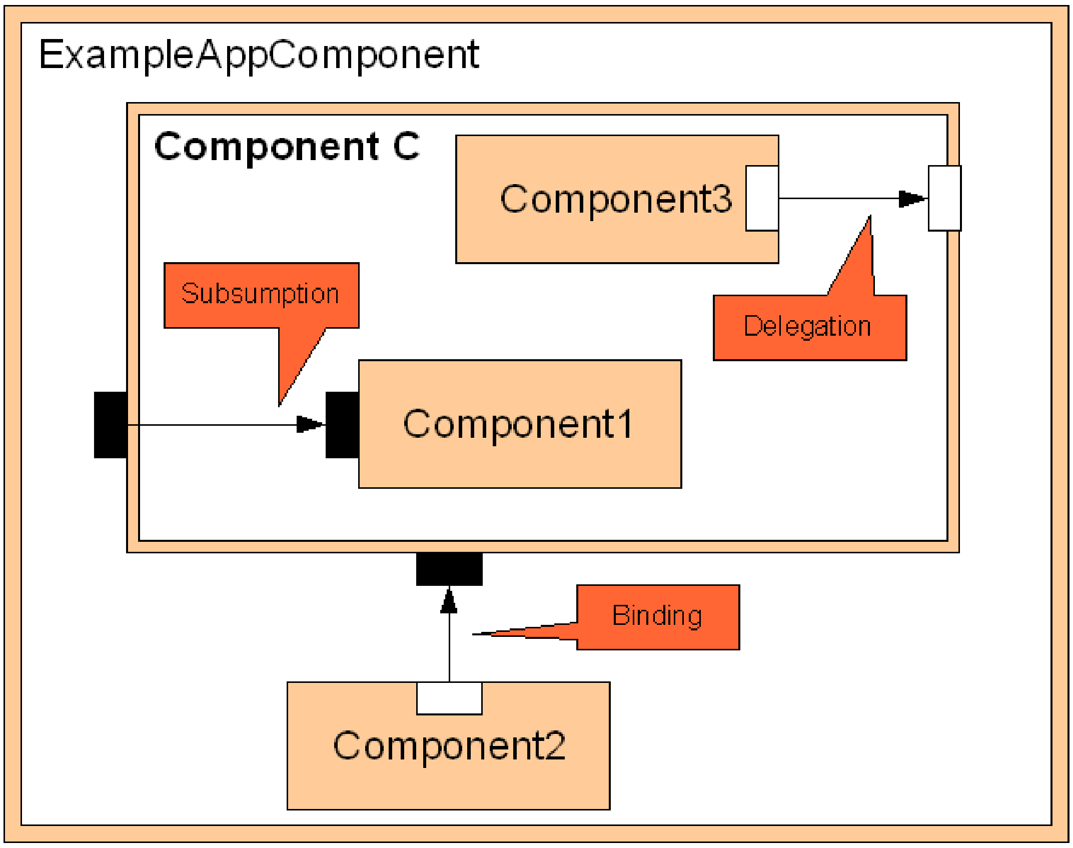
5.2.4. Bindings

After a component and all of its subcomponents are instantiated, they must be connected according to the instructions in the Deployment Plan. However, the components are not connected directly. Instead, for each one involved in a binding, a *connector unit* is created using the `DockConnectorManager` and later all the units belonging to a single connector are rebound using the `GlobalConnectorManager`. There are three types of bindings that may employ a component C:

1. Subsumption of a provided interface to a child component of C
2. Delegation of a required interface of a child component of C
3. Binding between C and some of its siblings

These bindings are displayed in the Figure 5.1, “Bindings in *SOFA 2* applications”.

Figure 5.1. Bindings in *SOFA 2* applications



If the information about a binding employing the current component was processed in its parent (it is a binding between the parent and the current component or between the current component and its sibling), the necessary data for connector unit creation is passed to the current component in the last parameter of the `instantiateComponent` method. It's a list of `ConnectorUnitBindable` objects from which the appropriate connector units are created and registered in the DockConnectorManager. Similarly, the current component must prepare this data for its subcomponents involved in a binding. This is done in the method `void fillBindingLists( Architecture arch, String appName, String componentName, String componentInstanceName, LinkedList<Bindable> cuList, LinkedList<String> connectorIDList, Hashtable<Frame, LinkedList<ConnectorUnitBindable>> tableChildCUs)`, which actually fills three lists:

1. A list of `ConnectorUnitBindables` describing the "parent" part of the bindings (that should be initialized during the instantiation of the current component).
2. A list of lists of `ConnectorUnitBindables` describing the "child" parts of the bindings (these will be initialized along with the particular subcomponent); each list describes the bindings for one subcomponent.
3. A list of connector IDs that the `GlobalConnectorManager` should rebind after the current component and all its subcomponents are instantiated (that is, at the end of the current call to the `instantiateComponent`).

5.2.5. Component creation

The component itself is instantiated by calling the `createComponent` method of the `ComponentFactory` class, which takes a list of `ConnectorUnitBindable` objects and binds them to the proper delegation chains. Each new component is assigned an unique identifier in the scope of the current dock, called `ComponentHandle`. Every subsequent operation refers to a specific component by using this handle. Next, the subcomponents of the current component are instantiated by recursive calls to the `instantiateComponent` method on the corresponding dock. If the instantiation is successful, all the connectors involving the current component are rebound. The last step is to check whether the current component is a factory. If it is, the relevant information is saved in the `FactoryInfo` structure which will be later used in the dynamic instantiation.

5.2.6. Finalization

If the application is successfully deployed, all the deployment docks involved in the process are sent a notification using the `notifyDeploymentSuccessfull(String deploymentPlanID)` method. In this method, all the components belonging to the given deployment plan and running on the current dock are marked as deployed. In case the deployment fails, a similar notification is sent using the `notifyDeploymentFailed(String deploymentPlanID)` method. In this method, all the components belonging to the given deployment plan are removed.

5.3. Controlling the components

The lifecycle of the components running on a dock can be controlled using the following methods of the class `DeploymentDockImpl`:

- `void startComponent(ComponentHandle component)`
- `void stopComponent(ComponentHandle component, boolean remove)`

The list of `ComponentHandle` objects representing the components can be retrieved using the methods

- `List<ComponentHandle> getRunningApplications()`

- `List<ComponentHandle> getRunningComponents()`

The former method lists only components with no parent (top-level components), the latter method list all components registered on the current dock. A single component can also be looked up by using the `ComponentHandle lookup( String name )` method. As an opposite to the `launchApplication( String appName, String appVersion )` method, there is also the `shutdownApplication( String appName, String appVersion )` method which stops and removes the given application from the current SOFAnode.

5.4. Thread management

In order to identify the threads crossing the address space boundaries, the Deployment Dock must support some kind of thread management.

Therefore, a separate call context is associated with every application thread in *SOFA 2*. This context is a string identifier unique in a SOFAnode and it is derived from the call context of the parent thread in such a way that by examining the call contexts of two threads it can always be determined whether one of the threads is an ancestor of the other one. This is necessary for the proper function of the lifecycle aspect. The `SOFAThreadHelper` class does the job of managing the call contexts. They are stored in `InheritedThreadLocal` variable, which keeps per thread value. Every thread has access only to the value belonging to it.

The newly created thread is assigned a call context of the parent which is extended with the dock name where the thread has been created and the serial number of the thread that the `SOFAThreadHelper` keeps track of. The exact syntax is following: `parent:dock_name.serial_number`.

5.5. Application shutdown

A *SOFA 2* application can be shut down in two different ways which however share the same code. To shut down a *SOFA 2* application the top-level instance component id within the `DeploymentDocks` is needed. It is passed to the `DeploymentDocks` which then recursively go through the components of the running application and remove them from the memory.

The first way of shutting down a *SOFA 2* application is calling the `sofa-shut.(bat/sh)` script (or its autoconfigured variation) and give it the top-level instance id as a parameter.

The second way is enabling the *SOFA 2* application to shut down itself from within using the `SOFASelfShutting` interface.

Chapter 6. Autoconfiguration

Since the configuration of the older versions of SOFA 2 was non-trivial when one wanted the SOFAnode to be spread over multiple computers, we introduced the possibility to run SOFA 2 autoconfigured. This is achieved by running a so called ZeroConfServer, which stores and distributes all the information it gathers from the SOFAnode - i.e. the address of the hosts, the ports which the daemons are running on, etc.

When designing the autoconfiguration handling classes, we wanted to make the autoconfiguration as transparent for the "autoconfiguredly" launched code as possible. This was achieved by introducing a so called `Launcher` - a class which is a preliminary class which is used to launch any part of SOFA 2 which should be autoconfigured. The autoconfiguration was obtained using the knowledge that all configuration of SOFA 2 was done using some properties (e.g. sofa.repository.host, sofa.registry.port, etc.) which were common to all the classes SOFA 2 uses for its runtime. Therefore most of the autoconfiguration is performed by the `Launcher` class which downloads the configuration information from the `ZeroConfServer` and transforms it into the arguments for the JVM in which it starts the given class that should be autoconfigured. Given this approach most of the SOFA 2 runtime classes don't know whether they are being started "autoconfiguredly" or using manual configuration because the autoconfiguration was reduced mainly to passing some Java properties (obtained from the `ZeroConfServer`) which the launched class would be normally passed nonetheless.

6.1. RunAutoconfiguredEnvironment

Even though the `ZeroConfServer` and `Launcher` classes take care of the autoconfiguration, they are just tools to pass the properties needed by the SOFA 2 classes. They also need to be published somehow and that's where the `RunAutoconfiguredEnvironment` class comes in. It is an almost precise copy of the `RunEnvironment` class but instead of just starting the servers needed for running a SOFAnode it also checks availability of the ports they should be running on. If the ports are unavailable, it finds the nearest free ports with greater numbers and assigns them to the servers instead of the original ones. This can be done thanks to publishing the configuration through the `ZeroConfServer` - the client processes get their knowledge of the configuration dynamically, so changing it at launching time doesn't matter.

When the repository, the deployment dock registry and the global connector manager are launched, a `ZeroConfServer` instance is launched and also published through the JMX technology for later use - e.g. exposing new properties, registering the `Launchers` and getting information about them.

6.2. ZeroConfServer

6.2.1. Introduction

The ZeroConfServer stores all the information about the SOFAnode configuration needed to run deployment docks, launch sofa applications and manage the whole SOFAnode. The needed configuration is passed to the ZeroConfServer when the SOFAnode is started. The ZeroConfServer then listens on a given UDP port and provides the configuration to anyone who asks.

6.2.2. Main functions of the `ZeroConfServer`

- **listen**

  Besides the constructor of the `ZeroConfServer`, the most important function would probably be the `listen` function which makes the `ZeroConfServer` start listening on a given port number as a UDP server where it waits for incoming connection (typically broadcasts) and replies with the configuration it stores.

- **setProperty**

Apart from just sitting on a port and replying to requests, the `ZeroConfServer` also needs to have a way of setting the properties it should provide. The `setProperty` function takes care of storing the data and also serializes it to a `byte` array for easy network transportation.

- **startJMX**

  The `ZeroConfServer` exposes a *JMX* interface for remote manipulation. For detail information on the *JMX* interface of the `ZeroConfServerMBean` see the *javadoc*.

  The `startJMX` function needs to be called to either register at a running registry or create a new one and register the `ZeroConfServerMBean` in it. The information on how to connect to the *MBean* is published along with the rest of the configuration the `ZeroConfServer` provides to the outside world.

- **registerLauncher**

  Another of the functions exposed by the `ZeroConfMBean` is the `registerLauncher` function. This function should be called only by the `Launchers` for which the function provides the means of telling the `ZeroConfServer` that they are there somewhere and information on how to connect to them.

  When this function is used correctly, the `ZeroConfServer` always has information about all the `Launchers` which are connected to it and can pass this information using the `getLauncherInfos` function further to anyone who wants to control the `Launchers`.

- **getLauncherInfos**

  The `getLauncherInfos` function first checks the availability of all the `Launchers` which are registered at the `ZeroConfServer` and then returns an array containing objects of the `LauncherInfo` class which only provides the information needed to connect to the `Launchers'` *JMX* interfaces.

  This functionality is mainly used by the MConsole, see Part VI, “MConsole Developer Guide”.

6.3. Launcher

6.3.1. Introduction

The Launcher class has two modes in which it can operate. First one is used in the scripts for running autoconfigured deployment docks, autoconfigured launching of sofa applications, etc. This mode gets the name of the class it should launch as one of its parameters, passes it all its parameters and provides it with the configuration downloaded from the ZeroConfServer.

Shortly, when you have a script which runs not configured, simply delete the configuration properties passed and write the Launcher class before the name of the launched class and change the call of setenv to setenv-auto. That's all needed to create an autoconfigured script from a manually configured one.

The second mode is executed when the `Launcher` is run without specifying a command to execute. When this option is used, the `Launcher` just registers itself at the `ZeroConfServer` and waits for incoming *JMX* commands. The daemonified `Launcher` is used when the user wants to control it for example from the MConsole (see Part VI, “MConsole Developer Guide”).

6.3.2. Main functions of the Launcher

- **launchConfiguredSofaClass**

  Probably the most important of the `Launcher's` methods is the `launchConfiguredSofaClass`. This method is used to launch a Java class in a separate *JVM* and pass it the configuration that the `Launcher` downloaded from the `ZeroConfServer` through the Java properties.

- **launch and launchClass**

  These two methods are quite similar to the `launchConfiguredSofaClass` method (which actually uses them). The `launch` method can be used to execute any command on the computer which the `Launcher` is running on and the `launchClass` is used for launching Java classes.

  The main difference between these two methods and the `launchConfiguredSofaClass` method is that they don't download the configuration from the `ZeroConfServer` and so launch the commands just as stated.

- **launchDeploymentDock**

  This method is a wrapper for the `launchConfiguredSofaClass` function which it uses to launch a *deployment dock* with a name it gets as a parameter. Its purpose is only to make launching *deployment docks* remotely easier and more programmer-friendly.

- **startJMX**

  For remote access of the `Launcher` we have chosen *JMX* and the `startJMX` method is used to register the `Launcher` at a running registry or start a new one.

- **registerAtZeroconfServer**

  When the `Launcher` gets its configuration from the `ZeroConfServer` it has to call the `registerAtZeroconfServer` method to make it know that it exists. Without this method the `ZeroConfServer` would have no knowledge of the running `Launchers` and therefore it couldn't pass it further.

- **listProcesses**

  This function returns the information about all the processes which are running within the `Launcher` (were started by it).

- **getAllDeploymentDocks**

  The `getAllDeploymentDocks` function is a wrapper around the `listProcesses` function and parses its output to return only the *deployment docks* which were started using the actual `Launcher`.

- **killProcess**

  Apart from just starting the processes, the `Launcher` can also kill the processes it launched. The `killProcess` needs one parameter - the id of the process started by the `Launcher`.

  Important

  The id of the process is the inner id that the `Launcher` holds, not the process id you can get from the operating system!

6.4. Configuration

6.4.1. Introduction

The configuration is handled using the Configuration class provided with the ZeroConfServer, which downloads the data from the ZeroConfServer, interprets them and then either returns its own instance holding the information and providing it through the get methods or sets the configuration values as Java properties (in which way all the parts of sofa are configured).

This approach provides a simple way to extend existing classes handling different parts of sofa simply by calling the appropriate method which loads the configuration into memory and the program then uses it the same way it would use the properties passed to it from the starting scripts when running not autoconfigured.

6.4.2. Main functions of the `Configuration` class

Most of the methods of the `Configuration` class are static and usually the `Configuration` class isn't instantiated using the new operator in the client code. The `Configuration` class can be used to carry any information represented by the *key-value pairs*. However, the usage for *SOFA 2* autoconfiguration may lead to not using instances of this class at all since the "parsed value" of the *SOFA 2* configuration is returned as `java.util.Properties`.

- **loadSofaConfiguration**

  This static function downloads the configuration from the `ZeroConfServer`, parses the properties needed by *SOFA 2* and sets them as the java properties of the current JVM.

  When wanting to manually load the *SOFA 2* properties, this is probably the best choice.

- **getSofaPropertiesFromServer**

  The `getSofaPropertiesFromServer` function is an equivalent of the `loadSofaConfiguration` but instead of loading the properties into the *JVM* it stores them in a `java.util.Properties` object which it returns.

- **getFromServer**

  Even though this function's name might emphasize that it should be used to get the `Configuration` of *SOFA 2* from the `ZeroConfServer`, it is more low-level than the other two methods and doesn't parse the configuration to standard form. It should be used only when the programmer knows what they are doing.

Chapter 7. Microarchitecture details

This chapter describes the microarchitecture implementation which concentrates on the component instantiation process. During this process the control part of the component has to be populated by the defined aspects.

7.1. Run-time representation

Microarchitecture deals primarily with component and aspect instantiation as well as with business code support. The run-time functionality was pushed into separate aspects as much as it was possible. *SOFA 2* microarchitecture has been implemented in a modular fashion and it is structured according to the microcomponent model. Most of the model entities do not have a run-time representation because the implementation uses the model objects where possible. The few entities represented in the run-time are described in the following text.

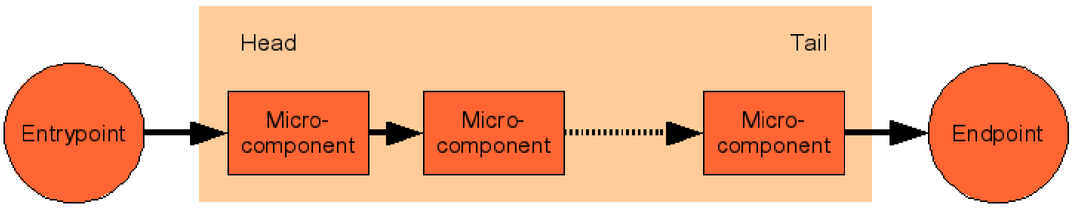
7.1.1. Component

The component and the frame that is being instantiated is represented by a `ComponentInstance` object. This object stores the parameters for microcomponent initialization. It also holds the component content for a primitive component and implements namespaces for component and interface selects. The `ComponentInstance` object is responsible for component instantiation and applying the aspects. After the component is instantiated the object releases the data it holds.

7.1.2. Interfaces

The business and control interfaces are represented by the `DelegationChain` class. This object stores the interface name and type of the delegated interface. The delegation chain also eases connecting delegating microcomponents together. The `MicroComponentInstance` model objects are inserted and they are internally instantiated by the `createMicrocomponent` method of the `MicroComponentFactory` class. The resulting microcomponent is connected to the delegation chain. After its instantiation, the microcomponent has no run-time representation (the microarchitecture code holds the content class of the microcomponent only during the component instantiation). The parameters passed to the `init` method of the microcomponent's content class are modified by the delegation chain - name and type parameters are added for the delegated interface. Also the class name of the delegated interface type is added.

Figure 7.1. Delegation chain



The Figure 7.1, “Delegation chain” shows the delegation chain containing a number of microcomponents. Delegating microcomponents can be appended to the end or inserted to the start of the chain. A reference to the endpoint has to be provided by calling the `setEndpoint` method. The endpoint is the interface that the tail microcomponent delegates to; it can be the component content's interface or a `Bindable` object wrapping the connector unit. The provided delegated interface of the head microcomponent can be obtained by the `getEntrypoint` method. However, when the `getEntrypoint` is called for the first time the delegation chain is locked and no other microcomponents can be inserted. On the other hand, the `getEntrypoint` has to be called to prepare the delegation chain for use. In singular situations where no microcomponents are inserted, the call to `getEntrypoint` returns the reference set by the `setEndpoint` method.

7.1.3. Collection interface mechanism

For collection interfaces the `CollectionChain` class is used in place of the `DelegationChain` class. The `CollectionChain` inherits from `DelegationChain` and modifies its behavior. The `MicroComponentInstance` objects inserted into the collection chain are not instantiated - instead, the `MicroComponentProxy` implementing the `SOFAMicroComponent` interface is instantiated. It stores the necessary information for future microcomponent instantiation. Most important information is about the connections to other microcomponents.

When a new instance of the collection chain is required to be created (i.e., for a new dynamic component obtained from a factory component), the `newInstance(Object endpoint)` method is called to create an instance of the `DelegationChain` class and to insert the microcomponents instantiated with the information stored in the `MicroComponentProxy` objects. Furthermore, it sets the endpoint to the given reference and initializes the created microcomponents.

7.2. Component instantiation

The instantiation of a component starts by calling the `createComponent` method of the `ComponentFactory` class. This method creates a new `ComponentInstance` object whose constructor performs all the necessary steps to build a functional component. These steps are described in the following list:

1. Business interfaces' delegation chains are created from the component's Frame definition.

2. If the component is a primitive one, its content class is loaded and instantiated.

3. Delegation chain endpoints are connected to the connector units or the component content.

4. One aspect is applied. Proceed to step 5. When finished, repeat for next aspect. After all aspects are processed go to step 7.

5. The *component selects* from the Aspect are examined (the component is tested to the query string using the `ComponentSelectEngine`) and control interfaces are created. Microcomponents of the component select are created by the `MicroComponentFactory` class. They are stored in the component namespace. Each microcomponent is connected with the component select bindings using only the component namespace. Proceed to step 6. Repeat for other component selects. Return to step 4.

6. The *interface selects* are examined (all interfaces are tested to the query string using the `InterfaceSelectEngine`) and microcomponents are created. They are stored in an interface namespace which is unique for every interface select. Delegating microcomponents are also inserted into the delegation chains. At last the microcomponents are connected using the interface select bindings and both the component and the interface namespaces. Repeat for all interface selects. Return to step 5.

7. The connector units and the content are connected to the delegation chains entrypoints.

8. The microcomponents are connected to the control interfaces. These connections were delayed when the component or interface bindings were processed.

9. Initialize all the created microcomponents. They were stored in a list along with their parameters.

At this point, the `ComponentInstance` constructor returns. The `ComponentFactory` object checks if the instantiation has succeeded. If so, it returns a reference to component's *Component* control interface.

Chapter 8. Bootstrap aspects

Bootstrap is the common name for essential aspects. They are bundled together with the microcomponents they use. There has to be at least the "Component" aspect applied to components in *SOFA 2* component system but the implementation other than the one from bootstrap project can be used.

8.1. Interceptor

The Interceptor is a microcomponent that notifies about events that are occurring on business or control interfaces. Bootstrap interceptors are generated by the `InterceptorGenerator` class using the ASM framework [http://asm.objectweb.org] to optimize performance. They consist of the `InterceptorController` class which implements the `SOFAMicroComponent` interface and a class generated to fit the delegated interface.

Generation of interceptors can be controlled by setting parameters for each interceptor microcomponent instance. The generated interceptor notifies about calls to methods when the *notifyCall* parameter is set to a non-null value and notifies about returns from method calls when *notifyReturn* parameter is set. These interceptors are called *CallInterceptor* and *ReturnInterceptor*, respectively. When an interceptor notifies about both calls and returns it is called *Interceptor*. The name of the generated class is derived from the name of the delegated interface as follows: After the interceptor name placed in the bootstrap package follows the appended classname of the delegated interface type with dots replaced by underscores (e.g. `org/objectweb/dsrg/sofa/bootstrap/ReturnInterceptor$com_foo_Dummy`).

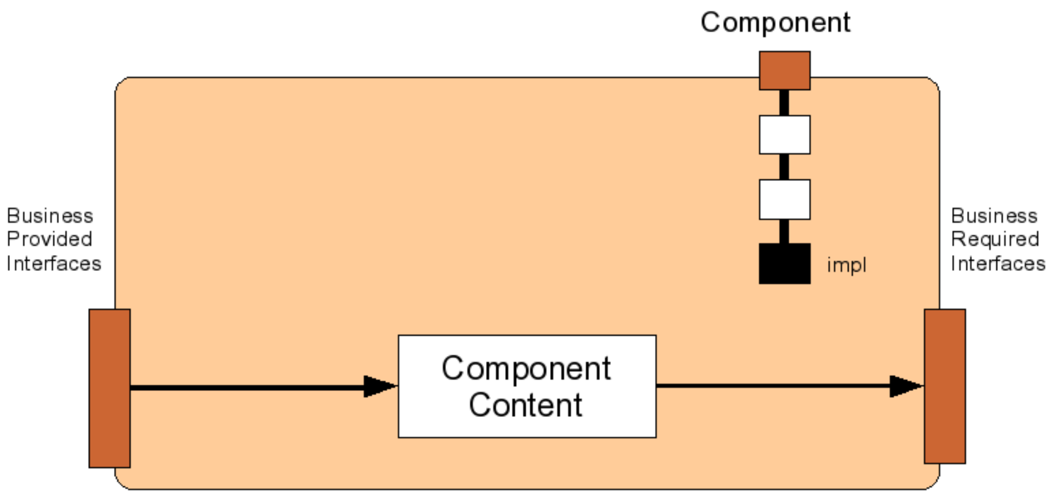
8.2. FactoryInterceptor

Factory interceptors are used to catch the return value of a factory call and to replace this return value with a reference to the entrypoint of the newly created collection interface instance. This job is not as performance critical as the job done by notifying interceptors, therefore standard Java reflection `Proxy` objects are used to intercept the calls to the factory interfaces. The Factory interceptor is implemented by the `FactoryInterceptor` class. The custom `Proxy` object is provided by the `FactoryInterceptorGenerator` class.

8.3. Component aspect

This aspect introduces control interface of `MIComponent` type called *Component*. The aspect is used to store and retrieve various information about a component. Once initialized (by the `init` method) it can provide references to control and business interfaces and other information. See `MIComponent` interface. This control interface is need for every component in the *SOFA 2* component system.

Figure 8.1. Model of the Component aspect

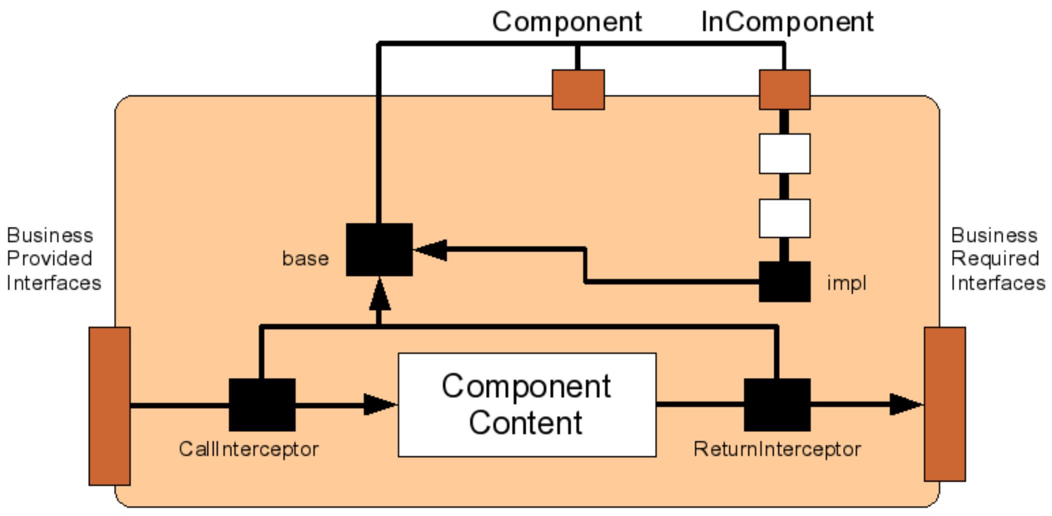


The Figure 8.1, “Model of the Component aspect” shows the architecture of the *Component* aspect. The *impl* microcomponent serves all requests on the *Component* control interface with data it stored at the time of its initialization.

8.4. InComponent aspect

This aspect introduces control interface of the `MIInComponent` type called *InComponent*. The purpose of this interface is to provide the business code with reference to the *Component* control interface. Interceptors are used to set the reference to the *InComponent* control interface using the `SOFAThreadHelper` class when a method on provided business interface is invoked or call to required business interface returns.

Figure 8.2. Model of the InComponent aspect



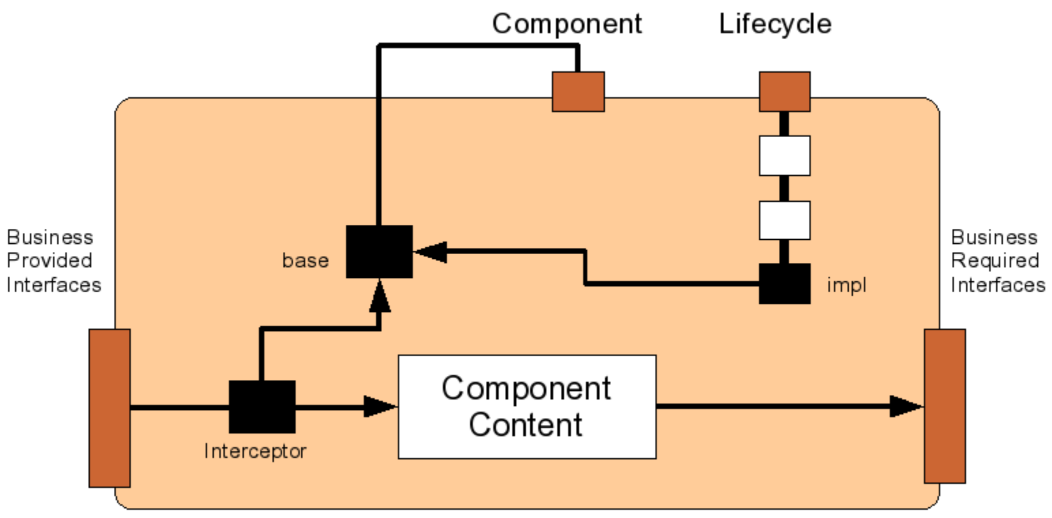
The Figure 8.2, “Model of the InComponent aspect” shows the architecture of the aspect. The interceptor notifies the *base* microcomponent when the program control enters a component. The *base* component sets reference to the *InComponent* control interface. The *impl* component delegates all method calls to the *base* microcomponent which implements the whole functionality. This is needed to overcome interface namespace visibility.

8.5. Lifecycle aspect

This aspect introduces control interface of the `MIInComponent` type called *InComponent*. The purpose of this interface is to control the component lifecycle. The call contexts are used to identify the threads entering and leaving the component.

A component has four states - *starting*, *started*, *stopping*, *stopped*. Only started components receive incoming calls. In other states the incoming threads are blocked. After instantiation, all the components are in the *stopped* state. A component is started by calling the `start` method on the *Lifecycle* interface. This method changes the state of the component to *starting* and calls the `start` method on the component content. After return, the component is advanced to the *started* state. Stopping running component is a two-phase process. First, the component is advanced to the *stopping* state by calling the `stop` method and the `stopping` method is called on the component content. Then, the method `waitStopped` is called. This method waits until all threads leave the component content. Then it calls the `stop` method on the component content which blocks it until the business code is ready to be unloaded. At this point the component state is changed to *stopped*. Alternatively, the `exit` method can be used to interrupt all blocked threads. This can be used to resolve deadlocks.

Figure 8.3. Model of the Lifecycle aspect



The Figure 8.3, “Model of the Lifecycle aspect” shows the architecture of the aspect. The *impl* microcomponent only delegates calls to the *base* microcomponent. The interceptor microcomponent notifies the *base* microcomponent about the calls and returns to the component. In the *base* microcomponent, the thread context is stored whenever a thread enters the component and is removed after it leaves. This mechanism is reentrant-safe. The thread that tries to reenter a component in the *stopping* state is permitted to enter otherwise a deadlock will occur. Also a child thread of the thread that would be able to reenter is permitted as the parent thread usually waits for the child threads.

Chapter 9. Building SOFA 2

9.1. Build Process

The original build process of SOFA 2 and its whole toolchain was based entirely on Apache Ant [http://ant.apache.org/] build system. This was sufficient in the project's early phases, but as the project has grown on and more and more subprojects were emerging, it became clear that this is not sufficient. There were lots of hidden dependencies and it was rather uneasy to manage them. Therefore, we have decided to improve the build process by utilizing the Apache Ivy [http://ant.apache.org/ivy/] dependency manager.

9.1.1. Requirements

The *SOFA 2* framework is written in the Java programming language and the Java Standard Edition 6.0 development kit is required to build it. For automated build system we have used Ant 1.6 which has to be installed prior to building the system with the Ivy plugin which will take care of needed external libraries.

9.1.2. Building SOFA 2

For centralization of the *SOFA 2* building process we introduced the `sofa-build` subproject which is present only for the purposes of *SOFA 2* code compilation and initial filling of the repository with examples.

To use the `sofa-build` subproject the `sofa`, `sofa-tools-api`, `cushion` and `sofa-examples` subprojects need to be present. To build *SOFA 2* along with the examples, run the `ant` command in the `sofa-build` directory. This command compiles all the needed projects, starts a repository and fills it with example data.

Chapter 10. Documentation

10.1. Overview

Documentation has to be written for users and programmers, which will use the SOFA 2 runtime. For source documenting the JavaDoc has been chosen. However, some form of a documentation system for generating printed documentation and HTML documentation had to be found too. We chose to use Docbook in SOFA 2 project because it is considered to be a stable format for documenting. Developers can extend the SOFA 2 documentation by updating the Docbook sources and exports to PDF can be easily regenerated. Following sections describe the documentation in detail.

10.2. Docbook

Docbook is a XML based documentation format. In simplicity it is only DTD. But XSL templates can be downloaded and using these stylesheets Docbook documents can be converted to various formats, for example RTF, DOC and especially PDF.

10.3. JavaDoc

For documenting the source code in Java JavaDoc has been chosen, because it is a standardized and widely used tool in Java community and there is no reason to look for any other tool. Documentation is generated in two versions: for users and for developers. Developers JavaDoc documentation contains all interfaces all methods anybody can use. But it contains methods meant only for internal usage too.

Part II. SOFA 2 Tools API

The sections in Part II present SOFA 2 Tools API which provides API that is generally used by other tools and applications.

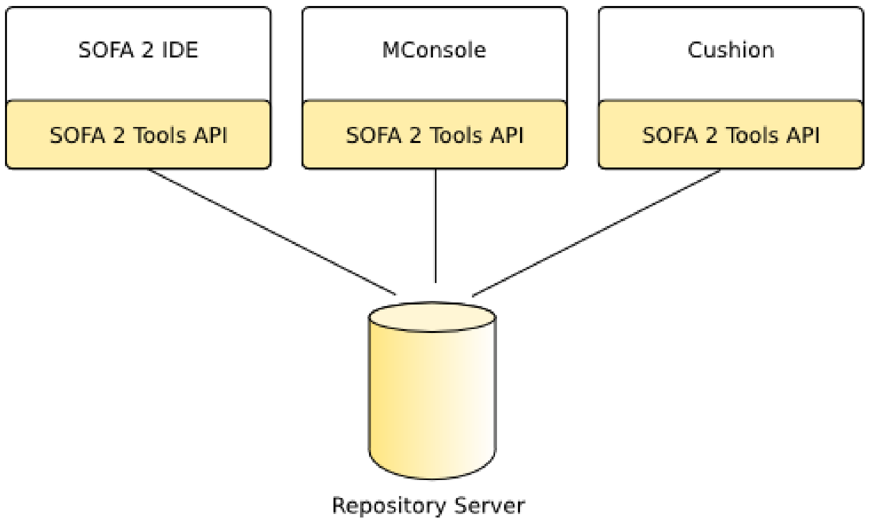
Chapter 1. Introduction

1.1. The Origin of SOFA 2 Tools API

As the SOFA and afterwards SOFA 2 project evolved, a command-line management tool called *Cushion* has been introduced. It provides set of actions covering essential development and deployment tasks. Later on, as SOFA 2 IDE and MConsole tools had approved needs of nearly the same actions as Cushion, the application logic of these actions have been separated into standalone module called SOFA 2 Tools API. The actions have been extended, improved and maintained within the module then and the original Cushion application became a thin shell wrapping it.

1.2. Role in the Overall Architecture of SO-FA 2 Framework

Figure 1.1. Placement of SOFA 2 Tools API in the SOFA 2 Framework



Basically, the SOFA 2 Tools API (further just *API*) provides interface to other tools standing along SOFA 2, as it is presented on Figure 1.1, “Placement of SOFA 2 Tools API in the SOFA 2 Framework”. The applications which utilize the *API* will be called *client*s of the *API*.

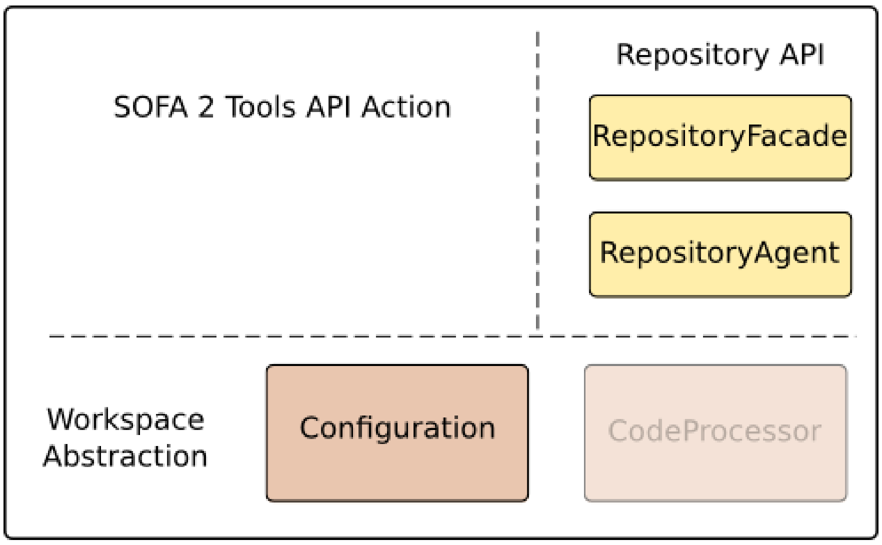
The *API* presents an *ADL* oriented view on entities of SOFA 2 model. The ADL is an simplified model in xml format, the *client* will mostly work with the ADL form of entities rather than entities of SOFA 2 model. Structure of the ADL format can be found in the **user's guide**, part about *Cushion*.

Chapter 2. Architecture

This chapter shows how the SOFA 2 Tools API is designed and structured.

2.1. Overview

Figure 2.1. SOFA 2 Tools API Action



From the *client*'s point of view, the API is separated into several actions. An action is a standalone class, with various `perform` methods. Actions typically interact with repository and *workspace*, a storage of *working entities*[1]. The actions also provide some outputs such as error, warning and informational messages. An instance of `EntityDescriptor` is generally the common argument of any `perform` method of any action, its role is to identify a SOFA 2 entity, which the action will work with. Figure 2.1, “SOFA 2 Tools API Action” shows theese concepts in a schematical way.

[1]The term "working entity" represents a SOFA 2 entity, that has been created or checked-out from repository, thus the *client* application is aware of it.

2.2. Structure

Figure 2.2. SOFA 2 Tools API Action Class Diagram

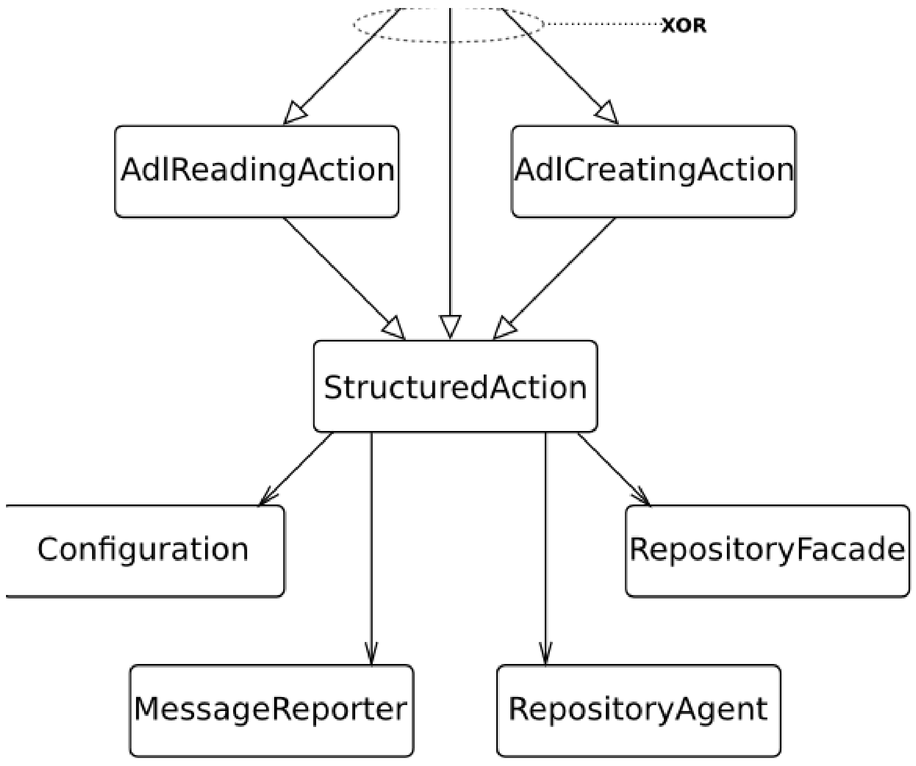


Figure 2.2, “SOFA 2 Tools API Action Class Diagram”shows common class structure of actions. The `StructuredAction` is a common base class for all actions. The `RepositoryFacade` and `RepositoryAgent` are members of *Repository API* (see Part I - SOFA 2, Section 3.5, “Using the Repository”).

2.2.1. Configuration

The `Configuration` interface declares several methods for workspace management, mainly for querying files and directories of working entities and also for adding working entities. Generally the implementation is supposed to be provided by *client*, nevertheless, SOFA 2 Tools API provides an implementation called `XMLConfiguration`, which is used in Cushion and SOFA 2 IDE.

The `XMLConfiguration` retains a set of working entities. For each entity it keeps its name, version identifier and a flag, which indicates, whether the entity is writable or read-only. It can be saved to a xml file named `.sofa2` and restored from the file again. The `XMLConfiguration` defines the *workspace* as working directory, where the `.sofa2` is stored.

2.2.2. CodeProcessor

The `CodeProcessor` is also an interface that helps API actions to manage source code of entities. The main purpose of this interface is to provide source code compilation, uploading and downloading source and binary code of entities. There is also an implementation available in SOFA 2 Tools API called `JavaCodeProcessor`. This implementation handles Java code, which is the only language supported by entire framework now.

The benefit of using `CodeProcessor` interface is that it makes API flexible due to implementation language of SOFA 2 applications. Support for another language can be added by introducing another `CodeProcessor`'s implementation.

2.2.3. EntityDescriptor

`EntityDescriptor` is simply an interface for an object that describes entity which should an action operate with. It is usually required as an argument of actions' `perform` method. There is also an implementation available in SOFA 2 Tools API called `EntityDescriptorImpl`, and there will be usually no reasons to re-implement it.

2.2.4. EntityInfoHelper

The `EntityInfoHelper` provides functionality for managing additional development informations related to an entity. There are two types of such informations which are listed below:

- Name of file with code of 3rd party *CodeBundle*
- Name of file with behavior protocol specification for *Frame*.

These informations are represented as *Info* objects associated with the entity. For particular name attributes of the *Info* objects, see the *javadoc* source documentation.

2.2.5. MessageReporter

As an action performs, it may reach some states which may be more on less considered as errors or should be at least reported as a warning. Although everything is all right, the action may wish to print some information in a form of an *info*.

For this functionality, API actions use `MessageReporter` interface for publishing `StructuredMessages`. The default implementation (`StdMessageReporter`) wrapps `java.io.PrintStream`, where the messages will be printed on.

2.3. Packages

This section provides an index of packages with brief descriptions for better orientation within *javadoc* source documentation.

The *javadoc* documentation can be generated from sources by running `ant jdoc` in the `sofa-tools-api` directory.

- **org.objectweb.dsrg.sofa.tools.api**

  API actions, *client*'s entry point

- **org.objectweb.dsrg.sofa.tools.api.codebundles**

  compilation and code bundle handling support
- **org.objectweb.dsrg.sofa.tools.api.config**

  workspace abstraction
- **org.objectweb.dsrg.sofa.tools.api.entity**

  entity description and helpers
- **org.objectweb.dsrg.sofa.tools.api.exceptions**

  exceptions used within API
- **org.objectweb.dsrg.sofa.tools.api.reporting**

  error, warnig and info reporting facility
- **org.objectweb.dsrg.sofa.tools.api.validation**

  entity (ADL) validation
- **org.objectweb.dsrg.sofa.tools.util**

  utilities, implementations of `Configuration` and `CodeProcessor`

2.4. Distribution Package Implementation

The distribution package is used by only two actions, `Export` and `Import`. It is a simple ZIP archive. It contains two or three files:

adl.xml	Definition of the entity described by SOFA 2 architecture description language file. See Part IV, “SOFA 2 ADL Developer Guide” for SOFA 2 ADL description.
contents.xml	Contains entity's name, version, type and lists all it's dependencies by their name and version. The `contents.xml` file is processed first by the import tool and the dependencies are checked. This way the import can satisfy the dependencies before actually creating the entity from the ADL file. The entity creation also requires an empty entity of corresponding type, therefor the type is mentioned in the `contents.xml` file.
`entity-name`.jar	Contains the entity's code, if the entity has one.

Chapter 3. Actions

This chapter describes SOFA 2 Tools API actions.

3.1. Overview of SOFA 2 API Actions

Following list enumerates *API* actions with their class names and a brief descriptions. For some particular actions, which are not very straightforward, a more detailed description is presented.

- `Checker`

  This action calls given behavior checker on given architecture. The given checker is instantiated within this action and the check itself is provided using `org.objectweb.dsrg.sofa.util.checker.BehaviorChecker`.

- `Checkout`

  This action checks entities out from a repository. It means that it locates the entities in the repostory, transforms them into the ADL representation and serializes them onto the workspace. The entities are also registered within the `Configuration`. In addition, if an entity has some related files such as source code, behavior description or structural diagram, they will be also downloaded into the workspace.

  The class extends `AdlCreatingAction`, which provides transformation from SOFA 2 entity model into ADL representation.

  The eventual source code of the entity is downloaded and unpacked into workspace using `CodeProcessor`. There is an exception for *third party code bundles*, where the code will not be unpacked, but stored as it is (*JAR* file) and the name of the file is obtained using `EntityInfoHelper`.

  Note that a *Frame* can contain behavior specification, which is intended to be stored in a separate file.

  The action provides two working modes; *recursive* and *single*. The recursive mode cheks given entity with all its dependencies while the single mode checks just the given entity. The *DependencyCalculator* class is used for the dependency lookup.

- `Commit`

  This action takes ADL representation of given entity, transforms it into the SOFA 2 model and updates the repository entity appropriately. The class extends `AdlReadingAction`, which provides transformation from ADL to repository model. First, the ADL document (adl.xml) is validated syntactically against the ADL schema (`org/objectweb/dsrg/sofa/tools/api/xsd/simpleadl.xsd`). Then it is transformed into xml representation of SOFA 2 model using XSL transformation (`org/objectweb/dsrg/sofa/tools/api/xslt/simple2repository.xslt`). After the transformation, the entity is validated with respect to other entities, for example reference consitency is checked. This validation is provided by `EntityValidator`.

  If the entity contain any information which is not part of the SOFA 2 model, this information is handled by `EntityInfoHelper`.

  This action provides commiting of single given entity and also *mass* commit of all working entities. The mass commit privdes exactly the same steps for each entity, but in following order from a global point of view:

  1. All entities are transformed into SOFA 2 model and additional informations are handled.

  2. All entities are validated using `EntityValidator`.

3. Changes are written into repository.

   Note that the source and byte-code of the entity is not uploaded within this action. See `UploadBundles` action.

- `Compile`

  This action provides compiling of entities' sources and also provides compilation of single entity or all working entities at once. The task may be logically separated into two subtasks; *dependency ordering* and the *compilation* itself.

  The *compilation* is provided by `CodeProcessor`, within the `processCode` method, which is called for each entity separately and in with all dependencies.

  The *dependency ordering* is provided by `CompileDependencyGraph`. It takes all entities to be compiled, constructs *dependency closure*, and orders all the entities topologically. Then the `Compile` action may compile entities with `CodeProcessor` in correct order and with all dependencies. Notice following feature of the *dependency closure* construction. Not all the dependencies are necessarilly working entites, thus the code bundles of (non-working) entities are downloaded into temporary storage, where they can be referenced for compilation (added in classpath).

- `Delete`

  This action deletes given entities from repository. Provides deletion of multiple entities, with or without their dependencies. The deletion itself is done using *RepositoryFacade*'s `delete` method (see Part I - SOFA 2, Section 3.5.2, “Repository Facade” and Section 3.5.3, “Deleting entities”).

  A possibility of confirmation of the deletion is provided through confirmation callback. There is a `Confirmable`, which may be optionally passed to the action. Then it is notified about each object which will be deleted and finally asked for confirmation.

- `Deploy`

  This action finalizes *DeploymentPlan* to be ready for launching. The local changes of deployment plan will be written onto repository and then the `DeploymentPlan`'s `deploy` method will be called. The method uses *congen*[1] to generate connectors.

- `Export`

  This action exports entities from repository to standalone packages. It can operate in recursive and non-recursive modes. In non-recursive mode it exports single entity to the destination directory. In recursive mode it also exports all dependencies, including transitive ones.

  For details of the packages' structure, please refer to Chapter 2 - Architecture, Section 2.4, “Distribution Package Implementation”.

- `Import`

  This action imports entities from standalone packages to repository. It always operates in recursive manner. Given a package to import, it first checks, whether all the dependencies for the package are satisfied in the repository. If not, it searches the directory of currently processed package for other packages. If the dependency can be satisfied by importing another package found in that directory, it will import the other package before the current one. This way it is possible to import whole application by a single call of the import tool. However, the user needs to know which entity is the top level one.

  For details of the packages' structure, please refer to Chapter 2 - Architecture, Section 2.4, “Distribution Package Implementation”.

- `New`

This action creates new entity in the repository as well as on the work space. The freshly created entity is an empty shell with name and version in the repository and is ready for updtes. In the workspace, a skeleton of ADL document is prepared using templates (see `org/objectweb/dsrg/sofa/tools/api/templates` directory). There are two modes of creation; *initial* and *next*. The *initial* creation mode creates entirely new entity with no ancestor. The *next* creation mode creates an successive version of an entity with given name and version.

- `PrepareAssembly`

  This action generates a skeleton of new *AssemblyDescriptor* based on a top-level architecture. It prepares an empty shell in the repository, and creates a skeleton of ADL. The *AssemblyDescrtiptor* describes a hierarchy of components, thus the ADL skeleton is made by traversing the hierarchy of components from the given top-level architecture over its subcomponents to primitive architectures (leafs of the hierarchy tree). Whenever the particular architecture is not known (only a frame reference represents a subcomponent), a placeholder is set in the ADL document to be filled in by user.

- `PrepareDeplPlan`

  This action generates a skeleton of new *deployment plan* based on a *assembly descriptor*. The assembly descriptor is traversed in a bottom-top manner, and the deployment plan is composed of the hierarchy nodes.

- `Print`

  This action prints out contents of repository. It provides listing versions of an entity given by name, listing all component types, its implementations and interfaces.

- `RepositoryDump`

  This action is intended for debugging purposes, it dumps all contents of the repository into a single file.

- `Status`

  This action lists content of workspace - working entities and their status.

- `Tag`

  This action adds a *tag* to given version of entity. The *tag* is an textual label for the version, it is represented as an `Info` named TAG referenced by the `Version` in the SOFA 2 model. The particular version of entity may be then referenced with the tag.

- `Update`

  This action updates given working entity or all of them. The class extends the `AdlCreatingAction`, the common base for `Checkout`, `Export` and `Update` It provides similar functionality as the `Checkout` action does (without registering the entities in the workspace - they are already present there).

- `UploadBundles`

  This action uploads code bundles onto repository. It also provides uploading code either of single entity or of all working entities.

  The code may be uploaded to *InterfaceType*, *MicroInterfaceType*, *Architecture* (a primitive one), *MicroComponent* or *CodeBundle* (a standalone one). Since only *CodeBundle*s may retain code in the repository, other entities will simply get an *CodeBundle* with their code.

  In the early stage of upload process of single entity, the code of the entity is packed into an single file (archive) using `CodeProcessor`'s `packCode` method. Then a *CodeBundle* is created (or obtained) in the repository and the archive is uploaded to it using *CodeBundleHelper* (see Part I - SOFA 2, Section 4.1.1, “Class name augmentation solution”). Finally, the *CodeBundle* is assigned to the entity (if the entity itself is not a standalone *CodeBundle*).

In case of uploading code of all entities, the process is exactly the same for each entity, but must be provided in correct order. The *CodeBundleHelper* performs also the *renaming* (see Part I - SOFA 2, Section 4.1.1, “Class name augmentation solution”), thus all dependencies must be uploaded before the entity itself. The `CompileDependencyGraph` is utilized for the ordering.

Part III. Cushion

The sections in Part III present Cushion which is command line tool for developing SOFA 2 applications.

Chapter 1. Developer's guide

1.1. Overview

Since *Cushion* mainly utilizies *SOFA 2 Tools API* (just API) in a straightforward way, we will just describe the process of API action invocation here. Each run of Cushion means execution of a action (except the *exec* action), where first argument from command-line presents the action name - command. The action name is taken from arguments and an *action wrapper* is then instantiated using *action registry*. Then a `perform` method of the *action wrapper* is called. The method takes successive arguments from command-line and converts them into a form, that the API action accepts. Then it instantiates API action, its *Configuration*[1] and calls its `perform` method, eventually catches exceptions from API.

1.2. Action Interface

The `ActionInterface` stands as a common interface for the *action wrapper*s. It contains method for performing the action, printing out help and usage messages. It also contains predefined constants for general return values of the `perform` method. Note that the `perform` itself should not print out any help or usage messages on error, it should rather return appropriate return value and the message is then printed from Cushion's main method.

1.3. Action Registry

All the Cushion's actions are managed using *ActionRegistry*, which is based on Java XML Bindnig [http://java.sun.com/xml/jaxb] (JAXB). This forms a uniform facility for maintaining available actions using single xml document and maps action name to its wrapper (appropriate implementation of `ActionInterface`). Naturally it also provides a list of action names for printing overall help message.

1.4. Parsing Command-line Arguments

In spite of some purely action-specific arguments and options, the main common argument is an entity identification. It consists of entity name and version or tag identifier. Some actions require just name, some require both parts in a mandatory or optional way. The `CommandLineParser` provides conversion from textual entity identification to API's `EntityDescriptor`, which is accepted by API actions. The conversion may be affected using modifiers to suite entity identification requirements of the action.

1.5. Script Execution

The `ExecuteScriptAction` is an exceptional action class, because it does not wrap any particular API action. Instead, it takes a file (given by argument) and performs each line as a standalone command. Thus for each line, it takes first word as an action name, following words as its arguments and executes appropriate action. Line starting with # is interpreted as a comment - it is ignored.

1.6. Other Features

Cushion contains some actions, that are not integrated in the API.

One of them is `RemoveAction`, which is not present in the API for its simplicity. It just excludes an entity from a list of working entities, so it operates only with the `XMLConfiguration`, calls its `remove` method.

[1]Note that Cushion uses `XMLConfiguration` which is an implementation of `Configuration` interface presented in presented in Section 2.2.1, “Configuration”.

Next one is `OsgiAction`. This action provides interface to integration of SOFA 2 components integration to the OSGI framework. The OSGI integration is not fully functional in the current state of SOFA 2 and it is out of the scope of this project.

Another group of features is formed by various behavioral and code conformance checkers. The **org.objectweb.dsrg.cushion.jpfcheck** package provides checking components' code and its protocol conformance using Java PathFinder [http://babelfish.arc.nasa.gov/trac/jpf] and the **org.objectweb.dsrg.cushion.behavior** package contains implementation of behavior checkers for *Behavior Protocols* and *Extended Behavior Protocols*. The formal methods and verification is out of scope of this work, thus we refer kind reader to following articles:
Behavior Protocols for Software Components[bp02]
Checking Software Components Behavior Using Behavior Protocols and Spin[bpspin07]
Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker[jpfcheck07]

1.7. Packages

This section presents package index for better orientation within the source codes and *javadoc* documentation.

- **org.objectweb.dsrg.cushion**

  contains application's main class `Cushion`, the action wrappers and the `CommandLineParser`

- **org.objectweb.dsrg.cushion.actionreg**

  contains action register.

- **org.objectweb.dsrg.cushion.behavior**

  contains behavior checkers.

- **org.objectweb.dsrg.cushion.jpfcheck**

  contains Java PathFinder based checker.

- **org.objectweb.dsrg.cushion.osgi**

  contains utilities for OSGI action.

References

[bp02] *Behavior Protocols for Software Components*. František Plášil. Stanislav Višňovský. 2002.

[bpspin07] *Checking Software Components Behavior Using Behavior Protocols and Spin*. Jan Kofroň. 2007.

[jpfcheck07] *Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker*. Pavel Parízek. František Plášil. Jan Kofroň. 2007.

Part IV. SOFA 2 ADL Developer Guide

The sections in Part IV present SOFA 2 ADL which is the model representation of SOFA 2 architecture description language.

Chapter 1. SOFA 2 ADL Overview

1.1. SOFA 2 ADL

The key part of entity from user point of view is the SOFA 2 architecture description language file. The SOFA 2 architecture description language is a XML file describing single top level entity. Before the start of the project, the ADL files were only used by Cushion as the only way to describe entity.

Because the cooperation of developer tools was the key requirement, we have decided to formally describe the ADL file by the Eclipse Modeling Framework [http://www.eclipse.org/modeling/emf/] metamodel. Moreover, the Eclipse Modeling Framework metamodel is able to create metamodel from existing XML Schema file and model instance can be therefore load and stored into an XML file.

Use of Eclipse Modeling Framework metamodel not only simplified the ADL files processing, but also allowed us to use other technologies which are based upon Eclipse Modeling Framework such as Eclipse Graphical Modeling Framework [http://www.eclipse.org/modeling/gmf/], Eclipse Textual Modeling Framework [http://www.eclipse.org/modeling/tmf/], JFace Data Binding [http://wiki.eclipse.org/index.php/JFace_Data_Binding], Operational QVT [http://www.eclipse.org/m2m/] transformations, and others.

In the early phases of projects, we have used the editors automatically generated from Eclipse Modeling Framework metamodel. These brought the basic edit support for ADL files which were sufficient in the beggining, but was not suitable for ordinary work. Therefore, we have decided to create new editors from scratch using JFace UI forms which will be similar to standard Eclipse Plugin editor and resembly its logic and semantics.

The Eclipse Modeling Framework model proved to be a good choice during the project as it was used not only in SOFA 2 IDE, but in emerged projects which stand on top of SOFA 2 as well.

Beside of the form editors, we have created another editor based on top of SOFA 2 ADL; the graphical architecture editor. The base part of the editor was created using the Eclipse Graphical Modeling Framework. This was further modified to support special properties of the SOFA 2 ADL metamodel which cannot be easily modeled by the Graphical Modeling Framework.

We have also extend the SOFA 2 ADL metamodel with some advanced features like entity resolvation which adds ability to traverse even to other models referenced from the model. This feature was therefore used in the Graphical Modeling Framework editor as well as in other tools which are being created on top of SOFA 2 ADL model as well as SOFA 2 IDE.

Entity resolvation also utilizes the Operational QVT transformation which is used to transform the SOFA 2 repository model to SOFA 2 ADL model.

Chapter 2. Programmer's Guide

SOFA 2 ADL allows user to create and edit SOFA 2 architecture description language files using hierarchical model.

SOFA 2 ADL supplies model and number of editors, dialogs and wizards to simplify SOFA 2 architecture description language files processing. If you are building a plug-in that interacts with SOFA 2 architecture description language files as part of its function, you may need to do one or more of the following things:

- Programmatically manipulate SOFA 2 ADL model, such as creating and modifying SOFA 2 entities definitions.
- Use existing SOFA 2 ADL editors.
- Add new functions and extensions to the SOFA 2 ADL itself.

The SOFA 2 ADL is structured into few major components:

SOFA 2 ADL	the model itself and infrastructure for processing the SOFA 2 architecture description language files.
SOFA 2 ADL Edit	the model item providers subsequently used for model editing support.
SOFA 2 ADL Editor	the model editors, dialogs and wizards infrastructure.
SOFA 2 ADL Diagram	the architecture graphical editor infrastructure.

2.1. SOFA 2 ADL

SOFA 2 ADL (**org.objectweb.dsrg.sofa.adl**) is the plug-in that contains the SOFA 2 ADL model and supporting infrastructure. You should always list this plug-in as a prerequisite when you are developing features using SOFA 2 ADL model.

SOFA 2 ADL packages give you access to the SOFA 2 ADL model objects and SOFA 2 ADL infrastructure. The SOFA 2 ADL packages include:

- **org.objectweb.dsrg.sofa.adl** contains the SOFA 2 ADL model.
- **org.objectweb.dsrg.sofa.adl.transformation** contains classes for transforming SOFA 2 repository to SOFA 2 ADL model.
- **org.objectweb.dsrg.sofa.adl.util** provides classes supporting SOFA 2 ADL model and infrastructure for model processing.

2.1.1. SOFA 2 ADL Model

The SOFA 2 ADL model is the set of classes that model the content of SOFA 2 architecture description language file content and allows to create and edit SOFA 2 entities it represents. The SOFA 2 ADL model classes are defined in **org.objectweb.dsrg.sofa.adl** package.

2.1.1.1.1. SOFA 2 ADL model processing

SOFA 2 ADL model can be created from SOFA 2 architecture description language file content. The model can be therefore processed and again saved to SOFA 2 architecture description language file. The following code snippet shows how to construct and save the model from/into file.

```
private void processSOFA2ADLModel(IFile file) {
  if (adl == null || !adl.exists()) {
    // the file does not exists
    return;
  }

  ResourceSet resourceSet = new ResourceSetImpl();
  Resource resource = resourceSet.getResource(URI
    .createPlatformResourceURI(file.getFullPath().toString(), false), true);
  SOFA2ADL model = (SOFA2ADL)resource.getContents().get(0);

  resource.save(new HashMap<Object, Object>());
}
```

2.2. SOFA 2 ADL Edit

SOFA 2 ADL Edit (**org.objectweb.dsrg.sofa.adl.edit**) is the plug-in providing SOFA 2 ADL model item providers used to edit the model content.

The SOFA 2 ADL Edit packages include:

- **org.objectweb.dsrg.sofa.adl.provider** contains SOFA 2 ADL model item providers.

2.3. SOFA 2 ADL Editor

SOFA 2 ADL Editor (**org.objectweb.dsrg.sofa.adl.editor**) is the plug-in implementing the SOFA 2 ADL model editors, dialogs and wizards that manipulates SOFA 2 ADL model elements.

The SOFA 2 ADL Editor packages include:

- **org.objectweb.dsrg.sofa.adl.presentation** contains SOFA 2 ADL Editor plugin implementation and the composed SOFA 2 ADL editor.
- **org.objectweb.dsrg.sofa.adl.presentation.dialogs** contains dialogs used in SOFA 2 ADL form editor pages.
- **org.objectweb.dsrg.sofa.adl.presentation.forms** contains SOFA 2 ADL editor form pages for processing SOFA 2 ADL model elements.
- **org.objectweb.dsrg.sofa.adl.presentation.parts** provides common parts used in SOFA 2 ADL editor form pages.
- **org.objectweb.dsrg.sofa.adl.presentation.wizards** contains wizards used in SOFA 2 ADL form editor pages as well as standalone wizards for processing SOFA 2 ADL model.

Chapter 3. Reference

3.1. Extension Points

The following extension points can be used to extend the capabilities of the SOFA 2 ADL model:

- **org.objectweb.dsrg.sofa.adl.entityLocators**

  This extension point allows client to contribute custom SOFA 2 ADL entity locators implementations.

  This is used while traversing entities through entity reference specified inside SOFA 2 ADL.

- **org.objectweb.dsrg.sofa.adl.implementationLocator**

  This extension point allows client to contribute custom implementation locator.

  This is used while browsing for implementation of components and interfaces.

- **org.objectweb.dsrg.sofa.adl.repositoryLocator**

  This extension point allows client to contribute custom repository locator.

  This is used while browsing repository for referenced entities.

Part V. SOFA 2 IDE Developer Guide

The sections in Part IV present SOFA 2 IDE which is graphical environment for developing SOFA 2 applications built as Eclipse IDE [http://www.eclipse.org/] plugin.

Chapter 1. SOFA 2 IDE Overview

1.1. SOFA 2 IDE

The main idea of SOFA 2 IDE is to resemble the ideas of Eclipse IDE in all the ways and to use the same logic and reuse the maximum of existing code and plugins.

The first version of SOFA 2 IDE was based on proprietary Eclipse Modeling Framework metamodel designed directly for usage with diagram editor, but without any relations to existing metamodels. This resulted in some unwanted features, such that the Cushion projects were incompatible with SOFA 2 IDE projects.

The current version of SOFA 2 IDE is therefore based on top of SOFA 2 ADL model as a common way of processing SOFA 2 architecture description language files which are used by all the SOFA 2 tools such as Cushion.

SOFA 2 IDE use two basic resource types, projects and entities. The project is an Eclipse project adapted to include new attributes related to SOFA 2. The entity is basically a folder named same as entity it contains, it contents therefore differs by the type of entity it contains. Eclipse representation of folder has been also adapted to contain all the additional information needed such as entity name, version, type, etc.

The project structure is basically defined by SOFA 2 Tools API and is therefore used by all tools based on top of it. For example Cushion, which also based upon this API, use the same structure. This gives the user power to use multiple tools at same time even on the same project. This is similar to what for example Subversion provides.

1.2. SOFA 2 Repository Team Provider

Standard Eclipse way to support version control systems is by providing the Eclipse Team Provider implementation. Because the SOFA 2 repository resembles version control system in many ways, we have decided to create SOFA 2 team provider implementation.

The overall architecture of SOFA 2 team provider was heavily inspired by Subversive [http://www.eclipse.org/subversive/] which is the official Subversion team provider implementation.

Overall implementation of SOFA 2 team provider is based on SOFA 2 Tools API using the actions it provides, encapsulating them with the Eclipse resources and providing them to user directly from Eclipse IDE.

Chapter 2. Programmer's Guide

SOFA 2 IDE allows users to create and edit SOFA 2 component model entities and supports complete SOFA 2 development process.

The SOFA 2 IDE makes use of many of the platform extension points and frameworks described in the Platform Plug-in Developer Guide. It's easiest to think of the SOFA 2 IDE as a set of plug-ins that add SOFA 2 specific behavior to the generic platform resource model and contribute SOFA 2 specific views, editors, and actions to the workbench.

This guide discusses the extension points and API provided by the SOFA 2 IDE. We assume that you already understand the concepts of plug-ins, extension points, workspace resources, and the workbench UI.

SOFA 2 IDE supplies full featured IDE for SOFA 2 component system, why would you need to use the SOFA 2 IDE API? If you are building a plug-in that interacts with SOFA 2 component system or resources as part of its function, you may need to do one or more of the following things:

- Programmatically manipulate SOFA 2 resources, such as creating projects, creating and editing SOFA 2 entities.
- Programmatically use the SOFA 2 repository.
- Add new functions and extensions to the SOFA 2 IDE itself.

The SOFA 2 IDE is structured into few major components:

SOFA 2 IDE	the headless infrastructure for compiling and manipulating SOFA 2 resources.
SOFA 2 IDE UI	the user interface extensions that provide the IDE.
SOFA 2 IDE Repository	the headless infrastructure providing SOFA 2 repository team provider implementation.
SOFA 2 IDE Repository UI	the user interface extensions for team provider implementation.
SOFA 2 IDE Cushion	the headless infrastructure providing Cushion support.
SOFA 2 IDE Cushion UI	the user interface extensions for Cushion support in IDE.
SOFA 2 IDE JDT	the headless infrastructure for JDT integration in SOFA 2 IDE as runtime platform.
SOFA 2 IDE JDT UI	the user interface extensions for JDT integration in IDE.

2.1. SOFA 2 IDE

SOFA 2 IDE (**org.objectweb.dsrg.sofa.eclipse**) is the plug-in that defines the core SOFA 2 elements and API. You should always list this plug-in as a prerequisite when you are developing SOFA 2 specific features.

SOFA 2 IDE packages give you access to the SOFA 2 model objects and headless SOFA 2 IDE infrastructure. The SOFA 2 IDE packages include:

- **org.objectweb.dsrg.sofa.eclipse** contains SOFA 2 IDE plugin implementation.
- **org.objectweb.dsrg.sofa.eclipse.commands** provides implementation of basic SOFA 2 project commands.

- **org.objectweb.dsrg.sofa.eclipse.configuration** provides classes supporting SOFA 2 project configuration.
- **org.objectweb.dsrg.sofa.eclipse.expressions** provides classes implementing expressions testers for SOFA 2 elements.
- **org.objectweb.dsrg.sofa.eclipse.platform** provides classes and interfaces for SOFA 2 runtime platform.
- **org.objectweb.dsrg.sofa.eclipse.project** defines the classes for SOFA 2 project support.
- **org.objectweb.dsrg.sofa.eclipse.resources** defines the classes that describe the SOFA 2 resource model.

2.1.1. SOFA 2 Model

The SOFA 2 model is the set of classes that model the entities associated with creating, editing, and building a SOFA 2 application. The SOFA 2 model classes are defined in **org.objectweb.dsrg.sofa.eclipse.resources** package. These classes implement SOFA 2 specific behavior for resource.

2.1.1.1. SOFA 2 Elements

The package **org.objectweb.dsrg.sofa.eclipse.resources** defines the classes that model the entities that compose a SOFA 2 application as resources. The resource structure is defined from the SOFA 2 Tools API workspace abstraction. The model is hierarchical. Elements of a application can be decomposed into entities.

Manipulating SOFA 2 entity is similar to manipulating resource objects. When you work with a SOFA 2 entity, you are actually working with a handle to some underlying model object.

The following table summarizes the different kinds of SOFA 2 elements:

Element	**Description**
`ISOFA2Resource`	Represents the SOFA 2 entity as a general resource.
`ISOFA2LocalResource`	Represents the SOFA 2 entity as a local resource, corresponding to the workspace.
`ISOFA2Project`	Represents the SOFA 2 project.

Some of the elements are shown in the SOFA 2 navigator view. Other elements correspond to the virtual items used in operation's implementation.

2.1.1.1.1. SOFA 2 elements and their resources

Many of the SOFA 2 elements correspond to generic resources in the workspace. When you want to create SOFA 2 elements from a generic resource, use the standard `IAdaptable` interface. The following code snippet shows how to get SOFA 2 elements from their corresponding resources.

```
private void createSOFA2ElementsFrom(IProject project, IFolder folder) {
  ISOFA2Project sofa2Project = (ISOFA2Project)project
    .getAdapter(ISOFA2Project.class);
  if (sofa2Project == null) {
    // the project does not have SOFA 2 nature
    return;
  }

  ISOFA2LocalResource resource = (ISOFA2LocalResource)folder
    .getAdapter(ISOFA2LocalResource.class);
  if (resource == null) {
    // the folder does not represent SOFA 2 entity
    return;
  }
}
```

2.1.1.1.2. SOFA 2 projects

When you create a SOFA 2 project from a simple project, the implementation will check to see if the project is configured with the SOFA 2 nature. The SOFA 2 IDE plug-in uses a project nature to designate a project as having SOFA 2 behavior. This nature (`org.objectweb.dsrg.sofa.eclipse.project.SOFA2ProjectNature`) is assigned to a project when the "New SOFA 2 project" wizard creates a project. If the SOFA 2 nature is not configured on a project, the `IProject#getAdapter(Class)` will return `null` when asked to adapt the project.

2.1.2. SOFA 2 Configuration

All SOFA 2 projects has associated SOFA 2 configuration. This is used by the SOFA 2 Tools API action implementations and contains information about SOFA 2 entities' resource contained in the SOFA 2 project. The SOFA 2 configuration is created inside a project when the "New SOFA 2 project" wizard creates a project itself.

SOFA 2 IDE contains its own implementation of SOFA 2 configuration based on standard implementation provided by SOFA 2 Tools API. This is represented by the `ISOFA2ProjectConfiguration` interface and can be resolved from the `ISOFA2Project` instance by using the `ISOFA2Project#getConfiguration()` method.

2.1.3. SOFA 2 Runtime Platform

SOFA 2 IDE implementation is aimed to independent on SOFA 2 runtime platform implementation. This is achieved by the complete separation of the platform implementation behind the `ISOFA2RuntimePlatform` interface.

Runtime platform implementations are usually implemented as a separate Eclipse bundles and registered using the **org.objectweb.dsrg.sofa.eclipse.runtimePlatforms** extension point.

Particular SOFA 2 runtime platform is assigned to a project when the "New SOFA 2 project" wizard creates a project and can be chosen by the user. This can be resolved from the `ISOFA2Project` instance by using the `ISOFA2Project#getPlatform()` method.

Exactly one runtime platform can be marked as default. This platform is used when user does not explicitly define runtime platform for newly created project.

2.2. SOFA 2 IDE UI

SOFA 2 IDE UI (**org.objectweb.dsrg.sofa.eclipse.ui**) is the plug-in implementing the SOFA 2 specific user interface classes that manipulate SOFA 2 elements.

The packages in the SOFA 2 IDE UI implement the SOFA 2-specific extensions to the workbench. The SOFA 2 IDE UI packages include:

- **org.objectweb.dsrg.sofa.eclipse.ui** contains SOFA 2 IDE UI plugin implementation.
- **org.objectweb.dsrg.sofa.eclipse.ui.actions** provides basic actions for SOFA 2 IDE UI.
- **org.objectweb.dsrg.sofa.eclipse.ui.decorator** provides decorators for SOFA 2 resources
- **org.objectweb.dsrg.sofa.eclipse.ui.extension** contains interfaces for SOFA 2 IDE UI extensions points.

- **org.objectweb.dsrg.sofa.eclipse.ui.navigator** contains classes related to SOFA 2 navigator implementation.
- **org.objectweb.dsrg.sofa.eclipse.ui.operations** provides classes supporting SOFA 2 project operations.
- **org.objectweb.dsrg.sofa.eclipse.ui.perspectives** provides classes implementing SOFA 2 perspectives.
- **org.objectweb.dsrg.sofa.eclipse.ui.platform** provides classes and interfaces for SOFA 2 runtime platform.
- **org.objectweb.dsrg.sofa.eclipse.ui.preferences** provides classes implementing SOFA 2 IDE preferences pages.
- **org.objectweb.dsrg.sofa.eclipse.ui.views** provides classes implementing SOFA 2 related views.
- **org.objectweb.dsrg.sofa.eclipse.ui.wizards** provides classes implementing SOFA 2 related wizards.

2.2.1. SOFA 2 Runtime UI Platform

SOFA 2 runtime platform implementation can also use the user interface to be provided by the information from user. For this purpose, `ISOFA2RuntimeUIPlatform` has been defined as a extension to the `ISOFA2RuntimePlatform` interface.

Implementations of the SOFA 2 runtime user interface platforms are also registered using the **org.objectweb.dsrg.sofa.eclipse.runtimePlatforms** extension point same as standard runtime platform implementation.

2.2.2. SOFA 2 Actions

The **org.objectweb.dsrg.sofa.eclipse.ui.actions** package provides actions related to SOFA 2 elements and workspace containing SOFA 2 entities.

Action `AddSOFA2NatureAction` is used to add *SOFA 2 Nature* to selected projects while action `RemoveSOFA2NatureAction` is used to remove it.

2.2.3. SOFA 2 Views

The **org.objectweb.dsrg.sofa.eclipse.ui.views** and **org.objectweb.dsrg.sofa.eclipse.ui.navigator** packages contains views that presents information related to SOFA 2 elements to the user.

The `SOFA2ResourceNavigator` is the main view providing view of workspace that contains SOFA 2 entities.

2.2.4. SOFA 2 Wizard Pages

The **org.objectweb.dsrg.sofa.eclipse.ui.wizards** package provides wizard pages for creating and configuring some of the SOFA 2 elements.

2.2.4.1. Creating new SOFA 2 elements

`SOFA2ProjectWizard` allows user to create new SOFA 2 projects. `SOFA2ProjectPage` allows to define new SOFA 2 project. Clients can use and configure this page in their own project wizard.

The concrete creation wizards can be used directly and generally are not intended to be subclassed.

2.2.4.2. Contributing a wizard page

Users can contribute their own wizard pages and operation into the `SOFA2ProjectWizard` by using the **org.objectweb.dsrg.sofa.eclipse.ui.project** extension point and implementing the `ISOFA2ProjectFactory` interface. Using this way, users can customize the SOFA 2 project creation.

2.3. SOFA 2 IDE Repository

SOFA 2 IDE Repository (**org.objectweb.dsrg.sofa.eclipse.repository**) is the plug-in that contains SOFA 2 Team Provider implementation and SOFA 2 repository resource elements.

The packages in the SOFA 2 IDE Repository provides access to SOFA 2 repository by implementing standard Eclipse Team Support interfaces.

- **org.objectweb.dsrg.sofa.eclipse.repository** contains SOFA 2 Team Provider implementation.
- **org.objectweb.dsrg.sofa.eclipse.repository.commands** provides abstract SOFA 2 repository commands.
- **org.objectweb.dsrg.sofa.eclipse.repository.commands.local** provides local SOFA 2 repository commands.
- **org.objectweb.dsrg.sofa.eclipse.repository.commands.remote** provides remote SOFA 2 repository commands.
- **org.objectweb.dsrg.sofa.eclipse.repository.history** contains classes for SOFA 2 version history access.
- **org.objectweb.dsrg.sofa.eclipse.repository.resources** defines the classes that describe the SOFA 2 repository model.

2.3.1. SOFA 2 Repository Model

The SOFA 2 repository model is the set of classes that model the elements associated with the SOFA 2 repository. The SOFA 2 repository model classes are defined in **org.objectweb.dsrg.sofa.eclipse.repository.resources** package.

2.3.1.1. SOFA 2 Elements

The package **org.objectweb.dsrg.sofa.eclipse.repository.resources** defines the classes that model the elements in the SOFA 2 repository. The element hierarchy is defined by the repository implementation. The model is hierarchical.

The following table summarizes the different kinds of SOFA 2 repository elements:

Element	**Description**
`ISOFA2RemoteResource`	Represents the SOFA 2 entity as a remote resource, corresponding to the repository element.
`ISOFA2RepositoryContainer`	Represents the SOFA 2 entity container which contains the SOFA 2 repository entities.
`ISOFA2RepositoryLocation`	Represents the SOFA 2 repository location.

Some of the elements are shown in the SOFA 2 repository view. Other elements correspond to the virtual items used in operation's implementation.

Registered repository locations can be listed and accessed using `SOFA2RepositoryManager` class.

2.4. SOFA 2 IDE Repository UI

SOFA 2 IDE Repository UI (**org.objectweb.dsrg.sofa.eclipse.repository.ui**) is the plug-in implementing the SOFA 2 Team Provider specific user interface classes that manipulate SOFA 2 repository elements.

The packages in the SOFA 2 IDE Repository UI implement the SOFA 2-specific extensions to the workbench. The SOFA 2 IDE Repository UI packages include:

- **org.objectweb.dsrg.sofa.eclipse.repository.ui** contains SOFA 2 IDE Repository UI plugin implementation.
- **org.objectweb.dsrg.sofa.eclipse.repository.ui.actions** provides abstract actions for SOFA 2 Repository IDE UI.
- **org.objectweb.dsrg.sofa.eclipse.repository.ui.actions.local** provides local repository actions for SOFA 2 Repository IDE UI.
- **org.objectweb.dsrg.sofa.eclipse.repository.ui.actions.remote** provides remote repository actions for SOFA 2 Repository IDE UI.
- **org.objectweb.dsrg.sofa.eclipse.repository.ui.decorator** provides decorators for SOFA 2 repository resources.
- **org.objectweb.dsrg.sofa.eclipse.repository.ui.extension** contains SOFA 2 extension points repository-related implementaions.
- **org.objectweb.dsrg.sofa.eclipse.repository.ui.history** provides classes implementing SOFA 2 history access.
- **org.objectweb.dsrg.sofa.eclipse.repository.ui.operations** provides classes supporting SOFA 2 repository operations.
- **org.objectweb.dsrg.sofa.eclipse.repository.ui.perspectives** provides classes implementing SOFA 2 repository perspectives.
- **org.objectweb.dsrg.sofa.eclipse.repository.ui.views** provides classes implementing SOFA 2 repository related views.
- **org.objectweb.dsrg.sofa.eclipse.repository.ui.views.model** contains classes that implements SOFA 2 repository views model.
- **org.objectweb.dsrg.sofa.eclipse.repository.ui.wizards** provides classes implementing SOFA 2 repository related wizards.

2.4.1. SOFA 2 Repository Actions

The **org.objectweb.dsrg.sofa.eclipse.repository.ui.actions** package provides actions related to SOFA 2 repository.

SOFA 2 repository actions are derived from `SOFA2Action` abstract action.

2.4.1.1. Local SOFA 2 repository actions

Local repository actions operate on local entities in SOFA 2 project associated with SOFA 2 repository. These are contained in **org.objectweb.dsrg.sofa.eclipse.repository.ui.actions.local** and derive from `SOFA2LocalAction` subclass.

Action	Description
`SOFA2CommitAction`	Action for committing one or more SOFA 2 entities into repository.

Action	Description
`SOFA2HistoryAction`	Action that shows history of selected local SOFA 2 entity.
`SOFA2TagAction`	Action that assigns tag to current version of selected SOFA 2 entity.
`SOFA2UpdateAction`	Action that update local version of SOFA 2 entity for changes from repository.
`SOFA2UploadAction`	Action for uploading files associated with entities to SOFA 2 repository.

2.4.1.2. Remote SOFA 2 repository actions

Remote repository actions operate on remote entities directly in SOFA 2 repository. These are contained in **org.objectweb.dsrg.sofa.eclipse.repository.ui.actions.remote** and derive from `SOFA2RemoteAction` subclass.

Action	Description
`SOFA2AddRepositoryAction`	Action for adding SOFA 2 repository location into workbench.
`SOFA2CopyNameAction`	Action for copying name of selected SOFA 2 entity into clipboard.
`SOFA2CopyVersionAction`	Action for copying version of selected SOFA 2 entity into clipboard.
`SOFA2DeleteAction`	Action that deletes selected SOFA 2 entity from repository.
`SOFA2DiscardRepositoryAction`	Action that discard selected SOFA 2 repository location from workbench.
`SOFA2ExportAction`	Action for exporting remote SOFA 2 entity into distribution package.
`SOFA2HistoryAction`	Action that shows history of selected remote SOFA 2 entity.
`SOFA2CheckoutAction`	Action that checkouts selected remote SOFA 2 entity into local workspace.
`SOFA2ImportAction`	Action for importing SOFA 2 entity contained in distribution package into repository.
`SOFA2RefreshAction`	Action that refresh selected SOFA 2 entity or element content.
`SOFA2TagAction`	Action that assigns tag to current version of selected SOFA 2 entity.

2.4.2. SOFA 2 Repository Views

The **org.objectweb.dsrg.sofa.eclipse.repository.ui.views** package contain views that presents SOFA 2 repository related information to the user.

The `SOFA2RepositoriesView` provides view of repository content and `SOFA2RepositoryBrowserView` allows user to browse content of selected repository SOFA 2 entity.

2.4.3. SOFA 2 Repository Wizard Pages

The **org.objectweb.dsrg.sofa.eclipse.repository.ui.wizards** package provides wizard pages for creating and configuring SOFA 2 repository elements and resources.

2.4.3.1. Creating new repository SOFA 2 elements

`SOFA2EntityWizard` allows user to create new SOFA 2 entity. `SOFA2EntityPage` allows to define new SOFA 2 entity properties. Newly created entity is automatically checked out into the selected project.

Particular SOFA 2 elements wizards are derived from `SOFA2EntityWizard` abstract wizard, these are

- `SOFA2InterfaceTypeWizard` for creating new SOFA 2 interface types
- `SOFA2FrameWizard` for creating new frames
- `SOFA2ArchitectureWizard` for creating new architectures
- `SOFA2AssemblyWizard` for creating new assemblies
- `SOFA2CodeBundleWizard` for creating new code bundles

The concrete creation wizards can be used directly and generally are not intended to be subclassed.

2.4.3.2. Using SOFA 2 repository elements

`SOFA2CheckoutWizard` allows user to checkout SOFA 2 entities into selected project. `SOFA2CheckoutPage` and `SOFA2CheckoutProjectPage` allows to specify checkout properties and to selected the project where entities will be checkouted.

`SOFA2CommitWizard` allows to commit SOFA 2 entities into repository, `SOFA2CommitPage` allows to select which entities will be commited.

Chapter 3. Reference

3.1. API Reference

For API reference, please see the Javadoc documentation.

3.2. Extension Points

The following extension points can be used to extend the capabilities of the SOFA 2 IDE infrastructure:

- **org.objectweb.dsrg.sofa.eclipse.runtimePlatforms**

  This extension point allows client to contribute custom SOFA 2 Runtime Platform implementation into SOFA 2 IDE

- **org.objectweb.dsrg.sofa.eclipse.ui.project**

  This extension point allows client to contribute factory for customization of SOFA 2 project wizard.

Part VI. MConsole Developer Guide

The sections in Part V present MConsole which is graphical environment for managing SOFA 2 runtime environment built as Eclipse RCP [http://www.eclipse.org/] application.

Chapter 1. MConsole Overview

1.1. MConsole

The main purpose of MConsole is to have the tool which could be used by developers as well as administrators to manage the SOFA 2 runtime environment and applications. Therefore we have decided to distibute it as Eclipse plugin as well as standalone Eclipse Rich Client Platform [http://www.eclipse.org/home/categories/rcp.php] based application.

While SOFA 2 IDE focuses only on development phase, the MConsole should focus only on deployment phase. Therefore it is user interface and logic differs from standard Eclipse logic and was inspire with applications with similar purpose such as Microsoft Management Console.

The MConsole, same as SOFA 2 IDE, utilizes the SOFA 2 ADL metamodel to access and process deployment plans. Also, it uses the provided form editors to give user a way to create new and edit existing deployment plans.

MConsole also provides way to manage SOFAnodes, start and stop their repositories, start new and stop existing deployment dock, run and shutdown applications. For this purpose, it also utilizes the Eclipse Zest [http://www.eclipse.org/gef/zest/] visualization toolkit to present the information in a way that user can easily understand.

MConsole utilizes the MBeans provided by SOFA 2 runtime to access its data and manage it.

Chapter 2. Programmer's Guide

MConsole allows users to manage SOFA 2 runtime environment and deploy SOFA 2 applications.

The MConsole makes use of many of the platform extension points and frameworks described in the Platform Plug-in Developer Guide. MConsole consist from set of plug-ins placed on top of Eclipse Rich Client Platform contributing SOFA 2 specific views and actions to the workbench for SOFA 2 environment management.

This guide discusses the extension points and API provided by the MConsole. We assume that you already understand the concepts of plug-ins, extension points, workspace resources, and the workbench UI.

The MConsole is structured into few major components:

MConsole	the headless infrastructure for manipulating, deployment and launching of SOFA 2 resources.
MConsole UI	the user interface extensions that provide the MConsole.
MConsole Application	the implementation providing standalone application based on Eclipse Rich Client Platform.

2.1. MConsole

MConsole (**org.objectweb.dsrg.sofa.mconsole**) is the plug-in that defines the core MConsole elements and API.

MConsole packages give you access to the SOFA 2 MConsole model objects and headless MConsole infrastructure. The MConsole packages include:

- **org.objectweb.dsrg.sofa.mconsole** contains MConsole plugin implementation.
- **org.objectweb.dsrg.sofa.mconsole.commands** provides implementation of basic MConsole management commands.
- **org.objectweb.dsrg.sofa.mconsole.configuration** provides classes supporting MConsole management configuration.
- **org.objectweb.dsrg.sofa.mconsole.expressions** provides classes implementing expressions testers for MConsole elements.
- **org.objectweb.dsrg.sofa.mconsole.launcher** provides MConsole application launcher classes.
- **org.objectweb.dsrg.sofa.mconsole.resources** defines the classes that describe the SOFA 2 MConsole resource model.

2.1.1. MConsole Model

The SOFA 2 MConsole model is the set of classes that model the entities associated with managing and deploying a SOFA 2 application. The SOFA 2 MConsole model classes are defined in **org.objectweb.dsrg.sofa.mconsole.resources**.

2.1.1.1. SOFA 2 MConsole Elements

The package **org.objectweb.dsrg.sofa.mconsole.resources** defines the classes that model the entities that compose a SOFA 2 application and SOFA 2 runtime environment. The model is hierarchical.

The following table summarizes the different kinds of SOFA 2 MConsole elements:

Element	Description
`IMConsoleSOFAnode`	Represents the SOFA 2 SOFAnode.
`IMConsoleDeploymentDockRegistry`	Represents the SOFA 2 deployment dock registry adherent to SOFAnode.
`IMConsoleDeploymentDock`	Represents the SOFA 2 deployment dock, registered in the dock registry.
`IMConsoleConnectorManager`	Represents the SOFA 2 connector manager adherent to SOFAnode.
`IMConsoleComponent`	Represents the SOFA 2 component executed in the runtime environment.
`IMConsoleRepository`	Represents the SOFA 2 repository adherent to SOFAnode.
`IMConsoleRepositoryContainer`	Represents the SOFA 2 entity container which contains the SOFA 2 repository entities.
`IMConsoleResource`	Represents the SOFA 2 entity as a resource, corresponding to the repository element.
`IMConsoleZeroConfServer`	Represents the SOFA 2 zero-configuration server.
`IMConsoleLauncher`	Represents the SOFA 2 zero-configuration launcher.

Some of the elements are shown in the MConsole navigator view. Other elements correspond to the virtual items used in operation's implementation.

Registered SOFAnodes can be listed and accessed using `MConsoleNodeManager` class.

2.2. MConsole UI

MConsole UI (**org.objectweb.dsrg.sofa.mconsole.ui**) is the plug-in implementing the MConsole specific user interface classes that manipulate SOFA 2 MConsole elements and manages SOFA 2 runtime environment.

The packages in the MConsole UI implement the SOFA 2-specific extensions to the workbench. The MConsole UI packages include:

- **org.objectweb.dsrg.sofa.mconsole.ui** contains MConsole UI plugin implementation.
- **org.objectweb.dsrg.sofa.mconsole.ui.actions** provides basic actions for MConsole UI.
- **org.objectweb.dsrg.sofa.mconsole.ui.decorator** provides decorators for SOFA 2 MConsole resources
- **org.objectweb.dsrg.sofa.mconsole.ui.launcher** provides MConsole UI application launcher classes.
- **org.objectweb.dsrg.sofa.mconsole.ui.navigator** contains classes related to MConsole navigator implementation.
- **org.objectweb.dsrg.sofa.mconsole.ui.operations** provides classes supporting MConsole management operations.
- **org.objectweb.dsrg.sofa.mconsole.ui.perspectives** provides classes implementing MConsole perspectives.
- **org.objectweb.dsrg.sofa.eclipse.ui.platform** provides classes and interfaces for SOFA 2 runtime platform.
- **org.objectweb.dsrg.sofa.mconsole.ui.views** provides classes implementing SOFA 2 related views.

- **org.objectweb.dsrg.sofa.mconsole.ui.views.model** contains classes that implements MConsole views model.
- **org.objectweb.dsrg.sofa.mconsole.ui.wizards** provides classes implementing MConsole wizards.

2.2.1. MConsole Actions

The **org.objectweb.dsrg.sofa.mconsole.ui.actions** package provides actions for SOFA 2 repository elements creation and usage as well as for SOFA 2 runtime environment management.

MConsole actions are derived from `MConsoleAction` abstract action.

2.2.1.1. Managing SOFA 2 repository elements

Action	Description
`MConsoleDeploymentPlanAction`	Action for creating new SOFA 2 deployment plan entity.
`MConsoleEditAction`	Action for editing of existing SOFA 2 deployment plan entity.
`MConsoleDeleteAction`	Action that deletes selected SOFA 2 entity from repository.
`MConsoleTagAction`	Action that assigns tag to current version of selected SOFA 2 entity.
`MConsoleCopyNameAction`	Action for copying name of selected SOFA 2 entity into clipboard.
`MConsoleCopyVersionAction`	Action for copying version of selected SOFA 2 entity into clipboard.

2.2.1.2. Managing SOFA 2 runtime environment

Action	Description
`MConsoleDeploymentDockAction`	Action for starting new SOFA 2 deployment dock.
`MConsoleRemoveAction`	Action that removes selected SOFAnode from workbench.
`MConsoleRefreshAction`	Action that refresh selected SOFAnode child content.

2.2.2. MConsole Views

The **org.objectweb.dsrg.sofa.mconsole.ui.navigator** and **org.objectweb.dsrg.sofa.mconsole.ui.views** packages contains views that presents information about SOFA 2 environment to the user.

The `MConsoleNavigator` is the main view offering global view of selected SOFAnodes and their content. The `MConsoleDiagramView` view offers visualization of the runtime environment while `MConsoleOutlineView` presents structure of the selected environment element.

2.2.3. MConsole Wizard Pages

The **org.objectweb.dsrg.sofa.mconsole.ui.wizards** package provides wizard pages for creating SOFA 2 repository elements and managing the SOFA 2 runtime environment.

2.2.3.1. Creating new repository SOFA 2 elements

`SOFA2DeploymentPlanWizard` allows user to create and edit new SOFA2 deployment plan, `SOFA2DeploymentPlanWizardPage` allows to define new SOFA2 deployment plan properties. Newly created deployment plan is automatically commited into the selected node's repository.

2.2.3.2. Managing SOFA 2 runtime environment

`MConsoleSOFAnodeWizard` allows user to add new SOFAnode into workbench, `MConsoleSOFAnodeWizardPage` allows to specify general SOFAnode properties while `MConsoleAutoconfiguredSOFAnodeWizardPage` allow to specify properties of auto configured SOFAnode and `MConsoleManuallyConfiguredSOFAnodeWizardPage` allows to specify properties of manually configured SOFAnode.

`MConsoleDeploymentDockWizard` allows to start new SOFA 2 deployment dock if launcher is available for autoconfigured SOFAnode, `MConsoleDeploymentDockWizardPage` allows to specify dock properties.

2.3. MConsole Application

MConsole Application (**org.objectweb.dsrg.sofa.mconsole.application**) is the plug-in that provides standalone application implementation based on Eclipse Rich Client Platform.

The MConsole Application packages include:

- **org.objectweb.dsrg.sofa.mconsole.application** contains MConsole Application implementation.
- **org.objectweb.dsrg.sofa.mconsole.application.intro** provides implementation of MConsole Application intro page.

Chapter 3. Reference

3.1. API Reference

For API reference, please see the Javadoc documentation.

3.2. Extension Points

The following extension points can be used to extend the capabilities of the MConsole infrastructure.