Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

David Babka

Dynamic reconfiguration in SOFA 2 component system

Department of Distributed and Dependable Systems

Supervisor: RNDr. Tomáš Bureš, Ph.D.

Study program: Computer science

Specialization: Software systems

2011

I would like to thank my supervisor RNDr. Tomáš Bureš, Ph.D., for numerous pieces of advice, corrections, his patience and the time he spent with me. Furthermore I would like to thank RNDr. Michal Malohlava and Mgr. Petr Hošek for their help with initial understanding of the SOFA2 component system and RNDr. Vlastimil Babka for his thorough review of the thesis.

Contents

Název práce: *Dynamické rekonfigurace v komponentovém systému SOFA2*
Autor: *David Babka*
Katedra (ústav): *Katedra distribuovaných a spolehlivých systémů*
Vedoucí diplomové práce: *RNDr. Tomáš Bureš, Ph.D.*
e-mail vedoucího: *tomas.bures@d3s.mff.cuni.cz*

Abstrakt:
SOFA2 je komponentový systém založený na vytváření a uchovávání komponent v distribuovaném prostředí. Tento systém podporuje komponentovou hierarchii, což znamená, že lze několik komponent sloučit do jedné komponenty. Součástí tohoto systému jsou koncepty, které popisují návrh možných dynamických rekonfigurací architektury, které jsou nezbytné pro běh jakékoliv reálné složitější aplikace. Tyto koncepty jsou výjimečné hlavně tím, že návrh jednotlivých dynamických rekonfigurací je vytvářen už v době návrhu architektury aplikace a tedy se jimi běh aplikace musí řídit. Tyto dynamické rekonfigurace spočívají ve vytváření/ničení dynamických komponent a vytváření/ničení propojení mezi jednotlivými komponentami, díky čemuž se aplikace mohou lépe adaptovat nově vzniklým situacím. Cílem této teze je ověřit návrh těchto konceptů pomocí implementace prototypu pro SOFA2 a ověřit jejich korektnost a použitelnost na testovacích aplikacích.

Klíčová slova: *SOFA2, komponenty, distribuované systémy, dynamické rekonfigurace, factory pattern, dynamicita.*

Title: *Dynamic reconfiguration in SOFA 2 component system*
Author: *David Babka*
Department: *Department of Distributed and Dependable Systems*
Supervisor: *RNDr. Tomáš Bureš, Ph.D.*
Supervisor's e-mail address: *tomas.bures@d3s.mff.cuni.cz*

Abstract:
SOFA 2 is a component system employing hierarchically composed components in distributed environment. It contains concepts, which allow for specifying dynamic reconfigurations of component architectures at runtime, which is essential for virtually any real-life application. The dynamic reconfigurations comprise creating/disposing components and creating/disposing connections between components. In contrast to majority of component systems, SOFA 2 is able to specify possible architectural reconfigurations in the application architecture at design time. This allows SOFA 2 runtime to follow the dynamic behavior of the application and reflect the behavior in architectural reconfigurations. The goal of this thesis is to reify these concepts of dynamic reconfigurations in the implementation of SOFA 2 and demonstrate their usage on a demo application.

Keywords: *SOFA2, components, distributed systems, dynamic reconfigurations, factory pattern, dynamicity*

1 Introduction

Component based development has become a very valuable part of software engineering. It promotes the concept of separating business logic into larger units containing similar functionality. These units are called components and they act as a black box with well-specified interface to access the functionality.

Dynamic reconfiguration in a component based application is the ability to modify its architecture at runtime. This ability creates a powerful tool to adapt its form according to any events that may occur, making the application more flexible and efficient.

There are two major parts, which form dynamic reconfiguration. One of these parts is the instantiation of new components and their destruction. The other, nonetheless important part is the creation and destruction of connections between components. Both parts are responsible for the evolution of component architecture during run-time. Component reconfiguration modifies the architecture by extending and reducing it and connection reconfiguration changes the architecture internals. These actions can completely redefine component architecture and without proper constrains can uncontrollably evolve it into unwanted shapes with possible violations of the component hierarchy.

In order to track the architecture evolution, the components have to be separated into static and dynamic. Static components are created at the beginning of the application life-time and are destroyed at its end. Their existence is needed during the whole run-time and they could be marked as the “main pillars” of the component architecture. Dynamic components, on the other hand, are temporary and their creation and life-time depends on events that occurred during run-time. Similar separation could be used for component connections. Not all component connections need to be static and exist through the whole run-time; some can be created and destroyed only when needed. These separations are very valuable, because they reveal the core of the application and the possible evolution of its architecture.

The simple restriction of events, which can and cannot cause dynamic reconfigurations, makes the component architecture even more transparent. These limitations and specifications expose the complete application behavior and guarantee, that the component architecture will remain intact according to its design.

1.1 Motivation

The concept of dynamic reconfiguration is a known issue solved by many existing component based development systems. The innovation brought to this concept is the ability to determine architectural evolution at design time and the preservation of the hierarchy at the component connection level. Even though the creation of constrains seems like a trivial problem, in component systems this requires a lot of management, which is why most of the existing implementations or proposals of dynamic reconfiguration implementations do not support this feature. The few, which were dealing with this kind of issue, chose one of the three following options.

The first option [**1**] was the path chosen by the most, which is to completely disallow the dynamic reconfiguration. The implementations relied on the fact that any dynamic architecture can be transformed into a static one. However working with static architectures is very limiting and does not provide the expected usability.

In the second option, the dynamic reconfigurations are provided, but they do not support component hierarchy. Dynamic reconfigurations in flat component hierarchy are much simpler, because they do not need to solve issues such as where to place newly instantiated components or how to connect two distant components. On the other hand the evolution gap still remains and in larger component systems, where the number of components increases, the flat hierarchy makes the architecture more difficult to read. That is why it is better to have the ability to create composite components, which would divide the architecture into larger entities.

The third option is the one, which provides both the dynamic reconfiguration and multiple component hierarchy. The proposed or implemented solutions of this case are very few and since there is no best practice to solve this issue, they differ in many ways. In multiple hierarchy models there exist at least five different approaches to handle dynamic reconfigurations, but each of them is either incomplete (meaning it does not support full dynamic reconfiguration), unusable, or suffers from evolution gap.

The following list briefly describes several of the related approaches.

- Semi-permeable dynamic components [**2**], which does not allow dynamic components to provide any services to other components.
- The use of formal rules [**3**] to specify dynamic reconfiguration is very thorough and can describe any kind of reconfiguration. The disadvantage of this approach is that specifying even a small architecture without any reconfiguration is very laborious and requires a lot of effort.
- Flattening the hierarchy [**4**] during deployment takes advantage of simplicity of dynamic reconfiguration in flat hierarchy, but the flattening of the model creates even a bigger evolution gap during dynamic reconfigurations.
- Shared components [**5**] [**6**] is an approach, which allows any composite component to directly connect to a shared component regardless the hierarchical position of the composite component. This event hides away the information about the owner of the shared component, thus creating an evolution gap.
- Uncontrolled reconfigurations, which lets the architecture reconfigure anywhere anytime without any constraints, creating a huge evolution gap.

As can be seen from all the approaches, there are many completely different variants and each is focused on a different aspect of dynamic reconfiguration. However, only the formal rules based approach satisfies the condition of eliminating all the evolution gaps from the dynamic reconfiguration, but it is too complex even for a small architecture. That is why this thesis focuses on finding a simpler and more usable approach to dynamic reconfiguration without the evolution gap.

Since this thesis focuses mainly on the dynamic reconfiguration, it requires a component development system for the evaluation. For this purpose, the component development system SOFA2 [**7**], which provides many features like ADL-based design, component aspects, model driven design, hierarchical architectures, etc., was chosen. SOFA2 does not fully support dynamic reconfiguration, but the design of the component system provides simple extendibility and therefore became a perfect environment for reification of the concept.

1.2 Goals of the thesis

This thesis focuses on implementing a prototype of dynamic reconfigurations for SOFA2 component based system and demonstrating its usage on a demo application.

The dynamic reconfiguration should comprehend the specification of the component architecture evolution at design time and thereby eliminating the above mentioned evolution gap caused by dynamic reconfigurations. Dynamic components and connections must be created with respect to component hierarchy and there should not be any differences in support among dynamic and static components.

The demo application will focus on presenting the usage of the dynamic reconfiguration and depicting all its features and advantages.

1.3 Overview of the thesis

The second chapter of this thesis is devoted to describing the backgrounds of the SOFA2 component system and the concept of the dynamic reconfiguration patterns required for the prototype. The third chapter is analyzing all the missing gaps of the dynamic reconfigurations, which are required for the implementation. The solution proposal, which is built upon the previous chapters, is specified in the fourth chapter. The fifth chapter is focused on the prototype implementations issues and solutions and is followed by the evaluation in the sixth chapter. The seventh chapter serves as a user guide describing how to invoke the dynamic reconfigurations in the applications using the prototype implementation. In the eighth chapter is detailed description of other solution proposals or implementations of the dynamic reconfiguration supplemented by the comparison with the thesis goals. Finally, the chapter nine concludes the thesis.

2 Background

2.1 SOFA2 component system

Since this thesis is not a stand-alone application, but rather the extension to already existing component based system called SOFA2, the relevant details of SOFA2 have to be described first. This chapter is focused on the explanation of the SOFA2 functionality required for the reconfiguration pattern and for the understanding of the thesis goals. The complete programmers guide and user guide of SOFA2 can be found in Appendix A and B on the enclosed CD.

2.1.1 Basics

The SOFA2 component system allows to build and run large distributed projects developed in Java. The distribution of a project is handled by the deployment docks, which serve as a host to components. The deployment docks can be initialized all over the network, but they require the same registry server and repository server in order to host the whole component application. The repository server contains all the data and meta-data of the application and the model required for the application startup and runtime. The registry server registers each deployment dock of the application in order to ease the communication between the deployment docks.

The deployment dock also plays an important role as the initiator of the component application. When triggered it recursively creates components of the application and, using the connector management classes, initializes their connections. When creating a composite component in distributed application, it may occur that the sub-components are assigned to different docks then the parent component. For this purpose the deployment dock firstly needs to check where to place the sub-components and then leave the responsibility of their creation to the other docks. Since the deployment dock is also the only entity, which can manage the components, it needs to remember all the necessary information about the components it contains in order to provide them to anyone who may ask.

The responsibility for the connections is part of the connection management classes, since the logic is too complex and the deployment dock is already overwhelmed with the component management logic. The connection management is handled by four classes. The first, `ConnectionService` is responsible for generating code for all connectors, which may be required for the application. The other three classes are used for connecting the connectors together and managing the connections, but each provides the management in different hierarchy level. Each class and the level it belongs to follows:

- `ConnectorsManager` – manages connectors only for components.
- `DockConnectorManager` – manages connectors and connections for the whole deployment dock.

- `GlobalConnectorManager` – manages connectors and connections for the whole application.

Even though the responsibility of these classes is separated into different levels, they all are dependent to each other. They only provide more specific logic for the given level.

2.1.2 Model

The model in SOFA2 component system is a set of classes, which completely define the structure of a component and its communication. For the implementation of the model the Eclipse Modeling Framework (EMF) was used [**8**], which allows the initialization of each class by well-defined XML specifications. These specifications have a great influence on the structure of the Architecture Description Language (ADL), because the ADL is nothing more than a simplified specification of the EMF model.

Communication

The communication between components is the most basic and most important part of the model. Without the component communication, the component based system would be useless. Since the rest of the model is influenced by the communication as well, it is essential to describe the form of the communication first.

The communication in SOFA2 component system is client/server based, which means that in each connection, which provides the communication, there must be one server component and at least one client component. The server component has to provide some service to the client component and that is why the connection endpoint leading from the server component is called business provided. Since the other endpoint requires the service for the component's functionality, it is called business required.

The services provided from the server component to the client component are defined by an interface. The interface specifies methods, which can be used by the client component and implemented by the server component. The interface has to be specified by each connection endpoint. This limits the creation of the connections, because the connection can be realized only when all the endpoints of the connection share the same interface and just one of them is business provided.

Since the connection "one to one" does not cover all the communication cases, the SOFA2 also supports the "one to many", "many to one" and "many to many".

The "one to many" case means that one component provides the service and many components are connected. This case is already resolved, because the number of client components is not limited and all can connect to the server component.

The case "many to one" occurs when many server components provide the implementation of a single interface to one client component. This case cannot be handled by one client endpoint, because each server component endpoint requires own connection. For this reason SOFA2 supports collection endpoints, where there can be created multiple client connectors of the same interface and each can be connected to different server endpoint. Since the mark of the collection endpoint will be used further in the thesis it is depicted on Figure 2.1 along with the “one to one” case for comparison.

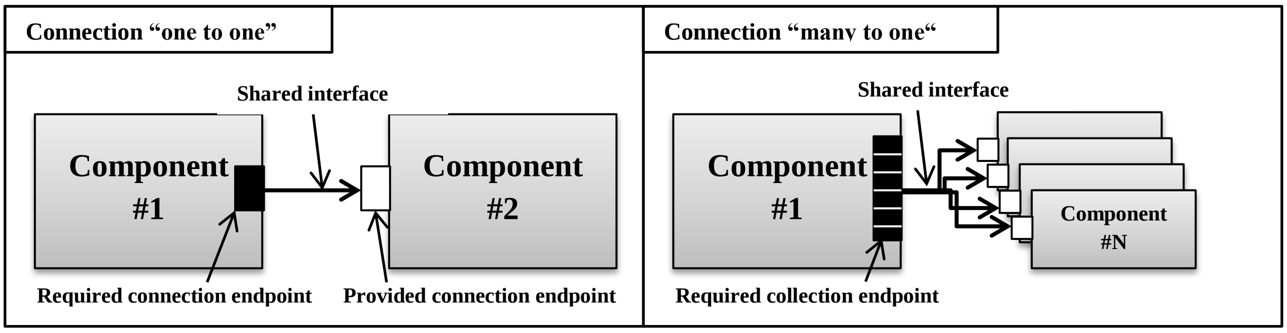

Figure 2.1 - Connection endpoint types

The last case “many to many” is supported by combining “many to one” and “one to many” connections and therefore does not have to be handled separately.

The communication in SOFA2 also specifies two types of connectors. So far there is used only the first basic type. It is created for each static connection and it is bound to deployment dock, component and interface. The binding ensures that the connector cannot be used by any other component or connection. The second type called shared connector is a preparation for not yet implemented dynamic reconfigurations. It is also bound to an interface and deployment dock, but it does not use component binding. Thanks to that, this connector can be freely used by any component from the deployment dock, which contains a frame with the specified interface.

Component structure

The structure of a component consists of a frame and architecture. The frame specifies all the connector interfaces, which are provided or required by the component and the architecture specifies the contents of the component.

Because SOFA2 defines that a composite components consists of a set of components connected together and only the primitive components are allowed to contain implementation, the architecture for these two types differs.

The architecture of a primitive component is very simple. It contains the information about the name of the architecture and the frame of the component, which is common to both types. However, in addition it specifies the path to the implementation of the component, which is omitted form the composite components.

The composite component architecture needs to specify all the sub-components it contains including their names, architectures, frames and the connections between them. The specification of the connection is done using the endpoints, which contains the information about the component name and the connector interface. For basic connection there have to be specified at least two endpoints, one for the server component and the rest for the client components. However, the connection may also specify only one endpoint. This case is for delegating connections from the sub-components to the composite component or for the subsuming connections from the composite component to the sub-components. These types of connections can be done only if the frame of the composite component specifies the same business required connector interface for the subsumption or business provided connector interface for the delegation. Both of these types are depicted on Figure 2.2.

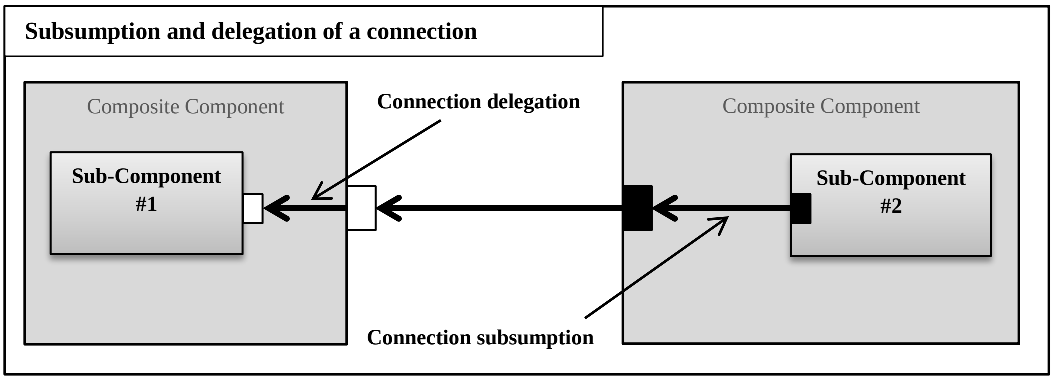

Figure 2.2 - Subsumption and delegation of a connection

Assembly

The assembly is used to describe the whole tree of components in a SOFA2 component application. It specifies the root component and all its sub-components it contains and the dynamic components it may create. All the composite sub-components and dynamic components have to specify their sub-components and dynamic components as well thus creating a tree-like structure. The assembly also contains the information about the frame and architecture of each component and is very useful to determine correct component creation order when the application starts.

Deployment plan

The deployment plan servers the purpose of assigning to each component of the SOFA2 application the proper deployment dock. The deployment plan has a tree like structure of components, sub-components and dynamic components, which makes it very similar to the assembly.

2.1.3 Aspects

The SOFA2 component system uses aspects as a way for extending its core logic. The aspects and all entities mentioned in this chapter are also part of the model, but due to the importance of the aspects for this thesis they are described in a special section.

SOFA2 contains special entities capable of interfering with the basic application execution. Aspects use these entities to apply additional logic of the extension right where they are needed. There are two types of these entities, micro-components and interceptors. Micro-components are capable of inserting their logic directly into the component, while an interceptor can intercept connection between components and modify it. Thanks to these entities there do not have to be done any core changes to the model in order to provide an extension.

The application of additional logic to the model is just a part of extension requirements. Since an extension may provide multiple functionalities and each may be required for different subset of components or connections, SOFA2 implements three additional entities to provide the specific selections. First, we describe annotations, because they are used also by the other two entities.

Annotations are used as a mark for an aspect to determine where to apply its logic. Each implementation of the annotation can contain additional data, which may be required by the extension for proper execution of its logic. The annotation can be applied only to frame and the connector interface, thus marking only the component or connection, which can be used for the interference with basic functionality.

The responsibility for the right selection of components and connections is left to the other two entities. The first entity is called `ComponentSelect` and can specify multiple instances of the second entity named `InterfaceSelect`. Both of these entities use special string queries to filter only the components or connections required for the application of the extension. The queries are separated into two parts. The first part filters by the general properties and the second part by name.

The general properties of the component select are:

- All – selects all components.
- Primitive – selects all components, which do not have any sub-components.
- Composite – selects all components, which contain sub-components
- Other – special component selectors implemented for SOFA2 extensions.

The general properties of the interface select are:

- All – selects all the connectors.
- Control – selects only connectors for micro-components.
- Business – selects all non-control connectors.
- Business provided – selects all server connectors of the component.
- Business required – selects all client connectors of the component.
- Other – special interface selectors implemented for the SOFA2 extensions.

The implementation of the special selector for the extension requires uniquely naming the selector and adding a condition, which determines if the component or connection is part of the selection. Usually the condition lies only in finding the required annotation on the frame or connector interface.

When the final selection is finished, the aspect inserts interceptors or micro-components to the result and thus providing the logic only to places, where it is required.

2.1.4 Interceptors

Interceptors are classes placed into the connection to enrich the communication with their own logic. To be more precise the component connection consists of multiple interceptors of the client component and multiple interceptors of the server component. Between the interceptors of the client and server component a middleware is placed, which is responsible for the distributed transfer of the method call. The collection of interceptors before and after the middleware is called delegation chain.

The delegation chains have to delegate the method call from the client component to the server component. In order to do that, they have to implement the interface of the connection. Since manual implementation of each interceptor for each connection would be a very large effort, the interceptors have to be generated or replaced with a proxy capable of react to any method, which may be invoked. Even though it is much more complicated, the more preferred way is to generate the interceptors using Java byte code. The proxy requires the use of reflection on each method call, which makes it much slower than the generation of the interceptor, which has to be done only once for each interface.

The delegation chain also specifies the entry point and the endpoint. The entry point of the client component is the object implementing the connector interface and the endpoint is the connector. The server component has this somewhat inverted, since its entry point is the connector and the endpoint is the object responsible for the provided logic. When the client calls a method from the connection, it actually calls the method of the first interceptor in the chain. The interceptor executes all the logic placed before the delegation of the method call and afterwards it delegates the call to the interceptor next in the delegation chain. When all the interceptors in the interceptor chain have finished delegating, the middleware takes over the responsibility and transfers the call to the delegation chain of the server components. The server components chain repeats the same procedure, until the call reaches the server component, which will finally execute the required method. After the execution of the method finishes, the interceptors continue to execute all the logic after the delegation. The reason this is described in such detail is to emphasize the execution direction of the interceptors, which differs before and after the delegation of the method call in the interceptor. The logic before the method call delegation is executed from client component to server component and the rest is executed in opposite direction. The directions and the communication structure with interceptors are depicted on Figure 2.3.

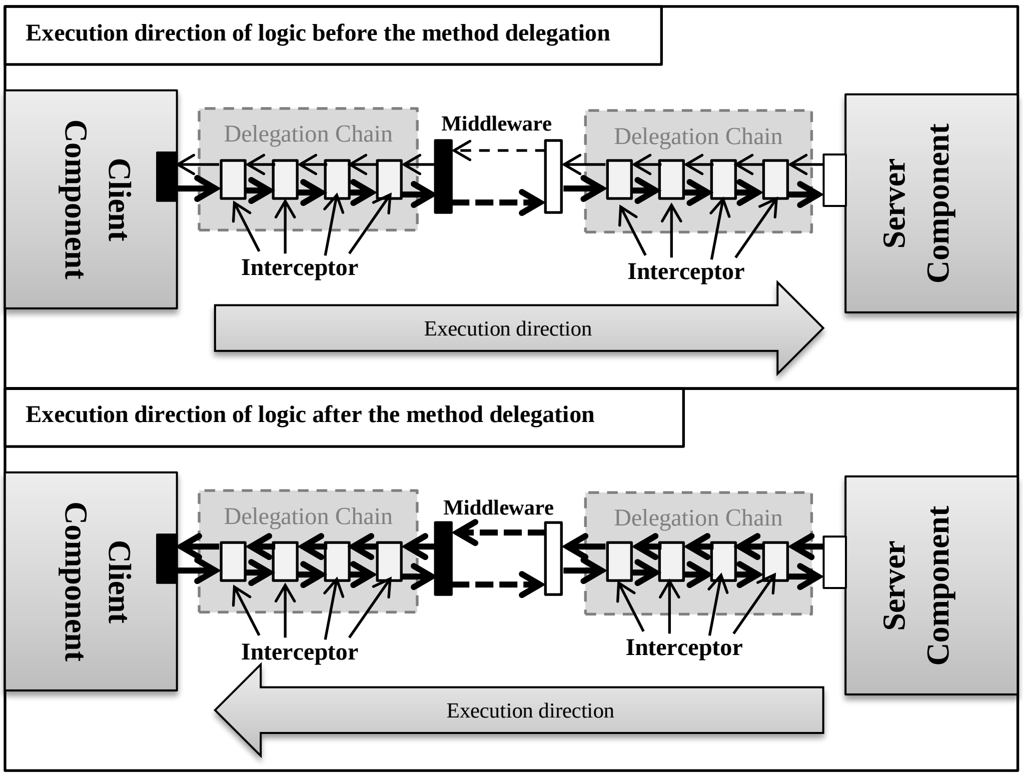

Figure 2.3 - The communication with interceptors

Until now we have only described delegation chains for a basic component to component connection. Since composite components use also delegation and subsumption connections, they must be specified as well.

When the delegation and subsumption types were described, there were also mentioned, that their connection requires specification of only one endpoint and that the frame of the composite component must specify the same connector as the delegating or subsuming component. Even though the specification of the second connector endpoint for the composite component does not exist, the connector is still internally generated in order to realize the connection. However, the connection is incomplete. The missing specification causes that the connection is without the delegation chain at the composite component side as can be seen on Figure 2.4. This is important when working with dependent interceptors and each is on different side of the connection. In component to component connections this would not be such an issue, but in delegation and subsumption connection, there will always be one of the complement interceptors missing and therefore the actions have to be handled differently.

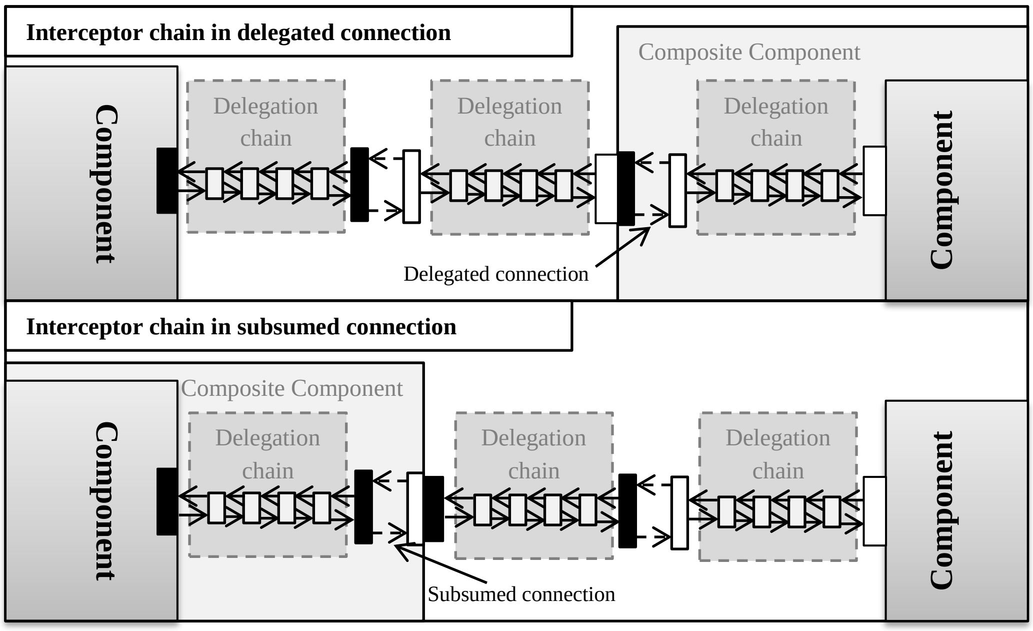

Figure 2.4 - Interceptor chains in delegated and subsumed connection

2.1.5 Micro-components

A micro-component in SOFA2 can be imagined as a small primitive components providing additional functionality to any component it is connected to. The connection between the component and micro-component is not direct. It is done through a base micro-component, which is part of each component and allows the access to the basic information and content of the component. The basic information contains for example the component's id, the frame and the delegation chains of the component. The content allows the access to the main class of the component, which can be modified using reflection in order to provide the extended logic.

The creation of micro-components occurs right after the component is created. For this purpose, all the aspects available for the created component are traversed and according to each aspect the micro-component and their connections are formed. The micro-component connections are called control and their creation is done using methods specified by the `SOFAMicroComponent` interface, which has to be implemented by every micro-component. This interface also specifies the initialization method, which is executed right after all the micro-components and their connections have been formed, and is used among other things to verify that all the required connections have been set. Since the micro-components can provide their logic to many other components, the micro-components can form a topologic structure. The example of the connected micro-components is depicted on Figure 2.5.

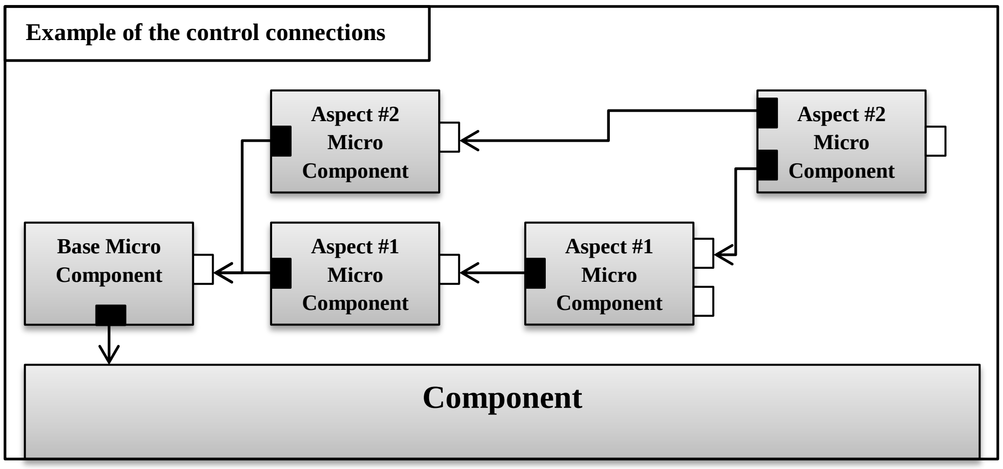
Example of the control connections
Aspect #2 Micro Component
Aspect #2 Micro Component
Base Micro Component
Aspect #1 Micro Component
Aspect #1 Micro Component
Component

Figure 2.5 - Example of the control connections

2.2 The dynamic reconfiguration proposal for SOFA2

The described dynamic reconfigurations in this chapter are predominantly based on the dynamic reconfiguration concept, which was a part of the SOFA2 specification. This concept is simple to use and eliminates all the evolution gaps in the dynamic reconfigurations, which are the main goals of the thesis. Since the concept has not yet been implemented, this thesis focuses on its implementation and evaluation.

2.2.1 Factory pattern

The factory pattern in a component based system has to provide to selected component the ability to create new dynamic components. The frame of each `factory` component needs to be marked with the `factory` annotation, which must also contain the information about the component, that may be created and the interface responsible for connecting the dynamic component. Thanks to the annotation the components will contain the additional functionality to create the dynamic components and the annotation will also serve as a sign of specific dynamic reconfiguration in the architecture, which is important for eliminating the evolution gaps.

The additional functionality should support two types of dynamic component creation. Even though the concept specifies only the creation of the dynamic component by the factory method, this thesis specifies one additional type.

The additional type of dynamic components creation is the possibility to create the dynamic component for the internal purposes of the `factory` component. This type can be seen on Figure 2.6 and is useful for example when the dynamic component has to be firstly initialized in order to be used. Since the responsibility for the initialization may be left on the `factory` component, there has to be established a connection to the dynamic component to access the initialization methods. When initialized, the `factory` component may distribute only the connection for the usage of the initialized dynamic component.

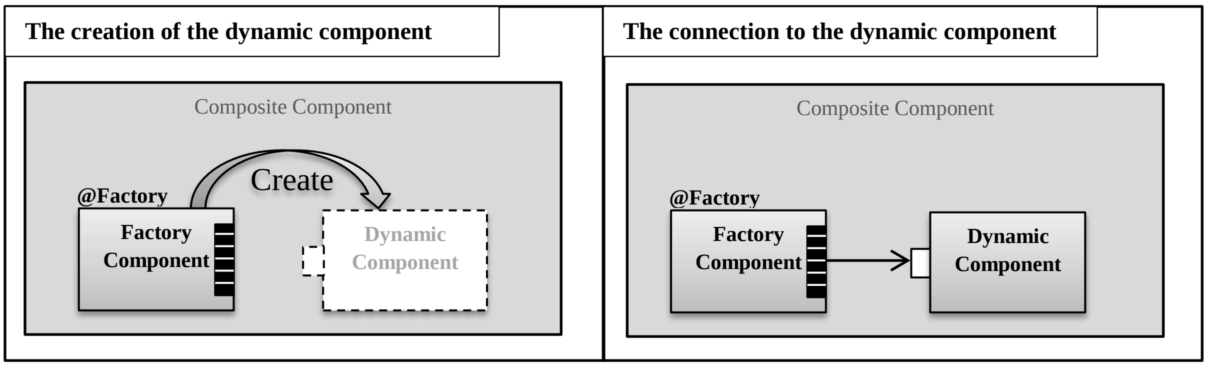

Figure 2.6 - The creation of dynamic component for internal purposes

The creation of the dynamic component using the factory method is the case, when the `factory` component creates the dynamic component only based on the method call and returns the connection to the dynamic component to the caller. The `factory` component takes only the role of the creator and therefore the connection from the `factory` component to the dynamic component will not be needed. Because of that the `factory` component does not even have to contain the business required collection connector for the dynamic components connection. However, all the components, which are connected to the `factory` component and can use the factory method, have to contain the mentioned connector. Moreover the connection to the `factory` component must be annotated with the same annotation, which was used to create the dynamic component, in order to complete the dynamic component connection.

One of the other responsibilities of the factory pattern is to determine the hierarchical position of the created dynamic components. In the case of creation of the dynamic component for internal purposes the decision is simple, since the component, which requested the creation, is also the `factory` component responsible for the creation. In this case the dynamic component is always placed as a sibling to the `factory` component. The second case is more complicated since the dynamic component can be placed also as a sibling to the component, which requested the components creation, as can be seen on Figure 2.7. The concept of dynamic reconfiguration suggests that the dynamic component should be placed as a sibling to the requesting component. This thesis also agrees with the proposed solution for the following reasons.

- The requesting component will be the first, who will initialize the connection to the dynamic component.
- The connection between the requesting component and the dynamic component is direct. If the dynamic component would be placed next to the `factory` component and the source of the request would not share the same parent as the `factory` component, the connection would require complicated redirecting through the composite components. This case is depicted along with the proposed solution on Figure 2.8.
- The `factory` component will not be over flooded with dynamic component siblings, because the dynamic components will be spread out to the requestors.

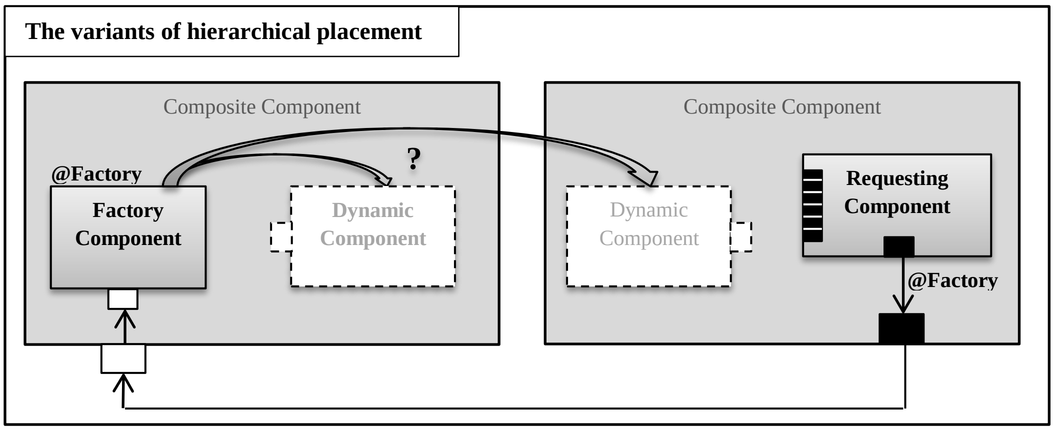

Figure 2.7 - The possible variants of the dynamic component hierarchical placement

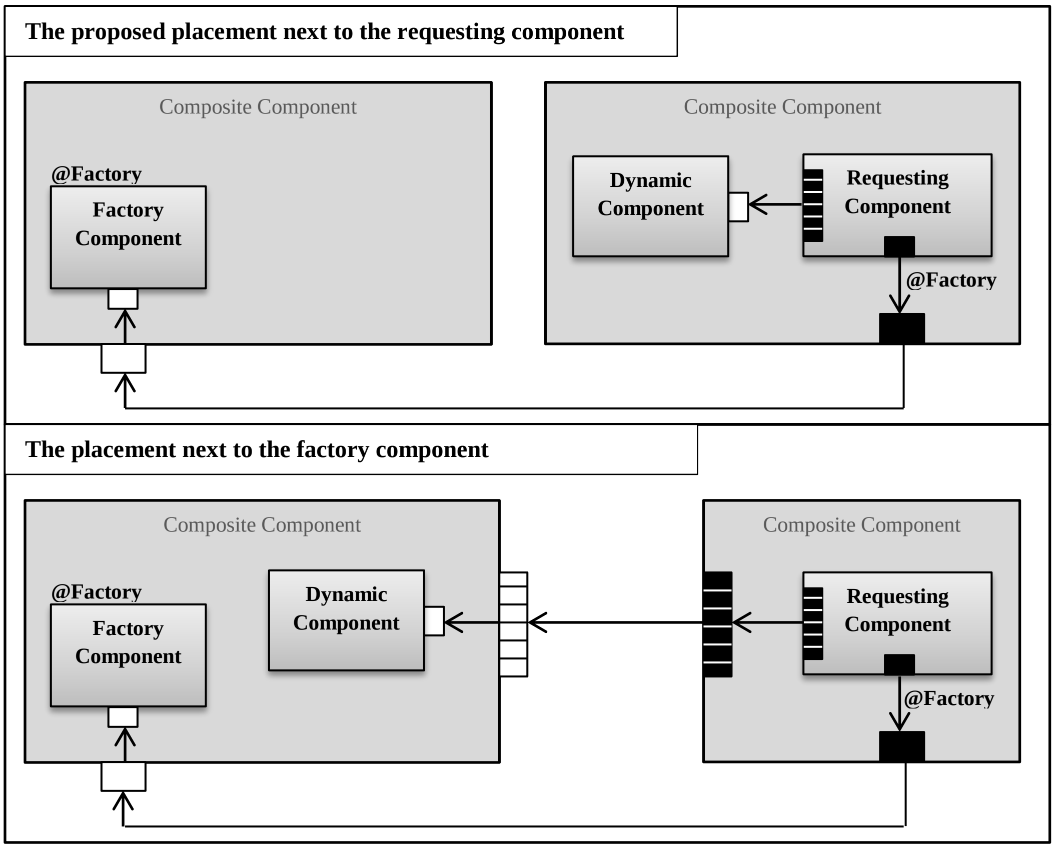

Figure 2.8 - The connection differences for the possible variants of hierarchical placement

2.2.2 Component destruction

The component destruction dynamic reconfiguration is very straightforward and therefore does not have to be described in such a detail. The logic of the functionality provision to the component is similar to factory pattern, but instead of the `factory` annotation, the `self-shutting` annotation is used. Components marked with this annotation will have access to the additional logic, which allows the component to destroy itself.

2.2.3 Dynamic connection reconfigurations

The dynamic connection reconfigurations in component based systems allow annotated connections the possibility to initialize or destroy connections between components. The annotations serve two purposes. The first is to mark the event responsible for dynamic reconfiguration and the second is to provide the data needed for the dynamic reconfiguration.

The create annotation

The dynamic reconfiguration concept specifies that the dynamic connection is realized by passing the provided object as a method parameter of component connection. When the method is called, from the provided object is created a business provided connector, which may be used to initialize the connection to any component, who may accept it.

The link annotation

The connector created from the provided object is just the first part of the dynamic connection creation. The parameter passed by the method to other component is useless unless it is affected by the annotation called `link`. The `link` annotation uses the parameter to connect its component to the business provided connector.

The unlink annotation

The exact opposite of the `link` is the `unlink` annotation, which is responsible for disconnecting the dynamic connection from the component. This connection can be also bound to a parameter and its advantage is, that any component, which obtained the unlinked parameter should still be able to link it.

The other possibility of this annotation is the method annotation, which can use only the methods of the dynamic connections. The result of this annotation variant is the same, except the unlinked connection is the same connection, where the annotation is placed.

The destroy annotation

The last connector interface annotation of the dynamic connection reconfiguration is called `destroy`. This annotation serves as the disposer of the dynamic business provided connector, created by the `create` annotation. Because of that, the `destroy` annotation can be only part of the `create` annotated component.

This annotation can also be specified in two variants. The first is bound to a method parameter and the second is bound to a method. The method parameter type is no different from the other. It expects that the contents of the parameter are any of the unlinked objects created by its component. The behavior of the second annotation is the destruction of the business provided connector, which is used for the current connection.

Dynamic connection reconfiguration in complex hierarchy

As was mentioned in the thesis goals, the dynamic connections have to respect the component hierarchy. In factory pattern this issue did not have to be handled, since the dynamic component will always be connected directly to the requesting component, but the dynamic connections have to support the connection between any two components regardless of their hierarchy level.

The dynamic reconfiguration concept suggests simple solution which is depicted on Figure 2.9. The solution specifies that the composite component needs to contain the same prerequisites as the sub-component, which delegates or subsumes the connection with the dynamic connection reconfiguration. This includes the collection connectors and the annotations. However, when there are more sub-components providing the same dynamic connection reconfiguration and the composite component have to delegate or subsume them all, than the composite component may stack them into one. This case can be seen on Figure 2.10.

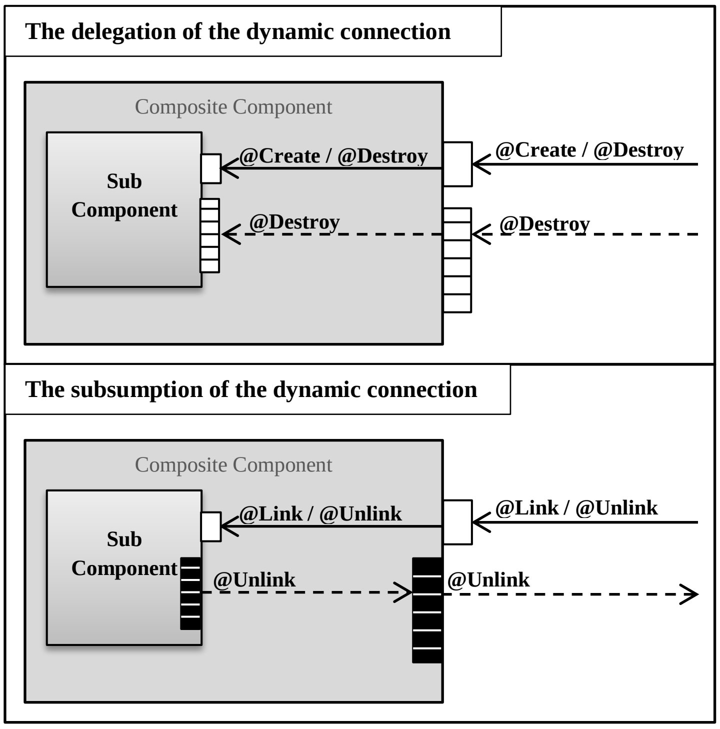

Figure 2.9 - The use of dynamic connection annotations in complex hierarchy

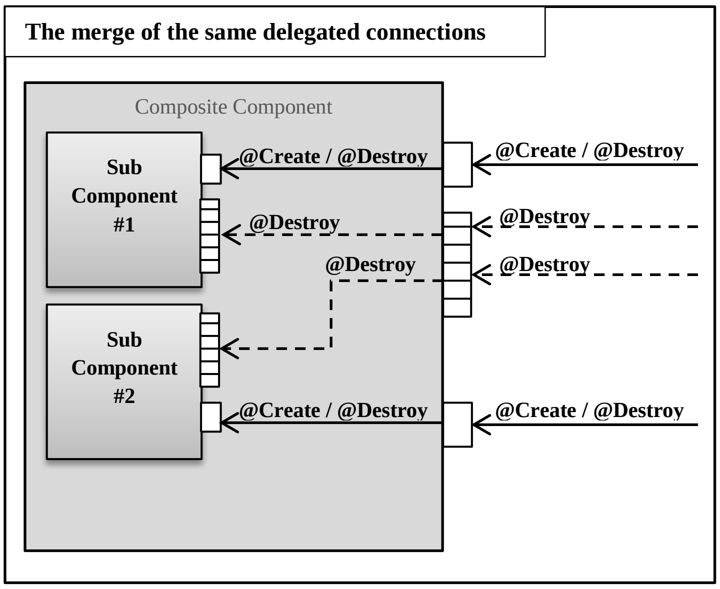

Figure 2.10 – The merge of the same dynamic connections through the composite component

2.3 Goals revisited

The goal of this thesis is to implement the aspects for SOFA2 component system according to the proposed dynamic reconfiguration patterns described in section 2.2 and evaluate them on a demo application.

3 Analysis

The whole analysis is based on the dynamic reconfigurations concept described in the background chapter. The purpose of this chapter is to fill the missing gaps, which are not part of the concept and their analysis is required for the implementation.

3.1 Factory pattern

The description of the factory pattern in the background chapter is very thorough. However, it omits details of a few procedures, which are essential for the implementation. The first of them is the dynamic component creation process. The second part is the process of realization connection between the dynamic component and the requesting component. The last process is the placement the dynamic component to the hierarchy.

3.1.1 Creation of the dynamic component

Since the dynamic component and the static component should not differ, the best way to create the dynamic component is by reusing the logic for the creation of the static component. This logic is part of the deployment dock, which is not accessible from the inside of the component. However, the provision of extended functionality directly to the component can be done by micro-components, which have the ability to access the deployment dock and with the use of the `factory` annotation data, they can gather all required parameters for the component creation.

3.1.2 Connection to the dynamic component

The dynamic reconfiguration concept for the factory pattern specifies, that the `factory` component will create the dynamic component and use its business provided connector for either the factory method or the realization of the connection to itself. The retrieval of the dynamic component connector according to the specified interface is simple and can be done when the dynamic component is created. The second connector required for the connection is shared and therefore has to be available at the deployment dock of the component responsible for the realization of the connection. This observation is very important, because the shared connector has to be firstly generated and placed into the deployment dock, which cannot be done without the knowledge of the components responsible for the connection realization.

Determining the responsibility of the connection realization in the case when the `factory` component creates the dynamic component and connects it to itself is simple. Since the connector does not leave the `factory` component through the whole process, the only available component, which may be responsible, is the `factory` component.

The use of the factory method makes it little bit complicated, because there are two possible variants. Since the design proposal did not specify, what should be sent through the method as a parameter, several variants are possible. The responsibility for the connection realization changes depending on the selected variant.

The first variant suggests that the dynamic components connector information will be sent through the factory method and the connection will be realized at the component, which requested the creation of the dynamic component. For this purpose, there would be required an interceptor capable of using the output parameter of the factory method to connect it to the complementary shared connector and then returning the connection delegation chains entry point instead. Since this type of interceptor will be used further in the thesis with different context it was given the name `output parameter modifier interceptor`.

The idea of the second variant is to connect the dynamic component to the shared connector as a part of the dynamic components creation, which leaves the responsibility for the connection realization to the `factory` component. This way the connection will be already initialized when sending the information about the shared connector through the factory method. The creation of the delegation chain with the shared connector as its endpoint will then be up to the output parameter modifier interceptor, which would only replace it with the delegation chains entry point and thus finishing the process.

Both variants are correct and usable. However, the second variant is more preferred, because all the shared connectors for the `factory` component will be placed into one deployment dock. Since the component application may be distributed and there may be more components using the factory method, then for each component would have to be generated the same shared connector to their deployment dock. Moreover the similar procedure will be used for the dynamic connection reconfigurations and the repeated use of the `link` and `unlink` may be very common therefore the simpler case of reconnecting is more welcomed.

3.1.3 Placement of the dynamic component to the hierarchy

The main issue associated with placement of the dynamic component to the hierarchy is how to determine when it happens. There are two possible answers for that question and both raise more and more related questions.

The first obvious suggestion is to place the dynamic component into the hierarchy during its creation. However, this solution would require the knowledge of the hierarchical placement of the requesting component and obtaining this kind of information is not supported by SOFA2. The additional implementation of this support would require either to send the information about the component along with the connection method call or to somehow remember all the business required connectors used for each business provided connector. Both of these proposals are very complicated, since the first would require modifying the generation of the connectors in order to extend the methods by another parameter and the second would work only for one client component.

The second suggestion is to wait until the dynamic component is created and then place it to the hierarchy as a part of the connection realization by the output parameter modifier interceptor. Since the component responsible for the realization of the connection is the requesting component and the dynamic component has to be placed right next to it, the future dynamic component position is known. The more pressing issue is to determine how to access the dynamic component and change its hierarchy, since the only part of the dynamic component available to the requesting component is the shared connector. The only way to make the dynamic component accessible is to somehow obtain at least the information about the deployment dock of the dynamic component and the component's id. This can be done either by extending the shared connector or extend the `GlobalConnectorManager` of additional API capable of providing this sort of information from any connector. Since the `GlobalConnectorManager` has to manage data about every connector and connection in the application, it might be better suited for this purpose than the shared connector, which does not know the destination of its connection until it is connected.

Another issue which must be resolved for this suggestion is where to put the dynamic component before it is placed in the hierarchy. Moreover what will be required in order to modify the hierarchy placement.

The first answer is simple. The temporary placement will be guided by the deployment plan and the assembly, where the dynamic component is specified under the `factory` component. Since the `factory` component has to be primitive, there will be no problem using it as a parent since it may contain only the dynamic components.

The placement of the dynamic component itself will require identifying the parent of the requesting component and adding the dynamic component to it as another child. Since the parent of the dynamic component is still set to the `factory` component, it must be changed to the requesting component's parent to avoid inconsistency. This also requires the change of the dynamic components hierarchical name according to the new parent and the change of the hierarchical name of all the components, which are part of the dynamic component sub-tree.

3.2 Dynamic connection reconfigurations

The proposal of the dynamic connection reconfiguration is not complete as well. It contains few unclear parts, which must be analyzed before the implementation can begin.

The most important part, which requires further analysis, is the identification of all the possible variants of the dynamic connection reconfiguration and determination of all the types of interceptors required for each variant.

In the end, there will be required more specific analysis of the interceptor logic for each annotation, since the concept specifies only the result of the annotations.

3.2.1 Interceptor logic

Since all the interceptors required for the reconfigurations are used only for the modification of connections between components, the ideal entity, which would handle this kind of responsibility, would be the `ConnectorsManager` or the `DockConnectorManager`. Since neither of them is freely accessible from the interceptor, the deployment dock has to be used as the delegate of the required actions.

Create interceptor for primitive components

The logic of the `create` interceptor is somewhat similar to the `factory` micro-component, except it does not create the dynamic component. Its purpose is to use the object specified by the `create` annotation and the business provided shared connector of the same interface and build a delegation chain from them. Since the direction of the method calls of the connection will lead from the connector to the provided object, the connector will be the entry point and the object will be the endpoint of the delegation chain.

The next step, which specifies the connection realization, was already analyzed by the `factory` component and therefore it does not have to be solved. There will be done the same procedure, which will create both business required and provided shared connectors and connect them together. The information about the business required connector will then be sent to the component instead of the provided object. The last requirement, which should not be omitted, is the generation of the shared connectors for the deployment dock. Since the creation needs to use both business provided and required shared connectors, both of them has to be generated by the `ConnectorDeploymentService`.

Link interceptor for primitive components

The logic of the `link` interceptor is exactly the same as the `factory` interceptors, except it does not have to perform the hierarchy modification. It only has to create a delegation chain from the business required shared connector and provide the entry point of the chain to the component.

Create interceptor for composite components

As was mentioned before in the background chapter, the connections in the complex hierarchy slightly differ from the basic connections. The first thing that needs to be recalled about the delegated and the subsumed connections requirements is that they need to follow some prerequisites. These prerequisites dictate that both the composite component and the sub-component will contain the same `create` annotation on the connectors. The second thing that needs to be recalled is, that the composite component does not have the delegation chain when delegating or subsuming. Because of that there will be repeated two `create` annotations in a row without any link. The example of the `create` annotation in complex hierarchy can be seen in Figure 3.1. However, this case does not have to be solved at all since the `link` annotation only creates delegation chain, which is absent in the delegation connection. The `create` interceptor at the composite component will therefore only use the business required connector as the provided object of its own `create` annotation.

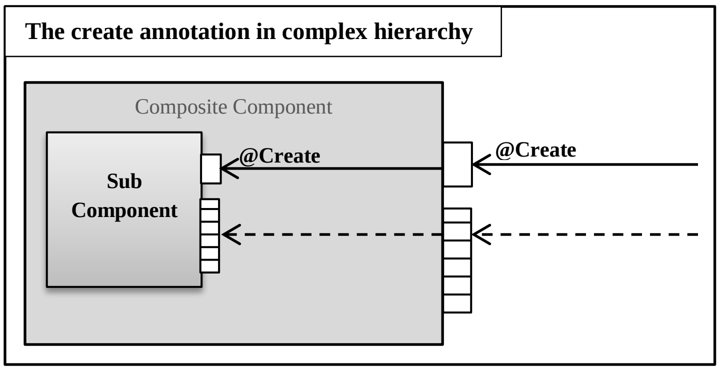

Figure 3.1 - The create annotation in complex hierarchy

Link interceptor for composite components

The `link` interceptor for composite components will have to solve the same issue as the `create` interceptor, except the delegation chain will omitted for the connector creation. Because of that, the same `link` from the primitive components, which would realize the connection to the composite component, has to be executed first. Then the resulting connector should be used to subsume the connection to the subcomponent.

Since the creation of the subsumption dynamic connection should follow the standards created for the basic dynamic connection, it may use the logic of the `create` interceptor. However, because of the missing delegation chain the logic of the `create` interceptor has to be slightly modified and the creation of the delegation skipped. The rest of the `create` logic will stay the same.

The last thing that should not be forgotten is the `ConnectorDeploymentService` and the generation of the shared connectors, since the `link` interceptor in composite components uses the logic of the `create` interceptor and therefore requires the shared connectors.

Unlink interceptor for primitive components

The responsibility of the `unlink` interceptor is to disconnect the delegation chain of the dynamic connection. In order to do that the delegation chain firstly has to be found. Since the only information provided for the identification of the delegation chain are either the id of the connector for the annotated dynamic connections or the delegation chains entry point for the specification of the parameter. However, neither of them provides any information leading to the delegation chain.

The solution for this issue would be the integration of the mapping to the `ConnectorsManager`. The mapping would use the id of the connector or the delegation chains entry point as a key leading to the delegation chain itself and it would have to be used when linking the connection to the component, which includes both the `factory` and the `link` interceptors.

Destroy interceptor for primitive components

The `destroy` interceptor is responsible for disconnecting both the connection and the delegation chain. The connection is disconnected through the `DockConnectorManager`, but the disconnection of the delegation chain suffers from the similar issue like the `unlink` interceptor. The SOFA2 does not provide any support to obtain the delegation chain attached to the business provided connector by using only the business required shared connector, therefore a solution has to be implemented.

Since the issues of the `destroy` and `unlink` interceptors are similar, so are their solutions. The integration of the additional mapping to the `ConnectorsManager`, which would map the business required shared connector to the delegation chain from its complementary connector endpoint, would handle the job. The only thing left is to apply the mapping when the `create` annotated interceptors are used.

Unlink interceptor for composite components

The `unlink` interceptor for composite component has to solve the same issue like the other dynamic connection reconfiguration interceptors and therefore they need to fill in for the `destroy` interceptor. Because of that, the analysis of this interceptor was delayed until both the `unlink` and `destroy` interceptors for primitive components have been described.

Since this interceptor needs to further delegate the `unlink` action, it firstly needs to destroy its own connection. The delegation chain for the destruction will not exist and therefore this step will be skipped. The process will continue with using the `ConnectorsManager` mapping to retrieve the provided object, which will be the business required shared connector. The final step would be using the full `unlink` interceptor logic on the required shared connector, which will complete the unlinking process.

Destroy interceptor for composite components

The `unlink` and the `destroy` interceptors for composite components are quite similar. The only difference is the delegation chain. For the `unlink` interceptor the delegation chain was missing for the destruction of the connection and the `destroy` interceptor will not have the delegation chain for the `unlink`. The process will therefore begin with using the destruction interceptor logic followed by the use of the mapping to obtain the business provided shared connector. The resulting connector information will be then used as a replacement of the parameter.

3.2.2 Reconfiguration variants

The variants of the dynamic connection reconfigurations are limited and all of them have to be analyzed in order to gather all the necessary interceptor types.

Reconfiguration through the output parameter

One of the first basic variants is the use of the output parameter for the dynamic reconfiguration. For this purpose we can use the already defined output parameter modifier interceptor, which applies the logic on the selected parameter and replaces it with any other object specified.

The modification of the output parameter can be used by any annotation of the dynamic connection reconfiguration. However, annotating the output parameter does not need to have always meaning. For example the `create` annotation using the output parameter cannot be used on business required connector, because the information about the created connector needs to be passed through the output parameter to the other component. The same goes for the `link` interceptor, which cannot be placed on the business required connector when modifying the output parameter, because it is used to provide the connection to the component and not to throw it away. To speed things up, all the correct usages of the dynamic reconfiguration using the output parameter are depicted on Figure 3.2.

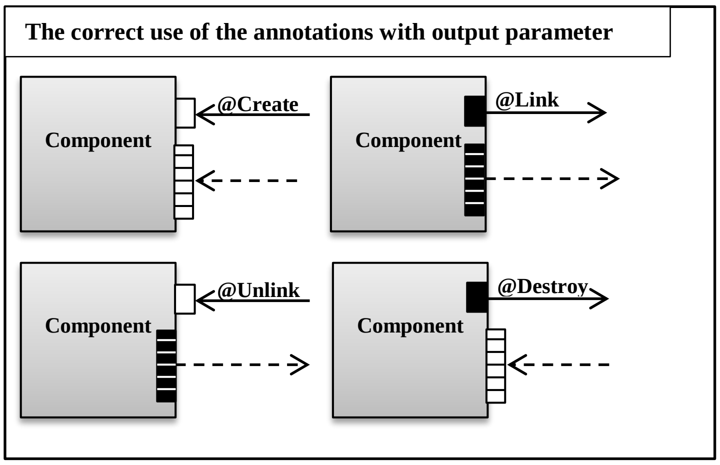

Figure 3.2 - The correct use of the annotations with output parameter

Reconfiguration through the input parameter

The second most basic variant available to all the dynamic connection reconfiguration annotations is the use of the input parameter. This variant also requires own interceptor type capable of applying its logic to any of the specified methods input parameters and replacing it with something else. The name given to this type of interceptor was the input parameter modifier interceptor.

The restrictions for the use of this interceptor type with the annotations are inverted to the use of the output parameter modifier type, since the direction of interceptor execution is opposite. This is caused, as it was mentioned in the backgrounds chapter, by applying the logic before delegating the method. The correct usage of the dynamic reconfiguration using the input parameter is depicted on Figure 3.3.

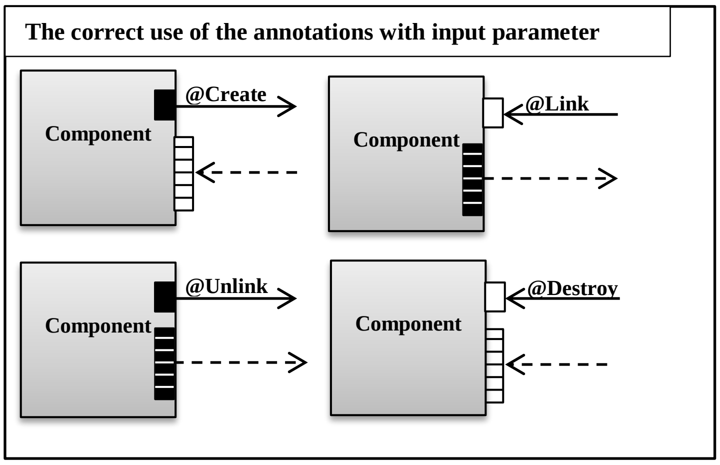

Figure 3.3 - The correct use of the annotations with input parameter

Reconfiguration by method

This variant is only available for the `unlink` and `destroy` annotations, because only those can be applied on own connectors. This case is required for immediate `unlink` and `destroy` by calling one method from the dynamic connection. Since the dynamic connection is destroyed by calling one of its methods, the annotations do not have to specify neither the parameter nor the interface.

The restrictions for this interceptor type are exactly as expected. The `unlink` can be placed only on business required connector and the `destroy` on the business provided. The reconfiguration by method can be seen on Figure 3.4.

The interceptor required for this reconfiguration needs to be notified by the method call, that it is time to execute its logic. This type of interceptor will be called the notification interceptor.

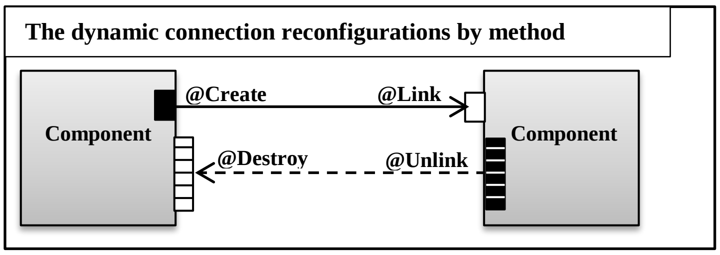

Figure 3.4 - The dynamic reconfiguration by method

Temporary reconfiguration by input parameter

The last reconfiguration variant is the temporary reconfigurations by input parameter. This reconfiguration can occur when the connection is needed only for the method execution. Such a feature can be provided only when the input parameter of the method is annotated by both the `create` and the `destroy` annotation on the business required end of the connection and by the `link` and `unlink` annotation on the other end. All these annotations also have to specify the same interface. The temporary connection is depicted on Figure 3.5.

The interceptor types required for this reconfiguration differ depending on the annotation. The `create` and `link` interceptors does not have to do anything unexpected, therefore they will retain the input parameter modifier interceptor type. The `unlink` and `destroy` cannot use the input parameter modifier interceptor type, because this type is executed before the method call, and the requirements are that it is sustained until the method finishes. Since any other interceptor type cannot handle this situation, there has to be created a new type, which would apply the logic on the method parameter after the method delegation finishes. This type will be called the input parameter handler.

This special case of dynamic reconfiguration also faces another issue, which is the maintenance of certain order of the interceptors. Since the destruction of the connector cannot happen before the connection is created and the same goes for unlink, which must wait until the connector is linked, there have to be ensured, that these interceptor will execute in specified sequence. Since the creation of the interceptors is dependent on the execution of the aspects, the possible solution would be to extend the aspects with a priority field. This field could provide an ordering, which would ensure that aspects with lower priority would be executed at the end and vice versa.

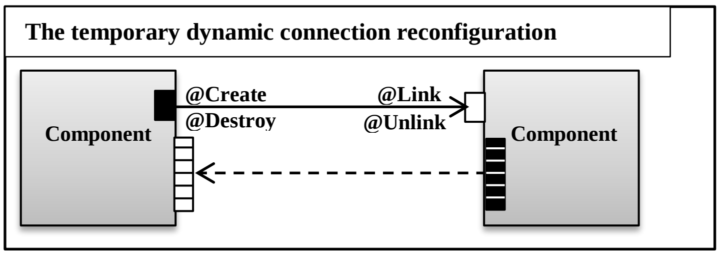

Figure 3.5 - The temporary dynamic connection reconfiguration

3.2.3 Annotation semantics

This section is focused on the explanation of the dynamic connection reconfiguration semantics from the component application developer's perspective.

Create annotation

The `create` annotation is specified in the ADL file of the components frame as a part of the components business provided or required interfaces. The `create` annotations has to specify in one of its attributes one method from the interface containing the annotation and the interface, which will the dynamic connection use for communication. If the containing interface is business required, the `create` annotation also has to specify, which of the methods input parameters will be used for the dynamic reconfiguration. Business provided interface will use for the reconfiguration the output parameter and because Java uses only one output parameter, there does not have to be any further specification.

In the run-time, when the method from the interface containing the create annotation is called and the annotations specifies the called method, then from the specified parameter is created new business provided collection connector with specified interface. The parameter of the method cannot be used until it is affected by either the `link` or the `destroy` annotations. The result of the `create` annotation is depicted on Figure 3.6.

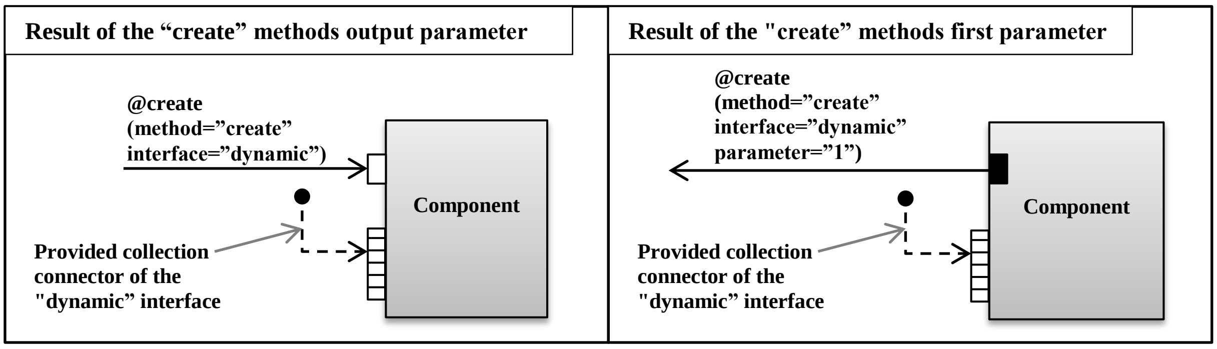

Figure 3.6 - The result of the create annotation

Link annotation

The `link` annotation is also specified in the ADL file of the components frame as a part of the components business provided or required interface. Even the method and interface specification match with the create annotation. However, the parameter specification is somewhat inverted. The parameter has to be specified only for the business provided interface containing the annotation, because only there the input parameters can be used.

In the run-time, the link annotation expects, that when the method specified by the annotation is called, then the specified parameter contains the object, which was already affected by the `create` annotation with the same interface. If it contains any different object, the reconfiguration will fail. From the parameter will be created new business required collection connector, which will be connected to the business provided collection connector created by the `create` annotation. When the parameter enters the component, it may be used to call any methods from the interface specified by the annotation. The method calls to the dynamic connection will execute the same methods on the object affected by the `create` annotation and return the same result.

Unlink annotation
The specification of the `unlink` annotation is equal to the specification of the `create` annotation, except the `unlink` specification contains one additional variant, where is specified only the method. This variant can be applied only if the interface containing the annotation is business required collection type.

The `unlink` annotation can only work on the dynamic connections, therefore if the method specified by the `unlink` annotation is called in the run-time and does not contain the reference to the dynamic connection with the specified interface in the parameter specifications, the reconfiguration will fail. Otherwise, it will destroy the business required collection connector created by the `link` annotation. The methods parameter affected by the `unlink` annotation is in the same state as the parameter affected by the `create` annotation and therefore cannot be used until the `link` or `destroy` annotation modifies its state.

The variant of the `unlink` annotation, which specifies only the method is triggered, when this method from the containing interface is called. This variant destroys the business required collection connector of the connection from the dynamic connection, which called the method. This variant has to be followed by the same `destroy` annotation on the other side of the connection.

Destroy annotation
The `destroy` annotation contains all the variants of the `link` annotations with one additional variant on the business provided collection interface, where can be specified this annotation only the method assigned.

When the method specified by this annotation is called in the run-time, the parameter must contain the object, which was originally affected by the create annotation of this component. Furthermore, the interface specified by the destroy annotation must match the interface in the create annotation. The last condition is, that the annotation from the dynamic connection reconfiguration annotations, which was the last to affect the parameter, has to be either the `create` annotation or the `unlink` annotation. When all these conditions are met the component destroys its business provided collection connector, which was created, when the parameter was firstly affected by the `create` annotation. The contents of the parameter entering the component will be the same reference, which was originally sent from this component.

The variant of this annotation, where only the method is specified, is similar to the same variant of the `unlink` annotation. The difference is that the business provided collection connector of the dynamic connection containing the called method is destroyed. The `destroy` annotation for this variant also requires to be preceded by the `unlink` annotation on the other end of the connection.

4 Solution proposal

This thesis defines five different aspects, which are described in following subchapters.

4.1 Factory aspect

The `factory` aspect specifies the following new entities to the SOFA2 model.

- Factory frame annotation
- Factory connector interface annotation
- Micro-component
- Interceptor

The purpose of the frame annotation is to identify components with the ability to create new dynamic components and the connector interface annotation identifies all the factory methods using the `factory` components.

4.1.1 Requirements

The proposed `factory` aspect requires the implementation of the following support:

ConnectorDeploymentService

- The generation of the business required shared connectors for each interface specified by the `factory` annotation, available at the `factory` components deployment dock.

GlobalConnectorManager

- The API capable of identifying the server component and its deployment dock by any connector from the application.

DeploymentDock

- The creation of the dynamic components.
- The realization of component connection from the provided connector.
- The ability to modify components hierarchy.

4.1.2 Micro-component

The micro-component will be connected to the base micro-component, because of the invocation of the micro-components logic into the components annotated field.

The requirements for the field are:

- It must be annotated by the `@SOFAFactoryProvider` annotation.
- It must be type of `SOFAComponentFactory`, which is the interface implemented by the micro-component.

The interface `SOFAComponentFactory` defines two methods. Both of these methods are used to gather parameters for the deployment dock, which provides the main logic.

The first method is only for creating the dynamic component and retrieving its connector. The second method does the same, except it continues with connecting the retrieved connector to the `factory` component and placing the dynamic component next to the `factory` component. The result of the second method will be the initialized connection.

4.1.3 Interceptor

The interceptor created for the `factory` aspect contains the ability to replace the output parameter of a method call with a different object. It will be placed only to a business required connectors with the `factory` annotations and it will use the dynamic components connector, which will be passed as an output parameter of the factory method, to initialize the connection between the requesting component and the dynamic component. When finished, the interceptor will move the dynamic component next to the requesting component.

4.2 Self-shutting aspect

This aspect specifies only one frame annotation named `self-shutting` and one micro-component. The micro-component will be initialized by the components marked with the `self-shutting` annotation and provide to fields annotated by `@SelfShutter` the logic of the component destruction. The logic will be implemented by the micro-component, which will use for this purpose the interface `SOFASelfShutting`, which will provide only one method called `shutdown`. This method will only use the deployment dock, which will then handle the components destruction.

4.3 Create aspect

The creation aspect specifies multiple variants of interceptors and the `create` and `link` connector interface annotations, which determine the type and the position of the interceptor. The content of both annotations is the method of the connection, parameter containing the provided object and the interface used for the connection.

The priority of the aspect will be set to number three in order to make room for the `unlink` aspect, which has higher priority, then the `create` aspect, but lower priority then the base aspects.

4.3.1 Requirements

Model

- The execution of aspects according to their priority. The lower the priority is the closer the interceptor will be to the component.

ConnectorDeploymentService

- The generation of both business provided and required shared connectors for each interface specified by the `create` annotation and interface specified by the `link` annotation in composite components. The shared connectors must be assigned to those deployment docks, which contained the components with the annotation specifying their interface.

ConnectorsManager

- Add to the management of the dynamic connections the mapping of the connector to the server delegation chain endpoint when creating the connection.
- Add to the management of the dynamic connections the mapping of the client delegation chains entry point to the client delegation chain and the connector when linking the connection.

DeploymentDock

- Functionality capable of creating the instance of the business provided shared connector along with the delegation chain and then connecting it with the business required shared connector.
- Support for the creation the delegation chain for the shared connector.

4.3.2 Interceptors for the create annotations in primitive components

Interceptor type usage

The type of interceptor is created only when all of the conditions specified below the type are met. This description will also be used for all the interceptor type usage chapters.

- Input parameter value modifier interceptor
  - The annotation specifies the input parameter.
  - The annotated interface connector is business provided.
- Output parameter value modifier interceptor
  - The annotation specifies the output parameter.
  - The annotated interface connector is business required.

Interceptors logic

The `create` interceptor is responsible for the replacement of the parameter specified by the annotation. It passes the parameter to the deployment dock, which creates a delegation chain with the parameter as its endpoint. As the entry point of the delegation chain will be used the business provided shared connector generated by the `ConnectorDeploymentService`.

The deployment dock will afterwards create a connection from the delegation chains entry point to the complementary business required shared connector. The business required shared connector will be returned to the interceptor as a replacement for the provided object.

4.3.3 Interceptors for the link annotations in primitive components

Interceptor type usage

- Input parameter value modifier interceptor
    - The annotation specifies the input parameter.
    - The annotated interface connector is business required.
- Output parameter value modifier interceptor
    - The annotation specifies the output parameter.
    - The annotated interface connector is business provided.

Interceptors logic

The interceptor replaces the connector passed by the parameter specified by the annotation, with the endpoint of the delegation chain generated by the deployment dock.

4.3.4 Interceptors for link or create annotations in composite components

Interceptor type usage

- Input parameter value modifier interceptor
    - The annotation specifies the input parameter.
- Output parameter value modifier interceptor
    - The annotation specifies the output parameter.

Interceptors logic

This interceptor merges the logic of the `create` and the `link` interceptors into one. First it executes the logic of the `link` interceptor and then it uses the object, which was created as the connector replacement is used as a providing object to the new connection. The only difference is, that when the annotation is `create`, than the `link` action will not create a delegation chain and vice versa.

4.4 Unlink aspect

The `unlink` aspect will be separated from the `destroy` aspect in order to use different priority, which will be set to number two.

It will specify multiple types of interceptors, which will be dependent on the `unlink` annotation. It also needs to keep track of the `link` annotation in order to provide the temporary connection functionality.

The `unlink` annotation contains data about the method, parameter and interface of the unlinked connection. However, only the method specification is mandatory.

4.4.1 Requirements

DeploymentDock

- The utilization of the `ConnectorsManager` dynamic connection mapping for quick access to the connection delegation chain.

- Support for disconnecting the delegation chains.

4.4.2 Interceptors for the unlink annotation

Interceptor types usage

- Input parameter value modifier interceptor
  - The annotation specifies the interface and the input parameter.
  - The annotated connector is business required.
- Output parameter value modifier interceptor
  - The annotation specifies the interface and the output parameter.
  - The annotated connector is business provided.
- Notification interceptor
  - The annotation specifies only the method.
  - The annotated connector is for collections and it is business required.
- Input parameter value handler interceptor
  - The annotation specifies the interface and the input parameter.
  - The annotated connector is business provided and contains the `link` annotation with the same interface, method and parameter specification.

Interceptors logic for primitive components

The notification interceptor will compare the method specified by the annotation and the delegated method. If the method names equals, it will use the connector id of its own connection to identify the delegation chain, which will be then disconnected.

The other interceptors will obtain the delegation chain thanks to the object passed by the parameter specified by the annotation. When found it will be disconnected and the endpoint of the delegation chain, which is the business required shared connector, will be used as a replacement.

Interceptors logic for composite components

The logic of the interceptor for composite components is similar, except it uses also the logic of the dynamic connection destruction, which is executed in precedence and without the delegation chain disconnection. The rest is kept the same.

4.5 Destroy aspect

The destruction aspect contains multiple variants of interceptors and the `destroy` annotations, which determines the position and the variant of interceptor. The destruction aspect also uses the `create` annotation to identify the special case of the creation of the temporary connections.

The contents of the annotation are the name of the method used, the optional specification of the parameter and the optional specification of the connection interface.

4.5.1 Interceptors for the destroy annotation

Interceptor types usage

- Input parameter value modifier interceptor
  - The annotation specifies the interface and the input parameter.
  - The annotated connector is business provided.
- Output parameter value modifier interceptor
  - The annotation specifies the interface and the output parameter.
  - The annotated connector is business required.
- Notify return interceptor
  - The annotation specifies only the method.
  - The annotated connector is for collections and it is business provided.
- Input parameter value handler interceptor
  - The annotation specifies the interface and the input parameter.
  - The annotated connector is business provided and contains the `create` annotation with the same interface, method and parameter specification.

Interceptors logic for primitive components

The notification interceptor compares the delegated method and the method specified by the annotation. When matched, it uses its own id of the connection to find the delegation chain and the connectors, which shall be unregistered from the `DockConnectorManager` along with the disconnection of the delegation chain.

The other interceptors uses the parameter specified by the annotation, where should be contained the business required shared connector. From the connector is identified the delegation chain and the business provided shared connector. Both connectors are unregistered and the delegation chain is disconnected.

Interceptor logic for composite components

The behavior of the interceptors in composite component is the merge of the `unlink` and the `destroy` interceptor functionalities. Firstly is used the logic of the `destroy` interceptor, which is followed by the execution of the `unlink` interceptor logic omitted by the delegation chain disconnection.

5 Prototype implementation

5.1 Factory pattern

The implementation of `factory` pattern required the following changes. Firstly there had to be extended the core model to support `factory` pattern annotation and for that matter the ADL parser for the correct transformations. The data from the annotations needed to be used to extend the connection service, which is responsible for generating connectors for the whole component application. The connection service uses the data to determine which component is a `factory` component and provides its deployment dock with corresponding shared connector. Now when the shared connectors are available, the focus can be moved to the process of creation the dynamic component. Because the deployment dock is responsible for the creation of static components and therefore has all the necessary resources, it is ideal also for creating dynamic components. The extension of the deployment dock is also required for the hierarchy issues, which needs to be solved when linking the newly created dynamic component. These issues also required the modification of the connector management, because for the proper placement to the hierarchy there needs to be a way to determine the host to the client endpoint only by the reference of the endpoint. The last modifications required were to provide the functionality to the developer, which is done by the `factory` aspect through micro-components and the interceptors.

All the mentioned modifications of SOFA2 are described in the following chapters.

5.1.1 Model

The addition of the annotation to the ADL required extending the core model of SOFA2. There had to be added a new class Factory, which is used to annotate interface connectors and frames of component responsible for creating and connecting to the dynamic component.

The core model of SOFA2 is created by Eclipse Modeling Framework (EMF) [**8**] and for the initialization of the model, a well-defined XML specification is used. Since the ADL of SOFA2 uses its own well defined XML specification different from the EMF, transformations from the ADL to EMF have to exist. For this purpose there are XSL transformations in the sofa-tools-api package. These XSL transformations along with the scheme defining the ADL needed to be modified in order to provide the `factory` annotation to the developers.

5.1.2 Connection deployment service

The connection deployment service was the only part, which could not be simply modified or extended. The logic was too complex for common understanding and the addition of more functionality would lead only to errors. To overcome this gap, the connection deployment service had to be completely recreated separating the functionality into simpler smaller classes with the intention to facilitate the extendibility.

The main purpose of the connection service is to provide connectors for all connections in the component application at deployment. This provision requires complex procedure which differs due to the type of connection and the events, which can occur to the component. The procedure has to traverse the deployment plan and at every component check for connections and events, which may require connector. There are three types of connectors:

- **Basic** – this is the most common connector, which is static and direct component to component.
- **Delegate or subsume** – this connector is a special type of static connector, which only passes the connection from composite component to its subcomponents or from the subcomponents to the composite component.
- **Shared –** this connector is used for dynamic connections and for that reason it is not bound to a component and connection, but to connecting interface and deployment dock. Any component, which lies in the same deployment dock can access this connector and use it.

Each type of the connector has to be treated little bit differently and the creation of the concrete type of connector depends on the events bound to the component. To make it little more difficult, all the connectors can either be required or provided and may differ in communication style.

The `ConnectorDeploymentService` needs to gather all these mentioned data and pass them to connector generator, which generates a compiled java code for each connector. The java code then needs to be packed into a jar file and uploaded to the repository and the information about the jar file must be brought to the deployment plan, to be easily accessible at the application start.

The proposed solution contains a hierarchy of `ConnectorSpecification` classes, `ConnectorSpecificationHandler` classes and a `CodeBundleUploader` class. Since working with all these classes require a lot of complex handling, all the classes are hidden under a façade, which provides all the functionality by a single method.

ConnectorSpecification classes

The hierarchy of connector specification classes was created to solve the issue of multiple connector types - therefore each type has its own class. All the connector specification classes inherit from an abstract base class, which has all the logic common to all the connectors. Their purpose is to obtain and provide all required information about these connectors, since the process of acquiring the information differs for each type. The manipulation with the ConnectorSpecification classes is done with their base class, which unites all the types into one class. The class diagram for the ConnectorSpecification hierarchy is shown on Figure 5.1.

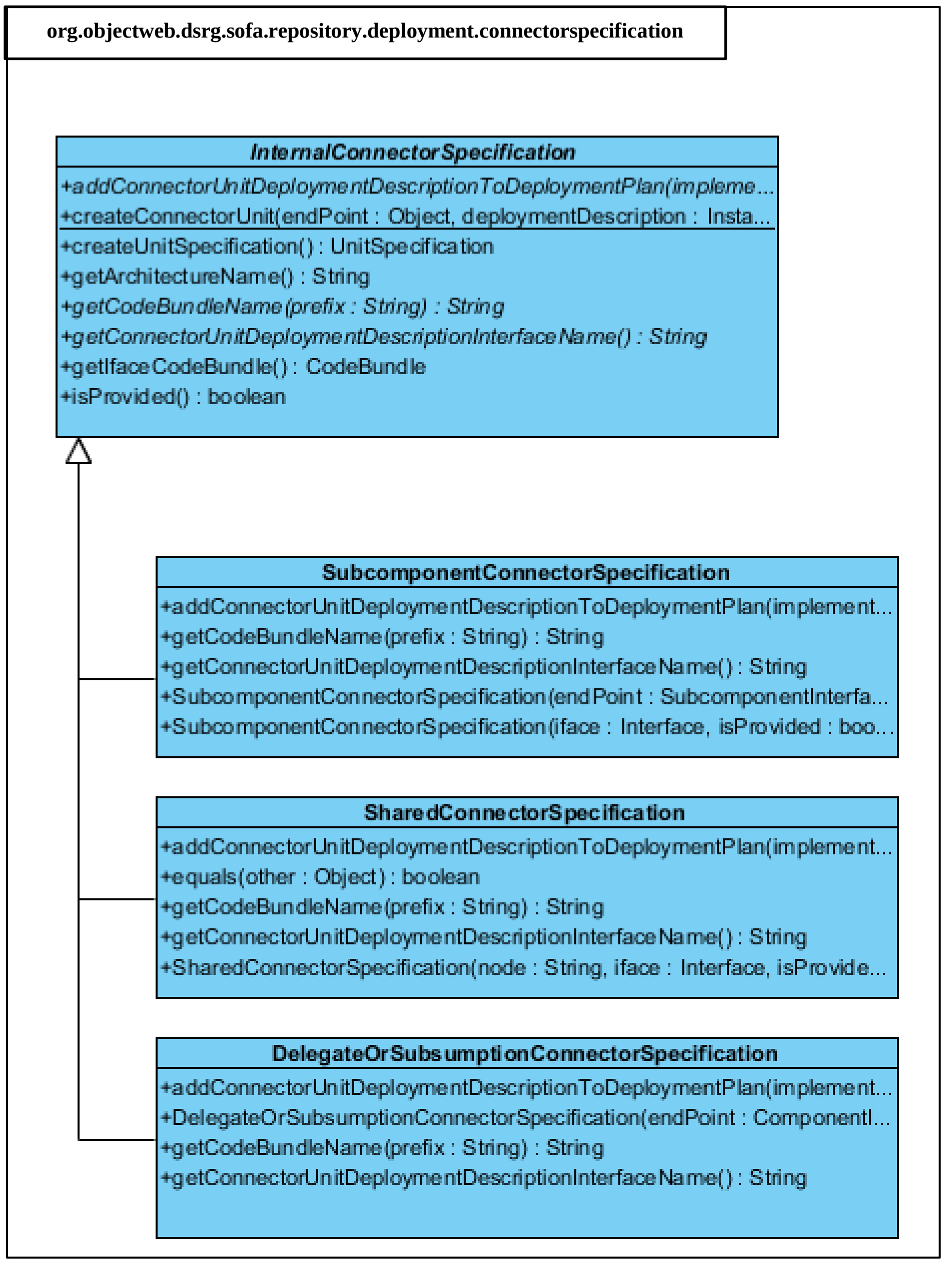
org.objectweb.dsrg.sofa.repository.deployment.connectorspecification
InternalConnectorSpecification
+addConnectorUnitDeploymentDescriptionToDeploymentPlan(impleme...
+createConnectorUnit(endPoint : Object, deploymentDescription : Insta...
+createUnitSpecification() : UnitSpecification
+getArchitectureName() : String
+getCodeBundleName(prefix : String) : String
+getConnectorUnitDeploymentDescriptionInterfaceName() : String
+getIfaceCodeBundle() : CodeBundle
+isProvided() : boolean
SubcomponentConnectorSpecification
+addConnectorUnitDeploymentDescriptionToDeploymentPlan(implement...
+getCodeBundleName(prefix : String) : String
+getConnectorUnitDeploymentDescriptionInterfaceName() : String
+SubcomponentConnectorSpecification(endPoint : SubcomponentInterfa...
+SubcomponentConnectorSpecification(iface : Interface, isProvided : boo...
SharedConnectorSpecification
+addConnectorUnitDeploymentDescriptionToDeploymentPlan(implement...
+equals(other : Object) : boolean
+getCodeBundleName(prefix : String) : String
+getConnectorUnitDeploymentDescriptionInterfaceName() : String
+SharedConnectorSpecification(node : String, iface : Interface, isProvide...
DelegateOrSubsumptionConnectorSpecification
+addConnectorUnitDeploymentDescriptionToDeploymentPlan(implement...
+DelegateOrSubsumptionConnectorSpecification(endPoint : ComponentI...
+getCodeBundleName(prefix : String) : String
+getConnectorUnitDeploymentDescriptionInterfaceName() : String

Figure 5.1 - ConnectorSpecification class diagram

ConnectorSpecificationHandler classes

The `ConnectorSpecificationHandler` classes were created as a strategy to provide connectors where they are needed. Each of these classes should be able to check if processing component has any events, which requires connectors, and decide which connector specifications should be created. The default connector specification handler is the `StaticConnectorSpecificationHandler`, which creates either the subcomponent connector specification or the delegate or subsumption connector specification depending on the type connection provided. The connector specification handlers also inherit from an abstract class, which simplifies the access to all the `ConnectorSpecifications` and their properties.

Thanks to this separation, the new functionality can be easily added, especially for the `factory` pattern. This class called `FactoryConnectorSpecificationHandler` is responsible for identifying all the dynamic components, which can be created by the currently processing component. The information is used to create one static connector specification to the main provided connection of each dynamic component and the complementary shared connector specification to the currently processing components deployment dock.

The whole hierarchy of the connector specification handler classes is depicted on Figure 5.2.

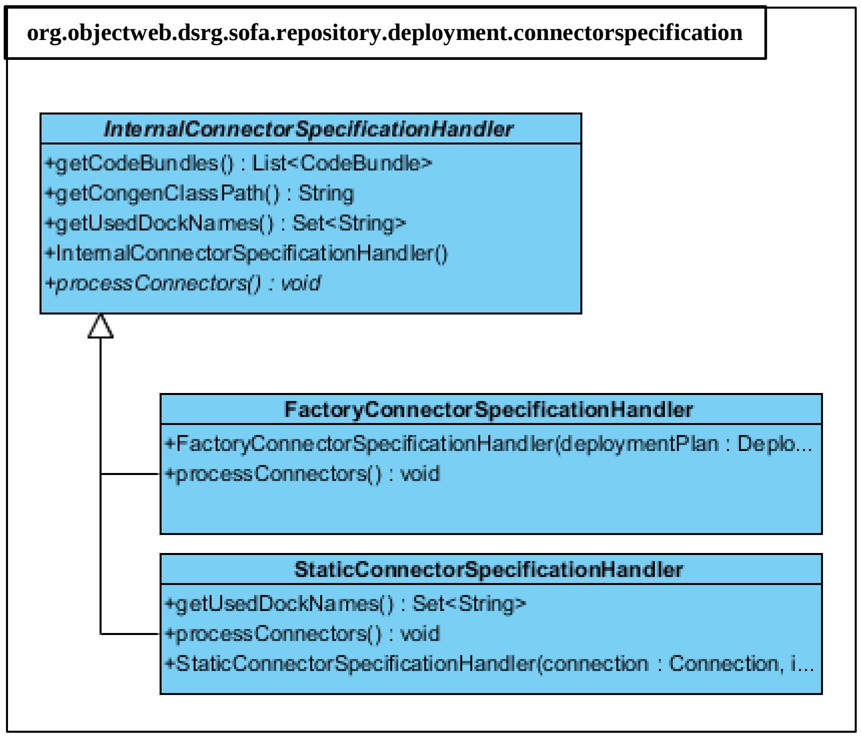
org.objectweb.dsrg.sofa.repository.deployment.connectorspecification
InternalConnectorSpecificationHandler
+getCodeBundles() : List<CodeBundle>
+getCongenClassPath() : String
+getUsedDockNames() : Set<String>
+InternalConnectorSpecificationHandler()
+processConnectors() : void
FactoryConnectorSpecificationHandler
+FactoryConnectorSpecificationHandler(deploymentPlan : Deplo...
+processConnectors() : void
StaticConnectorSpecificationHandler
+getUsedDockNames() : Set<String>
+processConnectors() : void
+StaticConnectorSpecificationHandler(connection : Connection, i...

Figure 5.2 - ConnectorSpecificationHandler class diagram

CodeBundleUploader class

This class, as the name suggest, is responsible for uploading the connector code bundles to the repository. The code bundle is a class of the core model, which references a compiled Java code and in this case it is used to access the code of the generated connector. It works with the connector specification classes and the generated connector code from the connector generator.

Firstly the code bundles need to be prepared, which means setting the right path to the compiled code (which is done using strict naming conventions) and finding the complement generated connector code. Then the code bundle needs to be placed correctly into the deployment plan, which is done by the connector specification classes itself, because they have the proper logic to do that. In the end the compiled Java code from the connector generators is placed into a jar file and uploaded into repository, where it can be accessed when needed.

ConnectorDeploymentServiceFacade class

This class is a façade for all the mentioned classes. It provides only one method, which is `DeployInstance`. This method recursively traverses the deployment plan and for each component creates a connector specification handler classes, which prepares the connection specifications. The data from the specifications and specifications handlers are passed to the connector generator, which generates Java code for each specification. The Java code is then passed to the `CodeBundleUploader` class along with the connector specification classes, which is responsible for the final actions of the whole procedure.

The simplified class diagram is depicted on Figure 5.3.

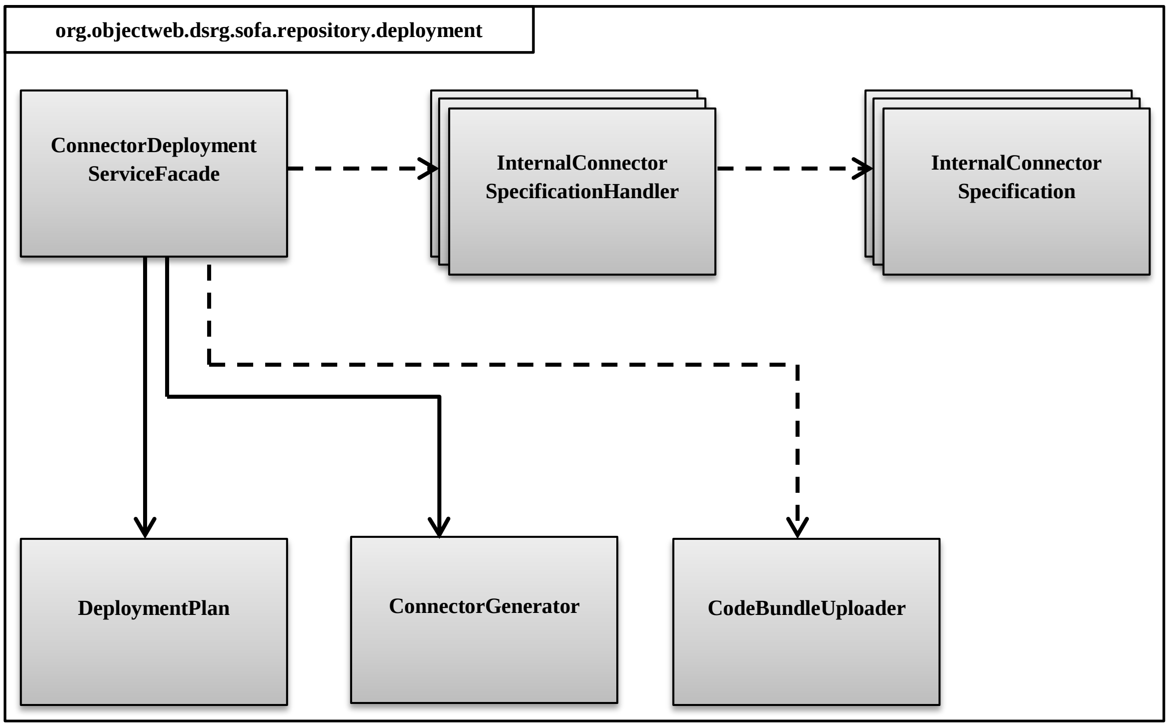
org.objectweb.dsrg.sofa.repository.deployment
ConnectorDeployment ServiceFacade
InternalConnector SpecificationHandler
InternalConnector Specification
DeploymentPlan
ConnectorGenerator
CodeBundleUploader

Figure 5.3 - Simplified ConnectorDeploymentServiceFacade class diagram

5.1.3 Global connector manager

Since it was decided to make the `factory` components responsible for realizing the connection, while the calling component only accepts the connected endpoint, there had to be extended the API of the global connector manager. Because the correct placement of the component to the hierarchy requires the knowledge about the dynamic component just from the client endpoint, there had to be created a system, which would track all the endpoints. This system would allow determining the component, which provides the connection, only by the connector id. Thanks to this system the caller component could identify the dynamic component and then change its hierarchy.

The first proposal was to extend the generated connectors and add to them more information about their destination. This proposal was declined, because there was found even more elegant solution. SOFA2 contains a global connector manager, which is responsible for binding the connectors with the same id together. Since all connectors have to be registered in this global connector manager to be bound, it is ideal for the tracking system. The only thing needed was to provide information about the connector and its component to the `subscribeToConnector` method and store this information to some special class within a hash map. The rest was to provide methods, which could retrieve the information from the special class to anyone, who has the connector id.

5.1.4 Deployment dock

The main logic of creating new dynamic components, accepting them by the calling component and the correct placement of the dynamic component to the hierarchy was placed into the deployment dock. This decision was completely natural, since the deployment dock is responsible for maintaining the component hierarchy and initialization of the component application. The newly implemented logic exposed the following six new methods.

InstantiateDynamicComponent method

This method uses almost all the relevant information, which can be passed by the `factory` micro-component, to create new dynamic component and return its component id. As was mentioned in the analysis, the creation of the dynamic component from within the `factory` component needs only two parameters, which are component name and the interface name. Both names must be specified in the ADL and must refer to an existing component and implemented interface. However this method requires only the id of the `factory` component, the name of the dynamic component and the index.

To create the dynamic component it has to be firstly found. This step is easy, because the `factory` component has in its deployment description all the information about dynamic components that it can create. The `factory` component deployment description can be found from the deployment dock by the `factory` components id and the required dynamic component is identified by the name.

The second step is to create templates for all the provided connectors of the dynamic component. This requires the type of the connection, which is always provided, the name of the interface, which is available from the frame of the dynamic component and the connector id. The connector id is a string, which is named using some strict naming convention. To ensure that the connector id will be unique for current connection, it is extended by the index parameter.

The last step is to find deployment dock, which was chosen as a container for the dynamic component and use its `instantiateComponent` method passing all the prepared data. This method is used for the initialization of the static components and therefore the dynamic components will not differ from the static.

GetDynamicComponentsProvidedConnection method

This method is the continuation of the `instantiateDynamicComponent` method. It is designated for the `factory` micro-component and retrieves the dynamic components connector according to specified interface name.

The only logic, which is within this method, is the following two actions. The first is the generation of the connector id, which will, thanks to the strict naming conventions and the same provided parameters, correspond to the dynamic components provided connector of the required interface. The second step is only to retrieve this connector using the connector id and connector management and return it to the micro-component.

ConnectDynamicInstanceWithCorrectComponentHierarchy method
This method is prepared for the `factory` interceptor, which is supposed to call it, when returning back from the factory method call. Its purpose is to create an interceptor chain and place the received connector at the beginning of the chain. The creation of the interceptor chain for collection connections is already implemented in SOFA2 and it is done by the delegation chain template, which is generated at the application start. This template simulates basic interceptor chain, but instead of really creating the interceptors, it only records the actions and when needed it use those actions to create exact copy of the chain. The end of the chain is then passed back as a return value of the called factory method instead of the connector.

As the name of the method suggest, it is also responsible for placing the dynamic component into hierarchy. For this task, firstly the dynamic component needs to be found. This is done thanks to the connector id of the returned connector and the extended logic of global connector management mentioned in the previous chapter. When the dynamic component is found, the hierarchy of itself and its subcomponents needs to be changed, since it is allowed to create a composite dynamic component. The second step is to find the parent of the caller component, because the parent component (and not the caller component) is the one adding the dynamic component to itself as a subcomponent and therefore should be responsible for the placement. In the end the `addComponentToHierarchy` method is called at the parent component deployment dock with the information about the parent and the dynamic component and it will handle the rest.

AddComponentToHierarchy method
The logic of this method is very simple. It only checks if the parent component, which is passed as a parameter along with the child component, contains the child component and if not, adds it to its children. Of course the placement of the component to the hierarchy does not stop here, the child components parent also needs to be changed and that could be done only from the deployment dock of the child component. For this purpose the following `changeComponentsParentComponent` method was created.

ChangeComponentsParentComponent method
This methods objective is to change the parent component of a component and to change the hierarchy name according to the current placement to self and to all the components found in the sub-tree of this component. The parameters given are the information about both the parent component and for the component, which needs to be modified. Even though the change of the parent component and the correction of the hierarchy name of the component could be done within the deployment dock of this component, the correction of the subcomponents hierarchy name could not. Because of that there had to be implemented the

last exposed method for deployment dock, which is responsible for the correcting hierarchy name of all the components in the sub-tree and which is called `repairComponentHierarchy`.

RepairComponentHierarchy method

This method is based on recursion and the only parameter it gets is the id of the parent component, which changed its hierarchy. This method traverses all the child components of the parent method and changes the hierarchy name according to their parent's name. When the modification is made, it finds the deployment dock of the subcomponent and calls there itself with the id of the subcomponent as a parameter. This method finishes, when there are no subcomponents to modify.

5.1.5 Micro-component

The `factory` micro-component is implemented to provide the creation of dynamic components into the components logic. The creation of the micro-component is allowed only to those components, which has provided connection or frame annotated by the `factory` annotations. This way is prevented to the uncontrolled dynamic reconfiguration.

The `factory` micro-component is connected to the base micro-component, which has access directly to the components content and data about the component, like components id. It is also connected to special micro-component called `MIComponentFactoryImpl`, which only delegates the logic, if any of the other micro-components would need to use it.

When the micro-component is initializing, it uses reflection to search for an annotated field with a special interface named `SOFAComponentFactory` and when found it instantiates it with self. This ensures that the component will have the logic available when initialized and the logic will be provided by the micro-component itself.

The `factory` micro-component provides to the components logic two methods with common logic. The first is called `createDynamicComponentWithConnection` and it is used only to create a dynamic component and retrieve connector endpoint, which can be then passed as a return value to the other components for acceptance and hierarchy placement. The second method named `createDynamicComponentConnectedToSelf` is using the first method to do exactly the same, except it does not retrieve the connector endpoint. Instead, it lets the `factory` component to accept it and place it in the hierarchy as a sibling. The initialized connection is returned back to the `factory` component and can be used for components internal purposes.

More to the provided factory methods is described in the following subsections.

CreateDynamicComponentWithConnection method

This method requires two parameters to be executed, namely the dynamic component's name and the interface name, which will be used for the connection and must be provided by the dynamic component. Since this method uses mostly the exposed logic of the deployment dock, which was described in section 5.1.4, it firstly needs to access the deployment dock, which is hosting the `factory` component. When the deployment dock is found, it uses its method `instantiateDynamicComponent`, which creates the dynamic component, and then the method `getDynamicComponentsProvidedConnection`, which retrieves the right connection endpoint. As the index parameter, which is required to both methods, a counter that is incremented after each call is used. Since the connector id generation uses the information about the `factory` component, it ensures that the generated connector id will always be unique. The connection endpoint is then returned back to the `factory` component, which can use it to return it through a factory method call.

CreateDynamicComponentConnectedToSelf method

As was described before, this method uses the `createDynamicComponentWithConnection` method to create the dynamic component and retrieve wanted connector. As the name suggest, after the creation of the dynamic component it is supposed to connect the dynamic component to self. For this purpose is also used a method from deployment dock named `connectDynamicInstanceWithCorrectComponentHierarchy`, which will do the rest of the logic. The only thing left is to return the result of the method call.

5.1.6 Interceptors

The creation of the interceptors is a little bit complex, since the interceptors have to implement any interface, which can be used for the connection. There are two possible ways to solve this issue. The first way is to create a proxy implementing the required interface and an invocation handler, which will listen to the method calls of the proxy and do the logic. However, this solution requires using a lot of reflection and since the interceptors may be used many times in every connection, it will have a bad impact on the performance. Much faster, but little more complicated solution is to generate the interceptor classes using Java byte code. This way the only performance cost will be the generation time and after that all the generated interceptors will act as regular classes. Because the use of the Java byte code is quite complicated, it is better to use it as little as possible. For this reason we have created the base interceptor generator class, which can generate the common logic to the interceptor and the rest must be implemented by its children.

The main idea of the interceptor was that it should use as little logic as possible. The main logic can be done by any other non-generated class, and therefore the interceptor is only supposed to call the class with the main logic, provide parameters to it and of course delegate the call. Because of that the base interceptor generator adds two fields and their public setters to the interceptors – the `functionalityProvider` and the `delegate` field. The only thing that needs to be added to these fields is the interface, which they implement. The base interceptor generator also implements to the interceptors the class header, the fields, the `MIInterceptor` methods the constructor and the `init` method, which only checks that both of the fields are provided. The extended children only need to implement the logic of the delegated methods. The class diagram for the base and the `FactoryInterceptorGenerator` is depicted on Figure 5.4.

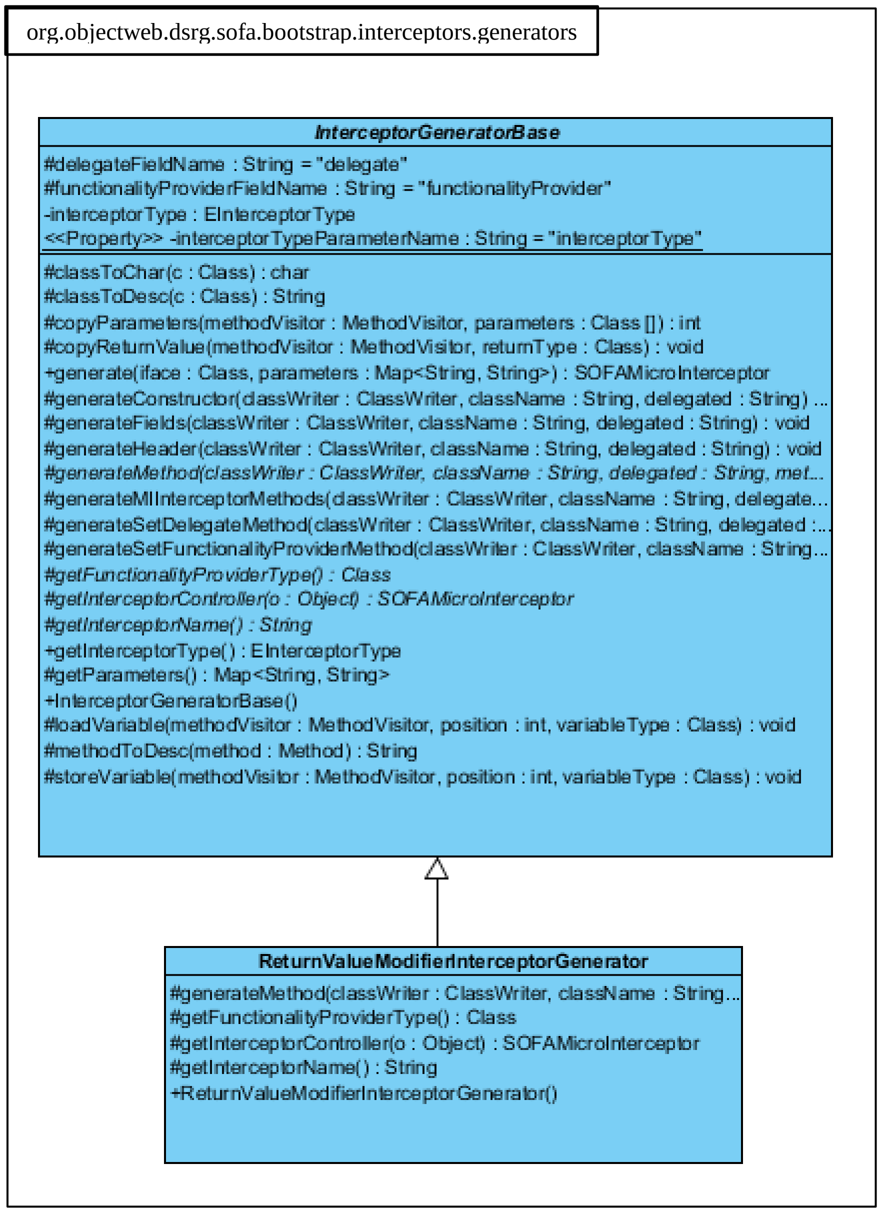
org.objectweb.dsrg.sofa.bootstrap.interceptors.generators
InterceptorGeneratorBase
#delegateFieldName : String = "delegate"
#functionalityProviderFieldName : String = "functionalityProvider"
-interceptorType : EInterceptorType
<<Property>> -interceptorTypeParameterName : String = "interceptorType"
#classToChar(c : Class) : char
#classToDesc(c : Class) : String
#copyParameters(methodVisitor : MethodVisitor, parameters : Class []) : int
#copyReturnValue(methodVisitor : MethodVisitor, returnType : Class) : void
+generate(iface : Class, parameters : Map<String, String>) : SOFAMicroInterceptor
#generateConstructor(classWriter : ClassWriter, className : String, delegated : String) ...
#generateFields(classWriter : ClassWriter, className : String, delegated : String) : void
#generateHeader(classWriter : ClassWriter, className : String, delegated : String) : void
#generateMethod(classWriter : ClassWriter, className : String, delegated : String, met...
#generateMIInterceptorMethods(classWriter : ClassWriter, className : String, delegate...
#generateSetDelegateMethod(classWriter : ClassWriter, className : String, delegated :...
#generateSetFunctionalityProviderMethod(classWriter : ClassWriter, className : String...
#getFunctionalityProviderType() : Class
#getInterceptorController(o : Object) : SOFAMicroInterceptor
#getInterceptorName() : String
+getInterceptorType() : EInterceptorType
#getParameters() : Map<String, String>
+InterceptorGeneratorBase()
#loadVariable(methodVisitor : MethodVisitor, position : int, variableType : Class) : void
#methodToDesc(method : Method) : String
#storeVariable(methodVisitor : MethodVisitor, position : int, variableType : Class) : void
ReturnValueModifierInterceptorGenerator
#generateMethod(classWriter : ClassWriter, className : String...
#getFunctionalityProviderType() : Class
#getInterceptorController(o : Object) : SOFAMicroInterceptor
#getInterceptorName() : String
+ReturnValueModifierInterceptorGenerator()

Figure 5.4 - Interceptor generators

Since the work with Java byte code is very complex, we used ASM [**9**], which is an Eclipse plugin with the ability to display Java code as Java byte code. This way it was much simpler to implement the generation of the interceptor classes.

The `FactoryInterceptorGenerator` name was changed to `ReturnValueModifierInterceptorGenerator`, because it better describes the purpose of the generator and because it can be used for other purposes, for example in the dynamic connections. The generated code inside the methods of the interceptor is very simple. It only delegates the call of the method and its return value is used as a parameter to the `modifyReturnValue` method of the `functionalityProvider`. The result of the method is then passed back to the interceptor caller. The generated Java code may look like the following code.

```
public Object interfaceMethodName(…parameters…) {
      return functionalityProvider.modifyReturnValue("interfaceMethodName",
            delegate.interfaceMethodName(…parameters…));
}
```

The `factory` functionality provider is called `FactoryInterceptorController`. This class firstly checks whether the name of the called method is specified in the connections `factory` annotation and if it is, it does exactly the same as the second step of the method `createDynamicComponentConnectedToSelf` described in chapter 5.1.5.

5.2 Dynamic connections

The implementation of dynamic connections was simplified thanks to the many changes made during the implementation of the factory pattern, which proved to be very useful. They also determined the way, which the implementation of the required changes for the dynamic connection should lead to. However, there were still a lot of issues, which required various modifications to the core code and which will be described in following subsections. But first follows a brief explanation of the modifications made and the reasons, why they were needed.

The first modification needed was to extend the core model. It required new annotations of the dynamic connections and the aspect class in the model had to implement the interface `IComparable`, because the order of the aspect execution could no longer be random. Next, the component connection management had to be changed. It had to provide mapping of the connectors to the connection references in the component. This was necessary to recognize which connector should be unlinked or which connection was destroyed. The last step was the implementation of new interceptors for each annotation action and the extension of the deployment dock, which was required for each action. In the end, the interface selector engine had to be modified, since it was not flexible enough to specify all the new possible events.

5.2.1 Model

The first modification to the core model was the addition of the `create`, `link`, `unlink` and `destroy` annotation classes, which required similar XML scheme and XSLT changes like the factory patterns core model. All the annotation classes resemble the `factory` annotation except they can be bound only to connection and they have new property about the parameter of the annotated method, which creates or destroys the dynamic connection.

The second modification was to the `Aspect` class. This class had to be made comparable, because some aspects can have a higher priority and must be executed in precedence. For this reason there was added new property `priority` to the class, which is used for the comparison. The lower the priority is the sooner is the aspect executed.

The order of the aspect execution also affects the order of the interceptor from the component to the connection. This is the main reason, why the ordering needed to be implemented. More information, why the ordering had to be implemented will be explained in section 5.2.5.

5.2.2 Connection deployment service

The modification of the connection deployment service was needed in order to provide the connectors to components which create the dynamic connection. Thanks to the new design of the service, there had to be created only one additional connector specification handler called `DynamicConnectorSpecificationHandler`.

The logic of the handler class is to traverse the annotations of the component connections and for each `create` annotation creates provided and required shared connector specification. However, creating the connectors just for the `create` annotation is not enough. When delegating or subsuming the `link` annotated connection from the composite component, the component has to do both `create` and `link` action, even though it does not have to be annotated with `create` annotation. That is why there also needs to be a check if the component is composite and if it is, then create the shared connector specification even for `link` annotated connections.

5.2.3 Component connector manager

The component connector manager in SOFA2 is an object assigned to every component with the responsibility for all its connections. That is why it was only natural to let this object manage the dynamic connections too.

The management of the dynamic connections requires the ability to create, link, unlink and destroy the connection. These actions required mapping of delegation chain entry point to the connector id for all required dynamic connections, mapping of connector id to the delegation chains endpoint for all provided dynamic connections and mapping of the connector id to the delegation chain itself. These mappings were required for quick identification of the relation between connector and delegation chain, when deleting or unlinking the connection.

CreateDynamicConnection method

The purpose of this method is to create a provided dynamic connection from any object passed as a parameter, which implements the interface of the connector. The method has to firstly retrieve the delegation chain template from the component, which is used to create the actual delegation chain. The rest of the method is simply filling the maps with information and binding the provided object to the connector, which is handled by the `ConnectorAdaptor` class.

AcceptDynamicConnection method

This method is using the connector passed as a parameter to connect it with a component and register it, along with the delegation chain and delegation chains entry point to the maps.

At first, it has to use the `ConnectorAdaptor` class, which can accept the connector a retrieve the connection endpoint. The rest is very similar to the `CreateDynamicConnection` method, where, if possible, the delegation chain is created using the connection endpoint and everything is added to the maps. In the end the delegation chain entry point or the connector itself for delegation and subsumption is passed back as the result of the method call.

UnlinkDynamicConnection method
This method is responsible for unlinking specific dynamic connection from the component. It has two versions with similar logic. The first version assumes that the connection is unlinked via parameter in method and therefore expects the parameter as its own. The second version is intended for unlinking by calling the annotated method on the dynamic connection. Since the connection, which has to be unlinked, is the same connection, which is used, the method is expecting as a parameter the current connection id.

The only difference between those two versions is only that the first version uses the mapping to identify the connection id from the connected object and the rest is practically the same.

Both methods firstly find using the connection id the delegation chain, which is connecting the component to the connector. If the connection is not delegated or subsumed and the delegation chain exists, then it is disconnected. The rest is simply unregistering the connector by the `ConnectorAdaptor` class, removing the information about the connection from the mapping and passing the mapped connector back to the caller.

DestroyDynamicConnection method
This method destroys the dynamic connection and thus prevents to any component to use this connection. Since this method is complementing the `unlinkDynamicConnection` method, it also requires two versions with different parameters and similar logic. The first version requires the connector, which is created when unlinking or creating the dynamic connection. The second version, when the connection is destroyed by method not the parameter, also requires the connection id.

In this case the second version requires additional handling, because for the destruction of the dynamic connection is everything mapped by the connector. To obtain the connector, there is used the `ConnectorAdaptor` class, which can access the `GlobalConnectionManager`, where all the connectors are registered to the connection id. By specifying the component and type of the connection, the `ConnectorAdaptor` can retrieve the connector. From now on is the procedure of both methods the same.

The procedure consists of obtaining the delegation chain from the mapping, if exists, and disconnecting it. The second step is to unregister the connector by the `ConnectorAdaptor` class, removing the mapped information about the connection and returning the object, which was mapped to the connector.

5.2.4 Deployment dock
The newly implemented methods exposed in the deployment dock have the same name and logic as the methods in the component connector manager, which was described in previous chapter 5.2.3. The methods in deployment dock only took advantage of the information about the components and the access to the connector manager to obtain required parameters for the call of the equivalent connector manager methods.

5.2.5 Interceptor types

Input parameter value modifier interceptor type

This interceptor type uses the logic of the functionality provider to replace the input parameters used for the delegated method with something else.

The functionality provider required for this interceptor must implement the `MIParameterValueModifierInterceptor` interface, which specifies method `modifyParameterValue`. The parameters of this method are the name of the method, the number of the parameter and the parameter itself.

The class responsible for the generation of the Java byte code is called `ParameterValueModifierInterceptorGenerator.`

For better understanding of the interceptor logic, the code similar to the generated Java byte code of the interceptor method is presented under the paragraph.

```
public Object interfaceMethodName(Object param1, Object param2, …) {
      param1 = functionalityProvider.modifyParameterValue(
            "interfaceMethodName", 1, param1);

      param2 = functionalityProvider.modifyParameterValue(
            "interfaceMethodName", 2, param2);

      . . .

      return delegate.interfaceMethodName(param1, param2, …);
}
```

Notification interceptor type

This interceptor notifies type is used to notify the functionality provider, that a method from tracked connection has been called. The generator responsible for generating this type of interceptors is called `NotifyInterceptorGenerator`, and the interface, which must be implemented by the functionality provider, is the `MINotificationInterceptor`. This interface specifies two methods. The first is `notifyCall` and the second is `notifyReturn`. Both methods contain only one parameter specifying the name of the method. The first one is called before the method delegation and the second one is called after the delegation.

The code similar to the generated Java byte code can be seen below.

```
public Object interfaceMethodName(…parameters…) {
      functionalityProvider.notifyCall("interfaceMethodName");
      Object result = delegate.interfaceMethodName(…parameters…);
      functionalityProvider.notifyReturn("interfaceMethodName");
      return result;
}
```

Input parameter value handler interceptor type

This interceptor type is very similar to the input parameter value modifier interceptor type. The difference between these two types is only in the timing of using the functionality provider. The parameter value handler waits until the method is completely delegated and then it starts to pass the parameters to the functionality provider. Since it does not have to modify anything, the result of the functionality provider logic can be void.

The functionality provider for this type has to implement the `MIParameterValueHandlerInterceptor` interface, which specifies only one method. The method is called `handleParameterValue` and its parameters are exactly the same as the input parameter modifier interceptor type.

The generation of the Jjava byte code is the responsibility of the `ParameterValueHandlerInterceptorGenerator` class. The similar code to the one it actually generates can be seen below.

```
public Object interfaceMethodName(Object param1, Object param2, …) {
      Object result = delegate.interfaceMethodName(param1, param2, …);

      functionalityProvider.handleParameterValue("interfaceMethodName", 1,
            param1);

      functionalityProvider.handleParameterValue("interfaceMethodName", 2,
            param2);

      . . .
}
```

5.2.6 Interceptors functionality providers

In the previous section was described in detail everything about the interceptor types and their generators. There was also mentioned, that these interceptors uses functionality providers, which has to implement corresponding interface. However, this was the only information, which has been possible to obtain from the previous section about functionality providers. That is why this section is focused on describing the interceptors functionality providers, which needed to be created and the logic they provide.

There are six newly created functionality providers. One is created for each dynamic connection annotation and two special types are created for the composite components.

The functionality providers for each annotation are very similar. They all implement the interface for the return value modification and parameter value modification functionality provider. They all are firstly initialized by retrieving the connection id and by traversing all the annotations of current connection. The annotations need to be traversed to obtain information about the method name and the parameter, which are responsible for the dynamic reconfiguration. When the actual method is called, then the functionality provider checks if the method name and the parameter matches the corresponding annotation and if it does, then the deployment dock is called to perform the changes.

The `destroy` and `unlink` functionality providers needs to implement among others the interface for the parameter value handling and notification functionality providers. Because it might happen that these functionality providers are used for basic destruction and the destruction of the temporary dynamic connection, there had to be added a check even for that. When the `modifyParameterValue` or `handleParameterValue` methods are called, there always needs to be a check for the existence of the creation (link) annotation with the same method name and the parameter. When found than the `handleParameterValue` method must be executed and vice versa.

The creation of the two special functionality providers for the composite components was needed, because of the difference between normal connection and delegated or subsumed connection. As it was already explained in the background chapter and depicted on **Error! eference source not found.** the delegation and subsumed connection does not contain delegation chain and therefore no interceptors can be used there. Because of that, the connection which is to be subsumed or delegated needs to provide both complementary actions.

The difference between the special type for composite component and the basic annotation based functionality provider is very slight. More accurate would be to say, that the creation functionality provider was created by merging the basic creation and `link` annotation functionality providers into one. The same goes for the destruction for composite component, which is the merge of basic `destroy` and `unlink` functionality providers.

5.2.7 Interface selector engine

The interface selector engine may be taken as filter for interfaces, which selects only those interfaces, which meets the specified requirements, from all the interfaces used by the component. The selected interfaces are than used for application of specified aspects logic. The creation of the interceptors, which is also part of the aspects logic, uses this engine to determine, whether it can be placed in the delegation chain of the interface. Since the number of possible variants increased from the dynamic reconfigurations, the interface selector engine could no longer keep up and had to be modified.

The logic implemented so far by the engine was two filters. The first filter was more general. It could for example select all the provided or required interfaces of the component. Any other wanted filter had to be named and implemented. The second filter was more precise. It could contain either star to select all the interfaces chosen by the previous filter or the name of the interface, which was required. Since the conditions for the dynamic connections were too many and more complex, it would be annoying to implement another filter for each condition, when the only difference was for example the change from required to provided.

The proposed solution was to create chain of filter classes, which would use the selected interfaces from the previous filter and apply own filter to them. This way it would be easy to select for example required `link` and `unlink` annotated interfaces without the creation of new filter.

The newly designed engine was created by extending two default abstract classes. The first class is `SimpleInterfaceSelector`. Any class extending this abstract class can be used as a last filter, because only filter, which can continue after this class is the interface name filter. The second abstract class is `CompositeInterfaceSelector`. This class is for filters, which can be followed by numerous other filters. After the selection of the filter is done, the abstract class initializes the following filter and provides it with required parameters. If the current filter is the last one, the interface name filter will be used and the selected interfaces will be returned to the caller. Because both of the abstract classes only requires to implement the `select()` method, the logic of the extended classes is very simple and straightforward. The addition of new filter now requires creating new filter class, which makes it very easy to manage the filters and makes the architecture more transparent.

Since the selection cannot be done without the interfaces, there were also created five base filters, which needs to be used first or the exception will be thrown. These filters are first in the following list of all newly implemented filters.

AllInterfaceSelector
This interface selector selects all business and control interfaces in the component.

BusinessInterfaceSelector
Interfaces from this selector are both required and provided business interfaces.

ControlInterfaceSelector
This is the only base interface selector, which does not support the additional filters. It selects all the control interfaces, which always has to be known and therefore the additional filters are not required.

ProvidedInterfaceSelector
Selects all business provided interfaces.

RequiredInterfaceSelector
Selects all business required interfaces.

CreateInterfaceSelector
This is the first non-based selector. It is composite and selects all interfaces with the `create` annotation.

CollectionInterfaceSelector
This is another composite selector, which selects all the interfaces, which has the `collection` attribute on true.

DestroyInterfaceSelector
Composite selector, which selects only interfaces with the `destroy` annotation.

FactoryInterfaceSelector
Composite selector, which selects only interfaces with the `factory` annotation.

LinkInterfaceSelector
Composite selector, which selects only interfaces with the `link` annotation.

UnlinkInterfaceSelector
Composite selector, which selects only interfaces with the `unlink` annotation.

6 Evaluation

The requirements for the demo application were to avoid complexity and to test all the features added by the dynamic reconfiguration. For these reasons two examples were created - one, which will evaluate the factory pattern with the component destruction and second which will test the variants of the dynamic connection reconfiguration.

The demo applications also have to prove, that the dynamic reconfiguration is capable of withstanding different hierarchy levels and also the component distribution. For this reason multiple assemblies and deployment plans had to be added in order to execute the same applications in different environments.

6.1 Factory example

The demo application created for the evaluation of the factory pattern with the component destruction consists of two static components and one dynamic. The first component is the `factory` component and it is providing connection to the other component called `FactoryTester`. The connection between the components specifies one factory method and one method, which tests the creation of dynamic component for self.

The logic of the dynamic component is very simple. It provides `IDynamicConnection` interface, which specifies a method for setting the name of the component, a method for logging to the system output and a method for destroying the dynamic component.

The main logic of the example is inside the `FactoryTester` component, which during its lifecycle repeatedly calls the factory method and uses the dynamic component to log a message as a proof, that the dynamic component is correctly created and connected. When the `FactoryTester` is finished with logging, it calls the dynamic components method for its destruction. The `FactoryTester` then continues with the use of the second method provided by the `factory` component, which does exactly the same thing, except the testing is done by the `factory` component. This procedure is repeated until the user shuts down the application or until it reaches maximum number of repetition.

The variants of the factory demo application are depicted on Figure 6.1 and Figure 6.2.

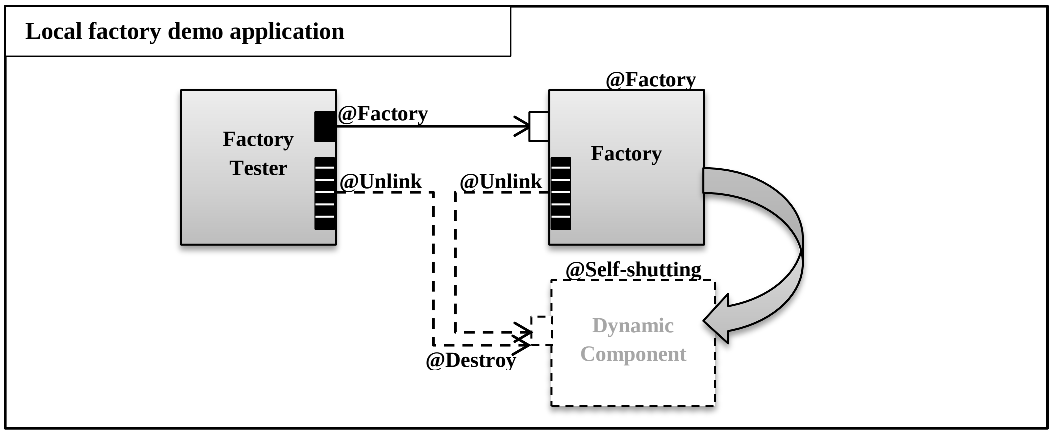

Figure 6.1 - Factory demo applications architecture

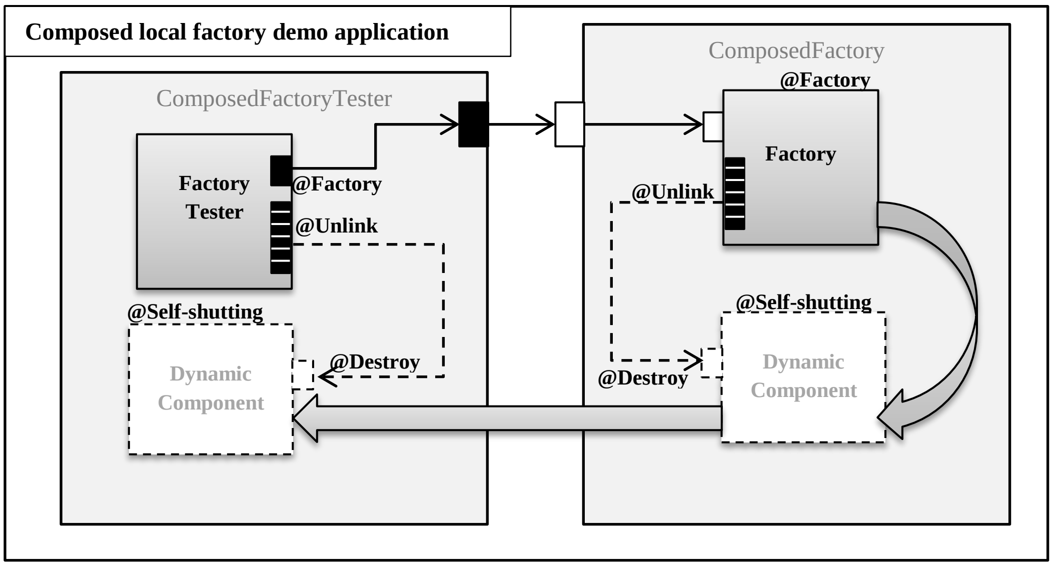

Figure 6.2 - Composed factory demo applications architecture

6.2 Dynamic connection reconfiguration example

The dynamic connection reconfiguration demo application is composed of four components, where one is a master, which only controls the other components' behavior. As a master it can execute any method of the slave components and thus control the whole dynamic reconfiguration.

The other three components are meant to exchange connectors among them. The first component is called `provider` and, as the name suggest, it uses itself to provide dynamic connection. For this purpose it uses one provided and one required interface. The provided interface contains the three methods with the following functionality:

- The creation of the dynamic connection by the output parameter.
- The destruction of the dynamic connection by the input parameter.

The required interface specifies the methods with following functionality:

- The creation of the dynamic connection by the input parameter.
- The destruction of the dynamic connection by the output parameter.
- The creation of the temporary connection.

The created dynamic connection also contains some simple functionality with the method for logging and the method for closing, which destroys the dynamic connection.

The other two components are used to test these methods and therefore each of the components is connected to the provider by one of the connections. The component, which is supposed to test the provided interface, is called `providedConnectionTester` and the second component is called `requiredConnectionTester`. The components are also connected with each other to test the connection transfer between the components with the interface called `connectionExchanger`. This interface specifies one method for exchanging the connection by input parameter and one for the output parameter.

The three mentioned components only implement the methods specified by the interfaces with additional logging of the currently executing actions. They also use the dynamic connection (when available) in order to prove that the connection between the components is correctly initialized. The main activity is up to the master component, which requires the connection to the all three components. The interfaces are called with the same name as the components with additional prefix "control". All the methods of the control interface are parameter-less and each of them executes one method specified by the required interface for the component. This means that for example the `controlProvider` interface specifies all the methods from the connection between the `provider` and the `requiredConnectionTester`. The architecture of the demo application is depicted on Figure 6.3 and Figure 6.4.

The master component follows a lifecycle, where it tests five different types of complete dynamic connection reconfigurations. All these dynamic reconfiguration tests are a part of a higher cycle, which finishes after a certain number of iterations have been done. The procedure of each type of reconfiguration is specified in the following subsections.

Reconfiguration by input parameter

- **provider** – creation of the dynamic connection by the input parameter.
- **requiredConnectionTester** – exchange of the dynamic connection by the input parameter
- **providedConnectionTester –** destruction of the dynamic connection by the input parameter.

Reconfiguration by output parameter

- **providedConnectionTester** – creation of the dynamic connection by the output parameter
- **requiredConnectionTester** – exchange of the dynamic connection by the output parameter
- **provider** – destruction of the dynamic connection by the output parameter

Reconfiguration by method – first variant

- **providedConnectionTester** – creation of the dynamic connection by the output parameter.
- **providedConnectionTester** – using the close method of the dynamic connection to destroy the connection.

Reconfiguration by method – second variant

- **provider** – creation of the dynamic connection by the input parameter.
- **requiredConnectionTester** – using the close method of the dynamic connection to destroy the connection.

Reconfiguration by temporary connection

- **provider –** creation of the temporary connection.

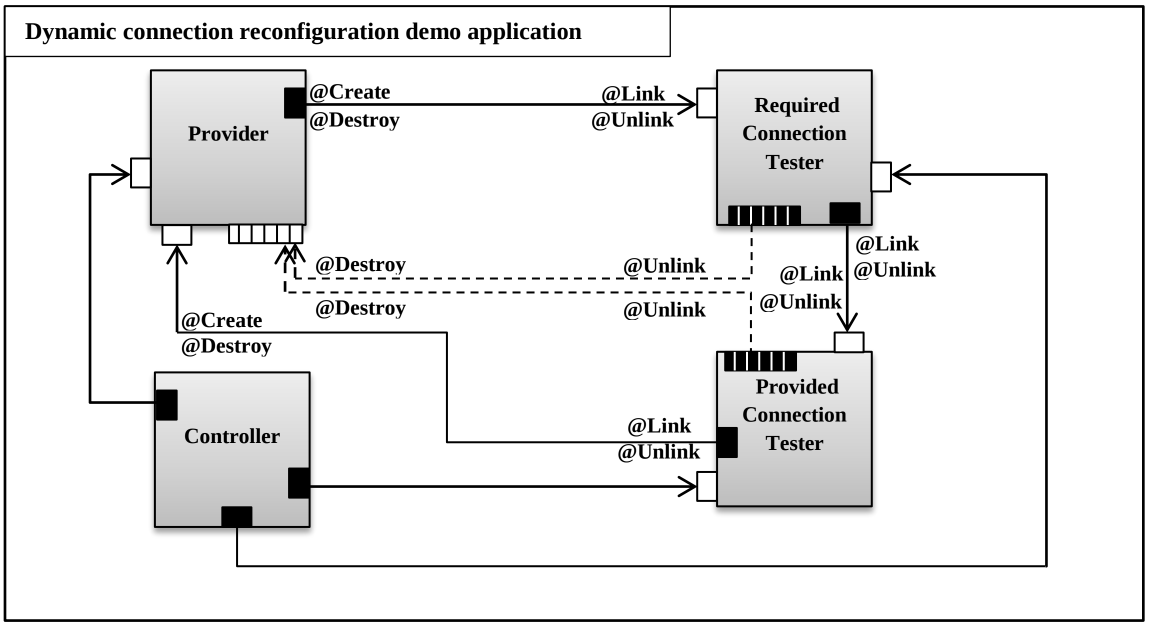

Figure 6.3 - Dynamic connection reconfiguration demo applications architecture

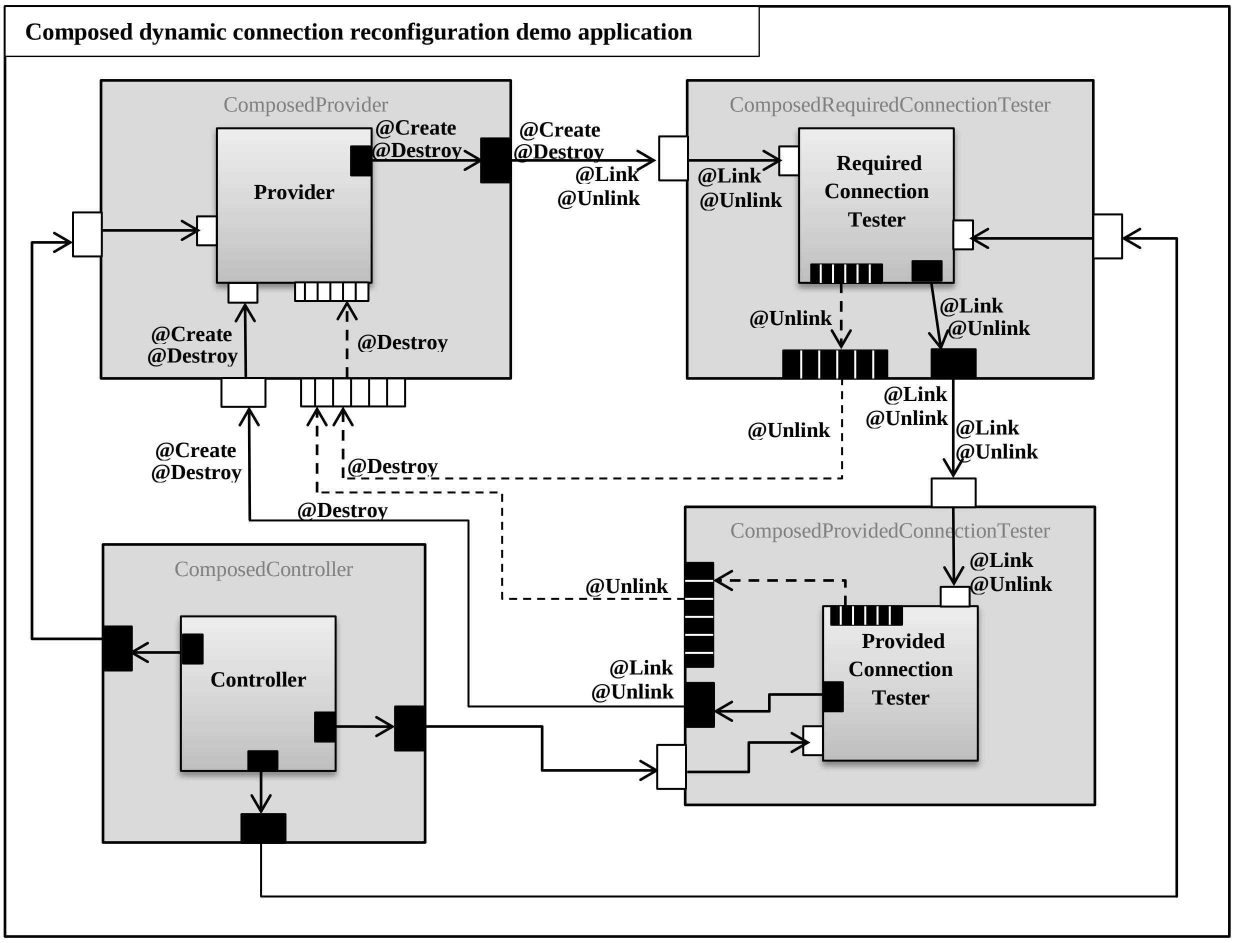

Figure 6.4 - Composed dynamic connection reconfiguration demo applications architecture

6.3 Performance results

Even though the evaluation of both presented demo applications was successful, there was still requirement of a proof, that the dynamic reconfigurations are suitable for regular usage. The only reason, which could cause the inapplicability of the dynamic reconfigurations, could be their long execution times. For this purpose the examples were extended by the duration measurement of each dynamic reconfiguration. Each dynamic reconfiguration variant in the basic dynamic reconfigurations examples was measured for at least thousand times and the results were written into a file. The result files were added to the thesis as an Appendix C and the mean values of each dynamic reconfiguration are presented in the following Table 6.1.

	Mean [ms]	Min [ms]	Max [ms]
Basic method call	0.246	0	4
Create by input parameter	8.369	7	401
Create by output parameter	8.3485	7	112
Create temporary connection	8.512	7	22
Exchange by input parameter	4.378	3	9
Exchange by output parameter	4.018	3	15
Destroy by input parameter	0.913	0	84
Destroy by output parameter	0.854	0	3
Destroy by method	0.776	0	6
Creation of dynamic component by factory method	223.235	185	4728
Creation of dynamic component for internal purposes	220.623	187	3181

Table 6.1 - The performance measurement of the dynamic reconfiguration

Used platform:
Windows 7 Enterprise SP1 64-bit Operating System
Intel® Core™ Duo CPU T9550 @ 2.66GHz
4GB RAM
Java™ SE Runtime Environment (build 1.6.0_21-b07)
Java HotSpot™ 64-Bit Server VM (build 17.0-b17, mixed mode)

As can be seen from the results the execution of the dynamic reconfigurations is slower compared to the basic method call, however, all the dynamic reconfigurations happened in under a second.

The slowest reconfiguration is the creation of `factory` component, which makes sense, since it requires a lot of handling. The execution time of the dynamic component creation is a lot dependent on the complexity of the dynamic component and therefore more complex components would cause a higher execution time.

The other results do not differ so much from the basic method call and probably would not be even noticed in a common application. However, these results are only from a local demo application without the use of component hierarchy, which would increase the execution time of both the basic method and the methods with dynamic reconfigurations. This is because for each delegation and subsumption between the components hierarchies there are the same amount of equal procedures, which have to be executed. Because of that, the execution times of the method with dynamic reconfiguration will always stay in the same ratio to the basic method execution time. Thanks to these results, there can be safely assumed, that the dynamic reconfigurations will be usable for any component application, which is not entirely dependent on speed.

7 Basic usage

The correct usage of the dynamic reconfiguration first requires the knowledge of SOFA2 and its ADL, which can be achieved by reading Appendix B the SOFA2 users guide.

7.1 Factory pattern

7.1.1 ADL files

This section describes the extended XML code that can be added to the ADL files in order to use the dynamic reconfigurations.

Prerequisites

The use of the `factory` annotation firstly requires the specification of the dynamic components in the deployment plan and the assembly.

An example of the dynamic components specified in the deployment plan ADL file is presented below. The `name` attribute describes the name of the dynamic component and the `node` attribute is specifies the name of the deployment dock, which will host the dynamic component.

```
<?xml version="1.0" encoding="UTF-8"?>
<depl-plan name="..." node="nodeA">
      . . .
      <depl-subc name="factory" node="nodeA">
            <depl-dyn-inst name="dynamicComponent1" node="nodeB"/>
            <depl-dyn-inst name="dynamicComponent2" node="nodeA"/>
            . . .
      </depl-subc>
      . . .
</depl-plan>
```

The following frame shows an example of dynamic component specifications in the assembly. The `name` attribute describes the name of the dynamic component and the `arch` attribute denotes the architecture, which will be used by the dynamic component.

```
<?xml version="1.0" encoding="UTF-8"?>
<assembly name="..." top-level-arch="...">
      . . .
      <subcomponent name="factory" arch="...">
            <dynamic-instance name="dynamicComponent1"
                  arch="..." />
            <dynamic-instance name="dynamicComponent2"
                  arch="..." />
            . . .
      </subcomponent>
      . . .
</assembly>
```

Frame annotation

The `factory` frame annotation contains following attributes:

- Instance-name (required)
  - The name of the dynamic component.
- Return-interface (required)
  - The interface, which will be used for the connection of the dynamic component.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<frame name="...">
      <factory instance-name="dynamicComponent1"
            return-itf="dynamicComponentInterface" />
      <factory instance-name="dynamicComponent2"
            return-itf="dynamicComponentInterface " />

      <!-- interface providing factory methods -->
      <provides name="factory" itf-type="..."
            collection="false">

      <!-- interface used for dynamic components connected to self -->
      <requires name="dynamicComponentInterface" itf-type="..."
            collection="true">

      . . .

</frame>
```

Interface annotation
The `factory` interface annotation contains following attributes:

- Method (required)
    - Specifies the factory method.
- Instance-name (optional)
    - The name of the dynamic component.
- Return-interface (required)
    - The interface, which will be used for the connection of the dynamic component.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<frame name="...">
      <!-- specification of the connector for the dynamic connection -->
      <requires name="dynamicComponentInterface" itf-type="..."
            collection="true"/>

      <!—interface providing the factory methods -->
      <requires name="factory" itf-type="..."
            collection="false">

            <!-- factory annotation -->
            <factory method="createDynamicComponent1"
                  instance-name=" dynamicComponent1"
                  return-itf="dynamicComponentInterface" />
            <factory method="createDynamicComponent2"
                  instance-name=" dynamicComponent2"
                  return-itf="dynamicComponentInterface" />
      </requires>

      . . .
</frame>
```

7.1.2 Component API

This section describes the API provided to the `factory` component by the micro-component.

Prerequisites
The use of the `factory` components API firstly requires a public field of the `SOFAComponentFactory` type and annotated by the `@FactoryProvider` annotation. This field needs to be in the component's main class and it will be used for invoking the logic of the dynamic component creation.

SOFAComponentFactory usage
The `SOFAComponentFactory` interface specifies two methods. The first method is called `createDynamicComponentWithConnection` and is used for the creation of the dynamic component and returning the connector, which is determined for the factory method. It has two `java.lang.String` parameters. The first parameter specifies the component name. The second parameter specifies the interface name of the connection required. The output

parameter of the method is business required shared connector converted to a `java.lang.Object`.

The second method is `createDynamicComponentConnectedToSelf` and it creates dynamic component connected to the `factory` component. It uses the same parameters as the previous method and the output parameter is `java.lang.Object`, which can be converted to the interface specified by the interface name parameter. After the conversion the returned object is used to communicate with the dynamic component.

If the field providing the component `factory` logic is used for a dynamic reconfiguration, which is not specified by the `factory` annotations, the result of the incorrect method will be a `null` value. The error message about the failure will be written to the log.

Example

```
package ...;

import org.objectweb.dsrg.sofa.SOFAComponentFactory;
import org.objectweb.dsrg.sofa.annotations.FactoryProvider;

. . .

public class Factory {

      @FactoryProvider
      public SOFAComponentFactory componentFactory;

      . . .

      public Object createDynamicComponent1() {
            return componentFactory.createDynamicComponentWithConnection(
                  dynamicComponent1, dynamicComponentInterface);
      }

      public Object createDynamicComponent2() {
            return componentFactory.createDynamicComponentWithConnection(
                  dynamicComponent2, dynamicComponentInterface);
      }

      private DynamicComponentInterface createDynamicComponent1ToSelf() {
            Object dynamicComponent =
                  componentFactory.createDynamicComponentConnectedToSelf(
                        dynamicComponent2, dynamicComponentInterface);
            return (DynamicComponentInterface) dynamicComponent;
      }

      . . .

}
```

7.2 Component destruction

7.2.1 Self-shutting annotation

This annotation does not contain any parameters and it is used as a frame attribute.

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<frame name="..." self-shutting="true">
. . .
</frame>
```

7.2.2 Component API

Prerequisites

The functionality of the `self-shutting` aspect is invoked to the object by the `SOFASelfShutting` public field, which needs to be annotated by the `@SelfShutter`. If these two conditions are met, then this object will contain the functionality of the `self-shutting` micro-component.

Usage

The `SOFASelfShutting` interface specifies only one method, which is shutdown. After calling this method, the component will destroy itself.

Example

```
package ...;

import org.objectweb.dsrg.sofa.SOFASelfShutting;
import org.objectweb.dsrg.sofa.annotations.SelfShutter;

. . .

public class ComponentForDestruction {

      @SelfShutter
      public SOFASelfShutting shutter;

      . . .

      public Object shutMeDown() {
            shutter.shutdown();
      }

      . . .
}
```

7.3 Dynamic connection reconfiguration

7.3.1 Create annotation

The `create` annotation contains following attributes:

- Method (required)
    - Specifies the method, which creates the dynamic connection.
- Interface (required)
    - The interface of the dynamic connection.
- Parameter (optional)
    - Default value is zero.
    - The number of the methods parameter, which contains the provided object. Zero is for output parameter and the numbers 1..N are for the input parameters.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<frame name="...">
      <!-- specification of the connector for the dynamic connection -->
      <provides name="dynamicConnectionInterface" itf-type="..."
            collection="true"/>

      <!-- creation via input parameter -->
      <requires name="..." itf-type="..." collection="false">
            <!-— create annotations -->
            <create method="createConnectionWithInputParameter"
                  interface="dynamicConnectionInterface" parameter="1" />
      </requires>

      <!-- creation via output parameter -->
      <provides name="..." itf-type="..." collection="false">
            <!-— create annotations -->
            <create method="createConnectionWithOutputParameter"
                  interface="dynamicConnectionInterface" />
      </provides>
      . . .
</frame>
```

7.3.2 Link annotation

The `link` annotation contains following attributes:

- Method (required)
    - Specifies the method, which links the dynamic connection.
- Interface (required)
    - The interface of the dynamic connection.
- Parameter (optional)
    - Default value is zero.
    - The number of the method parameter, which contains the business required shared connector for the link. Zero is for output parameter and the numbers 1..N are for the input parameters.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<frame name="...">
      <!-- specification of the connector for the dynamic connection -->
      <requires name="dynamicConnectionInterface" itf-type="..."
            collection="true"/>

      <!-- link via input parameter -->
      <provides name="..." itf-type="..." collection="false">
            <!-— link annotations -->
            <link method="linkConnectionWithInputParameter"
                  interface="dynamicConnectionInterface" parameter="1" />
      </provides>

      <!-- link via output parameter -->
      <requires name="..." itf-type="..." collection="false">
            <!-— link annotations -->
            <link method="linkConnectionWithOutputParameter"
                  interface="dynamicConnectionInterface" />
      </requires>
      . . .
</frame>
```

7.3.3 Unlink annotation

The `unlink` annotation contains following attributes:

- Method (required)
    - Specifies the method, which unlinks the dynamic connection.
- Interface (optional)
    - The interface of the dynamic connection.
    - Can be left out only when unlinking own connection.
- Parameter (optional)
    - Default value is zero.
    - The number of the method parameter, which contains the reference to the connected object. Zero is for output parameter and the numbers 1..N are for the input parameters.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<frame name="...">
      <!-- specification of the connector for the dynamic connection -->
      <requires name="dynamicConnectionInterface" itf-type="..."
            collection="true">
            <!-- unlink by calling method -->
            <unlink method="close" />
      </requires>

      <requires name="..." itf-type="..." collection="false">
            <!-— unlink via input parameter -->
            <unlink method="unlinkConnectionWithInputParameter"
                  interface="dynamicConnectionInterface" parameter="1" />
      </requires>
      <provides name="..." itf-type="..." collection="false">
            <!-- unlink via output parameter -->
            <unlink method="unlinkConnectionWithOutputParameter"
                  interface="dynamicConnectionInterface" />

            <!-- unlink temporary connection -->
            <link method="createTemporaryConnection"
                  interface="dynamicConnectionInterface" parameter="1" />
            <unlink method="createTemporaryConnection"
                  interface="dynamicConnectionInterface" parameter="1" />
      </provides>

      . . .
</frame>
```

7.3.4 Destroy annotation

The `destroy` annotation contains following attributes:

- Method (required)
  - Specifies the method, which destroys the dynamic connection.
- Interface (optional)
  - The interface of the dynamic connection.
  - Can be left out only when destroying own connection.
- Parameter (optional)
  - Default value is zero.
  - The number of the method parameter with the business required shared connector for destruction. Zero is for output parameter and the numbers 1..N are for the input parameters.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<frame name="...">
      <!-- specification of the connector for the dynamic connection -->
      <provides name="dynamicConnectionInterface" itf-type="..."
            collection="true">
            <!-- destroy by calling method -->
            <destroy method="close" />
      </provides>

      <provides name="..." itf-type="..." collection="false">
            <!-— destroy via input parameter -->
            <destroy method="destroyConnectionWithInputParameter"
                  interface="dynamicConnectionInterface" parameter="1" />
      </provides>
      <requires name="..." itf-type="..." collection="false">
            <!-- destroy via output parameter -->
            <destroy method="destroyConnectionWithOutputParameter"
                  interface="dynamicConnectionInterface" />

            <!-- destroy temporary connection -->
            <create method="createTemporaryConnection"
                  interface="dynamicConnectionInterface" parameter="1" />
            <destroy method="createTemporaryConnection"
                  interface="dynamicConnectionInterface" parameter="1" />
      </requires>

      . . .
</frame>
```

8 Related work

8.1 Darwin

Darwin [**10**] is one of the ADLs, that support component hierarchy and describe interfaces and component bindings. It provides dynamic reconfiguration using well specified dynamic components, lazy components and dynamic binding.

8.1.1 Lazy components

These components are acting as a static component with a slight difference in their instantiation time. They are not referenced directly. Between the lazy component and the calling component there is a pipe and this pipe is waiting for the first call to the lazy component. After the first call it takes care of its initialization and propagates the call to it. This way, the lazy component is initialized only when needed, which can effectively speed up the startup of the application.

The use of lazy components has its limitations. They must be well specified like static components, which mean bindings to other components must be set, and there cannot be instantiated an array of lazy components. The other limitation is that, once the lazy component is initiated, it loses its dynamicity and completely resembles the static component.

8.1.2 Direct dynamic instantiation

The other dynamic instantiation provided is called direct. It allows creating an array of dynamic components just by specifying the components bindings. This feature very closely resembles the desired dynamic reconfiguration, except for one issue, which makes the dynamic components unusable for some cases.

The problem lies in the inability to use provided bindings in dynamic components, except the one binding, which is responsible for creating the dynamic component. Darwin cannot express an array of required bindings to dynamic components and therefore provided bindings in a dynamic component cannot be used. This is very limiting because the dynamic components cannot be used as server components.

8.1.3 Dynamic binding

The bindings in Darwin are solved as many to one, in which means that many client components can be bound to one server component. The server components broadcast their signal from their services and client components are listening on specified port waiting for the signal. The same way is used to handle the dynamic bindings. There must be created specifications of bindings, which are exported (as in broadcasted) and then imported.

This resolves the issue of creating many dynamic components, which have required bindings and need to listen to static components. However this way it is possible to create dynamic bindings that do not respect the component hierarchy. Any component included in the architecture can import binding from any other component, which exports it. This behavior is forbidden by the requirements of the thesis and therefore cannot be used as an inspiration to implementation proposal.

8.2 Wright

One of the other known ADLs, which chose different approach to dynamic reconfigurations, is Wright [**1**]. Wright describes architectural configurations and architectural styles. Its characteristic is the use of explicit independent connector types, which are described by CPS[1]-like notation. However this notation is very limiting for the use of dynamic reconfiguration. The CPS is limited to systems with static process structure, which is why Wright chose to ignore dynamic reconfigurations and supports only static architectures.

As written in [**1**] dynamism can be simulated by a static architecture and therefore dynamic reconfiguration is not needed. The only difference is that dynamic components go through phases of instantiation and destruction at run-time and can be replaced by static components, which carries out the same purpose. This opinion is correct, but it comes with a lot of limitations. Due to creating many static components to simulate the dynamic ones, the application can become many times larger and will represent the worst possible case scenario of dynamic reconfiguration. This approach also dictates that there may be a large amount of components, that are not used during the run-time at all or just for a small part of the run-time. This inefficiency badly reflects on performance of the applications and can cause problems for large scale applications.

8.3 Chemical Abstract Machine (CHAM)

CHAM [**3**] is an ADL, which was inspired by chemicals and chemical reactions. Chemicals consist of molecules, which can participate in reactions. The reaction between two chemicals can occur only if some kind of rules is met and if the two chemicals are compatible. This concept can be transformed into architecture description by replacing the molecules with components and chemical reactions with component bindings, which have similar behavior. The only thing left is to specify the rules responsible for interaction between components.

Rules in CHAM are transformational and use Gamma (Γ) formalism. They consist of two parts. The first part represents the precondition and the second represents the transformation. The precondition usually represents a state, which must be fulfilled by the component, in order to undertake the transformation. The transformation represents any kind of dynamic reconfiguration or interaction between components. Rules are executed non-deterministically and in parallel, which means, that event though the condition in the rule is met, the transformation does not have to occur. This may happen because of the existence of another rule, which was non-deterministically preferred and changed the state of the component.

[1] Communication Sequential Processes (CSP) is a formal language for describing patterns of interaction in concurrent system. [**18**]

One of the other inspirations from nature were the states of the components. CHAM accepts three types of states: heating, cooling and inert. When the component is in heating state, than it decomposes, creating an output to other components. The cooling state is logically complementary to the heating state and the inert state represents, that the component is doing nothing and no rule can be applied to the component. These states ensure that components can receive or send information through the binding only when ready.

The architecture description performed by formal rules is very powerful. It can describe both events, which are responsible for dynamic reconfigurations, and the reconfigurations themselves. At first it may seem like there is no need to look for another type of ADL, because CHAM is able to solve all issues of dynamic reconfiguration. Formal rules do solve the issues, but they have a serious disadvantage. They are a little bit over-specified and require a large number of rules just to describe a simple architecture. Unfortunately software architecture dictates that the more rules it has, the less transparent is its behavior.

8.4 Object Management Group (OMG)

OMG [**4**] uses model driven architecture and has chosen a completely different approach than the other mentioned ADLs. Since it does not use dynamic reconfiguration constrains and does not completely support component hierarchy, it will be mentioned only briefly.

The reason, why component hierarchy is not completely supported, even though they can be hierarchically described in deployment plan, is because the hierarchy is flattened during deployment. The flattened architecture simplifies dynamic reconfigurations, because it takes away the need to decide where to create new components and bindings do not have to be connected via composite components. It also makes the evolution of the architecture less readable. Any dynamic reconfiguration made in the architecture may cause the disability to transform the flattened architecture back to hierarchical architecture. This makes the evolution gap even more significant, because it cannot be compared to the initial configuration.

8.5 ArchJava

ArchJava [**11**] is an extension to Java, which integrates ADL directly into Java implementation. The ADL of ArchJava adds new definitions such as components, ports and connection into Java, which are used to specify the architecture. Every class marked as a component may contain ports and connections, which bind subcomponents together. The port specifies required and provided methods of the component, where the provided methods must be implemented by the component and the required methods may be used in the implementation by the component. Ports, which are used to connect components, must be complementary, which means if one side of the port contains a required method, the other side must contain the same provided method. To ensure the respect to component hierarchy, a restriction was added, which allows the creation of connections between components only from a subcomponent to its immediate parent or a parent to its immediate subcomponent.

One of the biggest advantages of being an extension to Java is that ArchJava can use Java syntax and features to provide to the developer the same experience he is used to. ArchJava uses this advantage in dynamic reconfiguration. Developers, who worked with Java, are accustomed to using word “new” for creating classes. The same word can be used in a component to create a dynamic component at runtime. The created components do not have to be destroyed. They are garbage collected like any other class, when there is no connection leading to such component. ArchJava also supports the creation of dynamic connections at runtime. Connection is created the same way as a dynamic component using connecting ports as parameters. This way it enables a both-way communication from a component to an array of dynamic components, which is part of the requirements of the thesis and is not supported by the majority of existing approaches.

The main reason, why ArchJava does not fulfill goals of the thesis, is the constraint that dynamic connections can be only bound directly between two sibling components or by parent to subcomponent. This restriction disallows the creation of dynamic connection between cross hierarchy components, because there does not exist any way to delegate the connection.

The other less significant disadvantage is that ArchJava is not very extendable. Every extension requires nontrivial modifications to the compiler.

The last reason making the ArchJava not suitable for the thesis is the unwanted distribution of the software architecture specification. Because the architecture description is contained by every component, it cannot be kept at a single place. This way the architecture has to be traced through implementation files, which makes it more difficult to understand to behavior of the software.

8.6 Fractal

The component model in Fractal [**5**] [**6**] is very similar to SOFA2. It supports hierarchical component architecture, connecting components via provided and required interfaces and it is aspect based, which moves all additional extensibility into aspects keeping the model small and simple. The first difference between SOFA2 and Fractal is the application of the aspects. Unlike in SOFA2, which adds aspect functionality by annotating components or interfaces in ADL, the aspects in Fractal are added directly in code by inheriting aspect classes into business logic classes. This approach is obviously very limited, because Java does not support multiple inheritance and therefore only one aspect can be applied to a component. Fractal is very well aware of that and for this purpose it added a multiple inheritance support to Java, which allows applying as many aspects to component as needed.

The multiple inheritance support is done by generating special classes called “Mixin classes”. These classes are created by merging all inherited classes into one, which can access all their methods and fields. These classes are quite useful for example, when creating interceptors. They reduce the interceptor chains into one class, which provides all the required functionality.

Dynamic reconfigurations in Fractal are handled by aspects. They provide both creation/destruction of dynamic components and creation/destruction of dynamic bindings, which are handled by `GenericFactory` and `BindingController` aspects. Unfortunately Fractal also provides features like shared components and shortcut optimization, which violate the thesis goals.

A shared component is a component which can be directly bound to any component even though they are placed elsewhere in component hierarchy. This feature clearly violates the goal that all connections between components shall respect the hierarchy. It also creates an evolution gap in the architecture, because by binding more than one component to the shared component, it cannot be conclusively decided, which component is responsible for the instantiation of the shared component. This goal is also broken by shortcut optimization, which removes delegating and subsuming bindings between components and makes them direct in order to optimize the communication.

8.7 DiVA

DiVA [**12**] [**13**] is a project which provides framework for managing dynamic variability in flat component based systems. The framework is based on Aspect-Oriented and Model-Driven techniques.

The DiVA project is in many ways a similar to SOFA2. The concept of the implemented dynamic reconfigurations is also based on the aspects and the events responsible for the dynamic reconfiguration are specified in the design time of the architecture. The difference between the concept of dynamic reconfiguration for SOFA2 and DiVA is the concrete specification of the event responsible for the reconfiguration. The SOFA2 uses the connection endpoints specified by the frame of the component and the DiVA is using the whole interfaces. These methods provides full dynamic reconfiguration, however, the connection between the components have to be always realized. The connection annotations in SOFA2 have the advantage that they can only create the connector, which could be passed to any component and will not be connected, until it is affected by the `link` annotation.

The advantages of the DiVA project are in their validation and the support for rollback actions. The dynamic reconfiguration in SOFA2 assumes, that the components are well aware of what they are doing and when they destroy them self, then all the connections they provide have to be destroyed in order to avoid inconsistency.

8.8 MUSIC

The MUSIC [**14**] [**15**] (Mobile USers In Ubiquitous Computing) project is a middleware platform supporting the adaptation of the mobile applications in complex hierarchical model. The advantage of the MUSIC project is that it lets the component (mobile) assign a number of profiles specifying the dynamic reconfiguration. Then by the changes in the environment, there can be triggered the change of profile, which provides the reconfiguration.

9 Conclusion

The prototype implementation of the dynamic reconfigurations for SOFA2 component system fulfills all the expected requirements specified by the chapter 1.2 and 2.3. Thanks to the prototype the component system became more flexible and adaptable for the emerging application requirements. The annotations limiting the dynamic reconfigurations keep the applications evolution under control and thus fulfill the main goal of the thesis to remove the evolution gap. Thanks to that, the application behavior became more apparent just from the component architecture.

The prototype was evaluated on a few demo applications, which demonstrated that the dynamic reconfiguration prototype is stable and usable in the distributed applications and can handle complex hierarchy. Unlike the majority of the component systems, the prototype does not violate the components hierarchy by the dynamic reconfigurations, which makes the component system more clean and correct.

The usage of the dynamic reconfigurations was kept simple to avoid discouraging possible users and the speed of the dynamic reconfiguration was found acceptable for most component applications. The other features of SOFA2 and its basic functionality, except few renamed structures and optimizations, were kept intact and therefore any older applications should be still compatible with the prototype.

9.1 Future work

Since all the dynamic reconfiguration are completely supported, the future work may only focus on enhancements, such as the use of the strongly type parameters for the dynamic reconfigurations instead of `java.lang.Object`. This modification would slightly improve the work with the dynamic connections, since they would not have to be always converted to the interface they provide.

The other possible enhancement of the dynamic reconfiguration would be optimization of the dynamic component creation, since it is the slowest part of the dynamic reconfiguration.

10 Bibliography

[1] Allen Robert J., "A Formal Approach to Software Architecture," CMU-CS-97-144, Pittsburgh USA, May 1997.

[2] (2010, November) Darwin (ADL) - Wikipedia. [Online]. http://en.wikipedia.org/wiki/Darwin_(ADL)

[3] Paola Invevardi and Alexander L. Wolf, "Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model," IEEE Transactions on software engineering, vol. 21, no. 4, April 1995.

[4] "Deployment and Configuration of Component-based Distributed Applications Specification," OMG Document, formal/06-04-02, version 4, April 2006.

[5] Marc Léger, Thomas Ledoux, and Thierry Coupaye, "Reliable dynamic reconfigurations in the Fractal component model," ARM '07: International workshop on Adaptive and reflective middleware (6th), Proceedings, New York, USA, November 2007.

[6] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani, "The Fractal component model and its support in Java," Aug 2006.

[7] (2010, November) SOFA 2. [Online]. http://sofa.ow2.org

[8] (2011, February) Eclipse Modeling - EMF. [Online]. http://www.eclipse.org/modeling/emf/?project=emf

[9] (2011, February) ASM. [Online]. http://asm.ow2.org/

[10] Jeff Magee and Jeff Kramer, "Dynamic Structure in Software Architectures," San Francisco, USA, USA, October 1996.

[11] Jonathan Aldrich, Craig Chambers, and David Notkin, "ArchJava: Connecting Software Architecture to Implementation," Department of Computer Science and Engineering, University of Washington, Orlando, FL, USA, May 2002.

[12] B Morin, B Barais, O Nain, and Jean-Marc Jézéquel, "Taming Dynamically Adaptive Systems Using Models and Aspects," ICSE'09: 31st Internation Conference on Software Engineering, Vancouver, Canada, 2009.

[13] B Morin, O Barais, J Jézéquel, F Fleurey, and A Solberg, "Model@Run.Time to Support Dynamic Adaptation," IEEE Computer, 42(10):44-51, 2009.

[14] R Rouvoy, M Beauvois, L Lozano, J Lorenzo, and F Eliassen, "MUSIC: an Autonomous Platform Supporting Self-Adaptive," Proceedings of 1st International Middleware Workshop on Mobile Middleware: Embracing the Personal Communication Device (MobMid'08), ed. by Oriana Riva and Luís Veiga, pp. 6, Leuven, Belgium, 2008.

[15] M Alia, M Beauvois, Y Davin, R Rouvoy, and F Eliassen, "Components and Aspects Composition Planning for Ubiquitous Adaptive Services ," The 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Lille, France, 2010.

[16] Petr Hnětynka and František Plášil, "Dynamic Reconfiguration and Access to Services in Hierarchical Component Models," Dept. of SW Engineering, Charles University, Prague, Czech Republic, June 2006.

[17] (2010, November) Component-based software engineering - Wikipedia, the free encyclopedia. [Online]. http://en.wikipedia.org/wiki/Component-based_software_engineering

[18] (2010, November) Communicating sequential processes - Wikipedia, the free encyclopedia. [Online]. http://en.wikipedia.org/wiki/Communicating_sequential_processes

[19] Brice Morin, Franck Fleurey, Barais Olivier, and Jean-Marc Jézéquel, "Aspect-Oriented Modeling to Support Dynamic Adaptation," Forum Demo at AOSD'10, 2010.

11 Appendices

All appendices can be found on the enclosed CD.

11.1 Appendix A

SOFA2 programmers guide

`sofa2_programmers_guide.pdf`

11.2 Appendix B

SOF2 users guide

`sofa2_users_guide.pdf`

11.3 Appendix C

Dynamic reconfiguration measurement results

`measurement_results.zip`

11.4 Appendix D

The implementation of the dynamic reconfiguration prototype and the demo applications

`prototype.zip`

11.5 Appendix E

Electronic form of the thesis

`thesis.pdf`

11.6 Appendix F

The description of the enclosed CD contents

`contents.txt`