# JLCF User's and Developer's manual

Version: 1.0.0
Date: 22.02.2013
Author: Petros Pissias , http://jlcf.sourceforge.net

**Table Of Contents**

# 1 Introduction

## 1.1 What is JLCF

The Java Light Component Framework (JLCF) is a framework for developing modular applications for the java language. It allows designing an application using building blocks with well-defined inputs and outputs (aka software components). Using this well-defined modular approach, the application has a clear and explicit design which facilitates the development and maintenance of a software system.

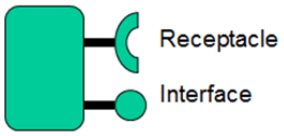
JLCF focuses on usability on a simple programming model and also provides advanced features such as the ability to replace components while the system is running (dynamic component replacement) in a way that tries to minimise the disturbance to the rest of the application.

## 1.2 JLCF Component Model

The following sections define the basic elements of JLCF.

### 1.2.1 Component

Components are the elementary building blocks of an application. Each component implements part of the application and formalizes the functionality it provides (through offered Interfaces) and the functionality it requires (through required interfaces (Receptacles) ).




All component-to-component communication must be done via their Receptacles and Interfaces.

#### *1.2.1.1 Interface*

A JLCF Interface is a plain Java interface. Components offer functionality through one or more Interfaces. This functionality can be used by another component or by a user of the framework.

#### *1.2.1.2 Receptacle*

A Receptacle is an interface requirement. Components require functionality through receptacles. During the instantiation of a component composition, Receptacles are connected by the framework to Interfaces in order to specify the provider of the functionality a component requires.

#### *1.2.1.3 Properties*

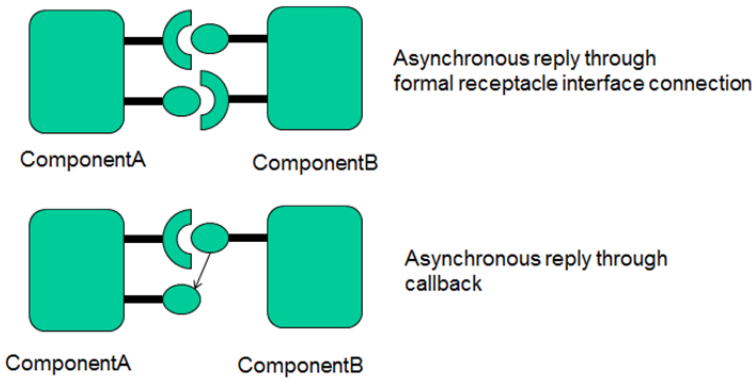
Each component can have one or more properties. These are name / value pairs that are specified externally and are passed to the component by the framework during its instantiation.

#### *1.2.1.4 Component Interaction patterns*

Components interact through their Interfaces and Receptacles using normal request / response synchronous method calls. Asynchronous calls between components may be implemented as 2 interactions between components, for example, `ComponentA` may call `ComponentB`, which will

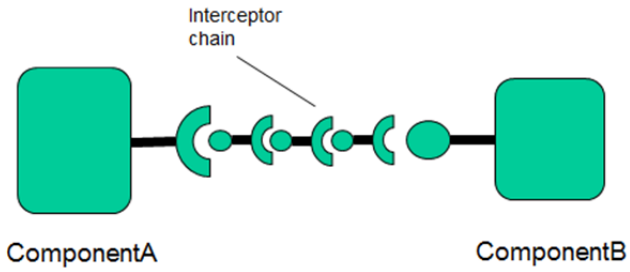
immediately return, will process the request and will later reply at an interface provided by `ComponentA`.

In addition, the framework provides the ability for one component to provide a callback reference when calling another component. The callback can be obtained on the target component and used to provide asynchronously one or more replies. Callbacks are further explained in the next sections.




### 1.2.2 Interceptor

An interceptor is a construct that is placed in receptacles and can intercept calls between components. An Interceptor can be used in order to intercept and change the target interface arguments or return data. Each receptacle may contain a chain of interceptors, which intercept the call one after the other.




### 1.2.3 Callback

A callback is an interaction pattern provided by the framework where a component may provide a callback address when calling another component. The Callback, which is a reference to a component interface, can be obtained from the target component and used to asynchronously provide one or more replies to the caller component.

### 1.2.4 Application Description

The application description is the configuration of the application in terms of a component composition, expressed in an xml document. This is a central concept of JLCF, all components and their interconnection are specified centrally in a configuration file. The framework is able to instantiate an application through an application description file which is provided by the framework user.

### 1.2.5 Distribution

The JLCF component mode is not a distributed component model. That is, it does not natively provide any mechanism interact with a JLCF Component based application running at another JVM locally or remotely. This part is intentionally left out and delegated to the application as there are many other frameworks and middleware that perform this task very well.

If there is a need to have communication between 2 remote JLCF applications, the communication between the boundary components of the 2 applications is done outside of the framework using any normal middleware or framework that is capable of doing this. This can be done either with interceptors or by having a proxy component that communicates with the remote side. These patterns are explained in more detail in the examples section.

## 1.3 Dynamic reconfiguration

While the application is running as a composition of components described in an application description, it is possible to instruct the framework to replace a component implementation with an alternative one. The application will continue to run and the framework will try to perform this reconfiguration transparently to the rest of the application.

At the current version, the framework only supports replacing one component at a time. If several components need to be replaced at runtime, several requests must be made to the framework sequentially.

# 2 User's Guide

## 2.1 General usage of the framework

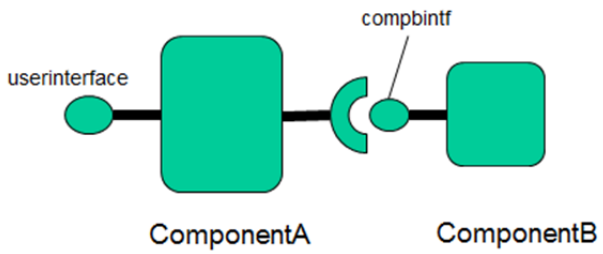
The definition and composition of the components that form an application is done in the *Application Description* file. This is an xml file which defines all components and their interconnections.

Typically a user of the framework will define an Application Description file, implement the components and will then instantiate and interact with the component composition through the API that the framework provides.

### 2.1.1 Application Description Files – Simple Example

A simple example of an Application Description below:

```
<?xml version="1.0" encoding="UTF-8"?>

<Application applicationName="Hello World Example" xmlns="http://jlcf.sourceforge.net/JLCFApplication"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

        <component implementationClass="ComponentA" name="compA">

                <interface name="userinterface" type="IComponentA" />

                <receptacle name="to_comp_B" >
                        <Reference path="compB/compbintf" />
                </receptacle>

                <property name="version" value="1.01"/>
        </component>


        <component implementationClass="ComponentB" name="compB">
                <interface name="compbintf" type="IComponentB"/>
        </component>
</Application>
```




This application would then be instantiated and used by the JLCF framework with the following API calls:

```
IJLCFContainer runtime = JLCFContainer.getInstance();
runtime.loadApplication("HelloWorldExample.xml");
```

and with the following API call, a user of the framework may obtain a reference to an a component interface and start using it.

```
//get reference to component provided interface
```

```
IComponentA compA = runtime.getComponentReference("compA/userinterface");
```

Regarding the component definition, this example file defines an application which has 2 components, starting with the simplest definition, compB is defined as follows:

```
<component implementationClass="ComponentB" name="compB">
        <interface name="compbintf" type="IComponentB"/>
</component>
```

In the component description we need to specify the implementation class that implements the component. This component offers a formal interface to other components or users of the framework. This is specified in the <interface> element. The interface has a name and type, which defines the Java Interface type. As with all formal interfaces of a component, the implementing class of a component needs to implement the corresponding Java interface.

The interface definition is

```
public interface IComponentB {

        public String someMethod(String input);

}
```

which is a plain old Java interface with no extra annotations.

The Component implementation class is listed below

```
public class ComponentB implements IComponentB {

        private final Logger logger = Logger.getLogger(getClass());

        public ComponentB() {
        }

        @InitMethod
        public void init() {
                logger.info("Init method called");
        }

        @Override
        public String someMethod(String input) {
                logger.info("method called, with input :"+input);
                return input.toUpperCase();

        }
}
```

Besides the `@InitMethod` annotation, the component implementation has nothing special, except that it implements the formal interface *`IComponentB`* that the component offers. The framework follows a POJO approach and tries to minimize the overhead for the component developer.

### 2.1.2 Component initialization

The component developer can indicate a method that the framework will call as soon as the component is instantiated. This method is marked with the `@InitMethod` annotation. Such a method is typically used to initialize the component and may call other components. Components are initialized in the order that they are defined in the Application Description file so if a component calls another

component during its initialization phase, the other component must be defined previously. The method marked with `@InitMethod` shall not have any arguments.

### 2.1.3 Properties and Receptacles

The other component definition, compA is shown below:

```
<component implementationClass="ComponentA" name="compA">

        <interface name="userinterface" type="IComponentA" />

        <receptacle name="compoB" >
                <Reference path="compB/compbintf" />
        </receptacle>

        <property name="version" value="1.01"/>
</component>
```

In addition to an interface that the component offers, *IComponentA*, it defines a Receptacle which is an interface requirement.

```
<receptacle name="compoB" >
        <Reference path="compB/compbintf" />
</receptacle>
```

The receptacle is named "compoB" and has a reference element which points to a target interface offered by another component.

The reference has a target path, which follows the format `<component_name / component_interface_name>`, which in the example above, the receptacle is pointing to interface `compbintf` of compB.

The component compB also defines a value for a property :

```
<property name="version" value="1.01"/>
```

The Receptacles and properties of a component are injected through the constructor of the class that implements the component, when the framework instantiates the component. Each Receptacle and Property must be annotated using the `@Property` and `@Receptacle` annotations in order for the framework to be able to correctly identify each one and inject the correct references.

The property types on the constructor can be: String, Integer, Float, Double or Boolean. The framework will automatically cast the value to the appropriate type.

```
public interface IComponentA {

        public String doTask(String task);

}

public class ComponentA implements IComponentA{

        private final IComponentB compB;
```

```
        private final String version;
        private final Logger logger = Logger.getLogger(getClass());

        public ComponentA(@Receptacle(name="compoB")  IComponentB compB,
                        @Property(name="version") String version) {
                this.compB = compB;
                this.version = version;
        }

        @InitMethod
        public void callme() {
                logger.info("component "+getClass().getName()+" version:"+version+" initialized");
        }

        @Override
        public String doTask(String input) {
                String compBInput = input;
                logger.info("doTask called. Calling component B with input="+compBInput);
                String ret = compB.someMethod(compBInput);
                logger.info("received reply from component B :"+ret);
                return ret;
        }
}
```

The implementation of Component compA is shown above.

The constructor

```
        public ComponentA(@Receptacle(name="compoB")  IComponentB compB,
                        @Property(name="version") String version
```

must define which fields are of type Property and a Receptacle and their respective names. The constructor must not contain other arguments besides Receptacles and Properties as the framework will not be able to instantiate an instance of the component.

### 2.1.4 Interceptors

The framework allows placing user defined interceptors between component connections.

Interceptors are specified at the receptacle definition of a component in the application description.

```
<component implementationClass="ComponentA" name="compA">

        <interface name="userinterface" type="IComponentA" />

        <receptacle name="compoB" >
                <Reference path="compB/compbintf"  />
                <Interceptor name="compbinterceptor" type="StringCaseInterceptor" />
        </receptacle>

</component>

<component implementationClass="ComponentB" name="compB">
        <interface name="compbintf" type="IComponentB"/>
</component>
```

The definition above specifies that an interceptor, implemented by the user provided class `StringCaseInterceptor`, will intercept all calls done from this component's receptacle.

Interceptors must extend the Interceptor class and implement the target interface of the receptacle, which in this case is `IComponentB`. The example below shows an interceptor that manipulates the String argument and return of a target method.

```
public class StringCaseInterceptor extends Interceptor implements IComponentB {


	@Override
	public String someMethod(String input) {
		IComponentB targetInterface = getTarget();
		String ret = targetInterface.someMethod(input.toUpperCase());
		return ret.toLowerCase();
	}
}
```

The programming model for interceptors is done by directly re-implementing the target interface and changing the arguments / return of the call. Access to the target interface is provided by the generic `getTarget` method which is provided by the `Interceptor` base class.

Interceptors have full freedom to do whatever they want with the input arguments of the method. It is even possible not to call the target interface and return something arbitrary.

### 2.1.5 Callbacks

When a component calls another component, via a receptacle – interface connection, it is possible to specify a callback interface that the target component may use. Callbacks are defined at the receptacle definition of a component, for example:

```
<receptacle name="compoB" >
	<Reference path="compB/compbintf" callbackReference="compA/callback"/>
</receptacle>
```

The definition above, will pass a callable reference to the target component that may be used to asynchronously reply one or more times or for whatever reason the application developer will decide. The callback reference can be obtained from the target component implementation via the generic methd `getCallback` provided by the framework.

```
ITestCallback cb = container.getCallback();
```

For example, below is an Interface method implementation that utilizes callbacks.

```
@Override
public void register() {
	ITestCallback cb = null;
	try {
		//obtain the callback, the framework provides the correct callback for this call
transparently
		cb = container.getCallback();
	} catch (ComponentReferenceException e) {
		logger.log(Level.ERROR, e.getMessage(), e);
		return;
	}
	if (cb != null) { //add the callback to this list of registered users
		registeredComponents.add(cb);
	}
}
```

The `container` reference is injected by the framework on instantiating the component.

The callback target interface (`ITestCallback` in this case) must be known in advance. Typically the target component will specify if it expects callbacks at an Interface and of which type. Then, calling components may specify which callbacks they will provide.

### 2.1.6 Dynamic Reconfiguration

The dynamic reconfiguration features of the framework allow to replace a component while the application is running. The framework follows a strategy in order to make the component reach a quiescent state. In such a state, the component is isolated from its surroundings and no calls will be made in its interfaces. In order to reach a quiescent state, the component must stop its internal processing (if any) in order to maintain a consistent internal state.

The algorithms of trying to assist a component to reach a quiescent state apply when the component is processing calls and at the same time there is a need to perform a reconfiguration. If the component is dormant (not serving and calls) and a reconfiguration is requested, it reaches a quiescent state immediately.

The API call in order to instruct the framework to perform a dynamic reconfiguration is

```
Pair<Boolean, String> singleComponentReconfguration(String component, String replacement, long millis)
```

The call will block until the reconfiguration succeeds or times out. The method expects the component name of the component to be replaced (compB for example), the new implementation class of the component and the timeout in milliseconds.

The call returns a pair containing a Boolean in order to indicate if the reconfiguration was successful and a String which contains additional information in case the reconfiguration fails.

A Component may optionally implement the *IReconfigurableComponent* interface

```
public interface IReconfigurableComponent {

        /**
         * Method called by the framework in order to inform a component to stop its internal processing
         * that may disrupt its quiscent state.
         * implementer should stop all threads that are able to make calls through receptacles.
         */
        public void stopAliveThreads();

        /**
         * Method called by the runtime in order to inform the component to proceed
         * its normal processing. This done when a reconfiguration cannot be reached within the desired
timeframe
         */
        public void proceed();

        /**
         * Method called by the framework when a quiescent state has been reached.
         * The components state is extracted in order to be passed to the new component.
         * @return the state of the component that should be passed on
         */
        public Object extractState();


        /**
```

```
         * Method called by the framework when a component is being replaced.
         * The old components state is inserted into the new component.
         * @param state the internal state
         */
        public void insertState(Object state);
}
```

in order to get notified when the runtime tries to bring the component in a quiescent state. The runtime will call the method `stopAliveThreads` when the reconfiguration process begins, in order to notify the developer that any internal processing must stop. If the reconfiguration attempt is unsuccessful, the framework will call the proceed method.

The interface also allows to transfer the old coponent's internal state to the new component, if it also implements the `IReconfigurableComponent` interface. If the component reaches a quiescent state, the framework will call the `extractState` method in order to get the internal state of the component. If the new component implements the interface, the framework will pass the data to the new component through the `insertState` method after it is instantiated. It is the responsibility of the new component to know the actual type of the data that is being passed.

The framework currently implements a heuristic mechanism on order to assist a component reaching a quiescent state. Besides keeping track all calls to the component interfaces in order to know if the component is currently serving calls, it keeps track of how long methods take to complete an selectively blocks calls directed to the component depending if the framework judges that they will complete in time. For example, if at a given time during the reconfiguration, the reconfiguration process will time out in 1500 milliseconds and the framework detects a method call that is expected to take 900 milliseconds to complete, it will allow the call to be made to the component.

There are other approaches for driving a component to a quiescent state but it is ultimately depending in the nature of the application and the current state of the application. If a component is serving a call that is blocked waiting for data that will never arrive, it will obviously not reach a quiescent state no matter what strategy the framework follows. The selected algorithm is optimistic, by allowing calls to be made to the component as much as possible. The idea is that these calls might help the component reach quiescent state. If a call is expected to take longer than the remaining time for the reconfiguration it is not allowed to proceed and is selectively blocked.

The framework is designed in order to allow other reconfiguration algorithms to be implemented. Please refer to the developer's manual section for more details.

### 2.1.7 Framework API

The `IJLCFContainer` interface that is used to interact with the framework is the following:

```
public interface IJLCFContainer {

        /**
         * Creates a complete application based on the application input file.
         * @param applicationFile applicationFile The input file describing the component composition
         * @throws ApplicationInstantiationException
         */
        public void loadApplication(String applicationFile) throws ApplicationInstantiationException;
```

```
        /**
         * Returns the callback associated with this call. The caller should
         * use the same thread that initiated the call and not call this method
         * from a new thread.
         *
         * @return the callback implementation
         * @throws ComponentReferenceException
         */
        public <T> T getCallback() throws ComponentReferenceException;


        /**
         * Generic method that returns a component interface to the requestor.
         * The requestors are: Component POJOs that want to get a callback address and Users of the
framework.
         * They are handled the same way.
         *
         * @param targetPath the target path, for example componentA/interfaceA
         * @return a component interface implementation that the user can call. This is actually a {@link
ReceptacleContextManager} with the target address. On the first call it will resolve the target address to
an actual component.
         * @throws ComponentReferenceException
         */
        public <T> T getComponentReference(String targetPath) throws ComponentReferenceException;

        /**
         * called by the user when a reconfiguration is needed
         * @param component the target component
         * @param millis the timeframe fo the reconfiguration
         * @return
         */

        /**
         * Starts a reconfiguration process.
         *
         * @param component the component name of the component to be replaced
         * @param replacement the class of the new component that will replace the old comopnent
         * @param millis the time-frame that the reconfiguration should be performed. If the time frame
elapses and the framework does not manage to replace the component the reconfiguration will fail.
         * @return pair of boolean indicating of the reconfiguration was successful and a String message
         * @throws Exception in case: The new component does not implement the formal interfaces of the
old component, in case the old component name cannot be found and in case the new component class cannot
be found.
         */
        public Pair<Boolean, String> singleComponentReconfguration(String component, String replacement,
long millis) throws Exception;
}
```

The framework provides methods in order to :

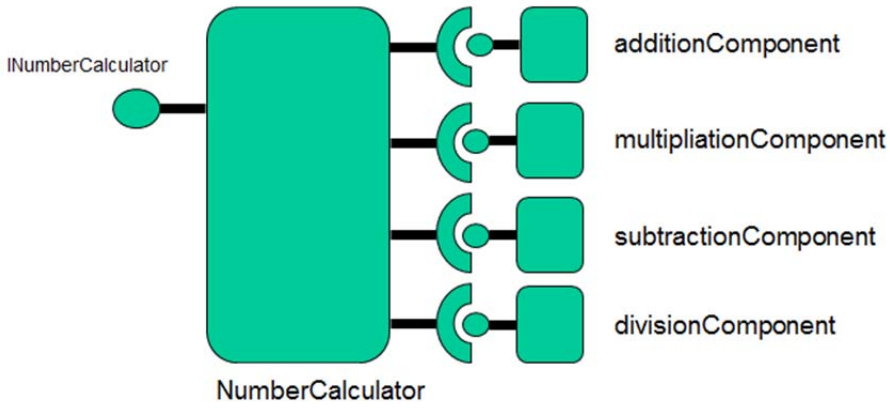
- Load an application (`loadApplication`)
- Get a component reference from a running application (`getComponentReference`)
- Get a callback reference – this method is used from within a component implementation (`getCallback`)
- Perform a component replacement (`singleComponentReconfiguration`)

## 2.2 Examples

The following examples are also available with the JLCF 1.0.0 download.

### 2.2.1 Simple Calculator

In this example, an application implementing a calculator will be created.




There are four components, implementing each operation (addition, multiplication, subtraction, division) and a central component that uses all of them and provides an interface to a user of the framework in order to do calculations.

The application description is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<Application applicationName="CalculatorExample" xmlns="http://jlcf.sourceforge.net/JLCFApplication"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">


        <component implementationClass="org.jlcf.example.simplecalculator.NumberCalculator"
name="NumberCalculator">
                <interface name="INumberCalculator"
type="org.jlcf.example.simplecalculator.INumberCalculator" />

                <receptacle name="addition">
                        <Reference path="additionComponent/addition"  />
                </receptacle>
                <receptacle name="multiplication">
                        <Reference path="multiplicationComponent/multiplication"  />
                </receptacle>
                <receptacle name="subtraction">
                        <Reference path="subtractionComponent/subtraction" />
                </receptacle>
                <receptacle name="division">
                        <Reference path="divisionComponent/division"  />
                </receptacle>
                <property name="version" value="1.01"/>
        </component>


        <component implementationClass="org.jlcf.example.simplecalculator.AdditionComponent"
name="additionComponent">
                <interface name="addition" type="org.jlcf.example.simplecalculator.IAddition" />
        </component>


        <component implementationClass="org.jlcf.example.simplecalculator.SubtractionComponent"
name="subtractionComponent">
                <interface name="subtraction" type="org.jlcf.example.simplecalculator.ISubtraction" />
```

```
        </component>


        <component implementationClass="org.jlcf.example.simplecalculator.AdditionComponent"
name="multiplicationComponent">
                <interface name="multiplication" type="org.jlcf.example.simplecalculator.IMultiplication"
/>
        </component>


        <component implementationClass="org.jlcf.example.simplecalculator.DivisionComponent"
name="divisionComponent">
                <interface name="division" type="org.jlcf.example.simplecalculator.IDivision" />
        </component>

</Application>
```

Each of the components that implement the operations offer a formal interface such as the one below,

```
public interface IAddition {

        public Double add(Double a, Double b);
}
```

Offered by the additionComponent, which has the following implementation;

```
public class AdditionComponent implements IAddition {

        @Override
        public Double add(Double a, Double b) {
                return a+b;
        }

}
```

As you can see from the implementation, the component is a POJO.

The NumberCalculator component, has four receptacles connected to the interfaces of each operation and offers the INumberCalculator interface for performing calculations.

```
public interface INumberCalculator {

        public Double doCalculation(String expression) throws Exception;

}
```

The NumberCalculator implementation is the following

```
public class NumberCalculator implements INumberCalculator{

	private final IMultiplication multComponent;
	private final IAddition addComponent;
	private final IDivision divComponent;
	private final ISubtraction subComponent;

	private final String version;

	private final Logger logger = Logger.getLogger(getClass());

	public NumberCalculator(
			@Receptacle(name="multiplication")  IMultiplication multComponent,
			@Receptacle(name="addition")  IAddition addComponent,
			@Receptacle(name="division")  IDivision divComponent,
			@Receptacle(name="subtraction")  ISubtraction subComponent,
			@Property(name="version") String version)

	{
		this.multComponent = multComponent;
		this.addComponent = addComponent;
		this.divComponent = divComponent;
		this.subComponent = subComponent;

		this.version = version;
	}

	@Override
	public Double doCalculation(String expression) throws Exception{
		//it is assumed that the expression string contains something in the form of X<operator>Y

		Double a,b;

		if (expression.contains("+")) { //get numbers and call addition component
			//parse expression and get a and b values
			…

			//get the receptacle and call the method on the other component
			return addComponent.add(a, b);

		}else if(expression.contains("-")) { //get numbers and call subtraction component
			//parse expression and get a and b values
			…

			//get the receptacle and call the method on the other component
			return subComponent.subtract(a, b);

			//parse expression and get a and b values
			…

			//get the receptacle and call the method on the other component
			return multComponent.multiply(a, b);

		}else if(expression.contains("/")) {
			//parse expression and get a and b values
			…

			//get the receptacle and call the method on the other component
			return divComponent.divide(a, b);
		}else {
			throw new Exception("Cannot parse exception");
		}
	}
```

```
	@InitMethod
	public void init() {
		logger.info("Init method called");
	}
}
```

The NumberCalculator constructor specifies the receptacles using the @Receptacle annotation and the component properties using the @Property annotation. Receptacles are identified by their name which must match the name of the Receptacle at the Application description. The framework will inject the receptacles when instantiating the component.

The component also implements a method that is annotated as an @InitMethod. The framework will call this method after instantiating the component. Typically init methods perform initializing actions on the component but in this case the method does not do anything.

The application is started as follows

```
//get instance of the JLCF runtime container
IJLCFContainer runtime = JLCFContainer.getInstance();

//load the application
runtime.loadApplication("resources/SimpleCalculatorExample.xml");

//obtain component interface reference
INumberCalculator numberCalcInterface    =
	runtime.getComponentReference("NumberCalculator/INumberCalculator");

//loop forever doing calculations
int i = 0, j = 4;
Double res;
while (true) {

	try {
		Thread.sleep(5000);
	}catch (InterruptedException yy) {

	}
	res = numberCalcInterface.doCalculation(i+"+"+j);
	logger.info(i+"+"+j+"="+res);

	i++;j++;
}
```

The first command, `JLCFContainer.getInstance()`, instantiates the framework and provides an interface that we can use to interact with the framework.

Next, the call `runtime.loadApplication("resources/SimpleCalculatorExample.xml");` will load the calculator application which is described as a component composition in the specified file.

Finally, the call `runtime.getComponentReference("NumberCalculator/INumberCalculator");` will return a reference to the Interface named "*INumberCalculator*" of the component named "NumberCalculator". This reference implements the formal component interface.

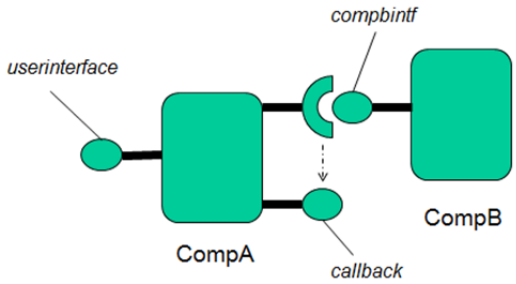
From then on, the example starts doing some calculations using the NumberCalculator component interface. The output of the example is:

```
2013-03-08 11:36:41,631 [main] INFO  org.jlcf.core.JLCFContainer  - JLCF container instance initialized
2013-03-08 11:36:41,631 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO
org.jlcf.core.util.AbstractQueueProcessorThread  - org.jlcf.core.util.AbstractQueueProcessorThread thread
started.
2013-03-08 11:36:41,756 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO
org.jlcf.core.JLCFFrameworkUtilities  - component:NumberCalculator instantiated
2013-03-08 11:36:41,756 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO
org.jlcf.core.JLCFFrameworkUtilities  - component:additionComponent instantiated
2013-03-08 11:36:41,756 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO
org.jlcf.core.JLCFFrameworkUtilities  - component:subtractionComponent instantiated
2013-03-08 11:36:41,756 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO
org.jlcf.core.JLCFFrameworkUtilities  - component:multiplicationComponent instantiated
2013-03-08 11:36:41,756 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO
org.jlcf.core.JLCFFrameworkUtilities  - component:divisionComponent instantiated
2013-03-08 11:36:41,756 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO
org.jlcf.core.JLCFContainerProcessor  - connecting NumberCalculator / addition ->
additionComponent/addition
2013-03-08 11:36:41,756 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO
org.jlcf.core.JLCFContainerProcessor  - connecting NumberCalculator / multiplication ->
multiplicationComponent/multiplication
2013-03-08 11:36:41,756 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO
org.jlcf.core.JLCFContainerProcessor  - connecting NumberCalculator / subtraction ->
subtractionComponent/subtraction
2013-03-08 11:36:41,756 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO
org.jlcf.core.JLCFContainerProcessor  - connecting NumberCalculator / division ->
divisionComponent/division
2013-03-08 11:36:41,756 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO
org.jlcf.example.simplecalculator.NumberCalculator  - Init method called
2013-03-08 11:36:46,764 [main] INFO  org.jlcf.example.simplecalculator.SimpleCalculatorExample  - 0+4=4.0
2013-03-08 11:36:51,788 [main] INFO  org.jlcf.example.simplecalculator.SimpleCalculatorExample  - 1+5=6.0
2013-03-08 11:36:56,797 [main] INFO  org.jlcf.example.simplecalculator.SimpleCalculatorExample  - 2+6=8.0
…
```

### 2.2.2 Callback

This example utilizes the callback interaction pattern provided by the framework

```
<?xml version="1.0" encoding="UTF-8"?>
<Application applicationName="Callback Example" xmlns="http://jlcf.sourceforge.net/JLCFApplication"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

	<component implementationClass="org.jlcf.example.callbacks.ComponentA" name="compA">

		<interface name="callback" type="org.jlcf.example.callbacks.ITestCallback" />
		<interface name="userinterface" type="org.jlcf.example.callbacks.IComponentA" />

		<receptacle name="compoB" >
			<Reference path="compB/compbintf" callbackReference="compA/callback"/>
		</receptacle>

		<property name="version" value="1.01"/>
	</component>


	<component implementationClass="org.jlcf.example.callbacks.ComponentB" name="compB">
		<interface name="compbintf" type="org.jlcf.example.callbacks.IComponentB"/>
	</component>

</Application>
```




The receptacle of compA specifies a callback reference, `callbackReference="compA/callback"`. At runtime, the framework will associate the callback address with the call context. On handling the call on compB, a reference to the callback interface can be obtained.

In order for the implementation of compB to obtain the callback reference, it needs to communicate with the JLCF framework. For that reason it utilizes an annotation (the `@ContainerRef` annotation) on its constructor that instructs the runtime to insert a reference when instantiating the component.

```
	//container reference
	private final IJLCFContainer container;

	public ComponentB(@ContainerRef IJLCFContainer container) {
		//initialize the list of clients
		registeredComponents = Collections.synchronizedList(new ArrayList<ITestCallback>());
		this.container = container;
	}
```

The callback can be obtained as follows on handling a call of the component's formal interface

```
	@Override
	//functional interface of the component. Used by other components that want to register for
receiving asynchronously information
	public void register() {
```

```
                ITestCallback cb = null;
                try {
                //obtain the callback,
                        cb = container.getCallback();
                } catch (ComponentReferenceException e) {
                        logger.log(Level.ERROR, e.getMessage(), e);
                        return;
                }
                if (cb != null) { //add the callback to this list of registered users
                        registeredComponents.add(cb);
                }
        }
```

In this example, compA registers for receiving information from compB. CompB stores the callback reference and periodically sends back information to compA. The full source code of the example is included in the JLCF distribution.

The output of the example is:

```
2013-03-09 09:24:46,157 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentB  - reply thread started
2013-03-09 09:24:46,172 [main] INFO  org.jlcf.example.callbacks.ComponentA  - version of component=1.01
doTask called. Registering to component B for receiving updates.
2013-03-09 09:24:48,169 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentB  - sending information to
all registered client callbacks
2013-03-09 09:24:48,169 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentA  - version of
component=1.01 received callback with information:100.0
2013-03-09 09:24:50,182 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentB  - sending information to
all registered client callbacks
2013-03-09 09:24:50,182 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentA  - version of
component=1.01 received callback with information:99.0
2013-03-09 09:24:52,196 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentB  - sending information to
all registered client callbacks
2013-03-09 09:24:52,196 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentA  - version of
component=1.01 received callback with information:98.0
2013-03-09 09:24:54,209 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentB  - sending information to
all registered client callbacks
2013-03-09 09:24:54,209 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentA  - version of
component=1.01 received callback with information:97.0
2013-03-09 09:24:56,223 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentB  - sending information to
all registered client callbacks
2013-03-09 09:24:56,223 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentA  - version of
component=1.01 received callback with information:96.0
2013-03-09 09:24:58,236 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentB  - sending information to
all registered client callbacks
2013-03-09 09:24:58,236 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentA  - version of
component=1.01 received callback with information:95.0
```

By a simple configuration change in the application description, we can add more components that register with compB. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<Application applicationName="Callback Example" xmlns="http://jlcf.sourceforge.net/JLCFApplication"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

        <component implementationClass="org.jlcf.example.callbacks.ComponentA" name="compA">

                <interface name="callback" type="org.jlcf.example.callbacks.ITestCallback" />
                <interface name="userinterface" type="org.jlcf.example.callbacks.IComponentA" />

                <receptacle name="compoB" >
                        <Reference path="compB/compbintf"
                                        callbackReference="compA/callback"/>
                </receptacle>
```

```
		<property name="version" value="flavour 1"/>
	</component>


	<component implementationClass="org.jlcf.example.callbacks.ComponentA" name="compA2">

		<interface name="callback" type="org.jlcf.example.callbacks.ITestCallback" />
		<interface name="userinterface" type="org.jlcf.example.callbacks.IComponentA" />

		<receptacle name="compoB" >
			<Reference path="compB/compbintf"
					callbackReference="compA2/callback"/>
		</receptacle>

		<property name="version" value="flavour 2"/>
	</component>

	<component implementationClass="org.jlcf.example.callbacks.ComponentA" name="compA3">

		<interface name="userinterface" type="org.jlcf.example.callbacks.IComponentA" />

		<receptacle name="compoB" >
			<Reference path="compB/compbintf"
					callbackReference="compA2/callback"/>     <!-- notice the
callback address, it is another component -->
		</receptacle>

		<property name="version" value="flavour 3"/>
	</component>

	<component implementationClass="org.jlcf.example.callbacks.ComponentB" name="compB">
		<interface name="compbintf" type="org.jlcf.example.callbacks.IComponentB"/>
	</component>

</Application>
```

In this example, we use 3 components that are connected to compB and provide difference callback references. The callback references do not have to be at an interface of the calling component itself, for example compA3 specifies a callback on compA2.

The output of this example is:

```
2013-03-09 09:29:11,069 [main] INFO  org.jlcf.example.callbacks.ComponentA  - version of component=flavour
1 doTask called. Registering to component B for receiving updates.
2013-03-09 09:29:11,069 [main] INFO  org.jlcf.example.callbacks.ComponentA  - version of component=flavour
2 doTask called. Registering to component B for receiving updates.
2013-03-09 09:29:11,069 [main] INFO  org.jlcf.example.callbacks.ComponentA  - version of component=flavour
3 doTask called. Registering to component B for receiving updates.
2013-03-09 09:29:13,066 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentB  - sending information to
all registered client callbacks
2013-03-09 09:29:13,066 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentA  - version of
component=flavour 1 received callback with information:100.0
2013-03-09 09:29:13,066 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentA  - version of
component=flavour 2 received callback with information:99.0
2013-03-09 09:29:13,066 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentA  - version of
component=flavour 2 received callback with information:98.0
2013-03-09 09:29:15,079 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentB  - sending information to
all registered client callbacks
2013-03-09 09:29:15,079 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentA  - version of
component=flavour 1 received callback with information:97.0
2013-03-09 09:29:15,079 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentA  - version of
component=flavour 2 received callback with information:96.0
2013-03-09 09:29:15,079 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentA  - version of
component=flavour 2 received callback with information:95.0
2013-03-09 09:29:17,093 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentB  - sending information to
all registered client callbacks
```

```
2013-03-09 09:29:17,093 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentA  - version of
component=flavour 1 received callback with information:94.0
2013-03-09 09:29:17,093 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentA  - version of
component=flavour 2 received callback with information:93.0
2013-03-09 09:29:17,093 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentA  - version of
component=flavour 2 received callback with information:92.0
2013-03-09 09:29:19,106 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentB  - sending information to
all registered client callbacks
2013-03-09 09:29:19,106 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentA  - version of
component=flavour 1 received callback with information:91.0
2013-03-09 09:29:19,106 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentA  - version of
component=flavour 2 received callback with information:90.0
2013-03-09 09:29:19,106 [Thread-2] INFO  org.jlcf.example.callbacks.ComponentA  - version of
component=flavour 2 received callback with information:89.0
```

### 2.2.3 Interceptor

This example uses an interceptor between components.

```
<?xml version="1.0" encoding="UTF-8"?>
<Application applicationName="Interceptor Example"
        xmlns="http://jlcf.sourceforge.net/JLCFApplication" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">

        <component implementationClass="org.jlcf.example.interceptors.ComponentA"
                name="compA">

                <interface name="userinterface" type="org.jlcf.example.interceptors.IComponentA" />

                <receptacle name="compoB">
                        <Reference path="compB/compbintf" />
                        <Interceptor name="compbinterc"
                                type="org.jlcf.example.interceptors.StringCaseInterceptor" />
                </receptacle>

                <property name="version" value="1.01" />
        </component>


        <component implementationClass="org.jlcf.example.interceptors.ComponentB"
                name="compB">
                <interface name="compbintf" type="org.jlcf.example.interceptors.IComponentB" />
        </component>

</Application>
```

Interceptors are defined at Receptacles and have full access to the inputs and outputs of method calls that belong to the interface of the target component. Interceptors must implement the target interface (*IComponentB* in this case) and extend a generic Interceptor class.

The interface of compB, IComponentB is the following

```
public interface IComponentB {

        public String someMethod(String input);

}
```

And the implementation is

```
        @Override
        public String someMethod(String input) {
                logger.info("method called, with input :"+input);
                return input+" My status is "+(status++);
        }
```

The interceptor StringCaseInterceptor implementation is

```
public class StringCaseInterceptor extends Interceptor implements IComponentB {

        private final Logger logger = Logger.getLogger(getClass().getName());

        @Override
        public String someMethod(String input) {
                logger.info("Intercepting call to someMethod, changing "+input+" to upper case");
                IComponentB targetInterface = getTarget();
                String ret = targetInterface.someMethod(input.toUpperCase());
                logger.info("Intercepting reply ("+ret+"), switching to lower case");
                return ret.toLowerCase();
```

```
        }
}
```

The interceptor accesses the target interface by calling the getTarget method that is provided by the Interceptor class. Then it has full access to the inputs and outputs of the target methods. In this example the interceptor simply changes the input string to upper case and the reply to lower case.

The output of the example is the following

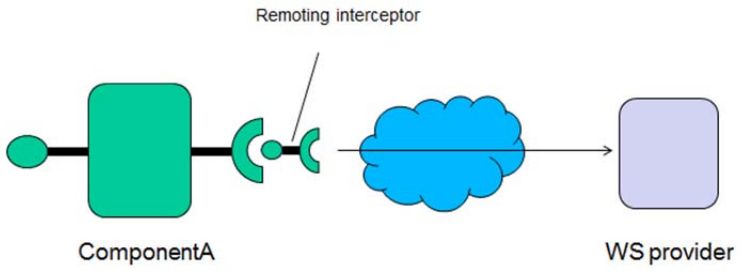
```
IJLCFContainer runtime = JLCFContainer.getInstance();
runtime.loadApplication("resources/InterceptorExample.xml");

IComponentA compA = runtime.getComponentReference("compA/userinterface");

while (true) {
        Thread.sleep(5000);
        compA.doTask("HellO");
}
```

```
2013-03-09 10:00:48,637 [main] INFO  org.jlcf.example.interceptors.ComponentA  - doTask called. Calling
component B with input=HellO
2013-03-09 10:00:48,637 [main] INFO  org.jlcf.example.interceptors.StringCaseInterceptor  - Intercepting
call to someMethod, changing HellO to upper case
2013-03-09 10:00:48,637 [main] INFO  org.jlcf.example.interceptors.ComponentB  - method called, with input
:HELLO
2013-03-09 10:00:48,637 [main] INFO  org.jlcf.example.interceptors.StringCaseInterceptor  - Intercepting
reply (HELLO My status is 100.0), switching to lower case
2013-03-09 10:00:48,637 [main] INFO  org.jlcf.example.interceptors.ComponentA  - received reply from
component B :hello my status is 100.0

2013-03-09 10:00:53,661 [main] INFO  org.jlcf.example.interceptors.ComponentA  - doTask called. Calling
component B with input=HellO
2013-03-09 10:00:53,661 [main] INFO  org.jlcf.example.interceptors.StringCaseInterceptor  - Intercepting
call to someMethod, changing HellO to upper case
2013-03-09 10:00:53,661 [main] INFO  org.jlcf.example.interceptors.ComponentB  - method called, with input
:HELLO
2013-03-09 10:00:53,661 [main] INFO  org.jlcf.example.interceptors.StringCaseInterceptor  - Intercepting
reply (HELLO My status is 101.0), switching to lower case
2013-03-09 10:00:53,661 [main] INFO  org.jlcf.example.interceptors.ComponentA  - received reply from
component B :hello my status is 101.0

2013-03-09 10:00:58,686 [main] INFO  org.jlcf.example.interceptors.ComponentA  - doTask called. Calling
component B with input=HellO
2013-03-09 10:00:58,686 [main] INFO  org.jlcf.example.interceptors.StringCaseInterceptor  - Intercepting
call to someMethod, changing HellO to upper case
2013-03-09 10:00:58,686 [main] INFO  org.jlcf.example.interceptors.ComponentB  - method called, with input
:HELLO
2013-03-09 10:00:58,686 [main] INFO  org.jlcf.example.interceptors.StringCaseInterceptor  - Intercepting
reply (HELLO My status is 102.0), switching to lower case
2013-03-09 10:00:58,686 [main] INFO  org.jlcf.example.interceptors.ComponentA  - received reply from
component B :hello my status is 102.0
```

### 2.2.4 Calling remote service with Interceptor

This example uses an interceptor to call a remote web service.

```
<?xml version="1.0" encoding="UTF-8"?>
<Application applicationName="Interceptor Example"
        xmlns="http://jlcf.sourceforge.net/JLCFApplication" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">

        <component implementationClass="org.jlcf.example.interceptors.remoting.ComponentA"
                name="compA">

                <interface name="userinterface"
                        type="org.jlcf.example.interceptors.remoting.IComponentA" />

                <receptacle name="remote">
                        <Reference path=""
                                type="org.jlcf.example.interceptors.remoting.IExternalInterface" />
                        <Interceptor name="compbinterc"
                                type="org.jlcf.example.interceptors.remoting.RemotingInterceptor" />
                </receptacle>

                <property name="version" value="1.01" />
        </component>


</Application>
```




The application description defines a component with a receptacle of type *IExternalInterface* which has an empty path. Receptacles normally have a path which is a target interface of a component, however in this special case the path is omitted as there is no target component. The interface type of the receptacle, *IExternalInterface*, is an arbitrary interface that we use to model the external service. It is up to the developer to define this interface. Depending on the middleware / remote service the remote service interface might have already been defined and thus could be re-used here. In any case, this interface is the way we see the external service in the application.

The implementation class of the interceptor is RemotingInterceptor.

In this example, the remote web-service is a simple web-service service exposed by java

```
@WebService
public interface RemoteWSInterface {
        @WebMethod String sayHello(String name);
}
```

The specific remote interface / transport / technology is not important in this example. The same pattern can be used to access any remote service, as long as it is accessible by your Java code.

The implementation of compA is the following:

```
public class ComponentA implements IComponentA{

	private final IExternalInterface remotingGateway;

	private final String version;

	private final Logger logger = Logger.getLogger(getClass());

	public ComponentA(
			@Receptacle(name="remote")  IExternalInterface remotingGateway,
			@Property(name="version") String version)
	{
		this.remotingGateway = remotingGateway;
		this.version = version;
	}

	@InitMethod
	public void callme() {
		logger.info("component "+getClass().getName()+" version:"+version+" initialized");
	}

	@Override
	public String doTask(String input) {
		String compBInput = input;
		logger.info("doTask called. Calling receptacle with input="+compBInput);
		String ret = remotingGateway.externalMethod(compBInput);
		logger.info("received reply :"+ret);
		return ret;
	}
}
```

And the implementation of IExternalInterface (which we use as a façade to the external service) is

```
public interface IExternalInterface {

	public String externalMethod(String argument);

}
```

The interceptor implementation is shown below

```
public class RemotingInterceptor extends Interceptor implements IExternalInterface {

	private final Logger logger = Logger.getLogger(getClass().getName());

	private final RemoteWSInterface remoteService = new
RemoteWSImplementationService().getRemoteWSImplementationPort();

	@Override
	public String externalMethod(String argument) {
		logger.info("Intercepting call to externalMethod, with input "+argument+" and forwarding
to web service");

		String ret = remoteService.sayHello("Hello");

		//proceess return if needed

		//return
		return ret;
	}
```

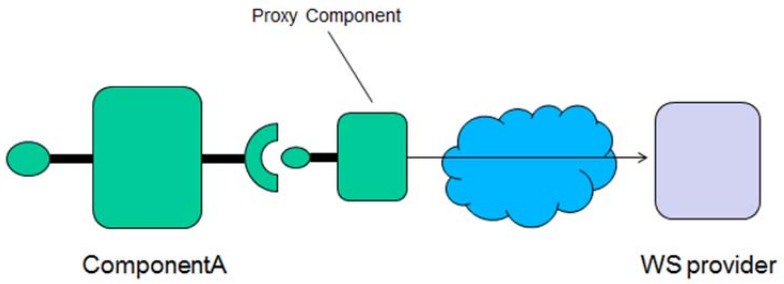
```
}
```

The interceptor calls the external web service and returns the data.

As noted before, in the general case, the interceptor would implement the logic in order to communicate with an external service using any kind of middleware.

### 2.2.5 Calling Remote Service with proxy component

Same as with interceptors, another way to interact with the external world is to use a proxy component.




The application description for this example is

```
<?xml version="1.0" encoding="UTF-8"?>
<Application applicationName="Interceptor Example"
        xmlns="http://jlcf.sourceforge.net/JLCFApplication" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">

        <component implementationClass="org.jlcf.example.remoting.ComponentA"
                name="compA">
                <interface name="userinterface" type="org.jlcf.example.remoting.IComponentA" />
                <receptacle name="remoteCompProxy">
                        <Reference path="remoteProxy/remoteinterface" />
                </receptacle>

                <property name="version" value="1.01" />
        </component>

        <component implementationClass="org.jlcf.example.remoting.RemoteProxyComponent"
                name="remoteProxy">
                <interface name="remoteinterface"
                        type="org.jlcf.example.remoting.IRemoteProxyComponent" />
        </component>

</Application>
```

In this scenario component `remoteProxy` is a boundary component that calls an external service.

Its implementation is very similar to the logic of the interceptor.

```
public class ComponentA implements IComponentA{

        private final IRemoteProxyComponent remProxyCpmp;
        private final Logger logger = Logger.getLogger(getClass());

        public ComponentA(@Receptacle(name="remoteCompProxy")  IRemoteProxyComponent remProxyCpmp) {
                this.remProxyCpmp = remProxyCpmp;
        }


        @Override
        public String doTask(String input) {
                String remProxyCpmpInput = input;
                logger.info("doTask called. Calling RemoteProxyComponent with input="+remProxyCpmpInput);
                String ret = remProxyCpmp.remoteCall(remProxyCpmpInput);
                logger.info("received reply from RemoteProxyComponent :"+ret);
                return ret;
```

```
        }
}
```

The output of the example is

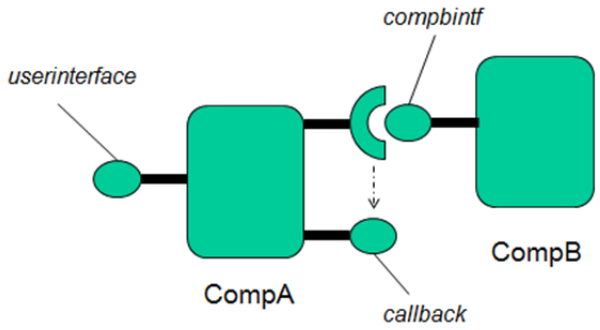
```
                //create the runtime container and get the single instance
                IJLCFContainer runtime = JLCFContainer.getInstance();
                //load the application
                runtime.loadApplication("resources/RemotingExample.xml");

                //get reference to ComponentA's "userinterface" interface
                IComponentA compA = runtime.getComponentReference("compA/userinterface");

                compA.doTask("HellO");
```

```
2013-03-09 10:46:32,403 [main] INFO  org.jlcf.example.remoting.ComponentA  - doTask called. Calling
RemoteProxyComponent with input=HellO
2013-03-09 10:46:32,403 [main] INFO  org.jlcf.example.remoting.RemoteProxyComponent  - doTask called.
Calling remote WS with input=HellO
2013-03-09 10:46:32,514 [main] INFO  org.jlcf.example.remoting.RemoteProxyComponent  - received reply from
remote WS :Hello from WS
2013-03-09 10:46:32,514 [main] INFO  org.jlcf.example.remoting.ComponentA  - received reply from
RemoteProxyComponent :Hello from WS
```

### 2.2.6 Dynamic Reconfiguration

In this example, an application is started and then a component is dynamically replaced with another implementation at runtime.

```
<?xml version="1.0" encoding="UTF-8"?>
<Application applicationName="Test"
        xmlns="http://jlcf.sourceforge.net/JLCFApplication" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">

        <component implementationClass="org.jlcf.example.reconfiguration.ComponentA"
                name="compA">
                <interface name="callback"
                        type="org.jlcf.example.reconfiguration.ITestCallback" />
                <interface name="userinterface"
                        type="org.jlcf.example.reconfiguration.IComponentA" />
                <receptacle name="compoB">
                        <Reference path="compB/compbintf" callbackReference="compA/callback" />
                </receptacle>

                <property name="version" value="1.01" />
        </component>


        <component implementationClass="org.jlcf.example.reconfiguration.ComponentB"
                name="compB">
                <interface name="compbintf"
                        type="org.jlcf.example.reconfiguration.IComponentB" />
        </component>

</Application>
```




The implementation of `compA` is `ComponentA`. At some point, the framework is asked to try and perform a reconfiguration, by changing the implementation class with a new one, `NewComponentA`

The application the very similar as the callback example. `CompA` registers with `CompB` and starts receiving callbacks.

The *`userinterface`* interface offered by `CompA` is

```
public interface IComponentA {

        public void register();

        public String doSomething(String input);

}
```

And the callback Interface

```
public interface ITestCallback {

	public String receiveCallback(String info);

}
```

The implementation of the interfaces on `ComponentA` is

```
	@Override
	public String receiveCallback(String info) {
		logger.info("component "+getClass().getName()+" received callback with
information:"+info);
		return "callback processed";
	}

	@Override
	public void register() {
		logger.info("register called. Registering to component B for receiving updates.");
		compB.register();
	}

	@Override
	public String doSomething(String input) {

		logger.info("received call to doSomething. Input:"+input);

		//simulate doing something that takes some time
		try {
			Thread.sleep(5000);
		} catch (InterruptedException e) {

		}
		logger.info("finished processing request for input:"+input);
		return "done";
	}
```

Method `doSomething`, simulates executing a time consuming task by sleeping and returning after 5 seconds.

The example scenario is the following:

Initially we call the register method, causing `compA` to register with `compB` and to start receiving callbacks from `compB`. Then we start a new thread every 0.5 seconds, which calls the `doSomething` method of `compA`. This causes `compA` to be active all the time. After 5 seconds we start a reconfiguration process asking the framework to replace the implementation class `ComponentA` with `NewComponentA`.

At the point the reconfiguration starts, `ComponentA` is serving 10 calls, with a new call coming every 0.5 seconds. The output of the example is the following

```
		final IJLCFContainer runtime = JLCFContainer.getInstance();
		runtime.loadApplication("resources/SimpleReconfigrationExample.xml");

		IComponentA compA = runtime.getComponentReference("compA/userinterface");

		compA.register();

		new Thread () {
```

```
                        public void run () {

                                //sleep for 5 seconds
                                try {
                                        Thread.sleep(5000);
                                } catch (InterruptedException e1) {

                                }
                                //by now the component is servicing at least 10 calls (see code at the
end)

                                //do a reconfiguration
                                try {
                                        logger.info("starting reconfiguration");
                                        //replace component compA with new implementation class
org.jlcf.example.reconfiguration.NewComponentA
                                        //set a timeout for the reconfiguration of 10 seconds
                                        runtime.singleComponentReconfguration("compA",
"org.jlcf.example.reconfiguration.NewComponentA", 10000);
                                } catch (Exception e) {
                                        logger.log(Level.ERROR, "exception during recofiguration", e);
                                }

                        }
                }.start();

                //do something on component A
                //every 500 millisecs we will call a method that needs 5000 millisecs to complete
                //this ensures that the component is fully busy while we do the reconfiguration
                while (true) {
                        //create new thread that calls component
                        Thread.sleep(500);
                        new ComponentCallerThread(compA).start();
                }
```

```
2013-03-09 11:32:20,773 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - thread started
2013-03-09 11:32:20,773 [main] INFO  org.jlcf.example.reconfiguration.ComponentA  - register called.
Registering to component B for receiving updates.
2013-03-09 11:32:21,288 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to
registered clients
2013-03-09 11:32:21,288 [Thread-4] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling
component method doSomething with input: INPUT1
2013-03-09 11:32:21,288 [Thread-4] INFO  org.jlcf.example.reconfiguration.ComponentA  - received call to
doSomething. Input:INPUT1
2013-03-09 11:32:21,288 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:100.0
2013-03-09 11:32:21,803 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to
registered clients
2013-03-09 11:32:21,803 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:99.0
2013-03-09 11:32:21,803 [Thread-5] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling
component method doSomething with input: INPUT2
2013-03-09 11:32:21,803 [Thread-5] INFO  org.jlcf.example.reconfiguration.ComponentA  - received call to
doSomething. Input:INPUT2
2013-03-09 11:32:22,303 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to
registered clients
2013-03-09 11:32:22,303 [Thread-6] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling
component method doSomething with input: INPUT3
2013-03-09 11:32:22,303 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:98.0
2013-03-09 11:32:22,303 [Thread-6] INFO  org.jlcf.example.reconfiguration.ComponentA  - received call to
doSomething. Input:INPUT3
2013-03-09 11:32:22,818 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to
registered clients
2013-03-09 11:32:22,818 [Thread-7] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling
component method doSomething with input: INPUT4
```

```
2013-03-09 11:32:22,818 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:97.0
2013-03-09 11:32:22,818 [Thread-7] INFO  org.jlcf.example.reconfiguration.ComponentA  - received call to
doSomething. Input:INPUT4
2013-03-09 11:32:23,318 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to
registered clients
2013-03-09 11:32:23,318 [Thread-8] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling
component method doSomething with input: INPUT5
2013-03-09 11:32:23,318 [Thread-8] INFO  org.jlcf.example.reconfiguration.ComponentA  - received call to
doSomething. Input:INPUT5
2013-03-09 11:32:23,318 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:96.0
….
….
….
```

At this point the reconfiguration request is processed by the framework

```
2013-03-09 11:32:25,785 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO
org.jlcf.core.dynrec.ConnectorTimingBasedReconfigurationManager  - connector received reconfiguration
message. reconfiguring:true
2013-03-09 11:32:25,785 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO
org.jlcf.core.dynrec.ConnectorTimingBasedReconfigurationManager  - connector received reconfiguration
message. reconfiguring:true
```

The framework at any given time monitors how many calls each component processes. When the reconfiguration process starts, an algorithm evaluates if a call should be made on a component or be temporarily blocked until the reconfiguration process terminates.

The current algorithm keeps track of how long methods take to complete and selectively blocks them if they are not likely to complete within the remaining time for the reconfiguration process.

All calls to method `receiveCallback` are allowed because they complete very fast. At some point the framework starts blocking calls to method `doSomething` because the remaining time for the reconfiguration is less than the expected time the call will take. Finally, as soon as the component processes all of its pending calls, it reaches a quiescent state and is replaced with the new implementation.

All the calls that were blocked during the reconfiguration are unblocked and reach the new component.

Below is the remaining output of the example

```
2013-03-09 11:32:25,863 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:25,863 [Thread-15] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT10
2013-03-09 11:32:25,863 [Thread-15] INFO  org.jlcf.example.reconfiguration.ComponentA  - received call to doSomething.
Input:INPUT10
2013-03-09 11:32:25,863 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:91.0
2013-03-09 11:32:26,301 [Thread-4] INFO  org.jlcf.example.reconfiguration.ComponentA  - finished processing request for
input:INPUT1
2013-03-09 11:32:26,363 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:26,363 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:90.0
2013-03-09 11:32:26,363 [Thread-16] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT11
2013-03-09 11:32:26,363 [Thread-16] INFO  org.jlcf.example.reconfiguration.ComponentA  - received call to doSomething.
Input:INPUT11
```

```
2013-03-09 11:32:26,816 [Thread-5] INFO  org.jlcf.example.reconfiguration.ComponentA  - finished processing request for
input:INPUT2
2013-03-09 11:32:26,878 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:26,878 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:89.0
2013-03-09 11:32:26,878 [Thread-17] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT12
2013-03-09 11:32:26,878 [Thread-17] INFO  org.jlcf.example.reconfiguration.ComponentA  - received call to doSomething.
Input:INPUT12
2013-03-09 11:32:27,316 [Thread-6] INFO  org.jlcf.example.reconfiguration.ComponentA  - finished processing request for
input:INPUT3
2013-03-09 11:32:27,378 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:27,378 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:88.0
2013-03-09 11:32:27,378 [Thread-18] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT13
2013-03-09 11:32:27,378 [Thread-18] INFO  org.jlcf.example.reconfiguration.ComponentA  - received call to doSomething.
Input:INPUT13
2013-03-09 11:32:27,831 [Thread-7] INFO  org.jlcf.example.reconfiguration.ComponentA  - finished processing request for
input:INPUT4
2013-03-09 11:32:27,893 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:27,893 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:87.0
2013-03-09 11:32:27,893 [Thread-19] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT14
2013-03-09 11:32:27,893 [Thread-19] INFO  org.jlcf.example.reconfiguration.ComponentA  - received call to doSomething.
Input:INPUT14
2013-03-09 11:32:28,331 [Thread-8] INFO  org.jlcf.example.reconfiguration.ComponentA  - finished processing request for
input:INPUT5
2013-03-09 11:32:28,393 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:28,393 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:86.0
2013-03-09 11:32:28,393 [Thread-20] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT15
2013-03-09 11:32:28,393 [Thread-20] INFO  org.jlcf.example.reconfiguration.ComponentA  - received call to doSomething.
Input:INPUT15
2013-03-09 11:32:28,846 [Thread-9] INFO  org.jlcf.example.reconfiguration.ComponentA  - finished processing request for
input:INPUT6
2013-03-09 11:32:28,908 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:28,908 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:85.0
2013-03-09 11:32:28,908 [Thread-21] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT16
2013-03-09 11:32:28,908 [Thread-21] INFO  org.jlcf.example.reconfiguration.ComponentA  - received call to doSomething.
Input:INPUT16
2013-03-09 11:32:29,346 [Thread-10] INFO  org.jlcf.example.reconfiguration.ComponentA  - finished processing request for
input:INPUT7
2013-03-09 11:32:29,408 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:29,408 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:84.0
2013-03-09 11:32:29,408 [Thread-22] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT17
2013-03-09 11:32:29,408 [Thread-22] INFO  org.jlcf.example.reconfiguration.ComponentA  - received call to doSomething.
Input:INPUT17
2013-03-09 11:32:29,861 [Thread-11] INFO  org.jlcf.example.reconfiguration.ComponentA  - finished processing request for
input:INPUT8
2013-03-09 11:32:29,923 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:29,923 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:83.0
2013-03-09 11:32:29,923 [Thread-23] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT18
2013-03-09 11:32:29,923 [Thread-23] INFO  org.jlcf.example.reconfiguration.ComponentA  - received call to doSomething.
Input:INPUT18
```

From this point the framework blocks calls to doSomething, as it estimates that they will not finish until the reconfiguration process completes.

```
2013-03-09 11:32:30,361 [Thread-12] INFO  org.jlcf.example.reconfiguration.ComponentA  - finished processing request for
input:INPUT9
```

```
2013-03-09 11:32:30,423 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:30,423 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:82.0
2013-03-09 11:32:30,423 [Thread-24] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT19
2013-03-09 11:32:30,423 [Thread-24] INFO  org.jlcf.example.reconfiguration.ComponentA  - received call to doSomething.
Input:INPUT19
2013-03-09 11:32:30,876 [Thread-15] INFO  org.jlcf.example.reconfiguration.ComponentA  - finished processing request for
input:INPUT10
2013-03-09 11:32:30,938 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:30,938 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:81.0
2013-03-09 11:32:30,938 [Thread-25] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT20
2013-03-09 11:32:31,376 [Thread-16] INFO  org.jlcf.example.reconfiguration.ComponentA  - finished processing request for
input:INPUT11
2013-03-09 11:32:31,438 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:31,438 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:80.0
2013-03-09 11:32:31,438 [Thread-26] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT21
2013-03-09 11:32:31,891 [Thread-17] INFO  org.jlcf.example.reconfiguration.ComponentA  - finished processing request for
input:INPUT12
2013-03-09 11:32:31,953 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:31,953 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:79.0
2013-03-09 11:32:31,953 [Thread-27] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT22
2013-03-09 11:32:32,391 [Thread-18] INFO  org.jlcf.example.reconfiguration.ComponentA  - finished processing request for
input:INPUT13
2013-03-09 11:32:32,453 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:32,453 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:78.0
2013-03-09 11:32:32,453 [Thread-28] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT23
2013-03-09 11:32:32,906 [Thread-19] INFO  org.jlcf.example.reconfiguration.ComponentA  - finished processing request for
input:INPUT14
2013-03-09 11:32:32,968 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:32,968 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:77.0
2013-03-09 11:32:32,968 [Thread-29] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT24
2013-03-09 11:32:33,406 [Thread-20] INFO  org.jlcf.example.reconfiguration.ComponentA  - finished processing request for
input:INPUT15
2013-03-09 11:32:33,468 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:33,468 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:76.0
2013-03-09 11:32:33,468 [Thread-30] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT25
2013-03-09 11:32:33,921 [Thread-21] INFO  org.jlcf.example.reconfiguration.ComponentA  - finished processing request for
input:INPUT16
2013-03-09 11:32:33,983 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:33,983 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:75.0
2013-03-09 11:32:33,983 [Thread-31] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT26
2013-03-09 11:32:34,421 [Thread-22] INFO  org.jlcf.example.reconfiguration.ComponentA  - finished processing request for
input:INPUT17
2013-03-09 11:32:34,483 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:34,483 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:74.0
2013-03-09 11:32:34,483 [Thread-32] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT27
2013-03-09 11:32:34,936 [Thread-23] INFO  org.jlcf.example.reconfiguration.ComponentA  - finished processing request for
input:INPUT18
2013-03-09 11:32:34,998 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:34,998 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentA  - component
org.jlcf.example.reconfiguration.ComponentA received callback with information:73.0
```

```
2013-03-09 11:32:34,998 [Thread-33] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT28
2013-03-09 11:32:35,436 [Thread-24] INFO  org.jlcf.example.reconfiguration.ComponentA  - finished processing request for
input:INPUT19
```

As soon as this call finishes, the component reaches a quiescent state.

```
2013-03-09 11:32:35,436 [Thread-24] INFO  org.jlcf.core.dynrec.SingleComponentReconfigurationManager  - component reached
quiescent state
2013-03-09 11:32:35,436 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO
org.jlcf.core.dynrec.SingleComponentReconfigurationManager  - got reply for the outcome of the reconfiguration:true
component reached quiescent state
2013-03-09 11:32:35,436 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO  org.jlcf.core.JLCFContainerProcessor
- reconfiguration response:true component reached quiescent state
2013-03-09 11:32:35,436 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO  org.jlcf.core.JLCFContainerProcessor
- reconfiguration was succesful, component reached quiescent state
2013-03-09 11:32:35,436 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO  org.jlcf.core.JLCFFrameworkUtilities
- component:compA instantiated with new implementation class:org.jlcf.example.reconfiguration.NewComponentA
2013-03-09 11:32:35,436 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO  org.jlcf.core.JLCFContainerProcessor
- connecting compA / compoB -> compB/compbintf
2013-03-09 11:32:35,436 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO
org.jlcf.example.reconfiguration.NewComponentA  - component org.jlcf.example.reconfiguration.NewComponentA version:1.01
initialized
2013-03-09 11:32:35,436 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO  org.jlcf.core.JLCFContainerProcessor
- calling setReconfiguring to false on connectors
2013-03-09 11:32:35,436 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO
org.jlcf.core.dynrec.ConnectorTimingBasedReconfigurationManager  - connector received reconfiguration message.
reconfiguring:false
2013-03-09 11:32:35,436 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO  org.jlcf.core.JLCFContainerProcessor
- calling setReconfiguring to false on connectors
2013-03-09 11:32:35,436 [AbstractQueueProcessorThread :JLCFContainerProcessor] INFO
org.jlcf.core.dynrec.ConnectorTimingBasedReconfigurationManager  - connector received reconfiguration message.
reconfiguring:false
```

The reconfiguration process finished successfully. Now all previously blocked calls are unblocked and reach the new component.

```
2013-03-09 11:32:35,452 [Thread-33] INFO  org.jlcf.example.reconfiguration.NewComponentA  - received call to doSomething.
Input:INPUT28
2013-03-09 11:32:35,452 [Thread-32] INFO  org.jlcf.example.reconfiguration.NewComponentA  - received call to doSomething.
Input:INPUT27
2013-03-09 11:32:35,452 [Thread-26] INFO  org.jlcf.example.reconfiguration.NewComponentA  - received call to doSomething.
Input:INPUT21
2013-03-09 11:32:35,452 [Thread-29] INFO  org.jlcf.example.reconfiguration.NewComponentA  - received call to doSomething.
Input:INPUT24
2013-03-09 11:32:35,452 [Thread-30] INFO  org.jlcf.example.reconfiguration.NewComponentA  - received call to doSomething.
Input:INPUT25
2013-03-09 11:32:35,452 [Thread-28] INFO  org.jlcf.example.reconfiguration.NewComponentA  - received call to doSomething.
Input:INPUT23
2013-03-09 11:32:35,452 [Thread-25] INFO  org.jlcf.example.reconfiguration.NewComponentA  - received call to doSomething.
Input:INPUT20
2013-03-09 11:32:35,452 [Thread-31] INFO  org.jlcf.example.reconfiguration.NewComponentA  - received call to doSomething.
Input:INPUT26
2013-03-09 11:32:35,452 [Thread-27] INFO  org.jlcf.example.reconfiguration.NewComponentA  - received call to doSomething.
Input:INPUT22
2013-03-09 11:32:35,498 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:35,498 [Thread-2] INFO  org.jlcf.example.reconfiguration.NewComponentA  - component
org.jlcf.example.reconfiguration.NewComponentA received callback with information:72.0
2013-03-09 11:32:35,498 [Thread-34] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT29
2013-03-09 11:32:35,498 [Thread-34] INFO  org.jlcf.example.reconfiguration.NewComponentA  - received call to doSomething.
Input:INPUT29
2013-03-09 11:32:36,013 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:36,013 [Thread-35] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT30
2013-03-09 11:32:36,013 [Thread-35] INFO  org.jlcf.example.reconfiguration.NewComponentA  - received call to doSomething.
Input:INPUT30
2013-03-09 11:32:36,013 [Thread-2] INFO  org.jlcf.example.reconfiguration.NewComponentA  - component
org.jlcf.example.reconfiguration.NewComponentA received callback with information:71.0
2013-03-09 11:32:36,467 [Thread-32] INFO  org.jlcf.example.reconfiguration.NewComponentA  - finished processing request
for input:INPUT27
2013-03-09 11:32:36,467 [Thread-26] INFO  org.jlcf.example.reconfiguration.NewComponentA  - finished processing request
for input:INPUT21
2013-03-09 11:32:36,467 [Thread-27] INFO  org.jlcf.example.reconfiguration.NewComponentA  - finished processing request
for input:INPUT22
```

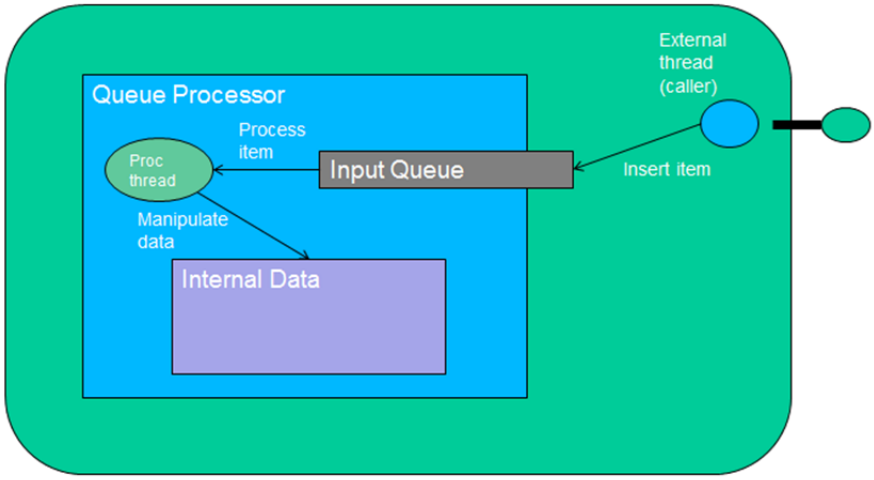
```
2013-03-09 11:32:36,467 [Thread-25] INFO  org.jlcf.example.reconfiguration.NewComponentA  - finished processing request
for input:INPUT20
2013-03-09 11:32:36,467 [Thread-31] INFO  org.jlcf.example.reconfiguration.NewComponentA  - finished processing request
for input:INPUT26
2013-03-09 11:32:36,467 [Thread-29] INFO  org.jlcf.example.reconfiguration.NewComponentA  - finished processing request
for input:INPUT24
2013-03-09 11:32:36,467 [Thread-33] INFO  org.jlcf.example.reconfiguration.NewComponentA  - finished processing request
for input:INPUT28
2013-03-09 11:32:36,467 [Thread-28] INFO  org.jlcf.example.reconfiguration.NewComponentA  - finished processing request
for input:INPUT23
2013-03-09 11:32:36,467 [Thread-30] INFO  org.jlcf.example.reconfiguration.NewComponentA  - finished processing request
for input:INPUT25
2013-03-09 11:32:36,513 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:36,513 [Thread-34] INFO  org.jlcf.example.reconfiguration.NewComponentA  - finished processing request
for input:INPUT29
2013-03-09 11:32:36,513 [Thread-2] INFO  org.jlcf.example.reconfiguration.NewComponentA  - component
org.jlcf.example.reconfiguration.NewComponentA received callback with information:70.0
2013-03-09 11:32:36,513 [Thread-36] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT31
2013-03-09 11:32:36,513 [Thread-36] INFO  org.jlcf.example.reconfiguration.NewComponentA  - received call to doSomething.
Input:INPUT31
2013-03-09 11:32:37,028 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:37,028 [Thread-37] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT32
2013-03-09 11:32:37,028 [Thread-35] INFO  org.jlcf.example.reconfiguration.NewComponentA  - finished processing request
for input:INPUT30
2013-03-09 11:32:37,028 [Thread-37] INFO  org.jlcf.example.reconfiguration.NewComponentA  - received call to doSomething.
Input:INPUT32
2013-03-09 11:32:37,028 [Thread-2] INFO  org.jlcf.example.reconfiguration.NewComponentA  - component
org.jlcf.example.reconfiguration.NewComponentA received callback with information:69.0
2013-03-09 11:32:37,528 [Thread-36] INFO  org.jlcf.example.reconfiguration.NewComponentA  - finished processing request
for input:INPUT31
2013-03-09 11:32:37,528 [Thread-38] INFO  org.jlcf.example.reconfiguration.ComponentCallerThread  - calling component
method doSomething with input: INPUT33
2013-03-09 11:32:37,528 [Thread-38] INFO  org.jlcf.example.reconfiguration.NewComponentA  - received call to doSomething.
Input:INPUT33
2013-03-09 11:32:37,544 [Thread-2] INFO  org.jlcf.example.reconfiguration.ComponentB  - sending data to registered
clients
2013-03-09 11:32:37,544 [Thread-2] INFO  org.jlcf.example.reconfiguration.NewComponentA  - component
org.jlcf.example.reconfiguration.NewComponentA received callback with information:68.0
```

### 2.2.7 Guidelines for implementing components

Although it is up to the component developer to choose how he will implement a component, this section aims at providing some guidelines for implementing complex systems with a component framework.

#### *2.2.7.1 The Abstract Queue processor*

The following pattern is used internally in the implementation of the framework and follows a principle where data is handled and processed by dedicated threads in order to avoid handling shared mutability, thread visibility and synchronization.




The principle is that the main logic and processing of the component is done in one or more Queue Processors. Calls arriving through the component interfaces insert requests to the input queue of the processor and are asynchronously processed. In case the caller needs to synchronously wait for a reply, the request itself may contain a reply queue that the caller will wait on.

This has the advantage that the internal data of the component are processed by a single thread, which does not need to care for synchronization and data visibility with other threads.

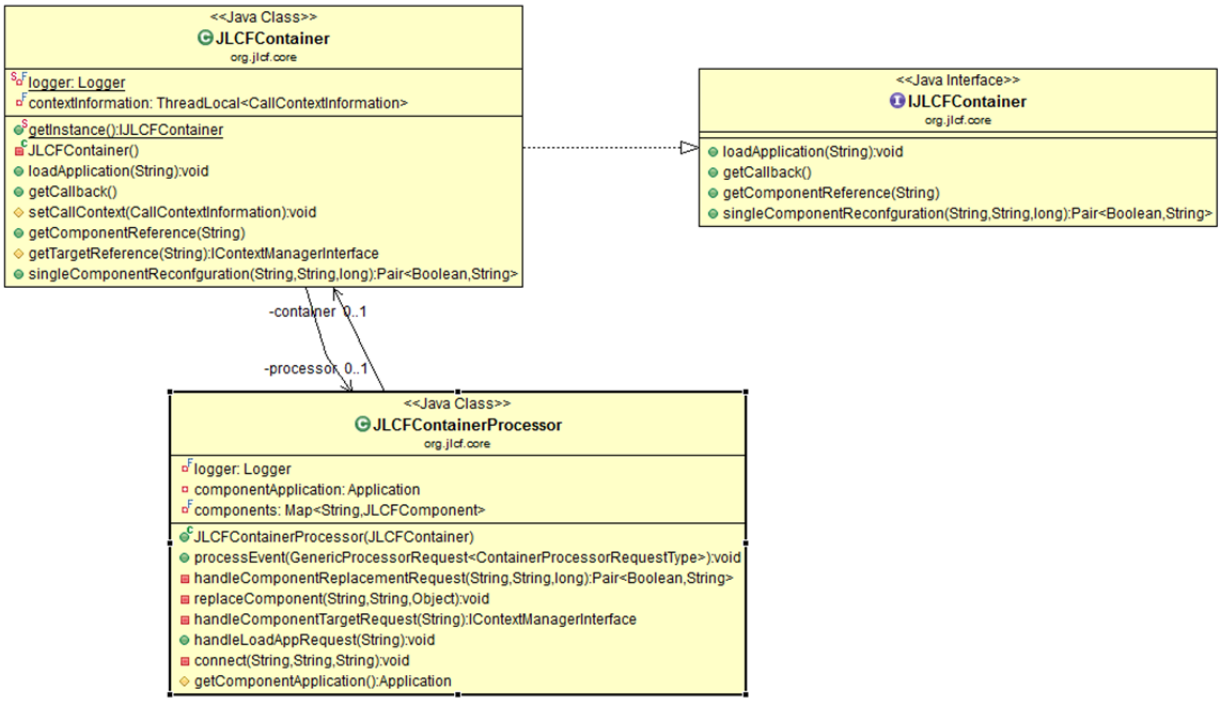
See the “advanced hello world” example that is included with JLCF for a concrete implementation of this pattern.

# 3 Developer's Guide

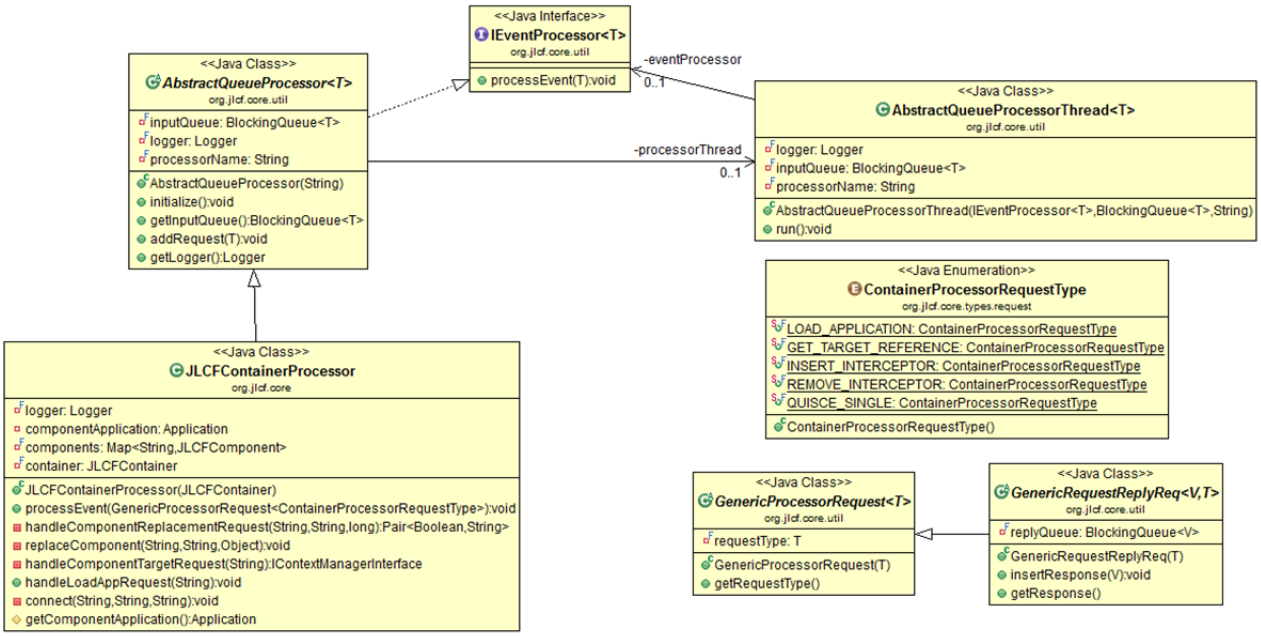
Besides this developer's guide, the source code is extensively documented.

## 3.1 System design

The central class of the system is JLCFContainer, which implements the formal interface of the framework and returns container instances that are able to load component applications.



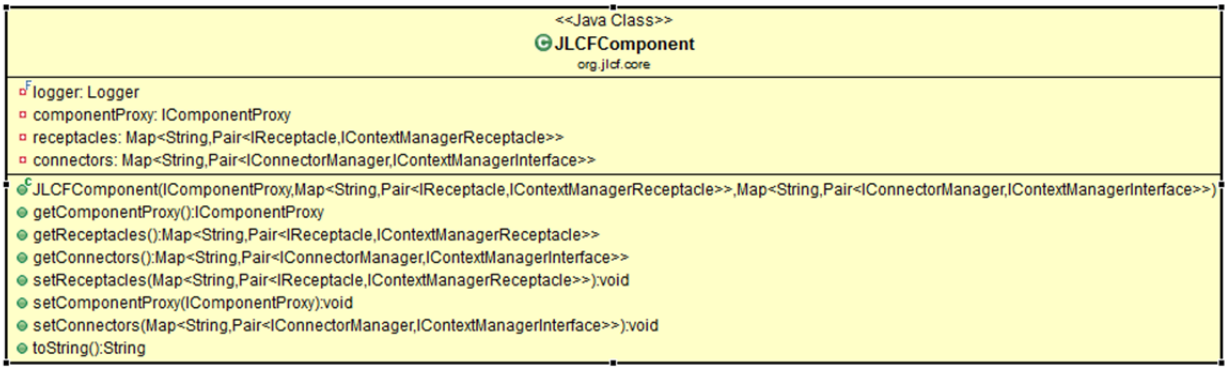

The main logic of the framework is implemented in JLCFContainerProcessor which extends the Abstract Queue Procesor pattern, explained in the examples section, and handles most user requests to the framework.




The Abstract Queue Processor is a generic class that is able to process a specific type of requests.

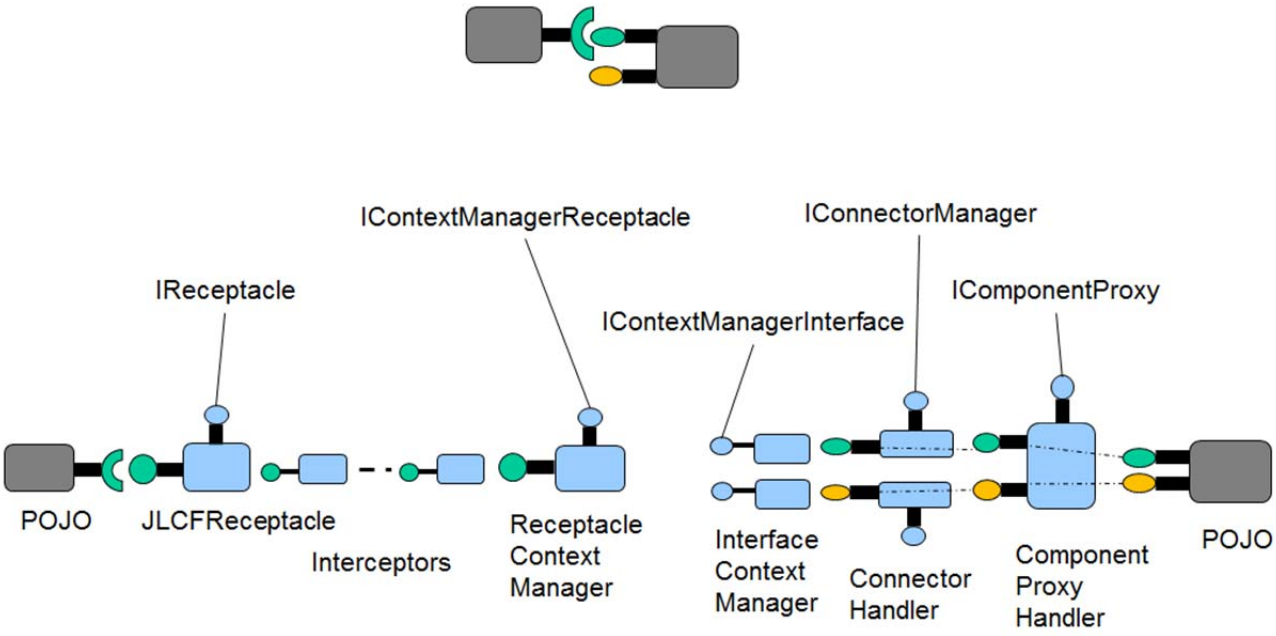
All requests that the JLCFContainreProcessor is able to process extend the GenericRequestReplyReq and can be found in package org.jlcf.core.types.request

JLCFContainerProcessor, on handling a “load application” request, which creates an application from an application description xml file, instantiates all components of the system and stores then on a map. The main class that holds all component information is JLCFComponent.




This class holds all data structures that are created by the framework and are related to a component.

When the framework creates a component, it creates the following data structures




So a simple receptacle – interface call passes through all the objects above. The functionality of each object is the following:

- JLCFReceptacle
  - This is an invocation handler that implements the target interface. The Proxy Object that is created is the object that is passed to the component POJO constructor. This holds the chain of (any) interceptors
- ReceptacleContextManager
  - The last interceptor (or the JLCFReceptacle directly if there are no interceptors) forwards the call to this object. This is an invocation handler that implements the target interface and manages the context of the call. The call context is extra information that is passed and it currently only contains the callback address (if any). Also, when a user of the framework calls the getComponentReference method, the framework returns an object of this type. The receptacle context manager resolves the reference to the target component interface and calls the Interface Context Manager of the target component interface.
- InterfaceContextManager
  - This object receives the call from the ReceptacleContextManager, extracts the callback path (if any) and forwards the call to the ConnectorHandler.
- ConnectorHandler
  - The ConnectorHandler is an invocation handler that is used in a proxy object that implements the target interface. There is one connector handler per formal component interface. The main purpose of the connector handler is to selectively block calls to the component during dynamic reconfiguration (when the component is being replaced). The connector handler forwards the call to the Componetn Proxy Handler.

- ComponentProxyHandler
    - The ComponentProxyHandler is an invocation handler that is used in a proxy object that implements all the component formal interfaces. It keeps track the number of active calls through the component interfaces and uses this information during dynamic reconfiguration (when the component is being replaced). It is also responsible of invoking the @InitMethod method on the component POJO.

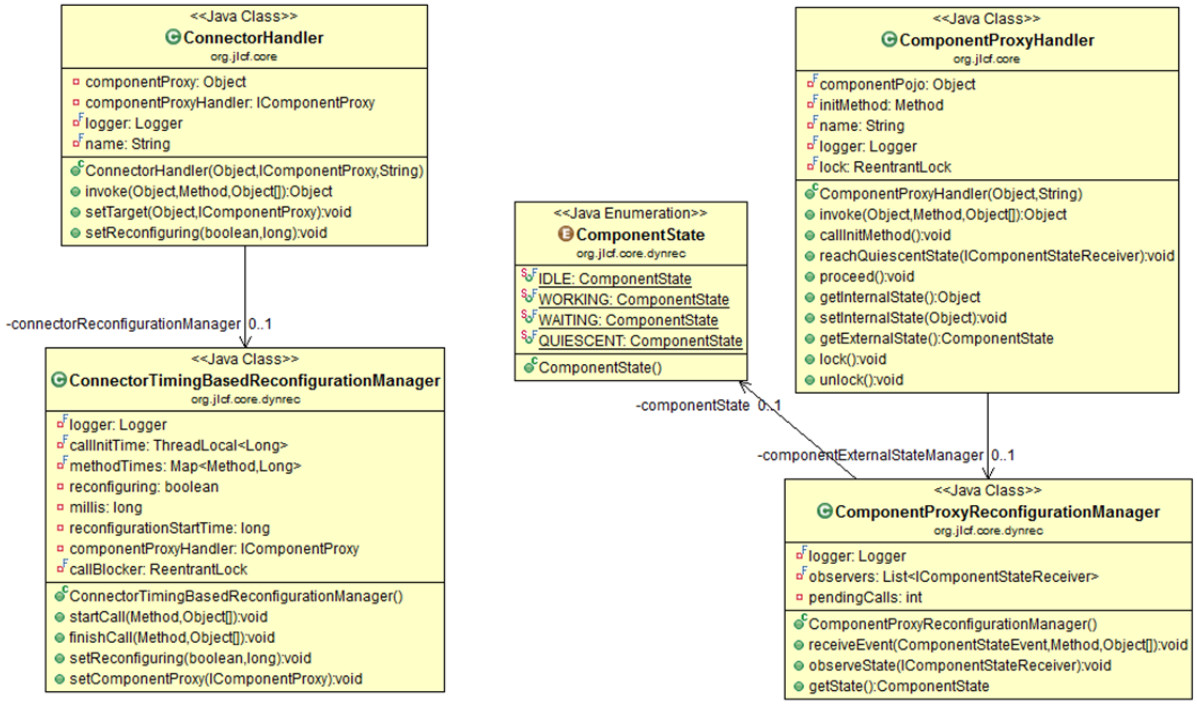
Finally the call reaches the component POJO which is implemented by a user of the framework.

Most of the logic of creating these object during a component instantiation is contained at theJLCFFrameworkUtilities helper class.

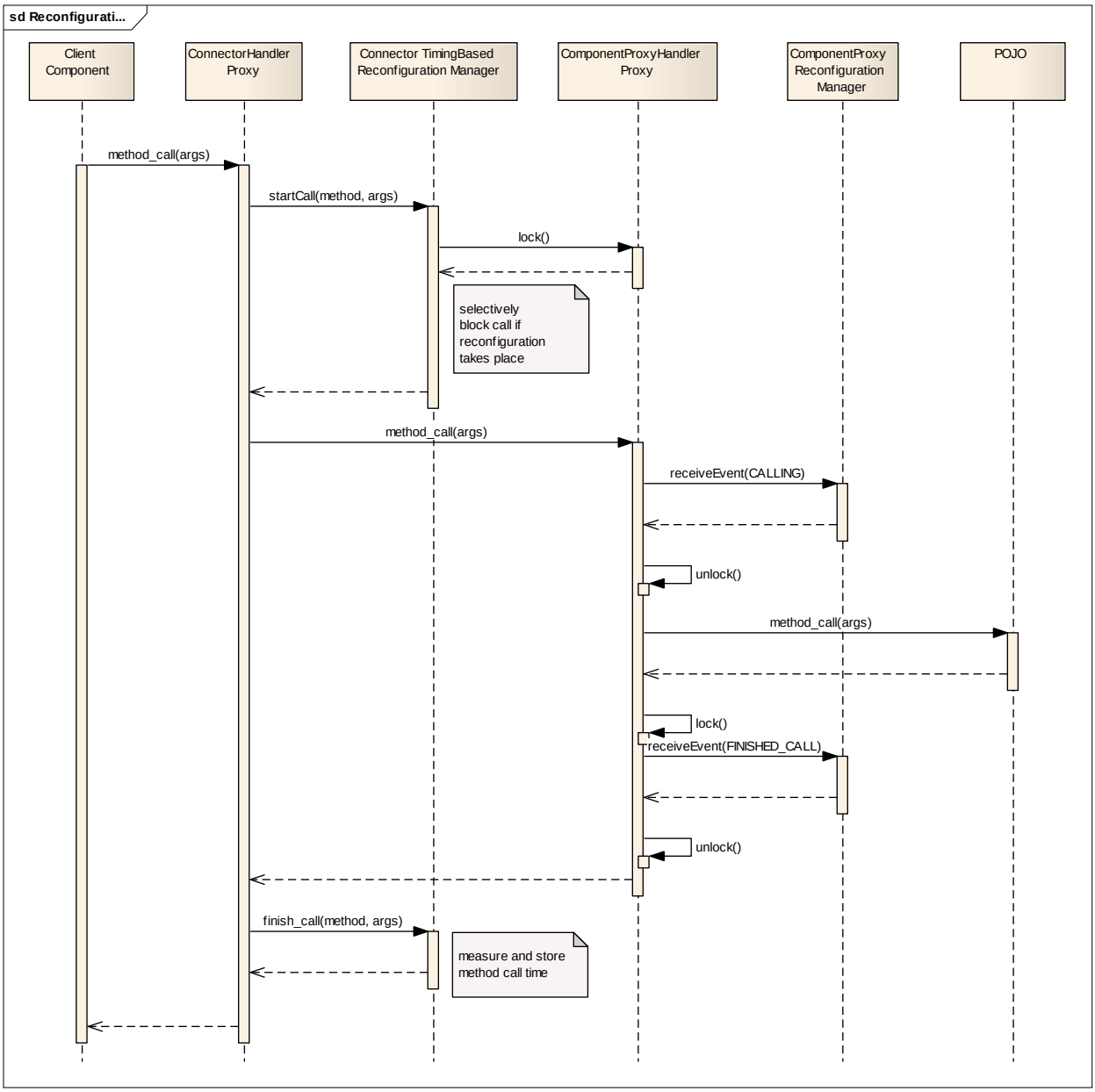
## 3.2 Dynamic reconfiguration internals

The current implementation of the dynamic reconfiguration features of the framework is simple and only supports a request to replace a single component dynamically.

The dynamic reconfiguration logic is implemented in the ***org.jlcf.core.dynrec*** package and the ConnectorHandler and the ComponentProxyHandler classes.



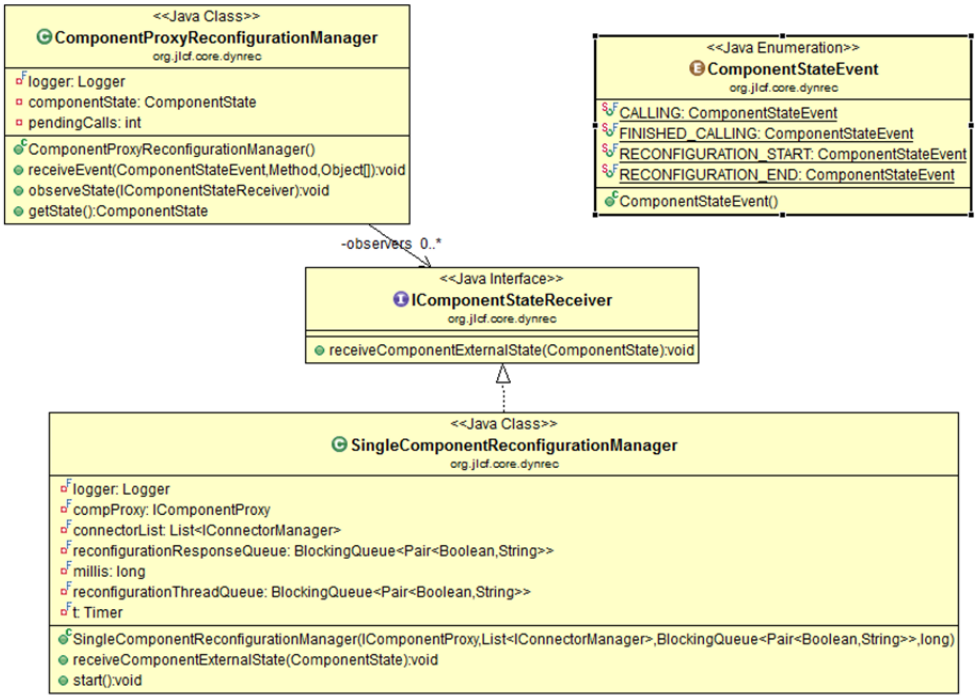

On each call on a component interface, regardless of whether a reconfiguration takes place or not, the following processing occurs:




The “Client Component” timeline, represents the rest of the object chain from the initiating POJO to the InterfaceContextManager object. On receiving a call, the connector proxy handler informs its reconfiguration manager, which acquires a lock on the component proxy handler. This lock ensures that the appropriate objects receive the start call notifications in a consistent way and at the same time does not limit concurrency on the target POJO, as it is unlocked at the Component Proxy Handler before the call is forwarded to the POJO. The connector implements logic in order to selectively block a call if reconfiguration is currently taking place. The call is then forwarded to the component proxy handler, which informs its own reconfiguration manager (which manages the state of the component and keeps track how many calls the component currently serves). It then unlocks the lock, allowing other calls to arrive at the component proxy handler and forwards the call to the POJO.

Once the method is executed at the POJO (which is the component implemented by a user of the framework) is aquires the lock, informs its reconfiguration manager and unlocks the lock. This locking mechanism is also needed in order to ensure consistent state transitions of the components external state as the component proxy reconfiguration manager is queried by connectors if a reconfiguration is currently taking place.

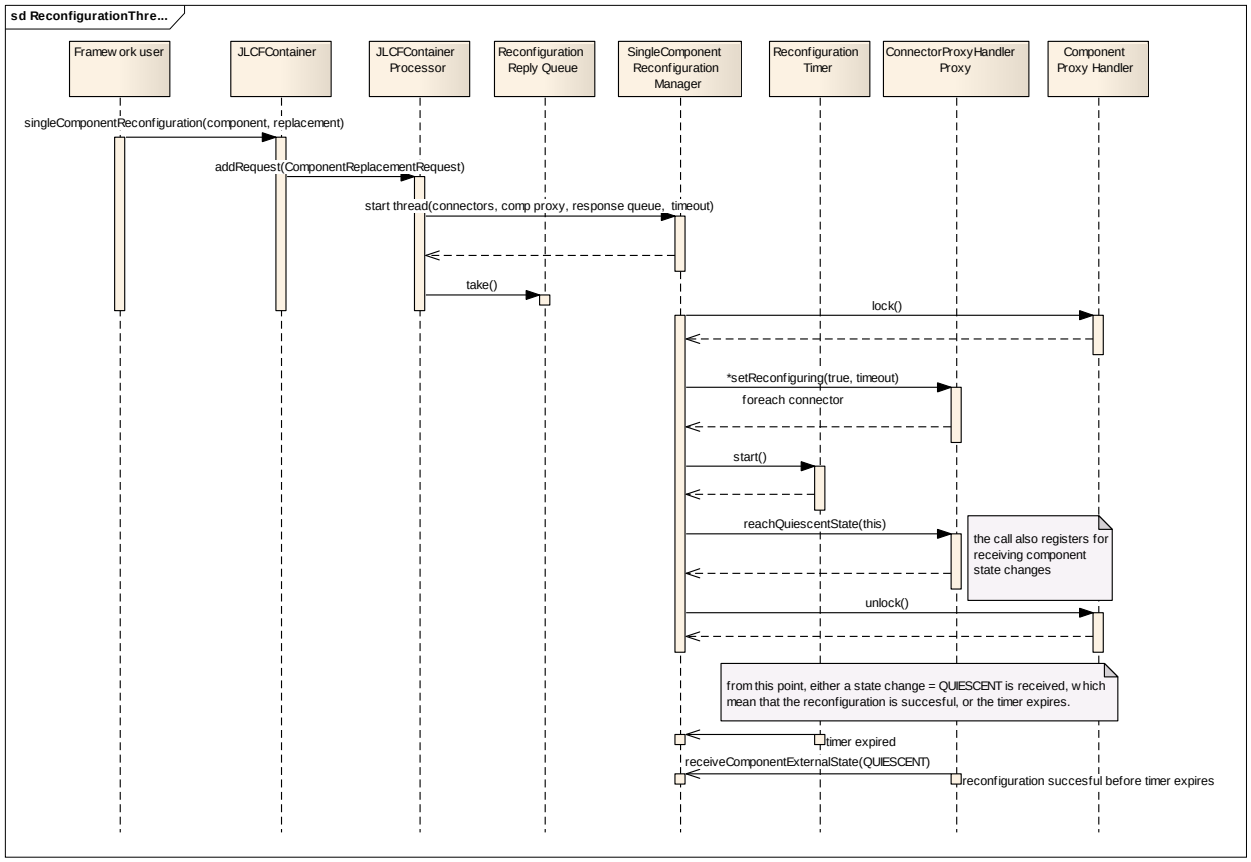
When a user sends a request to do a reconfiguration, it is handled by the JLCFContainerProcessor. The processor locates the target component and gets references to the component proxy handler and all the component connectors starts thread, the SingleComponentRecofigurationManager that is responsible of orchestrating the reconfiguration. It then blocks waiting for a reply indicating a successful or unsuccessful reconfiguration and informs the user.




The logic of the SingleComponentRecofigurationManager is the following:

It informs all connectors of the target component that a reconfiguration is about to start, it starts a timer that will expire after the user provided reconfiguration timeout and then instructs the component proxy handler to reach a quiescent state, by registering itself as a listener of the changes in the state of the component. Then, either the component reaches a quiescent state, in which case the reconfiguration process stops successfully, or the timer expires in which case the reconfiguration stops unsuccessfully.

The SingleComponentReconfigurationManager, registers for receiving state changes on the component proxy handler (which is driven by the receive_event calls made to the component proxy reconfiguration manager).




Following a successful reconfiguration, the JLCFContainerProcessor instantiates the new component POJO, transfers the state of the old component (if the POJO implements the optional IReconfiogurableComponent interface) and updates the reference of the component proxy handler in order to forward calls to the new component POJO implementation.

Then it informs all connectors and the component proxy handler that the reconfiguration has finished in order to proceed with the normal processing.

### 3.2.1 Extending dynamic reconfiguration

In order to change the algorithm for dynamic reconfiguration, or to implement additional primitives such as also allowing structural changes in a reconfiguration (replace a set of component with a different set of components) the main classes that will need modification are the ConnectorHandler and the ComponentProxyHandler.

In order to change the algorithm for driving a component reaching a quiescent state, the class ConnectorTimingBasedReconfiurationManager will need to be modified or a new class will need to be provided to the ConnectorHandler.

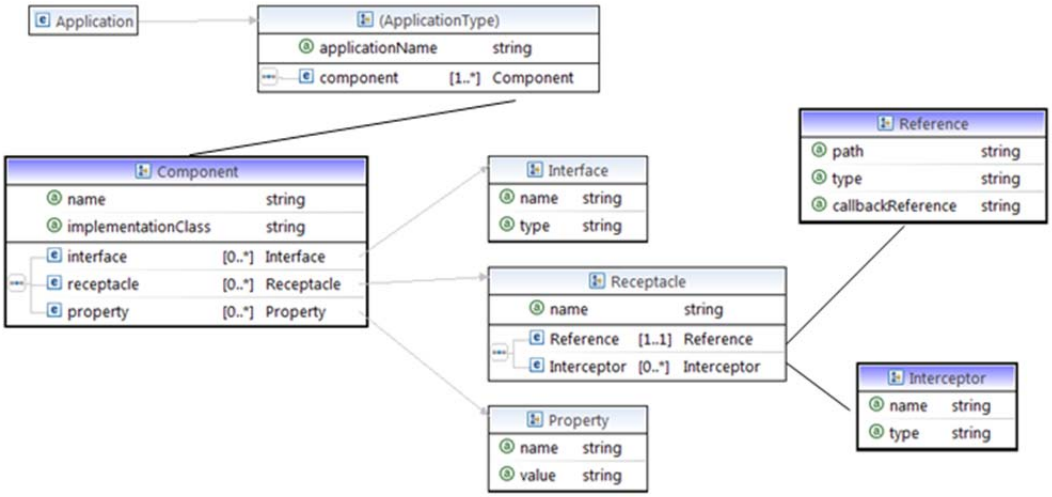
In order to add new dynamic reconfiguration primitives, a new framework request will need to be defined and then the rest of the current logic will need to be revised, as it is likely that new external component states will need to be defined and the blocking algorithm at connectors will need to be

revised, in order to take into account the states of the other components reconfiguring and not only the current component that the connector is pointing to.

# APPENDIX 1. Application Description Schema

The schema of the application description is the following

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://jlcf.sourceforge.net/JLCFApplication"
xmlns:tns="http://jlcf.sourceforge.net/JLCFApplication" elementFormDefault="qualified">

    <element name="Application">
            <complexType>
                <sequence>
                        <element name="component" type="tns:Component" maxOccurs="unbounded"
minOccurs="1"></element>
                </sequence>
                <attribute name="applicationName" type="string"></attribute>
            </complexType>
    </element>

    <complexType name="Component">
        <sequence>
                <element name="interface" type="tns:Interface" maxOccurs="unbounded"
minOccurs="0"></element>
                <element name="receptacle" type="tns:Receptacle" maxOccurs="unbounded"
minOccurs="0"></element>
                <element name="property" type="tns:Property" maxOccurs="unbounded"
minOccurs="0"></element>
        </sequence>
        <attribute name="name" type="string"></attribute>
        <attribute name="implementationClass" type="string"></attribute>
    </complexType>

    <complexType name="Reference">
        <attribute name="path" type="string"></attribute>
        <attribute name="type" type="string"></attribute>
                <attribute name="callbackReference" type="string"></attribute>
    </complexType>

    <complexType name="Receptacle">
        <sequence>
                <element name="Reference" type="tns:Reference"  maxOccurs="1" minOccurs="1"/>
                <element name="Interceptor" type="tns:Interceptor" maxOccurs="unbounded" minOccurs="0"/>
        </sequence>
        <attribute name="name" type="string"></attribute>
    </complexType>


    <complexType name="Interceptor">
        <attribute name="name" type="string"></attribute>
        <attribute name="type" type="string"></attribute>
    </complexType>

    <complexType name="Interface">
        <attribute name="name" type="string"></attribute>
        <attribute name="type" type="string"></attribute>
    </complexType>

    <complexType name="Property">
        <attribute name="name" type="string"></attribute>
        <attribute name="value" type="string"></attribute>
    </complexType>
</schema>
```


Application
(ApplicationType)
applicationName string
component [1..*] Component
Component
name string
implementationClass string
interface [0..*] Interface
receptacle [0..*] Receptacle
property [0..*] Property
Interface
name string
type string
Reference
path string
type string
callbackReference string
Receptacle
name string
Reference [1..1] Reference
Interceptor [0..*] Interceptor
Interceptor
name string
type string
Property
name string
value string