

EL4J

ReferenceDoc

Generated from Wiki page:
<http://wiki.elca.ch/twiki/el4j/bin/view/EL4J/ReferenceDoc>

Version	Date	Author(s)	Visa
1.11	27 Oct 2005 – 18:38	Main.PhilippHOser, PhilippHOser	

ELCA Informatique SA, Switzerland 2005

Table of Contents

1 Introduction.....	1
2 Installing modules.....	2
2.1 Introduction.....	2
2.2 Steps to install a module with EL4Ant.....	2
2.2.1 Details on the content of project.xml.....	3
3 Documentation for module core.....	4
3.1 Purpose.....	4
3.2 Support for EL4Ant modules on the level of Spring.....	4
3.2.1 Module abstraction of EL4Ant.....	4
3.2.2 ModuleApplicationContext.....	5
3.2.3 Convention on how to organize configuration.....	5
3.2.3.1 Examples.....	6
3.2.3.2 Using the execution units feature of the EL4Ant build system.....	7
3.2.3.3 Usage of configuration using this convention.....	8
3.3 Convenience Attributes for Transactions.....	8
3.3.1 Purpose.....	8
3.3.2 Introduction.....	8
3.3.3 Configuration.....	10
3.3.3.1 Overview.....	10
3.3.3.2 How to define an attribute.....	11
3.3.3.3 How to organize the transaction attributes.....	11
3.3.3.4 Transaction propagation behaviors.....	12
3.3.3.5 Transaction attribute classes.....	12
3.3.3.6 Adding spring beans to enable transactional behaviour.....	13
3.3.3.7 Programmatical transaction demarcation (start transaction, commit, rollback in code).....	14
3.3.3.8 setRollbackOnly is not equals to setReadOnly.....	15
3.3.4 Internal design.....	15
3.4 Attribute Convenience.....	15
3.4.1 Purpose.....	15
3.4.2 Configuration.....	15
3.4.3 Extension for more attributes.....	16
3.4.3.1 How to access all the attributes.....	16
3.4.4 Internal design.....	17
3.4.4.1 Classes.....	17
3.5 Additional Features.....	18
3.5.1 Configuration merging via property files.....	18
3.5.2 Bean locator.....	18
3.5.3 Bean type auto proxy creator.....	18
3.5.4 Exclusive bean name auto proxy creator.....	19
3.5.5 Abstract parent classes for the Typesafe Enumerations Pattern.....	19
3.5.6 Reject (Precondition checking).....	19
3.5.7 JNDI Property Configurers.....	19
3.5.8 DTO helpers.....	19
3.5.9 Primary key.....	20
3.5.10 SQL exception translation.....	20
3.6 Packages that implement the core module.....	20
4 Documentation for module web.....	21
4.1 Purpose.....	21
4.2 Features.....	21

Table of Contents

4 Documentation for module web	21
4.3 How to use	21
4.3.1 General configuration of the web module	21
4.4 Reference documentation for the Module-aware application contexts	21
4.4.1 Concept	21
4.4.1.1 ModuleDispatcherServlet	22
4.4.2 Build system integration	22
4.4.2.1 Adding files manually	22
4.4.2.2 Installing the build system hook	23
4.4.3 Limitations	23
4.4.4 MANIFEST.MF configuration section format	23
4.4.5 Implementation Alternative: Idea	24
4.4.6 Resources	24
5 Documentation for module remoting	25
5.1 Purpose	25
5.2 Introduction	25
5.3 How to use	25
5.3.1 Configuration	25
5.3.1.1 Recommended configuration file organisation	25
5.3.1.2 Configuration summary	27
5.3.1.3 How to use the Rmi protocol	29
5.3.1.4 How to use the Hessian protocol	29
5.3.1.5 How to use the Burlap protocol	32
5.3.1.6 How to use the Soap protocol	32
5.3.1.7 How to use the EJB protocol	38
5.3.1.8 Introduction to implicit context passing	38
5.3.2 Benchmark	39
5.3.3 Remoting semantics/ Quality of service of the remoting	40
5.3.3.1 Cardinality between client using the remoting and servants providing implementations	40
5.3.3.2 What happens when there is a timeout or another problem during remoting	40
5.4 Internal design	40
5.4.1 Sequences	41
5.4.1.1 Sequence diagramm from client side	41
5.4.1.2 Sequence diagramm from server side	43
5.4.2 Creating a new interface during runtime	45
5.4.3 Internal handling of the RMI protocol (in spring and EL4J)	45
5.4.4 To be done	45
5.5 Related frameworks	46
5.5.1 extrmi	46
5.5.2 Javaworld 2005 idea	46
6 Documentation for module EJB remoting	47
6.1 Purpose	47
6.2 Important concepts	47
6.3 How to use	47
6.3.1 Configuration	47
6.3.1.1 How to use the EJB protocol	47
6.3.1.2 How to use the build system plugin	49
6.3.1.3 How to use the EJB remoting module without the EL4Ant build system	49
6.4 References	50
6.5 Internal design	50

Table of Contents

6 Documentation for module EJB remoting	50
6.5.1 EJB generation.....	50
6.5.2 Adding support for another container.....	52
7 Documentation for module security.....	53
7.1 Purpose.....	53
7.2 Features.....	53
7.3 How to use.....	53
8 Documentation for module exception handling.....	54
8.1 Purpose.....	54
8.2 Important concepts.....	54
8.3 How to use.....	54
8.3.1 Configuration.....	54
8.3.1.1 Exception handlers.....	55
8.3.1.2 Example 1: Safety Facade for one Bean.....	55
8.3.1.3 Example 2: Context Exception Handler.....	56
8.3.1.4 Example 3: RoundRobinSwappableTargetExceptionHandler.....	57
8.3.1.5 Example 4: Using several exception handlers, each configured by a separate exception configuration.....	58
8.4 References.....	59
8.5 Internal design.....	59
8.5.1 Context Exception Handler.....	59
9 Documentation for module JMX.....	60
9.1 Purpose.....	60
9.2 Introduction to Java management eXtensions (JMX).....	60
9.3 Feature overview.....	60
9.4 Usage.....	61
9.4.1 Spring/JDK versioning issue.....	61
9.4.1.1 Spring versions 1.1 <=> 1.2.....	61
9.4.1.2 JDK versions 1.4.2 <=> 1.5.....	61
9.4.2 Basic Configuration (implicit publication).....	61
9.4.3 Connector.....	62
9.4.3.1 HtmlAdapter.....	62
9.4.3.2 JmxConnector.....	62
9.4.4 Example with one ApplicationContext.....	63
9.4.5 Configuration (explicit publication).....	63
9.4.6 Example with more than one ApplicationContext.....	64
9.5 References.....	65
10 Documentation for module light statistics.....	66
10.1 Purpose.....	66
10.2 Important concepts.....	66
10.2.1 Monitoring strategies.....	66
10.3 How to use.....	66
10.3.1 Configuration.....	66
10.3.2 Demo.....	66
10.3.3 How to set up the module-light_statistics for the ref-db sample application.....	66
10.3.3.1 binary-modules.xml.....	66
10.3.3.2 project.xml.....	66
10.3.3.3 Limit the set of intercepted beans.....	67
10.4 FAQ.....	67

Table of Contents

10 Documentation for module light statistics	
10.5 References.....	67
11 Exception handling guidelines.....	68
11.1 Topics.....	68
11.2 When to define what type of exceptions? Normal vs. abnormal results.....	68
11.2.1 Further examples.....	68
11.2.2 How to handle normal and abnormal cases.....	69
11.3 Implementing exceptions classes.....	69
11.4 Handling exceptions.....	69
11.4.1 Where to handle exceptions?.....	69
11.4.2 How to trace exceptions?.....	70
11.4.3 Rethrowing a new exception as the consequence of a caught exception.....	70
11.5 Related useful concepts and hints.....	70
11.5.1 Add attributes to the exception class.....	70
11.5.2 Mentioning unchecked exceptions in the Javadoc.....	70
11.5.3 Checking for pre-conditions in code.....	70
11.5.4 Exception-safe code.....	71
11.5.5 Handling SQL exceptions.....	71
11.5.6 Exceptions and transactions.....	71
11.5.7 SafetyFacade pattern.....	71
11.6 Antipatterns.....	72
11.7 References.....	72
12 Acknowledgments.....	74
13 References.....	75

1 Introduction

EL4J (<http://el4j.sourceforge.net/>), the Extension Library for the J2EE, adds incremental improvements to the Spring Java framework (<http://www.springframework.org/>). Among the improvements are (this is just the beginning):

- The ability to split applications in modules that each can provide their own code and configuration, with transitive dependencies between modules
- Simplified POJO remoting with implicit context passing, including support for SOAP and EJB
- A light daemon manager service
- Support to see the active beans and their configuration in JMX
- A light exception handling framework that implements a safety facade

Used libraries and tools

- Most libraries that are included in the spring framework
- EL4Ant (a light build system based on ant with modules and transitive dependencies, we will submit the project to sourceforge also)

EL4J is a package for Java developers – ready to start working. It is published under the GPL (<http://www.gnu.org/licenses/gpl.txt>) at sourceforge. Please contact info@elca.ch for a commercial license.

EL4J is already in use in 10+ projects within ELCA (<http://www.elca.ch>).

This documentation was auto-generated from content of our twiki. Some of the links still are undefined (due to the way we created it) and the content is still emerging.

2 Installing modules

EL4J (the framework) and applications using it are split in modules. One needs to install only the needed module and dependencies of modules are automatically taken into account. This section introduces how one can download modules. For more details on the module abstraction, please consult the corresponding section in the core module.

2.1 Introduction

To install a module in your project, use the [EL4Ant build system](#).

Each module of the framework EL4J is available as a binary release, stored on the download server. As its name indicates, a binary module contains compiled classes, so you can start using it directly. Each binary release of a module has a version number and it is a zip file with the filename `modulename_version.zip`. For example, the file named `module-web_1.0.zip` is a binary release for the `module-web` of version `1.0`.

2.2 Steps to install a module with EL4Ant

We shortly introduce here how to install EL4J with the EL4Ant build system, please refer to the EL4Ant build system for more details. Set your project description file (`project.xml`) up to something like the following:

```
<!-- sample project.xml file; adapt to your needs -->

<plugin name="binrelease.versions">
    <!-- list here any binary releases to know what versions to use -->
    <attribute name="binrelease.version.module-core" value="1.0"/>
    <attribute name="binrelease.version.module-web" value="1.0"/>
    <attribute name="binrelease.version.module-xpf" value="1.0"/>
</plugin>

<plugin name="binrelease"/>

<plugin name="onlinelib">
    <attribute name="onlinelib.repository" value="http://leaffy/java/lib/" />
    <attribute name="onlinelib.repository" value="http://leaffy/java/lib/src/" />
    <attribute name="onlinelib.repository" value="http://leaffy/java/el4j/modules/" />
</plugin>

<!-- add your module definitions here -->

<module name="sample-module" path="..."> <!-- a possible module definition -->
    ...
    <dependency module="module-core"/>      <!-- dependencies to other modules -->
    <dependency module="module-web"/>      <!-- both binary and source modules -->
    ...
</module>
```

Please refer to the next sections for more details on each entry of the `project.xml` file.

After having setup the project definition file, make a bootstrap of the EL4Ant build system (to generate the `build.xml` file and to download all the missing files):

```
ant -f bootstrap.xml
```

After this step has successfully completed, you are ready to start using EL4J!

2.2.1 Details on the content of project.xml

This section explains the content of the project.xml file.

The following XML section indicates EL4Ant to load the "binrelease" plugin. It allows downloading modules in binary form:

```
<plugin name="binrelease"/>
```

The binrelease plugin requires the plugin `<plugin name="binrelease.versions">...</plugin>`. Inside this element you must have an attribute with the version of each module you would like to use. For the version 1.0 of the module-web this would look like the following:

```
<attribute name="binrelease.version.module-web" value="1.0"/>
```

At the end, this XML section of the project.xml could look like the following:

```
<plugin name="binrelease.versions">
  <attribute name="binrelease.version.module-core" value="1.0"/>
  <attribute name="binrelease.version.module-web" value="1.0"/>
  <attribute name="binrelease.version.module-xpf" value="1.0"/>
</plugin>
<plugin name="binrelease"/>
```

A further required plugin is the onlinelib plugin. It downloads each required zip and jar file. A sample configuration looks like the following:

```
<plugin name="onlinelib">
  <attribute name="onlinelib.repository" value="http://leaffy/java/lib/" />
  <attribute name="onlinelib.repository" value="http://leaffy/java/lib/src/" />
  <attribute name="onlinelib.repository" value="http://leaffy/java/el4j/modules/" />
</plugin>
```

After having set up those plugins, you are ready to use each binary module you have listed in the plugin binrelease.versions as a "normal" module. Here's an example:

```
<module name="keyword-web" path="...">
  ...
  <dependency module="module-web"/>
  ...
</module>
```

The fact whether a reference to a used module is a binary release or a source release is completely transparent.

3 Documentation for module core

3.1 Purpose

The core module of EL4J contains support to split applications into separate modules. Each module can contain code, configuration and dependencies on jar files as well as on other modules. Dependencies are transitive. In addition, the core module contains helpers classes for attributes, transaction attributes, implicit context passing and others.

3.2 Support for EL4Ant modules on the level of Spring

The *module* support of the core module is provided in combination with the EL4Ant build system. The EL4Ant defines the module abstraction and the core module of EL4J makes use of it and supports it on the level of Spring.

Rationale for the module support:

- Modularity: be able to split your work in smaller sub-parts in order to reduce complexity, to simplify separate development, to reduce size of code by using only what is needed.
- Provide default configuration for modules: with spring, configuration can sometimes become complicated. We provide support for default spring configurations to modules.
- Dependency management (1): each module lists its requirements (other modules and jar files). These dependencies are then automatically managed (downloaded if needed, added to the classpath, added to deployment packages such as WAR, EAR or zip files)
- Dependency management (2): from each module only the resources of the dependent modules are visible (you can e.g. make certain server-side jar files invisible during the compilation of client-side code, in order to statically ensure they are not used)

The module support is based on the following:

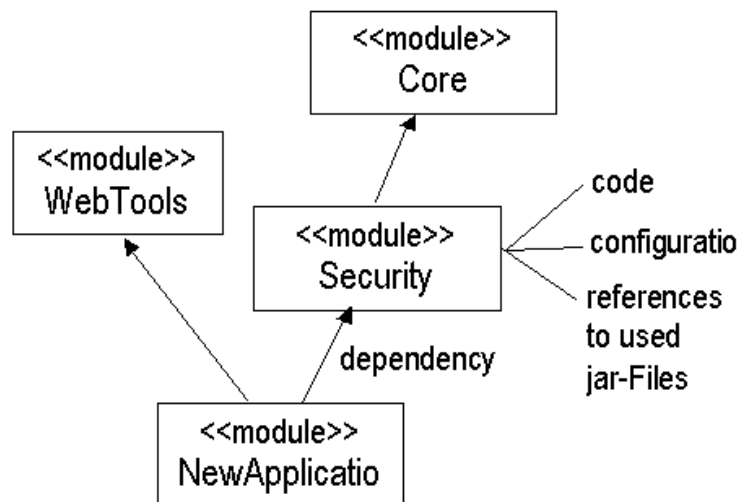
- the module abstraction of EL4Ant
- the `ModuleApplicationContext` (a wrapper for the standard Spring application context)
- a convention on how to organize configuration information within each module

These three parts are described in the next sections.

3.2.1 Module abstraction of EL4Ant

EL4Ant (<http://el4ant.sf.net>) is a light, Ant-based build system. It generates a standard build.xml file from a description of your project. With EL4Ant you can split your application or framework into *modules*. A module can contain code and configuration. Modules can define dependencies on jars and other modules. Dependencies on other modules are transitive (e.g. if A requires B and B requires C, A has implicitly also C available). EL4Ant can package your module into a jar file.

The following picture illustrates 4 modules with dependencies:



For more detail on how to setup modules and for more module features, we refer to the documentation of [EL4Ant](#).

3.2.2 ModuleApplicationContext

The **ModuleApplicationContext** is similar to the existing application contexts of Spring (i.e. `ClasspathXmlApplicationContext`). It is a light wrapper around the existing Spring application contexts.

The use of the **ModuleApplicationContext** is optional. We recommend it due to its following features:

- it finds all configuration files present in the modules, even if some J2EE-container present them differently (e.g. WLS)
- it solves issues with the order of loading configuration files in some J2EE-containers
- it complements the rest of the configuration support (e.g. via the configuration file exclusion list)
- it allows publishing all its Spring beans with their configuration (publication is possible e.g. to JMX).

The first two features are provided in collaboration with the module support of ELAnt. (EL4Ant lists the configuration files contained in each module into the Manifest file of modules. The **ModuleApplicationContext** then uses this information.)

The reference documentation of the **ModuleApplicationContext** is located under the [web module](#).

3.2.3 Convention on how to organize configuration

Our convention to organize config files helps to indicate what configuration should be automatically loaded when a module is active. One can also define different configuration scenarios among which one needs to choose one. A sample scenario is the choice of whether we run in a client or a server (e.g. for remoting or security) or what data access technology to use (e.g. ibatis or hibernate). NB: There is an easy way not to load mandatory configuration information.

The configuration files of a module are saved under a folder `/conf`. This `/conf` folder is divided into different subfolders:

- `/mandatory`: Here are all the xml and the property files which are always loaded into the `ApplicationContext` when the module is active.
- `/scenarios`: This is the parent folder for different scenarios. It does not contain any file, only subfolders. (e.g. a type of scenario would be 'authentication' and the scenarios of this type would be

stateless or stateful). Exactly one scenario of each type must be chosen. All possible combinations of the scenarios have to work.

- ◆ '/subfolder': For each type of scenarios (see below), there is a subfolder with a context-dependent name. One scenario of each subfolder must be chosen in order to execute the module. Note: Such a subfolder could contain further subfolders.
- '/optional': Here are optional xml and property files which are loaded if requested.
- In the '/conf/template' folder one can provide templates which can be helpful to efficiently develop applications or to understand the module.
 - ◆ '/subfolder': Subfolders can be introduced in order to structure the templates.

By loading all files in '/conf/mandatory' and one scenario of each type into the ApplicationContext, the module has to be executable. This constraint reduces the complexity for developers using this module.

3.2.3.1 Examples

Two examples are provided in order to illustrate the ideas of the above structure.

3.2.3.1.1 Example 1

The first example illustrates how the configuration structuring of the **ModuleSecurity** (old version) looks like:

- ch.elca.el4j.core.services.security:
 - ◆ '/conf/mandatory/':
 - ◊ security-attributes.xml
 - ◆ '/conf/scenarios/':
 - ◊ 'authentication':
 - stateless-authentication.xml
 - stateful-authentication.xml
 - ◊ 'logincontext':
 - db-logincontext.xml
 - nt-logincontext.xml
 - ◊ 'securityscope':
 - local-securityscope.xml
 - 'distributedsecurityscope':
 - client-distributedsecurityscope.xml
 - server-distributedsecurityscope.xml
 - web-securityscope.xml
 - ◆ '/conf/optional':
 - ◆ '/conf/templates':

Explanation: In security-attributes.xml, the attributes for the authorization interceptor is defined. Since it is always needed, it is put into the '/mandatory/' folder. There are 3 types of scenarios which the developer can choose from. Regarding the authentication there's the choice between a stateless and a stateful authentication. As a next thing it has to be defined which login context is chosen. Last, the security scope has to be defined, i.e. if the environment is set up locally, if it is distributed or if it is web based. In case the environment is distributed, we define a subfolder since there is more than one xml file defining these beans.

Important: in case of a distributed environment, the security module needs a remote protocol which has to be specified. Since in the distributed environment, the security module needs the **ModuleRemotingAndInterfaceEnrichment** module, the remote protocol is defined in that scope.

3.2.3.1.2 Example 2

A second example illustrates the Remoting And Interface Enrichment module (the current module is slightly different):

- ch.elca.el4j.core.services.remoting:
 - ◆ '/conf/mandatory/':
 - ◇ 'scope/':
 - client-config.xml
 - server-config.xml
 - ◇ 'protocol/':
 - rmi-protocol-config.xml
 - hessian-protocol-config.xml
 - burlap-protocol-config.xml
 - ◆ '/conf/optional/':
 - ◆ '/conf/templates/':
 - ◇ service-exporter-config.xml
 - ◇ service-importer-config.xml

Explanation: The developer has to choose exactly one possibility of both of the two scenarios. On the one hand, the scope has to be defined, i.e. if the ApplicationContext is loaded on a client or on a server. Then, the protocol has to be chosen, either rmi, burlap or hessian. Obviously, the remote protocol and its properties has to be the same, on the client and the server. Finally the exporter and the importer are stored under '/conf/templates/' since the content of these xml files highly depends on the specific implementation. Therefore, commented templates are provided.

Remark: it is still possible to load both the client and the server configs in case one would require to have both roles.

3.2.3.1.3 Example 3

This example illustrates the `ModuleJmx`:

- ch.elca.el4j.services.monitoring.jmx:
 - ◆ '/conf/mandatory/':
 - ◇ jmx.xml
 - ◇ htmlAdapter.xml
 - ◆ '/conf/scenarios/':
 - ◆ '/conf/optional/':
 - ◇ jmxConnector.xml
 - ◆ '/conf/templates/':

Explanation: Although the HTML adapter is just one option to access JMX data, it is considered to be the most used. Putting its configuration file in the mandatory folder loads it whenever the module is added as dependency. Users still can use the JMX connector and remove the HTML adapter using the `ModuleApplicationContext` with its ability to exclude configuration files explicitly.

3.2.3.2 Using the *execution units* feature of the EL4Ant build system

In the above text we have discussed the use of configuration scenarios to treat alternative cases. An orthogonal mechanism to support such orthogonal cases is the *execution unit* feature of EL4Ant. Execution units (Eus) allow to split the deliverables of a module in sub-sets (sample subsets are the interface and the implementation files, the files used in the client and server, or the files for gui and web user interfaces). For more details, please refer to the EL4Ant documentation.

Execution units can also be used to distinguish between alternative cases. One simply adds different configuration files for the separate execution units. The next example illustrates this:

3.2.3.2.1 Example 4

Configuration of the statistics module (it provides convenience to use the JAMon interceptor):

- `ch.elca.el4j.services.performance.jamon:`
 - ◆ `'/conf/mandatory/':`
 - ◇ `jamon.xml`
 - ◇ `jamon-jmx.xml`
 - ◆ `'/conf/scenarios/':`
 - ◆ `'/conf/optional/':`
 - ◆ `'/conf/templates/':`

Explanation: The module JAMon can be used together with the JMX module or stand-alone. While the former has a dependency on the JMX module and a JMX proxy configured in the `jamon-jmx.xml` file, the latter needs a web application container to display measurements. The dependency and the action to exclude the `jamon-jmx.xml` configuration file are defined in the module's specification in form of two different **execution units**.

3.2.3.3 Usage of configuration using this convention

This section presents how the security module (as defined above) could be used in an application. Note that EL4Ant adds the `conf` folder of each module automatically to the active classpath:

```
String[] configurationFiles = {"classpath*:mandatory/*.xml",
"scenarios/authentication/stateless-authentication.xml",
"scenarios/logincontext/db-logincontext.xml",
"scenarios/securityscope/local-securityscope.xml"};
```

```
ApplicationContext m_ac = new ClassPathXmlApplicationContext(configurationFiles);
```

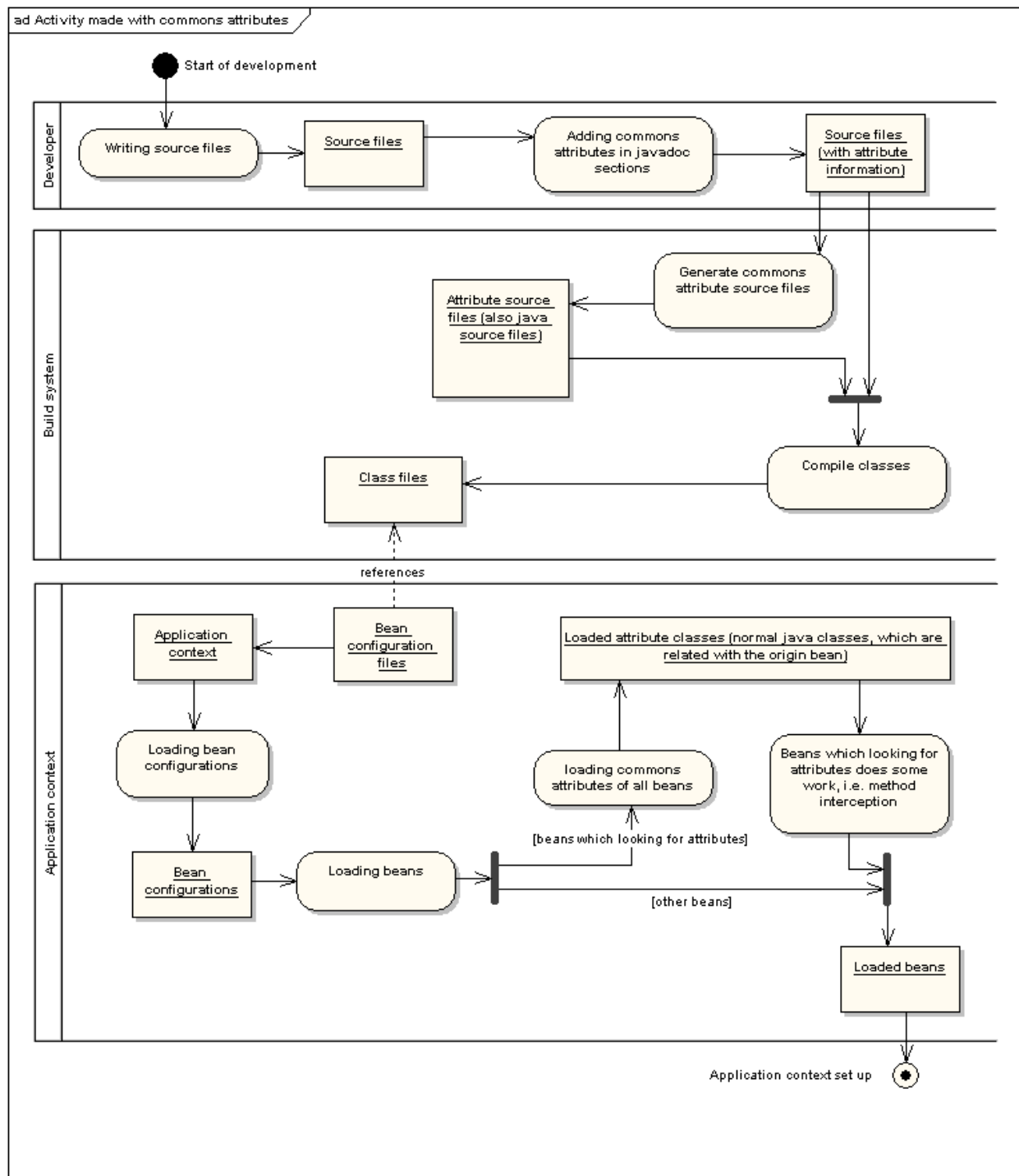
3.3 Convenience Attributes for Transactions

3.3.1 Purpose

The goal of the feature described in this section is to provide convenience attributes for transaction support. The convenience attributes allow some neat features to define attributes also on method interfaces, set the configuration automatically up (via the module support of EL4J) for the attributes support and require less writing. The use of these attributes is optional.

3.3.2 Introduction

The following picture defines how to work with attributes in spring:



The Spring framework offers a simple way to use transactions in combination with commons attributes. Commons attributes are defined in source code inside the javadoc section of a class or a method. To make these attributes at runtime available, we have to generate separate java source files for classes that contain such attributes. Afterwards all java source files have to be compiled. This is done by the build system.

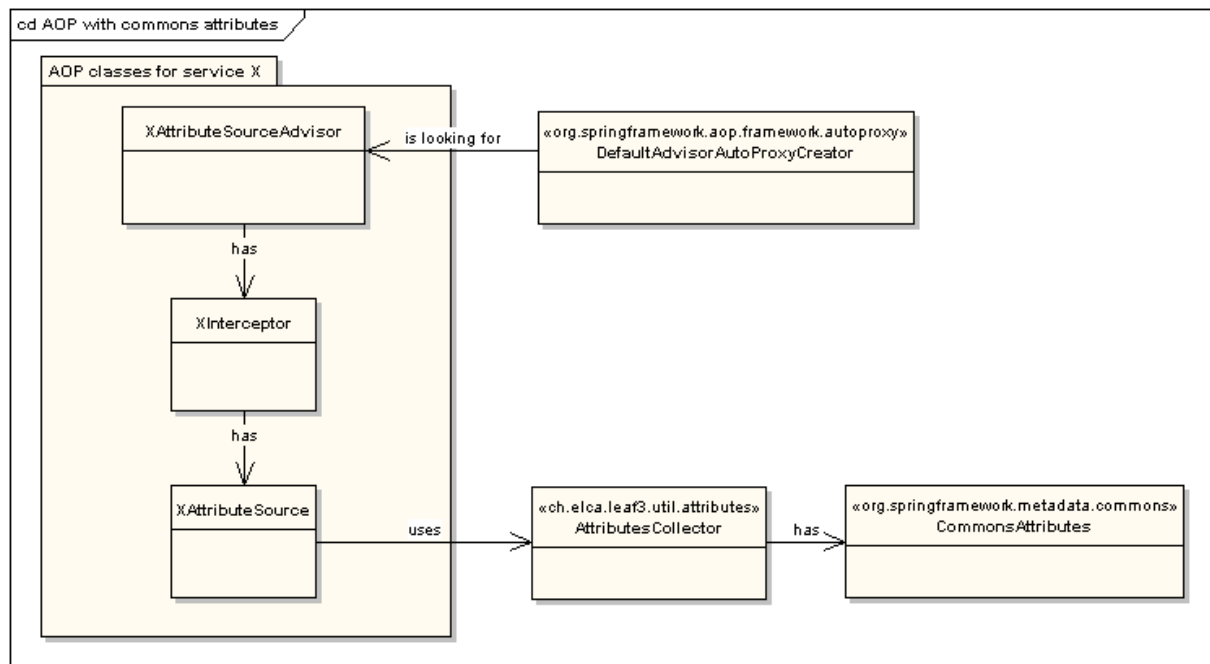
By using common attribute classes to access attributes of your classes you can do with attributes what you want. In many cases, attributes are used to enable method interceptors.

3.3.3 Configuration

To enable transactional behaviour in your project using spring you just have to add a few lines in one spring configuration file.

It is important that the spring configuration files are loaded within an application context. Otherwise automatic method interception will not work!

3.3.3.1 Overview



The top element is the `DefaultAdvisorAutoProxyCreator`. It collects all `Advisor`s and tells them to create a proxy for every bean the `Advisor` thinks it is necessary. A class that does this is the `XAttributeSourceAdvisor`. The `X` stands for the *responsibility* of the advisor, that is `Transaction` in our case. The proxy that will be added to each necessary bean is the `XInterceptor`. In our case this is a `TransactionInterceptor`. Every method call on a proxied bean will be passed through this interceptor. But to know which beans has to be proxied and how the transactions should be managed, the attributes are necessary. To get only the attributes that are interesting for the transactional behaviour, we have a class `XAttributeSource`, in our case the class `TransactionAttributeSource`. It gets all attributes of a given class objects and filters out the needed. At this time have only a one dimensional array of attributes (normal java objects). Sometimes it is necessary to link some kind of attributes with others. This is also done in the class `TransactionAttributeSource`.

The `AttributesCollector` collects not only the attributes that are written in the current class, but also the attributes which are defined in implemented interfaces. To illustrate the functioning of the `AttributeCollector`, let's see how he collects the attributes for a method of class `X`: It collects the attributes of the class `X` and combines them with the attributes implemented interfaces of class `X`. If a class `X` does not define any attributes, it adds the attributes of its parent class. For details of the attribute merging strategy, we refer to `AttributesCollector`. The class `CommonsAttributes` is a redirector to the implementation of commons attributes.

3.3.3.2 How to define an attribute

Defining an attribute is very easy. You just have to add a line in your javadoc that begins with `@@`. Here's an example:

```
/**
 * This is normal javadoc.
 *
 * @@attrib.transaction.RequiredReadOnly()
 */
public FileDTO getFileByKey(FilePK key) throws ObjectDoesNotExistException {
    ...
}
```

The class `attrib.transaction.RequiredReadOnly` is used as an attribute although it is a normal java class. If the attributes of this method are requested, an instance of class `attrib.transaction.RequiredReadOnly` will be returned. In this case, the default constructor of class `attrib.transaction.RequiredReadOnly` will be used, but it is possible to pass arguments like in the following example:

```
/**
 * Remove keyword. Primary key will be used.
 *
 * @param key
 *         Is the primary key of the keyword, which should be deleted.
 * @throws ObjectDoesNotExistException
 *
 * @@attrib.transaction.RollbackRule(ObjectDoesNotExistException.class)
 * @@attrib.transaction.RollbackRuleOnRuntimeException()
 * @@attrib.transaction.RollbackRuleOnError()
 */
public void removeKeyword(KeywordPK key) throws ObjectDoesNotExistException;
```

In the first attribute of this example, the constructor of class `attrib.transaction.RollbackRule` will receive a `java.lang.Class` as parameter. There is a further possibility to set values of setter methods, but this is not used by this module. For more information, please contact the [commons attributes homepage](#).

3.3.3.3 How to organize the transaction attributes

Attributes make both sense on the class and the method level (method-level attributes completely override those on the class) and attributes also make sense on the interface and the implementation level (overriding rules are not so simple in this case).

The following example shall illustrate the use of attributes on interfaces. It shows a transactional method on an interface. For someone using the interface, it is important to know, *in what cases* the transaction is rolled back.

```
/**
 * Remove keyword. Primary key will be used.
 *
 * @param key
 *         Is the primary key of the keyword, which should be deleted.
 * @throws ObjectDoesNotExistException
 *
 * @@attrib.transaction.RollbackRule(ObjectDoesNotExistException.class)
 * @@attrib.transaction.RollbackRuleOnRuntimeException()
 * @@attrib.transaction.RollbackRuleOnError()
 */
public void removeKeyword(KeywordPK key) throws ObjectDoesNotExistException;
```

The defined attributes indicate what exceptions make the method's transaction roll back:

- [ObjectDoesNotExistException](#)? (and sub exceptions)
- java.lang.Error (and sub exceptions)
- java.lang.RuntimeException (and sub exceptions)

Now we have defined what has to be happen on each exception case, but we did not define the transaction propagation behavior. Because the propagation behavior depends on the implementation, we have to define this attribute on implementation level. Here's the corresponding example:

```
/**
 * @see ServiceInterface
 *
 * @@attrib.transaction.RequiredRuleBased()
 */
public void removeKeyword(KeywordPK key) throws ObjectDoesNotExistException {
    ...
}
```

During runtime the RollbackRule attributes will be linked with the attribute `attrib.transaction.RequiredRuleBased` by the `TransactionAttributeSource`. This method can be overwritten for other propagation behaviors.

3.3.3.4 Transaction propagation behaviors

Here an overview of propagation behaviors:

- **required**: execute within a current transaction, create a new transaction if none exists.
- **requires new**: create a new transaction, suspending the current transaction if one exists.
- **supports**: execute within a current transaction, execute nontransactionally if none exists.
- **not supported**: execute nontransactionally, suspending the current transaction if one exists.
- **mandatory**: execute within a current transaction, throw an exception if none exists.
- **never**: execute nontransactionally, throw an exception if a transaction exists.

The default is **required**, which is typically the most appropriate. For more documentation, please refer to the spring or the EJB documentation.

3.3.3.5 Transaction attribute classes

Here are all available classes from package `attrib.transaction`:

Class name	Standalone
Required	✓
RequiredReadOnly	✓
RequiredRuleBased	
RequiresNew	✓
RequiresNewReadOnly	✓
RequiresNewRuleBased	
Supports	✓
SupportsReadOnly	✓
SupportsRuleBased	
NotSupported	✓
Mandatory	✓
MandatoryReadOnly	✓
MandatoryRuleBased	
Never	✓

Methods using a class as attribute that is marked as `Standalone` in the above table, do not require another class as attribute from package `attrib.transaction`. All the other classes (*RuleBased) must be used in combination with at least one `RollbackRule` or one `NoRollbackRule`. If no rollback rule is defined, the active transaction will not be rolled back in case of any exception, error or throwable.

The following table illustrates the different rollback rules:

Class name	Description
<code>RollbackRule</code>	This class needs a <code>java.lang.Class</code> as first parameter for the constructor. If on the declared method an exception of given <code>java.lang.Class</code> is thrown, the transaction will be rolled back.
<code>RollbackRuleOnError</code>	This class extends the class <code>RollbackRule</code> and rolls the transaction back, if a <code>java.lang.Error</code> is thrown.
<code>RollbackRuleOnRuntimeException</code>	This class extends the class <code>RollbackRule</code> and rolls the transaction back if a <code>java.lang.RuntimeException</code> is thrown.
<code>NoRollbackRule</code>	This class is the same but does the opposite of class <code>RollbackRule</code> .

The difference between the attributes `XYZRuleBased` (i.e. `RequiredRuleBased`) and attribute `XYZ` (i.e. `Required`) is explained in the following example. The transactional behaviour of method `a()` and `b()` is the same.

```
class XY {
    /**
     * @attrib.transaction.RollbackRuleOnRuntimeException()
     * @attrib.transaction.RollbackRuleOnError()
     * @attrib.transaction.RequiredRuleBased()
     */
    a();

    /**
     * @attrib.transaction.Required()
     */
    b();
}
```

The difference between attribute `XYZReadOnly` (i.e. `RequiredReadOnly`) and attribute `XYZ` (i.e. `Required`) is that in the read only case, the transaction will be optimized for read only. ***Read only does not mean that an exception will be thrown if a commit will be executed! Commit will work as usual.***

3.3.3.6 Adding spring beans to enable transactional behaviour

If you have added transaction attributes to the source code, you are now able enable transactions in your project. Just add the following spring bean configuration file:

```
<?xml version="1.0" encoding="ISO-8859-15"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!--
        General AOP definitions.
    -->
    <bean id="autoproxy"
        class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

    <!--
        Transaction configurations.
    -->
    <bean id="transactionAdvisor"
        class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
        <constructor-arg>
            <ref local="transactionInterceptor"/>
        </constructor-arg>
    </bean>
</beans>
```

```

        </constructor-arg>
    </bean>
    <bean id="transactionInterceptor"
        class="org.springframework.transaction.interceptor.TransactionInterceptor">
        <property name="transactionManager">
            <ref local="transactionManager"/>
        </property>
        <property name="transactionAttributeSource">
            <ref local="transactionAttributeSource"/>
        </property>
    </bean>
    <bean id="transactionAttributeSource"
        class="org.springframework.transaction.interceptor.AttributesTransactionAttributeSource">
        <constructor-arg>
            <ref local="attributesCollector"/>
        </constructor-arg>
    </bean>

    <!--
        General attribute definitions.
    -->
    <bean id="attributesCollector"
        class="ch.elca.el4j.util.attributes.AttributesCollector">
        <constructor-arg>
            <ref local="commonsAttributes"/>
        </constructor-arg>
    </bean>
    <bean id="commonsAttributes"
        class="org.springframework.metadata.commons.CommonsAttributes"/>

    <!--
        Individual configurations
    -->
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource">
            <ref bean="dataSource"/>
        </property>
    </bean>
</beans>

```

The latest section declares individual configurations. In this example, a `DataSourceTransactionManager` is used, so there should be a bean with name `dataSource` which implements the interface `javax.sql.DataSource`.

The following two configuration files of **module-core** have these beans already defined:

- `scenarios/db/rawDatabase.xml`
 - ◆ This file contains the transaction manager and the data source.
- `optional/interception/transactionCommonsAttributes.xml`
 - ◆ This file contains every other needed bean.

3.3.3.7 Programmatical transaction demarcation (start transaction, commit, rollback in code)

First, do not use this if it is not really necessary. Mostly you can separate your code in methods and use transaction attributes.

You can get the bean `transactionManager` that is defined in file `scenarios/db/rawDatabase.xml` of **module-core** and cast it to `org.springframework.transaction.PlatformTransactionManager`. On this class, you can call methods directly. Please use in addition the attributes, which are defined in **module-core** (attrib.transaction.*).

3.3.3.8 setRollbackOnly is not equals to setReadOnly

In this module there are attributes which name ends with **ReadOnly**. If this kind of attributes are used it is **still possible** to commit changes. This **property** will be only set in the JDBC properties to help intelligently implemented JDBC drivers to optimize connection creation. This means that a JDBC driver can, but must not read this property.

The **setRollbackOnly** method of class **org.springframework.transaction.TransactionStatus** is used to guarantee that the current transaction is rolled back. This property of class **TransactionStatus** can be set in code and the current **TransactionStatus** can be retrieved by invoking the static method **org.springframework.transaction.interceptor.TransactionAspectSupport.currentTransactionStatus()**.

3.3.4 Internal design

These classes just extend the corresponding attribute classes from spring.

3.4 Attribute Convenience

3.4.1 Purpose

In spring, attributes can be used to enable the right interceptors. However, adding support for a new attribute to spring requires to implement a few new classes. The new classes are typically quite redundant (and they are often implemented via cut and paste). It is the goal of the following helper classes (GenericAttributeAdvisor and a DefaultGenericAttributeSource) to alleviate this.

3.4.2 Configuration

Here's a sample configuration file:

```
<beans>
  <!--
    Define the autoproxy bean which looks for each advisor in this context.
  -->
  <bean id="autoproxy"
    class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

  <!-- Define the Advisor bean. -->
  <bean id="genericAttributeAdvisor"
    class="ch.elca.el4j.util.attributes.GenericAttributeAdvisor">
    <constructor-arg>
      <ref local="exampleInterceptor"/>
    </constructor-arg>
    <property name="interceptingAttributes">
      <list>
        <value>ch.elca.el4j.tests.util.attributes.ExampleAttributeOne</value>
      </list>
    </property>
  </bean>

  <!-- Define the interceptor to be used by the above defined advisor. -->
  <bean id="exampleInterceptor"
    class="ch.elca.el4j.tests.util.attributes.ExampleInterceptor">
  </bean>

  <!-- Define the bean which owns a method that should be intercepted. -->
```

```
<bean id="foo" class="ch.elca.el4j.tests.util.attributes.Foo"/>
</beans>
```

- The `autoproxy` bean is still needed since it looks for Advisors.
- The `genericAttributeAdvisor` bean extends the `org.springframework.aop.support.StaticMethodMatcherPointcutAdvisor`.
 - ◆ The Advice, e.g. a `MethodInterceptor` can be injected via the constructor argument. It is necessary to define one, otherwise, an exception is thrown.
 - ◆ The property `interceptingAttributes` takes a list of Attributes. The defined interceptor will be invoked if one of these Attributes is defined at a method/class.
- The `exampleInterceptor` bean extends `org.aopalliance.intercept.MethodInterceptor`. It also implements `ch.elca.el4j.util.attributes.AttributeSourceAware` which sets the Attribute Source of this Interceptor since the Interceptor needs to access the Attributes.
- The `foo` bean is a bean having a method `test(int)` where an `ExampleAttributeOne` is declared. Therefore, a call to `foo.test(int)` will invoke this Interceptor.

3.4.3 Extension for more attributes

The `DefaultGenericAttributeSource` is designed to handle one attribute. Assume there is a method with more than one attribute defined that matches the ones defined at the property `interceptingAttributes` of bean `genericAttributeAdvisor`. In this case `DefaultAttributeSource.getAttribute(Method, Class)` will return any one attribute in a nondeterministic way due to Spring interface definitions of `org.springframework.metadata.Attributes`.

If you need to access more than one or all of these attributes in your interceptor, please read the following chapter.

3.4.3.1 How to access all the attributes

The most convenient way is to add a member variable (here `m_associatedAttributes`) to an attribute (here `CollectingAttribute`) in order to save all the intercepting attributes in this variable and then return this attribute. The correct functionality can be achieved by overriding method `protected Object findAttribute(Collection atts)` of class `ch.elca.el4j.utils.attributes.DefaultGenericAttributeSource`. A code sample of how this overriding method could look like:

```
protected Object findAttribute(Collection atts) {
    if (getInterceptingAttributes() == null || getInterceptingAttributes().size() == 0) {
        String message = "There is no attribute defined which will be intercepted.";
        s_logger.error(message);
        throw new BaseRTException(message, (Object[]) null);
    }

    if (atts == null) {
        return null;
    }

    CollectingAttribute attribute = null;

    // Check whether there is a CollectingAttribute.
    for (Iterator itr = atts.iterator(); itr.hasNext() & attribute == null; ) {
        Object att = itr.next();
        if (att instanceof CollectingAttribute) {
            attribute = (CollectingAttribute) att;
        }
    }
}
```

```

// In case there is a CollectingAttribute, add the other matching attributes to the CollectingAttribute
if (attribute != null){
    List associatedAttributes = new LinkedList();
    for (Iterator it = atts.iterator(); it.hasNext(); ) {
        Object att = it.next();
        boolean found = false;
        for (Iterator intatts = getInterceptingAttributes().iterator(); intatts.hasNext() &!found;) {
            String className = (String) intatts.next();
            try {
                Class cl = Class.forName(className);
                if (cl.isInstance(att)) {
                    associatedAttributes.add(att);
                    found = true;
                }
            } catch (ClassNotFoundException e) {
                String message = "The class '" + className + "' does not exist.";
                s_logger.error(message);
                throw new BaseRuntimeException(message, e);
            }
        }
    }
    attribute.setAssociatedAttributes(associatedAttributes);
}

return attribute;
}

```

The following code in the interceptor's public `Object invoke(MethodInvocation methodInvocation)` method can extract the attributes:

```

CollectionAttribute att = null;
Object obj = m_attributeSource.getAttribute(methodInvocation.getMethod(), methodInvocation.getClass())

if (obj instanceof CollectionAttribute) {
    att = (CollectionAttribute) obj;
}

List attributes = att.getAssociatedAttributes();

```

In order to have access to the Attribute Source, this interceptor has to implement `ch.elca.el4j.util.attributes.AttributeSourceAware` that sets the Attribute Source.

3.4.4 Internal design

3.4.4.1 Classes

All classes of this feature are in the package `ch.elca.el4j.util.attributes`:

- **GenericAttributeAdvisor**: All advisors are collected by the `autoproxxy` bean. In case no `AttributeSource` is set in the configuration file. This class will generate an `AttributesCollector` which collects the `Attributes` in the classes and in the interfaces of a certain class. This `AttributesCollector` uses the `CommonsAttributes` implementation by default.
- **GenericAttributeSource**: The interface for the Attribute Source.
- **DefaultGenericAttributeSource**: The default implementation of the `GenericAttributeSource` interface. This class is suitable for most needs (see [Extension for more attributes](#)). It also includes caching of the `Attributes`.
- **AttributeSourceAware**: An interceptor that wants to access the `Attributes` defined at the joinpoint has to implement this interface.

3.5 Additional Features

3.5.1 Configuration merging via property files

The class `ch.elca.el4j.core.config.ListPropertyMergeConfigurer` can be used to add items to a list on an existing configuration. Here an example.

xml-config-file.xml:

```
<beans>
  <bean id="configurationTest"
    class="ch.elca.el4j.core.config.ListPropertyMergeConfigurer">
    <property name="location">
      <value>myconfig/mergeable-config-file.properties</value>
    </property>
  </bean>

  <bean id="listTest" class="ch.elca.el4j.tests.core.config.ListClass">
    <property name="abcList">
      <list>
        <value>item 0</value>
      </list>
    </property>
  </bean>
</beans>
```

mergeable-config-file.properties:

```
listTest.abcList=item 2, item 3
```

If the `xml-config-file.xml` is loaded in an application context the property `abcList` of bean `listTest` contains items 0, 2 and 3.

For more information have first a look at the javadoc of the spring class

`org.springframework.beans.factory.config.PropertyOverrideConfigurer` and then have a look at

<http://leaffy.elca.ch/java/el4j/snapshot/website/framework/javadoc/framework-modules/ch/elca/el4j/core/config/ListPropertyMergeConfigurer>

3.5.2 Bean locator

The class `ch.elca.el4j.core.beans.BeanLocator` can be used to get all beans in an application context, which are an instance of specific type (interface or class) or have a specific bean name. It is also possible to exclude beans. For more information have a look at

<http://leaffy.elca.ch/java/el4j/snapshot/website/framework/javadoc/framework-modules/ch/elca/el4j/core/beans/BeanLocator>

3.5.3 Bean type auto proxy creator

The class `ch.elca.el4j.core.aop.BeanTypeAutoProxyCreator` allows autoproxying beans by their type. It helps e.g. to use marker interfaces (such as [ServiceInterface](#)[?]/ DAO) that are then used more consistently than can bean naming conventions.

Using a pointcut with a class filter would solve the problem too. It requires writing a new static advisor that configures a `RootClassFilter` and that accepts a list of interceptors. Finally, a `DefaultAdvisorAutoProxyCreator` is required to proxy all classes. Using the `BeanTypeAutoProxyCreator` is much easier.

3.5.4 Exclusive bean name auto proxy creator

This auto proxy creator extends Spring's `BeanNameAutoProxyCreator`. It allows setting a list of name patterns of beans to exclude. The pattern can reference a distinct bean, a prefix or a bean name's suffix. If you don't declare an include pattern (i.e. using the `beanNames` property), all beans will be proxied, except the ones matching the exclude patterns. **Note** Exclusion patterns have higher priority.

Configuration Example

```
<bean id="exclusiveNameAutoProxy"
  class="ch.elca.el4j.core.aop.ExclusiveBeanNameAutoProxyCreator">
  <property name="exclusiveBeanNames"><value>foo*</value></property>
  <property name="interceptorNames">
    <list>
      <value>shortcutInterceptor</value>
    </list>
  </property>
</bean>
```

3.5.5 Abstract parent classes for the Typesafe Enumerations Pattern

An Enumeration is a type that can hold a value from a set of well defined values. We provide 2 super classes for the immutable enumeration pattern: one `java.lang.Comparable` and the other one not comparable. For an example, please have a look at the javadoc:

- <http://leaffy.elca.ch/java/el4j/snapshot/website/framework/javadoc/framework-modules/ch/elca/el4j/util/codingsupport/F>
- <http://leaffy.elca.ch/java/el4j/snapshot/website/framework/javadoc/framework-modules/ch/elca/el4j/util/codingsupport/F>

Remark: if you use only JDK 1.5, there is a more convenient enumeration mechanism in the core language.

3.5.6 Reject (Precondition checking)

As described in the [ExceptionHandlingGuidelines](#), we use the class `ch.elca.el4j.util.codingsupport.Reject` for precondition checking of a method. Have a look at the javadoc for an example:

<http://leaffy.elca.ch/java/el4j/snapshot/website/framework/javadoc/framework-modules/ch/elca/el4j/util/codingsupport/F>

3.5.7 JNDI Property Configurers

The JNDI property configurers get their values from a JNDI context. Default is `java:comp/env`. This can be overridden by setting the appropriate value in a `JndiConfigurationHelper`, which is injected into a JNDI property configurer.

For a `JndiPropertyPlaceholderConfigurer`, the values are queried one after another. There's no magic there. However, a `JndiPropertyOverrideConfigurer` needs to get the whole list of properties to override. The default strategy is to use a prefix. Default is `springConfig`. (notice the separating point at the end). Another possibility is to put override properties into a distinct context that allows you neglecting the prefixes (however you need to inject a configured `JndiConfigurationHelper` and you have to set the prefix to `null`).

3.5.8 DTO helpers

This package supports optimistic locking on the DTO level. Available is an abstract DTO that holds a primary key generator to realize the optimistic locking and an extended version of the abstract DTO that contains in addition the primary key named as `key` in form of a string. To have the primary key generator set for every DTO it is necessary to create DTOs by using the DTO factory, which can be found in this package too.

3.5.9 Primary key

This package contains an interface, which defines an `PrimaryKeyGenerator` with a method to generate a primary key as a string. Implemented is a `UuidPrimaryKeyGenerator` that always returns string primary keys with 32 characters [0–9a–z].

3.5.10 SQL exception translation

This package contains exceptions (subclasses of spring's [DataAccessExceptions](#)[?]). These exceptions complement the exception hierarchy of spring for duplicated values and too big values. When to throw which exception and for which database these configurations are valid can be found in this module's conf folder in file `sql-error-codes.xml`.

3.6 Packages that implement the core module

- `ch.elca.el4j.core.**`
- `ch.elca.el4j.services.persistence.generic.**`
- `ch.elca.el4j.services.persistence.ibatis.**` (could be moved into separate module)
- `ch.elca.el4j.services.monitoring.notification.CoreNotificationHelper`
- `ch.elca.el4j.util.**`
- `attrib.**`

Notes:

- `**` means all files from the current package and all sub packages.
- The full package structure of EL4J can be viewed [here](#).

4 Documentation for module web

4.1 Purpose

Web Module of EL4J. It includes struts, servlet-api, some commons libraries and a few own classes.

4.2 Features

The following features are included in this module:

- The `ModuleWebApplicationContext`. It decouples configuration location pattern interpretation from the current classloader.
- Implementation of the `SynchronizerToken`. This is useful for preventing duplicate form submissions. Further information under <http://www.javaworld.com/javaworld/javatips/jw-javatip136.html>. You can see an example of the Synchronizer Token in the `WebApplicationTemplate`.
- Xml Tag Transformer. Escapes xml tags in order to display them properly on web pages.

4.3 How to use

4.3.1 General configuration of the web module

The module web application context is used like its non-web counterpart (`ModuleApplicationContext`). For a sample usage of the other features, please refer to the web application template (the demo application).

4.4 Reference documentation for the Module-aware application contexts

The `ModuleWebApplicationContext` resolves issues that arise with web container class loaders. In contrast to standalone applications, web applications can't provide their classpath through a command line parameter or through environment parameters. The Servlet specification replaces the missing parameter with `Class-Path` entries in the `MANIFEST.MF`. Unfortunately, they're not respected by every servlet container.

The very same classloader issues appear also in environments other than web containers. The `ModuleApplicationContext` resolves absolutely the same problems using the same mechanisms. The following description applies to both application contexts.

4.4.1 Concept

Each jar from an EL4J module contains a manifest file with its module's name, its dependencies to other modules and a list of all configuration files it contains. The `ModuleWebApplicationContext` searches for all manifest files that are in the classpath, extracts their information and builds the complete module hierarchy. Then it creates a list of all provided configuration files, preserving the modules' hierarchy. The ordered list is used to fulfill any resource look-up queries.

In general, this resolves any problems with wildcard notation (e.g. `classpath*:mandatory/*.xml`): it's guaranteed, that all mandatory files of a module A are loaded before them from module B, if B depends on A). Further, some classloaders have problems recognizing jar files as jars and instead show them as zip files. Spring's pattern resolvers work with jars only, running into troubles if a jar is wrongly taken for a zip. Since the pattern resolver used together with the `ModuleWebApplicationContext` works on the internal module structure only, there's no dependency on the current environment's classloader.

The `ModuleWebApplicationContext` can resolve only files that are added to the corresponding attribute in the manifest file. In general, this is just a subset of resources that are loaded during the application's lifecycle.

Hence our custom pattern resolver that works on the internal module representation delegates each unsatisfied request to the standard Spring resource loading mechanism.

So, this solution uses the same infrastructure as the one defined in the servlet specification. However, the processing is done by EL4J, and hence doesn't depend on any servlet container and their specific behavior.

4.4.1.1 ModuleDispatcherServlet

To simplify the usage of the ModuleWebApplicationContext, there's the ModuleDispatcherServlet that configures a Spring DispatcherServlet. It behaves absolutely the same as the one of Spring. Additionally, it allows defining two lists of configuration files which are included and excluded respectively.

Note: You don't have to use any of them. Spring's DispatcherServlet configuration style is still available.

Example configuration making use of the include / exclude feature:

```
<servlet>
  <servlet-name>remotingtests</servlet-name>
  <servlet-class>
    ch.elca.el4j.web.context.ModuleDispatcherServlet
  </servlet-class>
  <load-on-startup>100</load-on-startup>
  <init-param>
    <param-name>inclusiveLocations</param-name>
    <param-value>WEB-INF/remotingtests-servlet.xml</param-value>
  </init-param>
  <init-param>
    <param-name>exclusiveLocations</param-name>
    <param-value>foobar.xml</param-value>
  </init-param>
</servlet>
```

Without the need of the exclusive list (the standard DispatcherServlet's naming convention is used, hence the `benchmark-servlet.xml` gets loaded in this context):

```
<servlet>
  <servlet-name>benchmark</servlet-name>
  <servlet-class>
    ch.elca.el4j.web.context.ModuleDispatcherServlet
  </servlet-class>
  <load-on-startup>100</load-on-startup>
</servlet>
```

4.4.2 Build system integration

We provide a hook task for the EL4Ant build system that gathers all the needed configuration information automatically and writes them into the manifest file. In the default mode, it simply collects all files that are controlled by the [resources plugin](#).

4.4.2.1 Adding files manually

While adding files automatically to the manifest file is most of the time comfortable, it's sometimes necessary to specify the list of files manually. Using an attribute in a module's or execution-unit's definition adds them to the manifest file. **Note:** Either you use the automatic or the manual way. It's not possible to append manually defined files to the ones gathered by the plugin.

```
<module name="module-light_statistics" path="module/light_statistics">
  <eu name="jmx">
    <!-- example configuration for manifest configuration files within an execution-unit -->
```

```

        <attribute name="manifestconfigsection.files" value="config/a.xml,mandatory/b.xml" />
    </eu>
</module>

```

4.4.2.2 Installing the build system hook

The EL4Ant build system plugin can be downloaded from [the EL4Ant plugins](#). Copy it to a directory that is reachable from your project (e.g. `etc/hooks/manifest-config-section-hook.xml`) and add the following snippet to your project's definition (`project.xml`):

```

<plugin name="manifest-config-section-hook"
    file="etc/hooks/manifest-config-section-hook.xml" />

```

4.4.3 Limitations

- Our custom resource pattern resolver handles wildcard classpath resources only, i.e. location patterns starting with `classpath*`: (with or without a wildcard pattern in the part following). Any other location pattern is resolved through delegation.
 - Overloading files with the same name is still a problem (e.g. file `a` in module `A` and file `a` in module `B` that depends on module `A`). It's not predictable which file is loaded. It's highly recommended to use unique file names. In some cases, this is not possible (e.g. the `log4j.xml`). Put these files into the `java` folder, if possible. They are correctly handled there by the build system.
 - Potentially, every response to a given request is incomplete, if answered by the custom module pattern resolver. The pattern resolver delegates unsatisfied requests only. Requests for which at least one resource is found are not handled by Spring's pattern resolver. Specifying configuration files manually may resolve the problem. See [the earlier section on adding files manually](#) for details.
-

4.4.4 MANIFEST.MF configuration section format

Of course, the manifest file can also be written by hand. Here is the format of the configuration section:

Add to the manifest of each module

1. the module's dependencies (`Dependencies`)
2. the list of all the configuration files it defines (`Files`)
3. the name of the module (actually the name of the jar file) (`Module`)

Example:

- 3 Modules: `A,B,C`
- `B` depend on `A`
- `C` depends on `A`

`B` contains `b1.xml`, `b2.xml`

`C` contains `c.xml`

`A` contains `a.xml`

The manifest of `A` contains

```

Name: config
Module: A
Files: a.xml
Dependencies:

```

The manifest of B contains

```
Name: config
Module: B
Files: b1.xml b2.xml
Dependencies: A
```

The manifest of C contains

```
Name: config
Module: C
Files: c.xml
Dependencies: A
```

4.4.5 Implementation Alternative: Idea

The resource pattern resolver delegates single-resource requests to one of Spring's pattern resolvers. That's because the configuration file list contained in a manifest file provides classpath-relative paths only. This paths could be made absolute using the manifest file's path as prefix. This would resolve problems with equally named resources. **Note:** loading all resources from the classpath using the `classpath*:` prefix requires top-down processing of the module hierarchy whereas loading of a single resource (i.e. using the `classpath:` prefix) bottom-up. Not sure if it works in all environments.

Example Manifest file location:

```
file:/C:/el4j/framework/lib/module-core_1.0.jar!/META-INF/MANIFEST.MF
```

Configuration files' prefix:

```
file:/C:/el4j/framework/lib/module-core_1.0.jar!/
```

4.4.6 Resources

- `ModuleWebApplicationContextToDo` specifies the problem more extensively
- `ModuleWebApplicationContextToDoSpecification` solution specification

5 Documentation for module remoting

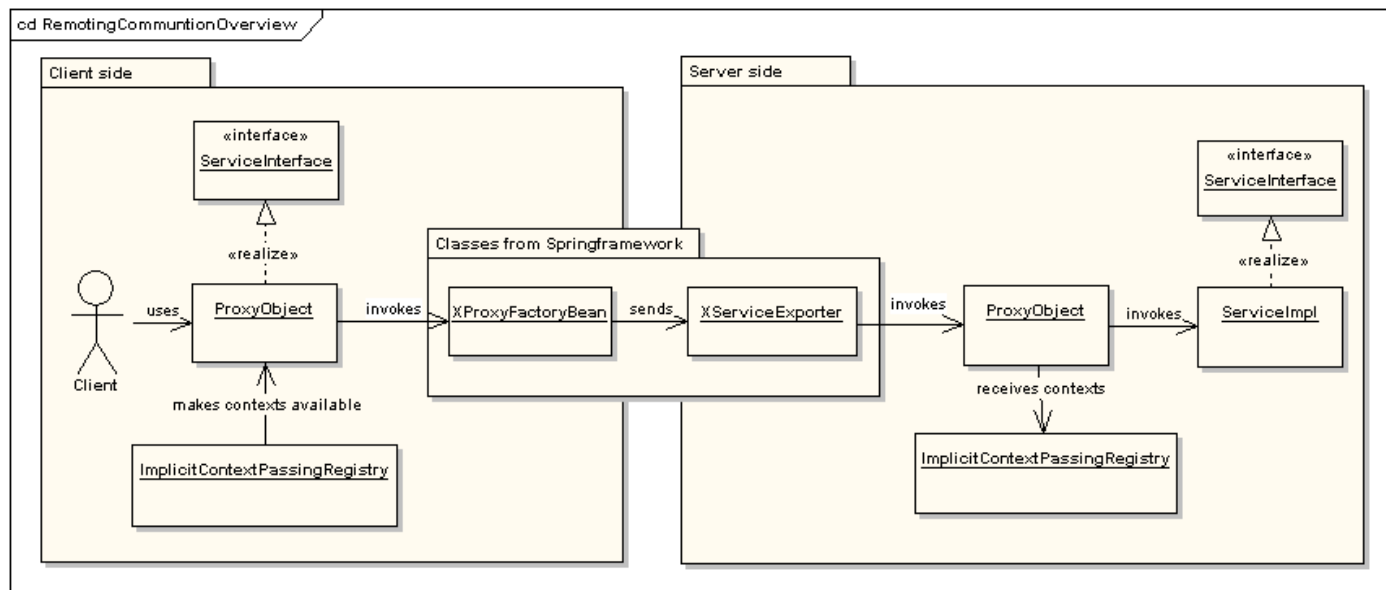
5.1 Purpose

Convenience module for spring POJO remoting: (1) allows **centralized protocol configuration**, (2) simplifies protocol switching (currently between **RMI**, **Hessian**, **Burlap**, **Soap** and **EJB**), and (3) transparently enriches interfaces for **automatic implicit context passing**.

5.2 Introduction

The Spring framework offers an easy way to distribute POJOs. Available protocols are Rmi, Hessian and Burlap. This module provides in addition implicit context passing. In addition, attention was paid to be able to distribute hundreds of services with a minimum of configuration.

The general idea is to internally use Spring's implementations and offer a proxy object to the outside. This is made on the client and on the server side (see picture).

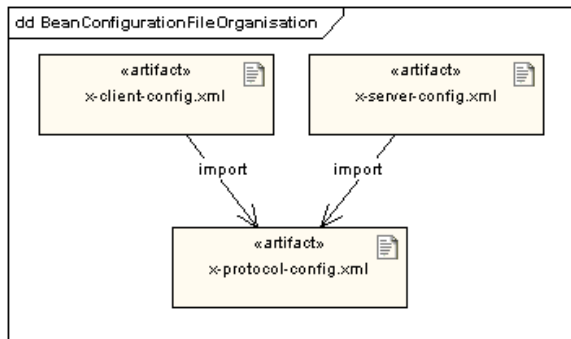


This module can also be used if you only develop the server or client side!

5.3 How to use

5.3.1 Configuration

5.3.1.1 Recommended configuration file organisation



Typically we have three configuration files. One for the server, one for the client and one which is shared between server and client. We present first the file that is shared between the server and the client, the `x-protocol-config.xml`. The `x` stands for the protocol such as `rmi`, `hessian` or `burlap`.

5.3.1.1.1 x-protocol-config.xml

This file contains the following for the protocol `rmi`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="remoteProtocol" class="ch.elca.el4j.services.remoting.protocol.Rmi">
        <property name="serviceHost">
            <value>localhost</value>
        </property>
        <property name="servicePort">
            <value>1099</value>
        </property>
        <property name="implicitContextPassingRegistry">
            <ref local="implicitContextPassingRegistry" />
        </property>
    </bean>
    <bean id="implicitContextPassingRegistry" class="ch.elca.el4j.tests.remoting.service.TestImplicitCont
</beans>
```

In this configuration file, we have only two beans defined. One bean for the remoting protocol and one for implicit context passing registry. Each bean that defines a remote protocol needs protocol-specific properties. In addition a reference to a class, which implements the interface `ImplicitContextPassingRegistry` is necessary, if you want to use the implicit context passing feature.

It is possible to have many beans that define a remoting protocol. In the example above it is the `rmi` remoting protocol. This requires the `serviceHost`, where the service is running and it also needs to know the `servicePort`. For the remoting protocol `rmi`, these two properties are mandatory. The other two predefined protocols (`hessian` and `burlap`) need additionally the property `contextPath` that defines in which webserver context the service is running.

5.3.1.1.2 x-client-config.xml

This file contains the following for the protocol `rmi` when we want to get access to the remote calculator bean.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
```

```

<import resource="rmi-protocol-config.xml"/>

<bean id="calculator" class="ch.elca.el4j.services.remoting.RemotingProxyFactoryBean">
  <property name="remoteProtocol">
    <ref bean="remoteProtocol" />
  </property>
  <property name="serviceInterface">
    <value>ch.elca.el4j.tests.remoting.service.Calculator</value>
  </property>
</bean>
</beans>

```

The first element imports the previous discussed ***x-protocol-config.xml*** file. In this way, we can set the property `remoteProtocol` to a bean that is defined in the file ***x-protocol-config.xml***. The second property `serviceInterface` has to be the business interface. These two properties are mandatory.

5.3.1.1.3 x-server-config.xml

This file contains the following for the protocol `rmi`:

```

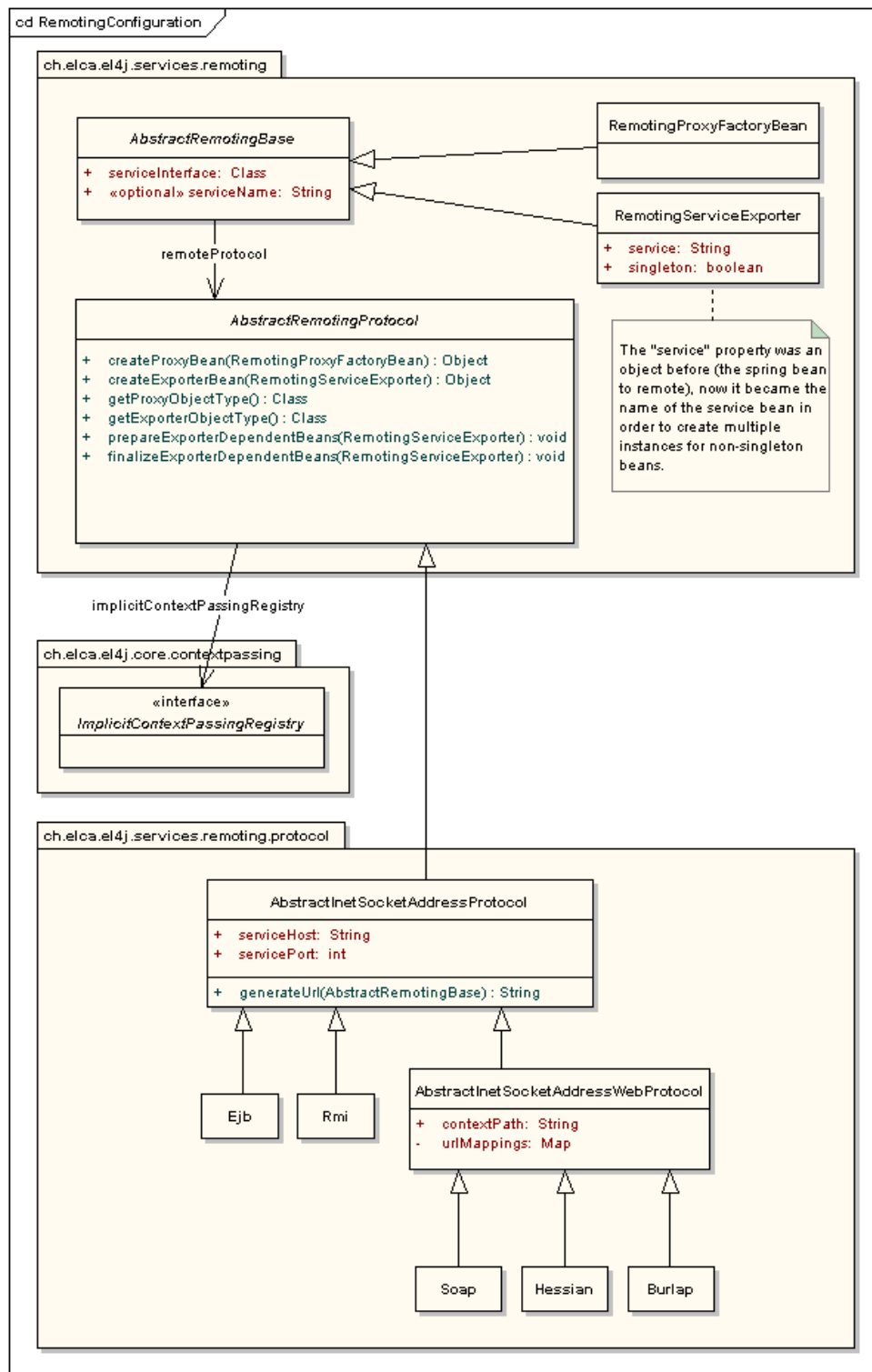
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <import resource="rmi-protocol-config.xml"/>

  <bean id="calculatorExporter" class="ch.elca.el4j.services.remoting.RemotingServiceExporter">
    <property name="remoteProtocol">
      <ref bean="remoteProtocol" />
    </property>
    <property name="serviceInterface">
      <value>ch.elca.el4j.tests.remoting.service.Calculator</value>
    </property>
    <property name="service">
      <ref local="calculatorImpl" />
    </property>
  </bean>
  <bean id="calculatorImpl" class="ch.elca.el4j.tests.remoting.service.impl.CalculatorImpl" />
</beans>

```

The first element imports also the ***x-protocol-config.xml*** file, like the client config does. The second property is also the `serviceInterface`. The difference to the client configuration is that the server configuration needs a reference to the service implementation. The bean for this implementation can be found as second bean definition in this configuration file. These three properties are mandatory.

5.3.1.2 Configuration summary



This picture describes the configuration information needed. On top you can find the base class **AbstractRemotingBase** that shares the common part between client (**RemotingProxyFactoryBean**) and server side (**RemotingServiceExporter**). This base class always needs to know the service interface and it also needs a reference to a class that extends **AbstractRemotingProtocol** such as **Rmi**, **Hessian** or **Burlap**.

While the class **RemotingProxyFactoryBean** does not need something more, the class **RemotingServiceExporter** needs additionally to the properties from the extended class a reference to the

implemented service. The service must naturally implement the `serviceInterface`.

The property `serviceName` of the base class is optional. It only must be set manually, if the given `serviceInterface` is used twice or more on the same server. If the property `serviceName` is not set, what is normally the case, it will be generated out of the name of the `serviceInterface` and the suffix `.remoteservice`. The suffix `.remoteservice` is needed in webserver to be able to know which requests have to be redirected to the `DispatcherServlet` from Spring. More details follows below.

The class **`AbstractRemotingProtocol`** can have a reference to a class that implements the interface **`ImplicitContextPassingRegistry`**. If such a reference exist, the implicit context passing will be enabled.

The abstract class **`AbstractInetSocketAddressProtocol`** has two required properties. The first is the `serviceHost` which must be the host and the second is the `servicePort` which is the port, where the service is running. **`Rmi`** directly extends this class.

Protocols that are running in a webserver must additionally know in which `contextPath` they are running. This is solved by the abstract class **`AbstractInetSocketAddressWebProtocol`**. This property `contextPath` is mandatory. Inside the webserver, the mapping of services is done automatically by this abstract class. There are two classes that directly extend this abstract class, the **`Hessian`** and the **`Burlap`** protocol.

5.3.1.3 How to use the **`Rmi`** protocol

The introduction the remoting module in the previous section of the document was made with RMI. So please refer there for general information about remoting with RMI. For additional constraints and implementation details about the RMI remoting, please refer to the last subchapter of this section.

Important points:

- If on host `serviceHost` no rmi registry is running on port `servicePort`, Spring will automatically start a rmi registry.
- The server side must naturally be started before the client side.

5.3.1.4 How to use the **`Hessian`** protocol

The usage of the **`Hessian`** protocol on the client side is the same as the **`Rmi`** protocol.

The server side must be started in a webserver. To realize this, take the following steps.

5.3.1.4.1 Create a web-deployable module

By using the **`el4ant`** build system you can create a module that can be deployed on a webserver such as tomcat. First you have to have the plugin for tomcat installed. This could look like the following snippet:

```
<plugin name="j2ee-web-tomcat">
  <attribute name="j2ee-web.container" value="tomcat"/>
  <attribute name="j2ee-web.mode" value="directory"/>
  <attribute name="j2ee-web.home" value="../../external-tools/tomcat"/>
  <attribute name="j2ee-web.port" value="8080"/>
  <attribute name="j2ee-web.manager.username" value="admin"/>
  <attribute name="j2ee-web.manager.password" value="password"/>

  <attribute name="j2ee-war.unpacked" value="true"/>
</plugin>
```

It is highly recommended to define the attribute `j2ee-web.home` relatively to your EL4J project to have the file in your CVS/SVN operating system independent.

After you have added this plugin you can define your module with the following lines:

```
<module name="mymodulename" path="here/is/my/module">
  ...
  <attribute name="runtime.runnable" value="true"/>
  <attribute name="j2ee.war.application"/>
  <attribute name="runtime.command.creator" value=" runtime.command.creator.web"/>
  ...
</module>
```

Of course you have to add at least a dependency to the `module-remoting_and_interface_enrichment` in this module.

Now you can deploy the module and start tomcat via the corresponding ant task, generated by el4ant.

5.3.1.4.2 Register Spring's DispatcherServlet

To register a servlet you have to create a folder **webapp** in your newly created module and in this folder a folder with name **WEB-INF**. Now you have to create a file with name **web.xml** and the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">

  <servlet>
    <servlet-name>remote</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>remote</servlet-name>
    <url-pattern>*.remoteservice</url-pattern>
  </servlet-mapping>
</web-app>
```

If you already have a **web.xml** file, just add the two elements `servlet` and `servlet-mapping`. If you have got already a servlet with name `remote` you have to change this name in your newly added two elements `servlet` and `servlet-mapping`.

Declarations:

- The element `load-on-startup` tells the webserver in which order he has to load the servlets. The servlet with the lowest number will be loaded as first and so on. In our example we have only one servlet, so it does not matter which number it has.
- The element `url-pattern` tells the webserver that every request, whose request path ends with `.remoteservice`, should be sent to the servlet with name `remote`.

5.3.1.4.3 Loading Spring configuration file(s)

Internally, the `DispatcherServlet` is looking for the xml file that is in the **WEB-INF** folder and whose name begins with the name of the servlet and ends with `-servlet.xml`. If you have not changed the name of the servlet, the `DispatcherServlet` will look for the file `remote-servlet.xml`. We create now such a file. The content could look like the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <import resource="hessian-protocol-config.xml"/>

  <bean id="calculatorExporter" class="ch.elca.el4j.services.remoting.RemotingServiceExporter">
    <property name="remoteProtocol">
      <ref bean="remoteProtocol" />
    </property>
    <property name="serviceInterface">
      <value>ch.elca.el4j.tests.remoting.service.Calculator</value>
    </property>
    <property name="service">
      <ref local="calculatorImpl" />
    </property>
  </bean>
  <bean id="calculatorImpl" class="ch.elca.el4j.tests.remoting.service.impl.CalculatorImpl" />
</beans>
```

The content of this file is exactly the same as for the `Rmi` protocol except that the `import` points to another file. This similarity is by choice, it makes it trivial to switch between different protocols.

Now we have to copy the file ***hessian-protocol-config.xml*** that is already configured by the client into the folder ***WEB-INF***. The content of file `hessian-protocol-config.xml` could look like the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="remoteProtocol" class="ch.elca.el4j.services.remoting.protocol.Hessian">
    <property name="serviceHost">
      <value>yourserver</value>
    </property>
    <property name="servicePort">
      <value>8080</value>
    </property>
    <property name="contextPath">
      <value>yourcontextpath</value>
    </property>
    <property name="implicitContextPassingRegistry">
      <ref local="implicitContextPassingRegistry" />
    </property>
  </bean>
  <bean id="implicitContextPassingRegistry" class="ch.elca.el4j.core.contextpassing.DefaultImplicitCont
</beans>
```

Declarations:

- The name of the bean that defines the remoting protocol does not have to be `remoteProtocol`. But when the name of it will be changed, all xml files that import this xml file have to be adapted.

5.3.1.4.4 Needed module classes and libraries

All needed module classes and libraries will be deployed if you execute the `deploy ant` target of the created module. If you execute this target a second time, the module will be redeployed.

5.3.1.4.5 Reloading context

Normally the reloading of the context will be automatically done, if you are executing the ant target of the module. But sometimes it can be helpful (e.g. if you want to test something) to reload the context manually. If you are using Tomcat, you can reload your context by using the **Tomcat Manager** (<http://serviceHost:servicePort/manager/html>). You have to login with your account you had created during the installation of Tomcat. By default this is `admin` for the username and `password` for the password. Now you can click on the corresponding link of your context to reload it.

5.3.1.4.6 Test your service and find logging information

Now we are ready to test the service. Open a web browser and enter the address, where the service should be.

Example:

Property	Value
serviceHost	myserver
servicePort	8080
contextPath	remotetest
serviceInterface	ch.elca.el4j.tests.remoting.service.Calculator

For the values above the address would be the following:

<http://myserver:8080/remotetest/ch.elca.el4j.tests.remoting.service.Calculator.remoteservice>

The result of this GET request should not be a `The requested resource is not available` (HTTP status 404). You should receive an `Internal error` (HTTP status 500). If you can see a stack trace, you should see that there is a message like `HessianServiceExporter only supports POST requests`. If you receive something like that, your service might be running correctly.

Whether it runs correctly or not you can have a look at the console output of your webserver. If you are using Tomcat normally you will find the `stdout.log` in folder `logs` of your Tomcat installation. The file `stdout.log` will be deleted on each restart of Tomcat.

5.3.1.5 How to use the Burlap protocol

The usage of the Burlap protocol is exactly the same as for the Hessian protocol. Just read the last subchapter and replace the word `Hessian` with `Burlap`.

5.3.1.6 How to use the Soap protocol

The Soap protocol must also be used in a webserver like the Hessian and Burlap protocols. Please read first the **How to use the Hessian protocol** before beginning with Soap.

5.3.1.6.1 JAX-RPC

JAX-RPC (API for XML-based Remote Procedure Call – <http://java.sun.com/xml/jaxrpc>) is the standard given by java to build RPC-illusion type web services. It uses the Soap protocol (<http://www.w3.org/TR/soap/>). JAX-RPC is only the definition of web services. One implementation of JAX-RPC is Axis (<http://ws.apache.org/axis/>). In **module-remoting_and_interface_enrichment** we use Axis.

5.3.1.6.2 Axis (implementation of JAX-RPC)

Axis is used to connect the XML and the java world. Therefore we need serialization and deserialization. Primitive types like `int`, `long`, `double`, `String` are no problem. They can be covered to XML schema type definitions (`xsd`). More complex types like a `java.util.HashMap` or java beans must be handled via

serializers and deserializers. In package `org.apache.axis.encoding.ser` there are several type handlers for frequently used types. Custom serializer and deserializer will be explained later.

5.3.1.6.3 Soap style and usage

The used soap style is ***wrapped*** (aka ***rpc illusion***) and the usage is ***literal***. For more information please contact [MartinZeltner](#).

5.3.1.6.4 Server side setup

5.3.1.6.4.1 web.xml

As in Hessian and Burlap protocols we have to use a `web.xml` file. This could look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">

  <!-- Used by all protocols -->
  <servlet>
    <servlet-name>remote</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>100</load-on-startup>
  </servlet>

  <!-- Used by Soap protocol -->
  <listener>
    <listener-class>org.apache.axis.transport.http.AxisHTTPSessionListener</listener-class>
  </listener>
  <servlet>
    <servlet-name>AxisServlet</servlet-name>
    <servlet-class>org.apache.axis.transport.http.AxisServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <!-- Used to see all deployed services with its methods as list -->
    <servlet-name>AxisServlet</servlet-name>
    <url-pattern>/servlet/AxisServlet</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <!-- Url to deployed services -->
    <servlet-name>AxisServlet</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>
  <session-config>
    <!-- Default to 5 minute session timeouts -->
    <session-timeout>5</session-timeout>
  </session-config>
</web-app>
```

The `DispatcherServlet` is already known. What stands out is that this servlet has the startup number 100. Servlets with numbers less than 100 will be started before. One such servlet is the `AxisServlet`. With Soap the `AxisServlet` processes each request, not the `DispatcherServlet`. All servlet mappings points to the `AxisServlet` and not to the `DispatcherServlet`. An important servlet mapping beside the `/services/*` is the `/servlet/AxisServlet`. With the help of this, it is possible to get an overview of all deployed services. The services itself will be deployed in `/services/NameOfTheDeployedService`. The listener `AxisHTTPSessionListener` and the session timeout config are used for session management.

5.3.1.6.4.2 remote-servlet.xml

As Hessian and Burlap protocols you have to have a file with the name `remote-servlet.xml` in the same directory as the `web.xml` is. The name of this file depends on the `DispatcherServlet` name in `web.xml` such as with Hessian or Burlap. Such a file could look like the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <import resource="soap-protocol-config.xml"/>
    <import resource="soap-server-config.xml"/>
</beans>
```

This file only imports two config files: One for the protocol and one for the server configuration.

5.3.1.6.4.3 soap-protocol-config.xml

Let us first have a look at the protocol configuration file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="soapProtocol" class="ch.elca.el4j.services.remoting.protocol.Soap">
        <property name="serviceHost">
            <value>yourserver</value>
        </property>
        <property name="servicePort">
            <value>8080</value>
        </property>
        <property name="contextPath">
            <value>yourcontextpath</value>
        </property>
        <property name="protocolSpecificConfiguration">
            <ref local="soapProtocolSpecificConfiguration"/>
        </property>
        <property name="exceptionTranslationEnabled">
            <value>true</value>
        </property>
        <property name="exceptionManager">
            <ref local="soapExceptionHandler"/>
        </property>
        <property name="implicitContextPassingRegistry">
            <ref local="implicitContextPassingRegistry"/>
        </property>
    </bean>

    <bean id="soapProtocolSpecificConfiguration"
        class="ch.elca.el4j.services.remoting.protocol.soap.SoapSpecificConfiguration">
        <property name="typeMappings">
            <list>
                <bean class="ch.elca.el4j.services.remoting.protocol.soap.axis.encoding.BeanTypeMapping">
                    <property name="types">
                        <list>
                            <value>my.package.MyValueObject</value>
                        </list>
                    </property>
                </bean>
            </list>
        </property>
    </bean>

    <bean id="soapExceptionHandler"
        class="ch.elca.el4j.services.remoting.protocol.soap.axis.faulthandling.SoapExceptionHandler">
        <property name="defaultHandler">
            <bean class="ch.elca.el4j.services.remoting.protocol.soap.axis.faulthandling.DefaultHandler">
```

```

        <property name="sendServerSideStackTraceActive">
            <value>true</value>
        </property>
    </bean>
</property>
<property name="soapExceptionHandlers">
    <map>
        <entry key="my.package.MyExceptionWithStringArgumentConstructor">
            <bean class="ch.elca.el4j.services.remoting.protocol.soap.axis.faulthandling.StringAr
                <property name="sendServerSideStackTraceActive">
                    <value>true</value>
                </property>
            </bean>
        </entry>
    </map>
</property>
<property name="allowedTranslations">
    <set>
        <value>ch.elca.el4j.core.exceptions.BaseException</value>
        <value>ch.elca.el4j.core.exceptions.BaseRTException</value>
    </set>
</property>
</bean>

<bean id="implicitContextPassingRegistry" class="ch.elca.el4j.core.contextpassing.DefaultImplicitCont
</beans>

```

This main bean of this file is the ***ch.elca.el4j.services.remoting.protocol.Soap***. This bean has three other important properties beside the well known properties `serviceHost`, `servicePort`, `contextPath` and `implicitContextPassingRegistry`.

Protocol-specific configuration

One of these properties is the ***protocolSpecificConfiguration*** which expects a bean of type `ch.elca.el4j.services.remoting.protocol.soap.SoapSpecificConfiguration`. This is a configuration object that contains configuration data that is only needed in special cases. It contains the following three properties:

- ***namespaceUri***

This property is used to set the namespace uri manually. By default this is the service URL.

- ***allowedMethods***

This property is used to define which methods of services should be exported. This must be a comma separated list of method names. Per default all methods will be exported (property value *).

- ***typeMappings***

This property receives a list of beans of class

`ch.elca.el4j.services.remoting.protocol.soap.axis.encoding.TypeMapping`. Type mappings are used to serialize and deserialize types that are not standard. This class has the following three properties:

- ◆ ***types***

Contains a list of `java.lang.Class` classes, which must be mapped. It is a list because the same type mapping can be used for several types.

- ◆ ***serializerFactory***

Contains a serializer factory that implements the interface

`javax.xml.rpc.encoding.SerializerFactory`. A lot of serializer factories can be found in package `org.apache.axis.encoding.ser`. For example there is the serializer factory `MapSerializerFactory` which can be used for types like `java.util.HashMap`.

- ◆ ***deserializerFactory***

This is the counterpart of the `serializerFactory`. This factory must implement the

interface `javax.xml.rpc.encoding.DeserializerFactory`. A lot of deserializer factories can also be found in package `org.apache.axis.encoding.ser`. For example there can be found the deserializer factory `MapDeserializerFactory` which must be used for types like `java.util.HashMap`.

For the most frequently used type mappings, the mapping of java beans, there is the class `ch.elca.el4j.services.remoting.protocol.soap.axis.encoding.BeanTypeMapping` which extends the class `ch.elca.el4j.services.remoting.protocol.soap.axis.encoding.TypeMapping`. The `BeanTypeMapping` differs from the `TypeMapping` in that it has preset serializer (`org.apache.axis.encoding.ser.BaseSerializerFactory`) and deserializer (`org.apache.axis.encoding.ser.BaseDeserializerFactory`) factory.

Pay attention to the fact that at the end every complex type must be matched to several primitive xsd (xml schema type definition) types such as mentioned above.

Exception translations

One thing that can not be disregarded is the exception handling. The property **`exceptionTranslationEnabled`** is set by default to `true`.

- **`exceptionTranslationEnabled = true`**
Thrown exceptions can be left as they are. If exceptions must **NOT** be instantiated with a non-argument-constructor, the next property `soapExceptionManager` must be adapted. Exception translation is only recommended if you would like to communicate between java JVMs, otherwise you should turn the exception translation off.
- **`exceptionTranslationEnabled = false`**
In this case the next explained property `exceptionManager` is not used! Thrown exceptions must respect the following rules:
 - ◆ Each exception must extend `java.rmi.RemoteException`. This is defined in jaxrpc specification (see <http://java.sun.com/xml/downloads/jaxrpc.html>).
 - ◆ Exceptions must be serializable (and deserializable). The most appropriate way is to add getter and setter methods to each exception, so they can be serialized and deserialized as beans. See the **`typeMappings`** above. Exceptions can also have getter methods and an appropriate constructor for them, but this is not the recommended way. If a constructor can be found, where the parameter types matches, the constructor will be preferred over the setter methods. So, they recommended way is to implement the default constructor and write getter **and** setter methods.
 - ◆ You need to setup an appropriate exception manager (see below).

Exception manager

The property **`exceptionManager`** must be set to an instance of class `ch.elca.el4j.services.remoting.protocol.soap.axis.faulthandling.SoapExceptionManager`. The properties of this class must be set to classes that implement the interface `ch.elca.el4j.services.remoting.protocol.soap.axis.faulthandling.SoapExceptionHandler`. `ch.elca.el4j.services.remoting.protocol.soap.axis.faulthandling.DefaultHandler` is such a class. This class can handle exceptions that **must** be instantiated via the default constructor. The handler `ch.elca.el4j.services.remoting.protocol.soap.axis.faulthandling.StringArgumentHandler` extends the `DefaultHandler` and can handle exceptions which must be instantiated by using a constructor with several `java.lang.String` attributes. But to know how to get the information out of an exception on server side and be able to instantiate it on client side is very exceptionspecific. The `StringArgumentHandler` can only handle exceptions of type `ch.elca.el4j.core.exceptions.BaseException`, because it knows it can get the information from method `getFormatParameters`, but they must be strings. Now back to the properties of

SoapExceptionHandler.

- **defaultHandler**

Needs a class which implements the `SoapExceptionHandler` interface. The default is class `ch.elca.el4j.services.remoting.protocol.soap.axis.faulthandling.DefaultHandler`. This handler will be used if no specific handler can be found.

- **soapExceptionHandlers**

Needs a map of classes that implements the `SoapExceptionHandler` interface. The key of each entry represents the class name of the exception, which must be translated with the given handler. By default the map is empty.

- **allowedTranslations**

Allowed translation contains a set of class names, which are allowed to be translated. If a thrown exception is not in this list, it will never be translated, even if there is a handler defined for it.

The `DefaultHandler` class has further the possibility to send the server side stack trace with translated exception. If the property **sendServerSideStackTraceActive** is set to true, you can get the server side stack trace on client side as a string via the method

`ch.elca.el4j.services.remoting.protocol.soap.SoapHelper.getLastServerSideStackTrace()`.

This method returns always the last server side stack trace of the current thread. By default the property **sendServerSideStackTraceActive** is set to true.

5.3.1.6.4.4 soap-server-config.xml

The server-side configuration file could look like the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <import resource="soap-protocol-config.xml"/>

    <bean id="soapCalculatorExporter" class="ch.elca.el4j.services.remoting.RemotingServiceExporter">
        <property name="remoteProtocol">
            <ref bean="soapProtocol" />
        </property>
        <property name="serviceInterface">
            <value>ch.elca.el4j.tests.remoting.service.Calculator</value>
        </property>
        <property name="service">
            <ref local="soapCalculatorImpl" />
        </property>
    </bean>

    <bean id="soapCalculatorImpl" class="ch.elca.el4j.tests.remoting.service.impl.CalculatorImpl" />
</beans>
```

As you can see, the server side configuration looks like with other remoting protocols. There is no difference except the choice of the soap protocol of course.

5.3.1.6.5 Client side setup

The client side looks like other remote protocols too. Here an example:

5.3.1.6.5.1 soap-client-config.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <import resource="soap-protocol-config.xml"/>

    <bean id="calculator" class="ch.elca.el4j.services.remoting.RemotingProxyFactoryBean">
        <property name="remoteProtocol">
```

```

        <ref bean="soapProtocol" />
    </property>
    <property name="serviceInterface">
        <value>ch.elca.el4j.tests.remoting.service.Calculator</value>
    </property>
</bean>
</beans>

```

Note: *It is very important to import the same `soap-protocol-config.xml` as used on server side.*

5.3.1.6.6 Test the Soap service

After you have deployed your Soap service, you should be able to do a call on it. As a first step you have to get the **WSDL** (Web Service Description Language) file. The url to your service is built like the following:

<http://serverhost:serverport/contextpath/services/serviceName>

- *serverhost* is the host where your web server is running.
- *serverport* is the port where your web server is running.
- *contextpath* is the name of the web context, where the Soap service is running.
- *serviceName* is the name of the service, defined in homonymous property of `RemotingServiceExporter` and `RemotingProxyFactoryBean`. If this property is not defined, the *serviceName* is the name of the service interface plus ".remoteservice" appended at the end.

For the example above the service url would be the following:

<http://yourserver:8080/yourcontextpath/services/ch.elca.el4j.tests.remoting.service.Calculator.remoteservice>

To get now the wsdl file, you just have to append **?wsdl** to your service url.

<http://yourserver:8080/yourcontextpath/services/ch.elca.el4j.tests.remoting.service.Calculator.remoteservice?wsdl>

5.3.1.7 How to use the EJB protocol

The EJB protocol support is available in the `ModuleRemotingEjb`.

5.3.1.8 Introduction to implicit context passing

The implicit context allows passing context data along with normal method calls. The term `implicit context` refers to any kind of object that should be included in a call. It is included in a service call in the calling direction, not in the response. Therefore changes made on a server do not affect the client's implicit context.

In practice, there are different ways to implement implicit context passing. The easiest way is if the used communication protocol supports it: one can simply add the implicit context to the remote invocations. However, in the Java context, many protocols do not directly support implicit context passing. Our solution is to add the implicit context in the form of a `Map` as the last argument of methods. Behind the existing interface, we add transparently a shadow interface that has the additional parameter added. Please refer to the internal design section for more details on this.

Implicit context passing is entirely optional, it can be enabled by defining a context passing registry on the level of the protocol definition.

A service that wants to have some `implicit context` passed, must implement the interface `ch.elca.el4j.remoting.contextpassing.ImplicitContextPasser`. This passer has two responsibilities: to get the data to pass along with the call on the client side, and to push the received data to the service before the real invocation on the server side. One instance of this context passer has to be registered to an `ch.elca.el4j.remoting.contextpassing.ImplicitContextPassingRegistry` on the client side and a second to the registry on the server side. Before a method call is made, the implicit

context to include in that call is assembled by the client's registry. Every registered `AbstractImplicitContextPasser` is called to deliver its data. The same thing happens on the server side when the remote call is received, every passer is called by the registry to push its data to the service. This is done completely transparent for the service and the client, if the configuration is properly set up.

On server **and** client side the configuration could look like the following (only a part from the bean configuration file):

```
<bean id="implicitContextPassingRegistry"
      class="ch.elca.el4j.core.contextpassing.DefaultImplicitContextPassingRegistry"/>

<bean id="authenticationServiceContextPasser"
      class="ch.elca.myproject.MyImplicitContextPasserOne">
  <property name="implicitContextPassingRegistry">
    <ref local="implicitContextPassingRegistry"/>
  </property>
</bean>

<bean id="authenticationServiceContextPasser"
      class="ch.elca.myproject.MyImplicitContextPasserTwo">
  <property name="implicitContextPassingRegistry">
    <ref local="implicitContextPassingRegistry"/>
  </property>
</bean>
```

In this example we have two classes that extend the class `AbstractImplicitContextPasser`. On the client side, the `DefaultImplicitContextPassingRegistry` gets the `Serializable` object from both `AbstractImplicitContextPasser` and on server side the `DefaultImplicitContextPassingRegistry` puts the `Serializable` object to the `AbstractImplicitContextPasser` where it has been received the object.

5.3.2 Benchmark

The module ***module-remoting_and_interface_enrichment-demos*** contains a benchmark for the protocols Rmi, Hessian and Burlap. The benchmark compares each protocol with and without context passing. With context passing the `RemotingProxyFactoryBean` and the `RemotingServiceExporter` from this module will be used. Without context passing the classes from Spring will be used directly. By the way, these Spring classes are used behind the scene of this module, so the results of benchmarks without context information should be faster than benchmarks with context information.

The following is the console output of the benchmark was running on a Pentium IV with a 2.8 GHz processor and 1 GB of memory:

```
Please wait, benchmarks are running...
Benchmark 1 of 7 with name 'rmiWithoutContextCalculator' is running... done.
Benchmark 2 of 7 with name 'rmiWithContextCalculator' is running... done.
Benchmark 3 of 7 with name 'hessianWithoutContextCalculator' is running... done.
Benchmark 4 of 7 with name 'hessianWithContextCalculator' is running... done.
Benchmark 5 of 7 with name 'burlapWithoutContextCalculator' is running... done.
Benchmark 6 of 7 with name 'burlapWithContextCalculator' is running... done.
Benchmark 7 of 7 with name 'soapWithContextCalculator' is running... done.
```

Name of test	*Method 1 [ms]*	*Method 2 [ms]*	*Method 3 [ms]*
rmiWithoutContextCalculator	1.41	1.72	6.72
rmiWithContextCalculator	1.25	1.87	8.44
hessianWithoutContextCalculator	0.63	1.87	19.69
hessianWithContextCalculator	0.78	1.88	17.81

burlapWithoutContextCalculator	0.93	1.88	18.6	
burlapWithContextCalculator	0.78	2.03	18.75	
soapWithContextCalculator	9.37	15.32	32.34	

Legend: Method 1: double getArea(double a, double b)
 Method 2: void throwMeAnException() throws CalculatorException
 Method 3: int countNumberOfUppercaseLetters(String textOfSize60kB)

To **execute the benchmark on your machine** you can run the demo yourself. For more information please contact [MartinZeltner](#).

5.3.3 Remoting semantics/ Quality of service of the remoting

5.3.3.1 Cardinality between client using the remoting and servants providing implementations

This section discusses how the clients and client requests are mapped to servant objects and how servant objects need to be implemented. The *servant object* is the object that runs on the server-side of the remoting and implements the real functionality. Basically we allow either a many to 1 mapping of clients to servant objects (Singleton in table below) and a 1 to 1 mapping (Client-activated in table below). A servant object remoted as a *Singleton* can optionally be pooled on the server side (this could then be extended to something similar to the stateless session bean semantics of EJB). In order to set this up, please refer to the spring reference manual. The semantics of the EJB remoting is slightly different. It is required that you understand what you are doing when switching between EJB and other remoting protocols.

Singleton objects that are not pooled need either be reentrant or be properly synchronized (some use the term "reentrant" in a way that these 2 things are equivalent (as a properly synchronized class is naively reentrant with this signification of reentrant)).

The following table summarizes this. On the left hand side, it shows the *desired semantics* and how the servant POJOs are implemented, the right hand side indicates how this semantics is realized with each protocol:

Desired semantics /implementation			Rmi	Hessian	Burlap	Soap	EJB
Singleton	POJO is reentrant		Standard use				N/A
	POJO is not reentrant	synchronized	By using an interceptor in or synchronizing in code				
		pooled	By using spring's pooling target source (see spring doc)				Stateless
Client-activated			TODO: Is currently not implemented.				Statefull

5.3.3.2 What happens when there is a timeout or another problem during remoting

The following document describes what happens in more details. It has been contributed by [MSM](#) from the Orchestra project. To understand their context: they use this EL4J remoting to communicate between processes and other projects. They run their code within the [ModuleDaemonManager](#) (this explains some of their behavior). The exceptions shown in section 2.5 are thrown because creating 1200 tickets takes about 20 minutes (and 20 minutes is bigger than the timeout value). Thank you, Marc!

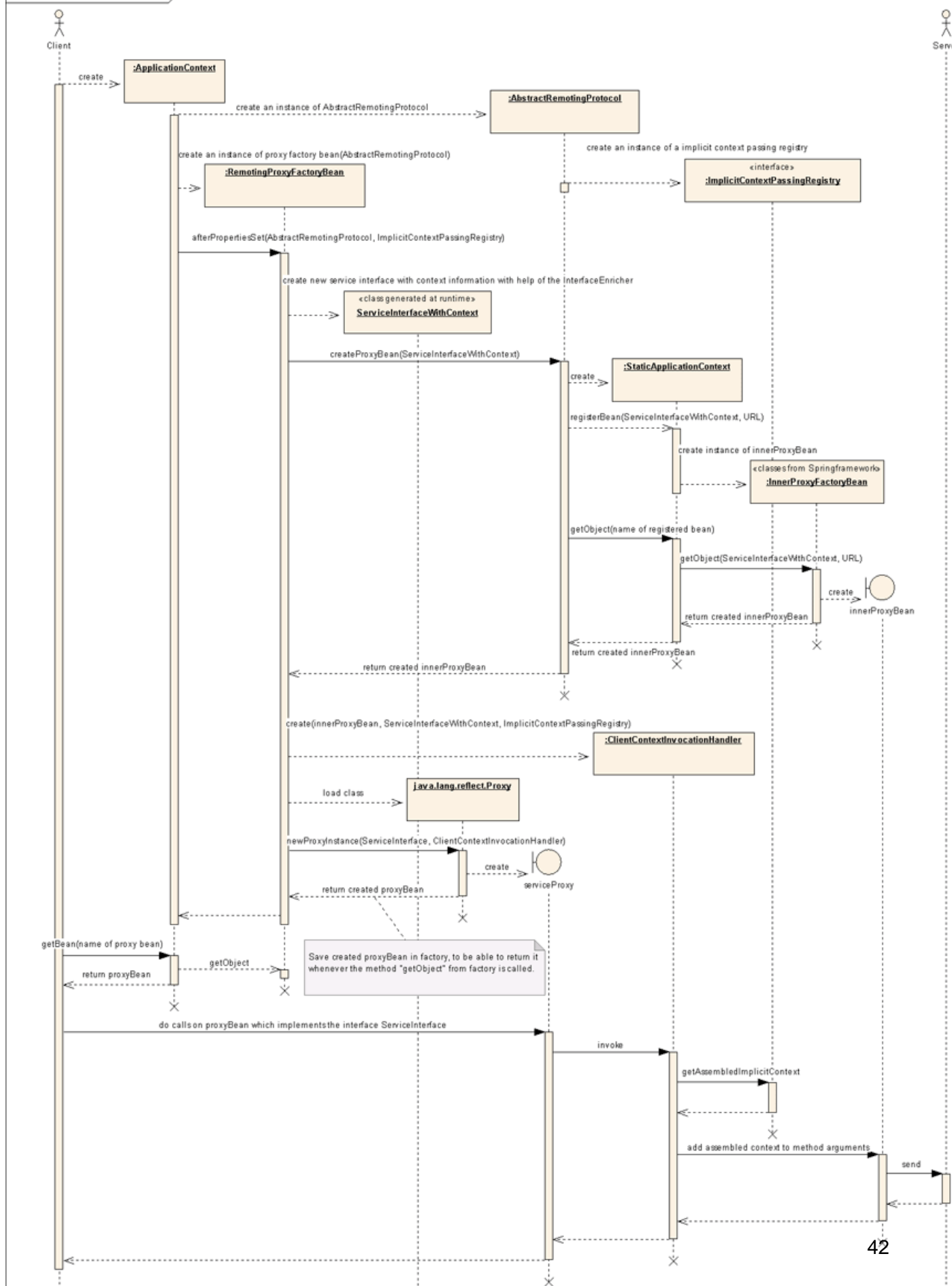
[RemoteServiceBehaviour_10.doc](#).

5.4 Internal design

5.4.1 Sequences

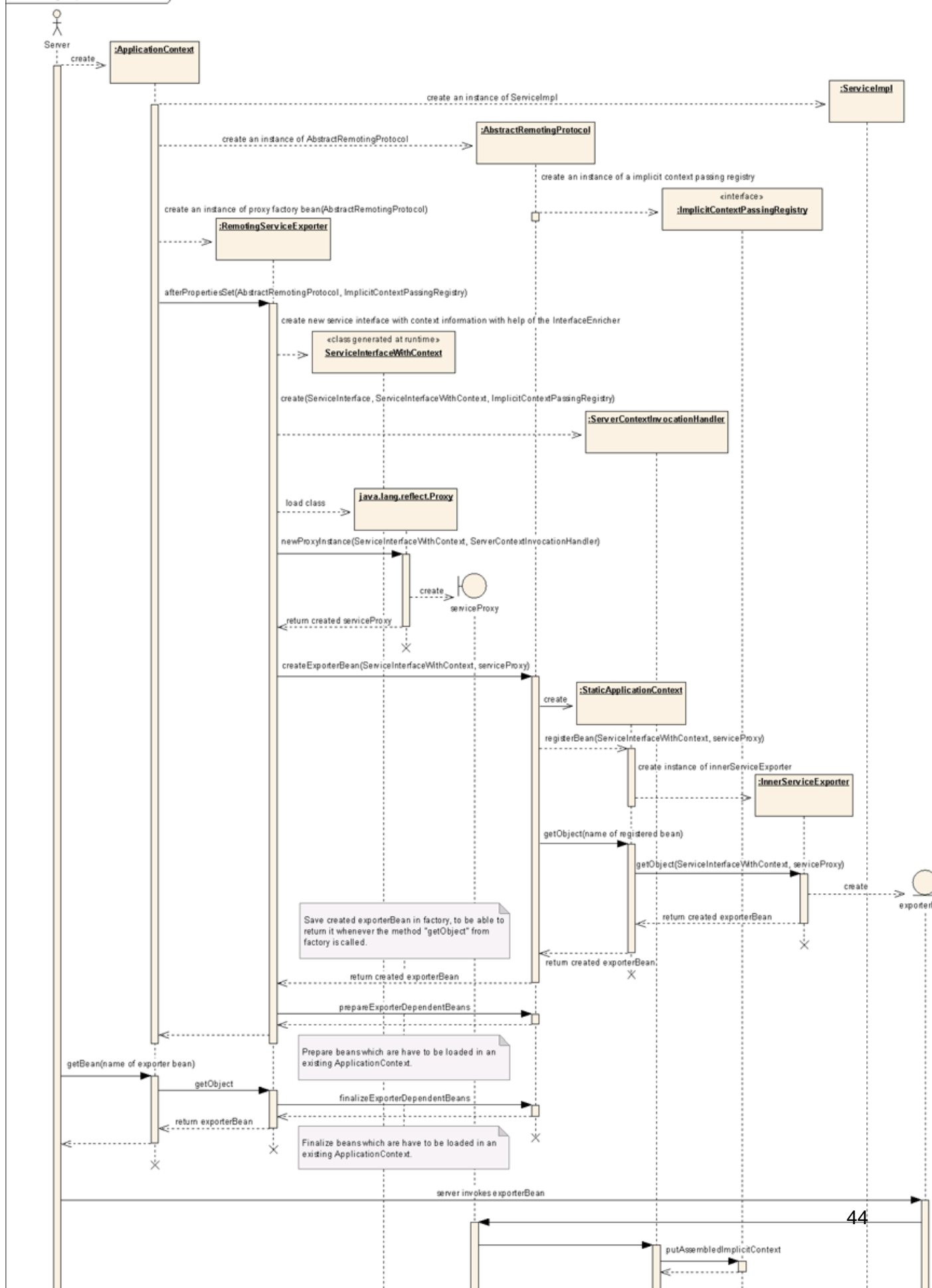
5.4.1.1 Sequence diagramm from client side

sd Remoting sequence on client side



5.4.1.2 Sequence diagramm from server side

sd Remoting sequence on server side



5.4.2 Creating a new interface during runtime

In this module, the implicit context can **optionally** be passed from client to server without changing the existing code. This is done by creating a new interface during runtime that slightly changes, **decorates** the service existing interface. But it is important that the created interface has no dependency to the service interface and vice versa. This enrichment is done with the help of the BCEL (Byte Code Engineering Library).

All classes for the interface enrichment are in package `ch.elca.el4j.util.interfaceenrichment` in the **module-core**. The class `InterfaceEnricher` offers methods to create such a new interface. One method is the `createShadowInterfaceAndLoadItDirectly` with parameters `serviceInterface`, `interfaceEnricher` and `classLoader`. The `serviceInterface` is the interface which has to be enriched, the `interfaceEnricher` is a class which implements the interface `EnrichmentDecorator` and the `classLoader` is the `ClassLoader` where the new class has to be loaded. The usage of this classes is explained in its javadoc.

By default, we do the interface enrichment during runtime, if possible. This uses the same mechanism as the CGLIB. The advantage of this is that it can be made transparent in most cases. In contexts where runtime enrichment is not applicable, the interface enrichment also supports interface enrichment during build time.

5.4.3 Internal handling of the RMI protocol (in spring and EL4J)

Perhaps you have recognized that the business interface does not extend the class `java.rmi.Remote` and the methods do not have to throw a `java.rmi.RemoteException`. This is normally mandatory to be able to use the RMI protocol. Additionally, **prior** to Java 1.5 you have to run the **RMIC (RMI-Compiler)** during build time for the service that implements the service interface.

If you have a service interface that fulfills the RMI requirements and you are using these classes in combination with the `RmiProxyFactoryBean` and `RmiServiceExporter`, the **real** RMI service will be exported. That means that everybody can access the service, whether it uses springs or EL4J's remoting facility or not.

If you have got a service interface that does not extend `java.rmi.Remote` and does not throw a `java.rmi.RemoteException` on each method, Spring will not publish the service directly via RMI. Spring uses Java's reflection to send calls through a generic invoke method. In the framework it has a RMI invoker, which tunnels every request through the method `invoke`. The RMI invoker extends `java.rmi.Remote` and the method `invoke` throws a `java.rmi.RemoteException`. The Stub and skeleton are already prebuild for this RMI invoker. (This is the default spring semantics.)

EL4J adds some more flexibility: via the interface decoration, it can wrap a non-RMI-conformant interface with a conformant interface. This support is again transparent for the user. This work similarly in the case of EJB. In the current implementation, this generates a double-indirection of interfaces. The last interface is visible to RMI, the first interface is visible to the user.

Business Interface --> Shadow Interface 1 (RMI-conformant) --> Shadow interface 2 (RMI-conformant and with implicit context passing)

5.4.4 To be done

The module should be able to export a rmi service with its service interface, but the service interface should not have any dependencies to RMI. To solve this problem we could create an ant task to call the interface enricher and let him generate and save the generated interface to disk. The interface enricher can already do that. So we could be able to wrap the service implementation with a class, which implements the generated service interface and could redirect method invocations. At the end we could also use the `rmic` to create stub and skeleton for Java 1.4 and below.

5.5 Related frameworks

5.5.1 extrmi

A further framework which also pass the context transparently is the `extrmi`. It can be found at sourceforge <http://sourceforge.net/projects/wenbozhu/>

- In this solution the implicit context is passed by using `java.lang.reflect`. So this is the same way like this module does it in the worst case. Why worst case? If the remoting is done by reflection the server side published interface is always the same. In a first way this sounds very good, but if you would like to access the server without using the given client stub you have not got any chance.
- Another negative point is that this framework does not help you to simplify switching between remoting protocols. You always have to adapt the business classes to the needs of the used remoting protocol.

Article reference: http://www.javaworld.com/javaworld/jw-04-2005/jw-0404-rmi_p.html

5.5.2 Javaworld 2005 idea

http://www.javaworld.com/javaworld/jw-03-2005/jw-0314-usersession_p.html

6 Documentation for module EJB remoting

6.1 Purpose

Convenience module to expose spring beans as **EJB session beans**. This module extends the `ModuleRemotingAndInterfaceEnrichment`, i.e. supports the same remoting features and allows switching from one of the other protocols to EJB and vice versa.

6.2 Important concepts

EL4J provides its own remoting services infrastructure (`ModuleRemotingAndInterfaceEnrichment`) that simplifies the use of remoting protocols supported by Spring. In addition, it adds support to pass an implicit context between the remote parties.

This module allows deploying Spring beans in EJB compliant containers, wrapping them transparently into session beans. Since most application containers do not allow creating enterprise beans at runtime, they have to be generated at deploy time and packed into an EAR. This is done transparently by the build system, if the needed plugin is activated.

6.3 How to use

6.3.1 Configuration

The protocol configuration is analogous to the [description in the easy remoting module](#).

6.3.1.1 How to use the EJB protocol

Different from the other remoting protocols, where all remoting specific objects are created dynamically at runtime, most EJB containers demand beans to be available at deploy-time. This requires to generate session bean wrappers during the build process. `EL4Ant` supplies a module using XDoclet which automates this step. Apart from this speciality one can do as much as with the other protocols. However there are some constraints given by the EJB world. Currently, JBoss, WebLogic and WebSphere (not much tested yet) are supported.

6.3.1.1.1 Protocol definition

Additionally to the protocol-independent properties, EJB protocol definitions have another one specifying the JNDI environment. This is a Properties object defining the initial JNDI context. There are already such environment definitions for the three supported EJB containers, stored in the plugin's scenarios.

The EJB protocol uses a configuration object in order to specify service-related configurations. It contains a number of properties describing the EJB session bean's lifecycle, a mapping from EJB to service bean methods and a map to supply additional XDoclet tags.

6.3.1.1.2 Constraints

- Service beans wrapped in stateful session beans must live a **prototype** lifecycle. Beans wrapped in stateless session beans can be either declared as singletons or prototypes (be aware of what this means, you can create contention points!)
- The service bean must implement `java.lang.Serializable` and all members it references (directly or indirectly) too. Note: using `writeObject` and `readObject` may help dealing with complicated situations.
- Exceptions thrown by service beans must be subclasses of `java.lang.Exception` (JBoss allows using `java.lang.Exception`, but WebLogic doesn't. see EJB 2.1 spec 18.1.1).

6.3.1.1.3 Method Mappings

EJB method name	property name	additional info / constraints
ejbActivate()	activate	A method with <code>void</code> as return type and an empty argument list. All checked exceptions are wrapped into an unchecked one that aren't propagated to the client.
ejbPassivate()	passivate	
ejbRemove()	remove	
setSessionContext(SessionContext ctx)	sessionContext	A method that takes a <code>javax.ejb.SessionContext</code> as parameter
create(Object[] args)	create	There's only one custom create method available that takes an object array as parameter. Parameters are supplied through the <code>createArgument</code> property, a list. Any checked exceptions are wrapped in a <code>javax.ejb.CreateException</code>
afterBegin()	afterBegin	A method with <code>void</code> as return type and an empty argument list. All checked exceptions are wrapped into an unchecked to keep them on server side.
beforeCompletion()	beforeCompletion	
afterCompletion(boolean commit)	afterCompletion	A method that takes a <code>boolean</code> argument. All Exceptions are kept on server side.

6.3.1.1.4 Exceptions

All exceptions defined on the service interface are sent to the client. Additionally, all runtime exceptions thrown by the service bean are forwarded to the client too (wrapping and unwrapping is done transparently). We chose to do this for developer convenience. If the exception class does not exist on the client side, the string of the exception message is displayed. All other exceptions stay on the server side. Especially exceptions thrown during activation and passivation are wrapped into runtime exceptions and stay on server-side. The tests (`module-remoting_ejb-tets`) provide some examples.

6.3.1.1.5 Adding additional XDoclet tags

The EJB remoting plugin allows adding additional XDoclet tags or to override existing ones. They are specified through the configuration object that has to be specified on the `RemotingServiceExporter` as well as on the `RemotingBeanFactory`. Additional tags are provided by the `docletTags` property, which is a map and conform the following naming scheme:

- **class** add XDoclet tags on class level
- **null** add XDoclet tags to all methods
- **<methodName>** add XDoclet tags to all methods with the given name
- **<methodName(java.lang.String, int)>** add XDoclet tags to the method with the given signature (Note: consists of the method name and the parameters' fully qualified types only — no return type or variable names)

6.3.1.1.5.1 Example

```
<property name="docletTags">
  <map>
    <entry key="class">
      <value>@ejb.util generate="logical"</value>
    </entry>
    <entry>
      <key><null/></key>
      <value>@ejb.do-whatever foo="bar"</value>
    </entry>
    <entry key="foo">
      <value>@ejb.dao call="helloWorld"</value>
    </entry>
    <entry key="bar(java.lang.String, org.foo.bar.Foobar, int)">
```

```

        <value>@jboss.persistence datasource="foo" read-only="false"</value>
    </entry>
    <entry key="passivate">
        <list>
            <value>@test arg="doit"</value>
            <value>@ejb.interface-method view-type="both"</value>
        </list>
    </entry>
</map>
</property>

```

6.3.1.2 How to use the build system plugin

The EJB build system plugin adds two hooks to the project that generate the needed session beans transparently. In order to get them activated, one has to add the following attributes:

```

<attribute name="runtime.runnable" value="true"/>
<attribute name="j2ee.ear.application"/>
<attribute name="remoting.ejb" value="true"/>
<attribute name="remoting.ejb.inclusiveLocations" value="classpath*:mandatory/env.xml,classpath*/gui/ser
<attribute name="remoting.ejb.exclusiveLocations" value="classpath*:gui/client-config.xml"/> <!-- configu

```

For using the plugin you have to copy the `remoting_ejb.jar` into your build system's lib directory. And it has to be added to the project in order to use it. You can add it to the `plugins.xml` (there are no attributes to set):

```

<plugin name="ejb" file="buildsystem/remoting_ejb/remoting_ejb.xml"/>

```

6.3.1.2.1 Known issues

- **Weblogic 8.1** Weblogic is not able to resolve Spring wildcard-locations, where the asterisks is in the second part. You have to enumerate the configuration locations explicitly or use the module application context.
 - ◆ valid use of wildcards in Weblogic:
 - ◊ classpath:foo/bar.xml
 - ◊ classpath*:foo/bar.xml
 - ◆ invalid use of wildcards in Weblogic:
 - ◊ classpath:foo/*.xml
 - ◊ classpath*:foo/*.xml

6.3.1.3 How to use the EJB remoting module without the EL4Ant build system

There's no need to use the EL4Ant build system in order to use the EJB remoting module. You need to perform the following steps:

1. compile the source code
2. run the EJB wrapper generator which creates an annotated Java source file.
3. run The XDoclet task for your application server that will use the annotated Java file created before to generate all the needed EJB specific files.
4. create an appropriate application.xml
5. pack the classes and all required libraries into an EAR file

In principle there's no binding to a specific build system at all. But using Ant simplifies the whole process (e.g. XDoclet Ant tasks).

6.3.1.3.1 Todo

The EJB wrapper generator's core is contained in the EJB remoting module, whereas the binding to Velocity, which is used to generate the wrapper, is located in the build system plugin (see internal design for details).

Providing a separate jar file that contains all the needed libraries (Spring, Velocity, commons-logging, ...) and that supports direct usage through the command line would simplify the above process.

6.4 References

- `ModuleRemotingAndInterfaceEnrichment`
 - EJB 2.1 specification <http://java.sun.com/j2ee> <http://jcp.org/en/jsr/detail?id=153>
 - XDoclet <http://xdoclet.sourceforge.net/>
 - Velocity <http://jakarta.apache.org/velocity/>
-

6.5 Internal design

6.5.1 EJB generation

The session beans are generated using a hook of the build system. This hook uses Velocity to create session beans that delegate the invocations to the POJOs that implement the services. The generated Java files contain javadoc comments that are read by XDoclet to create the real EJB-aware classes. Next, the hook compiles the generated classes and puts them in the module's class path. Finally, it generates the deployment descriptor (`application.xml`).

The buildsystem requires to start the complete application to be able to create the templates. Since there's no easy mechanism to filter the configuration files that are needed to start the application in a Spring container, the user has to provide them with the `remoting.ejb.inclusiveLocations` and `remoting.ejb.exclusiveLocations` attributes.

Each application container has it's own deployment descriptor. The remoting EJB plugin gets the container that is currently used from the actual J2EE-EJB plugin. This plugin also provides all the tasks to manage the server (e.g. starting, stopping, deploying) and to create EARs. Each remoting EJB application creates its EAR file. Of course, this file can contain more than one EJB application using build system dependencies.

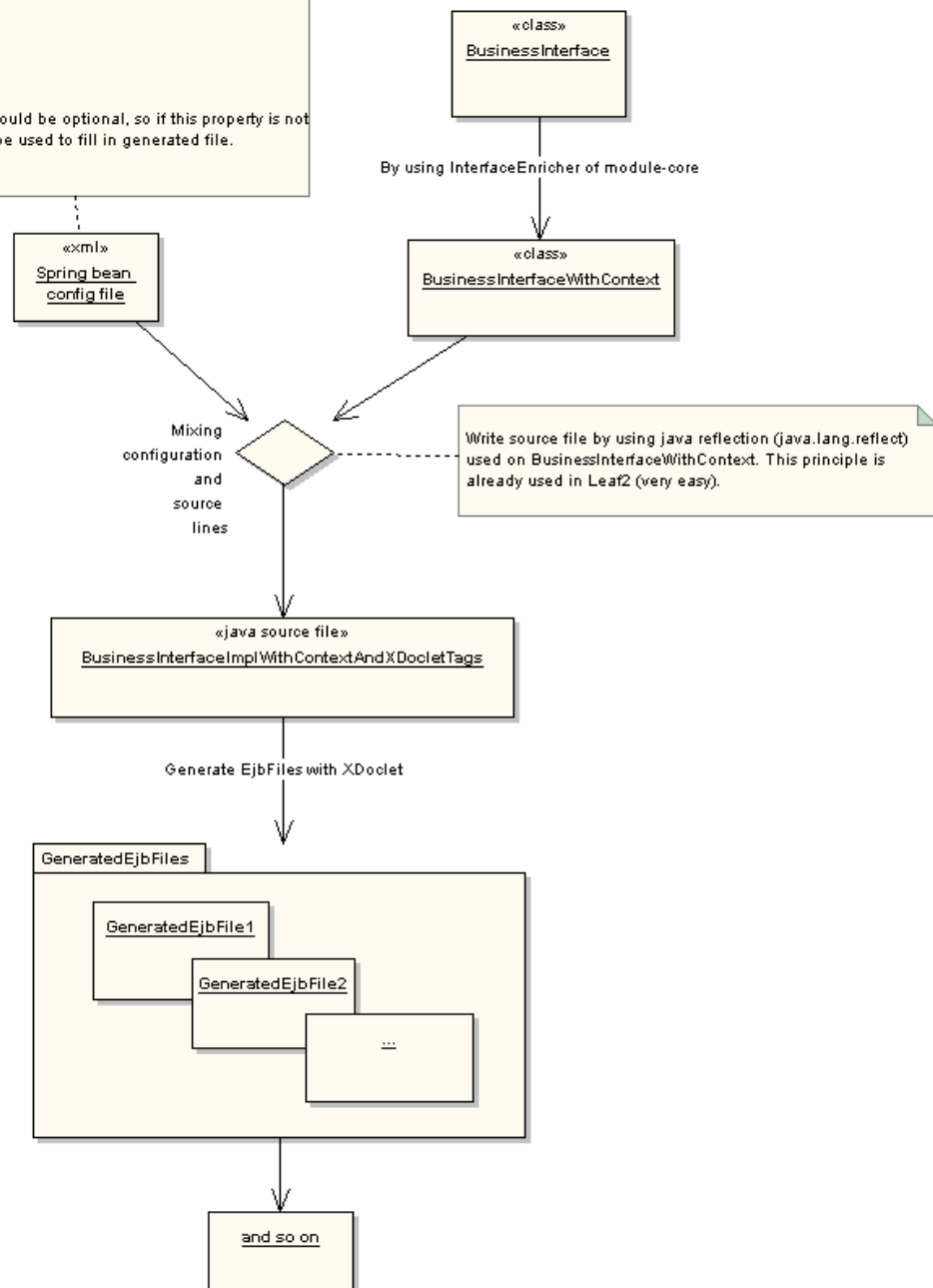
Important: To avoid dependencies, we use reflection to get access to classes residing in the framework module. For simplicity, there's a facade to generate the beans that is in the EJB remoting module. Its interfaces are copied to the build system plugin where we just need to get the facade's implementation by reflection. All other operations are performed on the interface.

ud Ejb remoting and interface enrichment

This configuration file contains xdoclet config in a way like:

```
...
<bean id="ejbProtocol"
class="ch.elca.leaf3.services.remoting.protocol.Ejb">
  <property name="serviceHost">
    <value>localhost</value>
  </property>
  <property name="servicePort">
    <value>1234</value>
  </property>
  <property name="implicitContextPassingRegistry">
    <ref local="implicitContextPassingRegistry" />
  </property>
  <property name="docletTags">
    <map>
      <entry key="set">@jboss...</key>
      ...
    </map>
  </property>
</bean>
...
```

The property "docletTags" should be optional, so if this property is not set, default xdoclet tags will be used to fill in generated file.



6.5.2 Adding support for another container

Adding support for a different container is straightforward. There are two things to do:

1. add a new ant file for XDocleting the generated files. You can just copy an existing one, do a search and replace on the container's name and adapt the container specific XDoclet target.
2. add the plugin dependency to the `ejb/ear.xml` file.

7 Documentation for module security

7.1 Purpose

The security module provides authentication and authorization for EL4J applications. The core part of this module is the [Acegi Security System for Spring](#). Attribute-enabled interceptors are used to perform access controls.

7.2 Features

Besides the [Acegi Security System](#) library, this module contains an `AuthenticationServiceContextPasser` and an `AuthenticationService` which allows the user to transparently log in to a server and transparently invoke the server's methods. For a demonstration of this feature, please consult the `module-security-tests`.

7.3 How to use

Please refer to the demo application for now.

8 Documentation for module exception handling

8.1 Purpose

This module provides configurable exception handlers that allow separating the exception handling from the actual business logic. There are two exception handlers: a safety facade that handles technical exceptions for collections of POJOs and a context exception handler that allows handling exceptions in function of what context is active. These exceptions handlers complement the EL4J exception handling guidelines.

8.2 Important concepts

EL4J supports two frameworks to handle exceptions, the **Safety Facade** and the **Context Exception Handler**. Both handle exceptions of several beans and both use exactly the same core framework. The former is intended to be used nearby a service to handle implementation-specific and technical exceptions. Instead of handling such *abnormal* exceptions in the business code, the handling of abnormal exceptions is delegated to the safety facade. This simplifies the use of the wrapped service, as one can concentrate on its core functionality. In addition, it separates the concerns of its core business functionality and abnormal cases. The latter context exception handler is used to handle exceptions in different ways, depending on the current context (e.g. show errors in message boxes if in a gui context or log them into a file if in server context). Both exception handler frameworks can be used to build complex exception handler hierarchies consisting of different risk communities, as described in the [ExceptionHandlingGuidelines](#).

Both exception handling frameworks are added to a project whenever they are needed. The handlers are configured in Spring configuration files, where you just change the names of the original beans and where you add new proxied versions of them. This still allows accessing the bare beans, without going through an exception handling facade, which is needed to build risk communities.

As already mentioned, the context exception handler handles exceptions according to the current context. It is set through a static method and is valid for the thread's whole life or until it's set to another value. It's considered a mistake if the context has not been set before the context exception handler treats an exception, hence a `MissingContextException` (unchecked) is thrown.

8.3 How to use

8.3.1 Configuration

Both the safety facade and the context exception handler are configured with a list of exception configurations. Each exception configuration associates a set of exceptions with its exception handler (see below for more details on exception handlers). The `ExceptionConfiguration` interface contains two methods, one for checking whether the configuration is able to handle the given situation, the other returns the configuration's exception handler. There are two default `ExceptionConfiguration` implementations:

- **`ClassExceptionConfiguration`** just checks the caught exception's type.
- **`MethodNameExceptionConfiguration`** checks the caught exception's type as well as the name of the method that threw it.

To configure a safety facade, one configures a list of exception configurations (please refer to the example below).

A context exception handler is configured with a map: The key of the map represents the context, the value the context's list of exception configurations (the format of the list of exception configurations (for each context) is the same as above).

8.3.1.1 Exception handlers

There are a number of exception handlers covering the most common cases:

- **RethrowExceptionHandler** forwards the exception to the caller.
- **SimpleLogExceptionHandler** logs the exception and the invocation description that raised it on trace level.
- **SimpleExceptionTransformerExceptionHandler** transforms the caught exception into the one specified by an exception class. The handler tries to fill it with the original exception's message, cause and stack trace.
- **SequenceExceptionHandler** allows declaring a list of exception handlers which are invoked one after another until one succeeds (=does not throw another exception). If all fail it returns the last caught exception.
- **RetryExceptionHandler** retries the very same invocation several times. The last caught exception is rethrown if all retries didn't succeed.
- **RoundRobinSwappableTargetExceptionHandler** retries the invocation on different targets, that is exchanged each time the current one fails. The handler modifies the proxy too, let it use the exchanged target for new invocations. This allows automatically reconfiguring the system at runtime (e.g. change the data source if the current one isn't reachable anymore).

All of them extend the `AbstractExceptionHandler` that provides a logging abstraction which allows users to set whether proxy generated log messages are registered as if they're coming from the exception handler (default) or whether they are reported as if they're coming from the proxied class.

Additionally, there are a number of abstract handlers making it easier to build custom strategies. The **AbstractReconfigureExceptionHandler** for example helps to reconfigure a bean (e.g. making it use another collaborator).

8.3.1.2 Example 1: Safety Facade for one Bean

This is the simplest form of a safety facade: The actual bean is renamed, an exception configuration is provided and the safety facade is created. There are no changes in the Java code, as long as no unsafe bean access is needed.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean id="unsafeA" class="ch.elca.el4j.tests.services.exceptionhandler.A"/>

  <bean id="A" class="ch.elca.el4j.services.exceptionhandler.SafetyFacadeFactoryBean">
    <property name="target"><ref local="unsafeA"/></property>
    <property name="exceptionConfigurations">
      <list>
        <bean class="ch.elca.el4j.services.exceptionhandler.ClassExceptionConfiguration">
          <property name="exceptionTypes">
            <list>
              <value>java.lang.ArithmeticException</value>
            </list>
          </property>
          <property name="exceptionHandler">
            <bean class="ch.elca.el4j.services.exceptionhandler.handler.SimpleLogExceptionHandler">
              <property name="useDynamicLogger"><value>true</value></property>
            </bean>
          </property>
        </bean>
      </list>
    </property>
  </bean>
</beans>
```

8.3.1.3 Example 2: Context Exception Handler

The context exception handler is initialized the same way as the safety facade. However, there is an additional indirection (the map) to setup different policies for each context.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean id="unsafeA" class="ch.elca.el4j.tests.services.exceptionhandler.AImpl"/>

  <bean id="A" class="ch.elca.el4j.services.exceptionhandler.ContextExceptionHandlerFactoryBean">
    <property name="target"><ref local="unsafeA"/></property>
    <property name="policies">
      <map>
        <entry key="gui">
          <list>
            <bean class="ch.elca.el4j.services.exceptionhandler.ClassExceptionConfiguration">
              <property name="exceptionTypes">
                <list>
                  <value>java.lang.ArithmeticException</value>
                </list>
              </property>
              <property name="exceptionHandler">
                <bean class="ch.elca.el4j.tests.services.exceptionhandler.MessageBoxExcep
              </property>
            </bean>
          </list>
        </entry>

        <entry key="batch">
          <list>
            <bean class="ch.elca.el4j.services.exceptionhandler.ClassExceptionConfiguration">
              <property name="exceptionTypes">
                <list>
                  <value>java.lang.ArithmeticException</value>
                </list>
              </property>
              <property name="exceptionHandler">
                <bean class="ch.elca.el4j.tests.services.exceptionhandler.LogExceptonHand
                  <property name="useDynamicLogger"><value>true</value></property>
                </bean>
              </property>
            </bean>
          </list>
        </entry>
      </map>
    </property>
  </bean>
</beans>
```

Corresponding Java snippet (**Note:** set the context to a valid value. Otherwise, a `MissingContextException` (unchecked) is thrown.):

```
A m_a = getA();
ContextExceptionHandlerInterceptor.setContext("gui"); // set the current thread's context
m_a.div(1, 0); // handles any exceptions using the gui policy
ContextExceptionHandlerInterceptor.setContext("batch"); // set the current thread's context
m_a.div(1, 0); // handles any exceptions using the batch policy
m_a.div(1, 0); // handles any exceptions using the batch policy
```

8.3.1.4 Example 3: RoundRobinSwappableTargetExceptionHandler

This example shows the round robin swappable target exception handler. **Note** the handler requires a `HotSwappableTargetSource` in order to reconfigure the proxy.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <bean id="unsafeA" class="ch.elca.el4j.tests.services.exceptionhandler.A"/>
    <bean id="B" class="ch.elca.el4j.tests.services.exceptionhandler.B"/>

    <bean id="swapper" class="org.springframework.aop.target.HotSwappableTargetSource">
        <constructor-arg><ref local="unsafeA"/></constructor-arg>
    </bean>

    <bean id="A" class="ch.elca.el4j.services.exceptionhandler.SafetyFacadeFactoryBean">
        <property name="target"><ref local="swapper"/></property>
        <property name="exceptionConfigurations">
            <list>
                <bean class="ch.elca.el4j.services.exceptionhandler.MethodNameExceptionConfiguration">
                    <property name="methodNames">
                        <list>
                            <value>concat</value>
                        </list>
                    </property>
                    <property name="exceptionTypes">
                        <list>
                            <value>java.lang.UnsupportedOperationException</value>
                        </list>
                    </property>
                    <property name="exceptionHandler">
                        <bean class="ch.elca.el4j.services.exceptionhandler.handler.RoundRobinSwappableTargetExceptionHandler">
                            <property name="swapper"><ref local="swapper"/></property>
                            <property name="targets">
                                <list>
                                    <ref local="unsafeA"/>
                                    <ref local="B"/>
                                </list>
                            </property>
                        </bean>
                    </property>
                </bean>
            </list>
        </property>
    </bean>
</beans>
```

Using a `ProxyFactoryBean` and an explicit interceptor to do the same work as the `SafetyFacadeFactoryBean` above would look like this

```
<bean name="safetyFacade" class="ch.elca.el4j.services.exceptionhandler.SafetyFacadeInterceptor">
    <property name="exceptionConfigurations">
        <list>
            <ref local="roundRobinSwappableTargetExceptionConfiguration"/>
        </list>
    </property>
</bean>

<bean id="A" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="targetSource"><ref local="swapper"/></property>
    <property name="interceptorNames">
        <list>
            <idref bean="safetyFacade"/>
        </list>
    </property>
</bean>
```

```

    </property>
</bean>

```

8.3.1.5 Example 4: Using several exception handlers, each configured by a separate exception configuration

Several exception handlers are configured by multiple exception configurations.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <bean id="unsafeA" class="ch.elca.el4j.tests.services.exceptionhandler.AImpl"/>

    <bean id="A" class="ch.elca.el4j.services.exceptionhandler.SafetyFacadeFactoryBean">
        <property name="target"><ref local="unsafeA"/></property>
        <property name="exceptionConfigurations">
            <list>
                <bean class="ch.elca.el4j.services.exceptionhandler.ClassExceptionConfiguration">
                    <property name="exceptionTypes">
                        <list>
                            <value>java.lang.ArithmeticException</value>
                        </list>
                    </property>
                    <property name="exceptionHandler">
                        <bean class="ch.elca.el4j.services.exceptionhandler.handler.SequenceExceptionHandler">
                            <property name="exceptionHandlers">
                                <list>
                                    <bean class="ch.elca.el4j.services.exceptionhandler.handler.SimpleLogger">
                                        <property name="useDynamicLogger"><value>true</value></property>
                                    </bean>

                                    <bean class="ch.elca.el4j.services.exceptionhandler.handler.RetryExceptionHandler">
                                        <property name="retries"><value>5</value></property>
                                        <property name="sleepMillis"><value>0</value></property>
                                        <property name="useDynamicLogger"><value>true</value></property>
                                    </bean>

                                    <bean class="ch.elca.el4j.services.exceptionhandler.handler.SimpleExceptionHandler">
                                        <property name="transformedExceptionClass">
                                            <value>java.lang.RuntimeException</value>
                                        </property>
                                    </bean>
                                </list>
                            </property>
                        </bean>
                    </property>
                </bean>
            </list>
        </property>
    </bean>

    <bean class="ch.elca.el4j.services.exceptionhandler.ClassExceptionConfiguration">
        <property name="exceptionTypes">
            <list>
                <value>java.lang.UnsupportedOperationException</value>
            </list>
        </property>
        <property name="exceptionHandler">
            <bean class="ch.elca.el4j.tests.services.exceptionhandler.ReconfigureExceptionHandler">
                <property name="c"><ref local="C"/></property>
            </bean>
        </property>
    </bean>
</list>
</property>
</bean>
</beans>

```

8.4 References

1. Moderne Softwarearchitektur — Umsichtig planen, robust bauen mit Quasar, Johannes Siedersleben, dpunkt.verlag, 2004, ISBN 3-89864-292-5

8.5 Internal design

Each secured bean is hidden behind a proxy that wraps each invocation into a try-catch block. If the invocation that is delegated to the bare bean throws an exception, the facade looks up an appropriate exception handler and delegates the handling to it. The interceptors are instantiated with one of the two convenience factories, the

`ch.elca.el4j.services.exceptionhandler.SafetyFacadeFactoryBean` and the `ch.elca.el4j.services.exceptionhandler.ContextExceptionHandlerFactoryBean`. These two factories create the interceptor transparently. They extend Spring's `AdvisedSupport` and simply add the exception handling interceptor. Using the `ProxyFactoryBean` provides access to the interceptor and allows adding additional interceptors (however this is not recommended since the exception handler interceptor should wrap the complete unsafe bean).

Important Although it's possible to use Spring's auto proxy features, it's not recommended because it hides the unsafe bean, making it impossible to build risk communities.

There are three common properties, independent of whether you use a safety facade or a context exception handler and independent from the way you create the proxies (convenience factory or `ProxyFactoryBean`):

Property	Description	Default value	
		Safety Facade	Context Exception Handler
<code>defaultBehaviourConsume</code>	<code>true</code> consumes any exceptions that are not handled by an exception handler. <code>false</code> rethrows unhandled exceptions to the caller.	true	true
<code>forwardSignatureExceptions</code>	<code>true</code> forwards any exceptions which are defined in the invoked method's signature. <code>false</code> forces to handle these exceptions by the handlers too.	true	true
<code>handleRTSignatureExceptions</code>	Declares whether unchecked exceptions that are listed in a method's signature should go through an exception handler or whether they are forwarded to the caller. <code>true</code> for handle, <code>false</code> for forward.	true	true

8.5.1 Context Exception Handler

The `ContextExceptionHandlerInterceptor` uses a `ThreadLocal` to store the current context. There's no mechanism that resets the context transparently, preventing pooled threads to use a wrong context.

Setting the appropriate context is the programmer's responsibility.

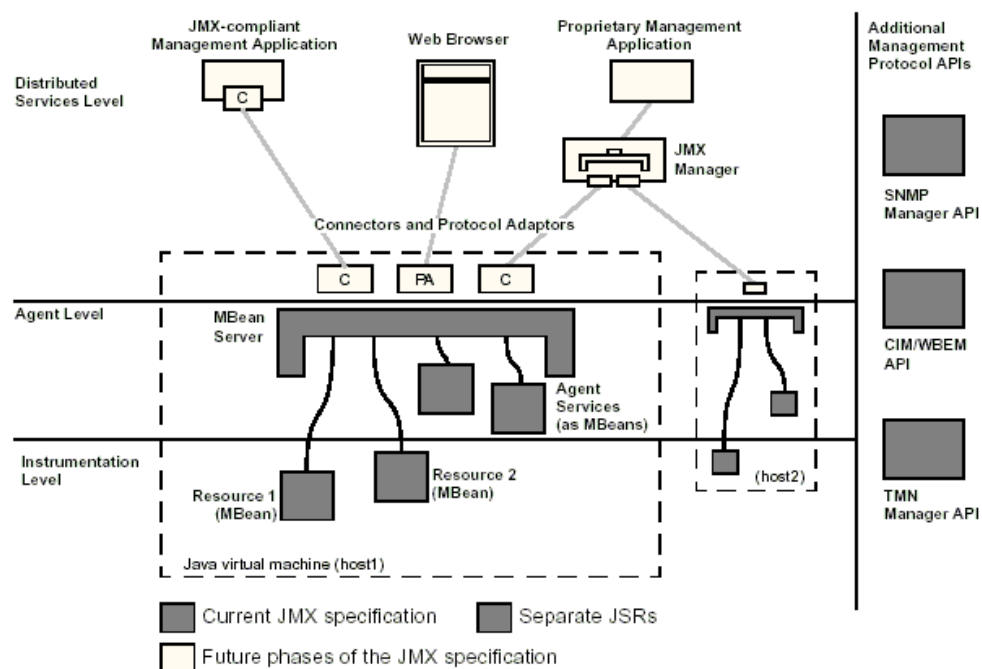
9 Documentation for module JMX

9.1 Purpose

The module *jmx* supports developers in understanding spring applications by providing an automatic (implicit) view of the currently loaded spring beans and their configurations. This becomes even more interesting as Spring (and EL4J) allow splitting configuration information in many files, making it sometimes hard to figure out what config applies.

9.2 Introduction to Java management eXtensions (JMX)

The following picture shows the components of JMX:



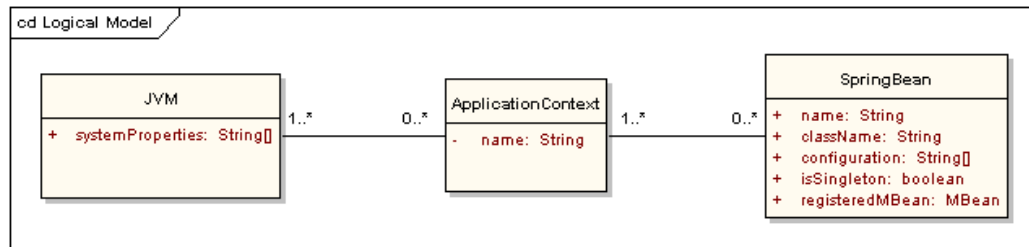
The central part of JMX is the *MBean Server*. *Managed Beans*, special Java objects that a developer wants to have controlled during runtime, are registered at the MBean Server. These Managed Beans or Mbeans are typically proxies for other components in the JVM one wants to monitor. These MBeans can be manipulated via JMX at runtime, i.e. their attributes can be read and edited and their methods can be invoked. Finally there are connectors that allows to access MBeans from remote.

9.3 Feature overview

We provide 2 ways to publish Spring Mbeans to JMX:

- Implicit publishing: publish all spring beans and their config automatically (we use the `ModuleApplicationContext` for this)
- Explicit publishing (as Spring provides it normally)

The following class diagram illustrates the mbeans we publish implicitly:



Besides all the spring beans, the *jmx* package also creates a JVM proxy in order to display the system properties etc. Furthermore, each ApplicationContext will be mirrored by a proxy that also provides links to all the loaded beans in it.

9.4 Usage

9.4.1 Spring/JDK versioning issue

The usage of the module depends on the used Spring and JDK versions. By default the module works with Spring 1.2 and JDK 1.4.2.

9.4.1.1 Spring versions 1.1 <--> 1.2

Spring supports JMX since version 1.2. If you are using Spring 1.1, you have to include a library with the missing files. This can be done by adding the following dependency in the `module.xml` file of the JMX module:

```
<dependency jar="spring-jmx-1.1.4.jar"/>
```

Difference in module-jmx:

Refactoring of `org.springframework.jmx.JmxMBeanAdapter` (Spring 1.1 extension) into `org.springframework.jmx.export.MBeanExporter` (Spring 1.2). Therefore you have to adapt the `beans.xml` as is described at [Editing an MBean](#) by replacing the corresponding class name. Everything else remains unchanged.

9.4.1.2 JDK versions 1.4.2 <--> 1.5

Since JDK 1.5, JMX is supported. If you are using JDK 1.5, you have to exclude the following 4 libraries in the `module.xml`, i.e. deleting these lines.

```
<dependency jar="jmxremote-1.4.2.jar"/>
<dependency jar="jmxremote_optional-1.4.2.jar"/>
<dependency jar="jmxri-1.4.2.jar"/>
<dependency jar="jmxtools-1.4.2.jar"/>
```

There is no difference in using the JMX module.

9.4.2 Basic Configuration (implicit publication)

The **JMX** package has to be included in the build path of your project which can be achieved by setting a dependency in your project to the module-jmx. First of all you need the `jmx.xml` which you can find at `mandatory/jmx.xml`. If you load the Application Context with one of your config locations equal to `classpath*:mandatory/*.xml`, then `jmx.xml` is loaded.

Here is a possible configuration file `jmx.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean id="mBeanServer" class="ch.elca.el4j.services.monitoring.jmx.MBeanServerFactoryBean">
    <property name="defaultDomain">
      <value>foobar1</value>
    </property>
  </bean>
  <bean id="jmxLoader" class="ch.elca.el4j.services.monitoring.jmx.Loader">
    <property name="server">
      <ref bean="mBeanServer"/>
    </property>
  </bean>
</beans>
```

- The bean `mBeanServer` creates a `MBeanServer` on the defined `defaultDomain`. Since the `MBeanServerFactoryBean` ensures that there is only one `MBeanServer` per domain, you can register as many `ApplicationContexts` at the same `MBeanServer` as you want, or easily assign each `ApplicationContext` a different `MBeanServer` by defining an unique domain for each `MBeanServer`. If you do not define this property, the `MBeanServer` on the domain "defaultDomain" will be taken.
- The bean `jmxLoader` defines the loader which is responsible for setting up the whole JMX world.

9.4.3 Connector

If you want to use **JMX** in a project, then you have to define what kind of connector you want to set up. At the moment, EL4J provides a `HtmlAdapter` and a `JMXConnector`.

9.4.3.1 HtmlAdapter

The bean `htmlAdapter` is a HTTP connector that allows observing the `MBean Server` of the property `mbeanServer`. ***This adapter is installed by default.*** The page can be called with `http://localhost:9092`. If no port is defined, the default one is 9092. The `Html Adapter` is defined as follows:

```
<bean id="htmlAdapter" class="ch.elca.el4j.jmx.HtmlAdapterFactoryBean">
  <property name="mbeanServer">
    <ref bean="mBeanServer"/>
  </property>
  <property name="port">
    <value>9092</value>
  </property>
</bean>
```

9.4.3.2 JmxConnector

The bean `jmxConnector` is a JMX connector based on the JSR-160 jmxmp protocol. Any client tool able to handle this protocol can be used to work with this MBeans. One such tool is **MC4J**. The bean definition provided is the following:

```
<bean id="jmxConnector" class="org.springframework.jmx.support.ConnectorServerFactoryBean">
  <property name="server">
    <ref bean="mBeanServer"/>
  </property>
  <!-- This is the default URL anyway -->
  <property name="serviceUrl">
    <value>service:jmx:jmxmp://localhost:9876</value>
  </property>
</bean>
```

Note: This class is only available as of Spring 1.2. This connector is optional (is it in the optional conf directory).

9.4.4 Example with one ApplicationContext

Here is a possible Test Class that uses *jmx*:

```
public class TestClass {

    public static void main(String[] args) {

        ApplicationContext ac = new ClassPathXmlApplicationContext(new String[] {"classpath*:mandatory/*.xml"});

        System.out.println("Waiting forever...");
        try {
            Thread.sleep(Long.MAX_VALUE);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

9.4.5 Configuration (explicit publication)

If you want to edit fields or invoke operations of a spring bean, e.g. on the bean `Fool`, then you have to explicitly register it via `mBeanExporter`. The automatically created proxies for Spring beans do not allow editing their fields. The `beans.xml` configuration file could look like this:

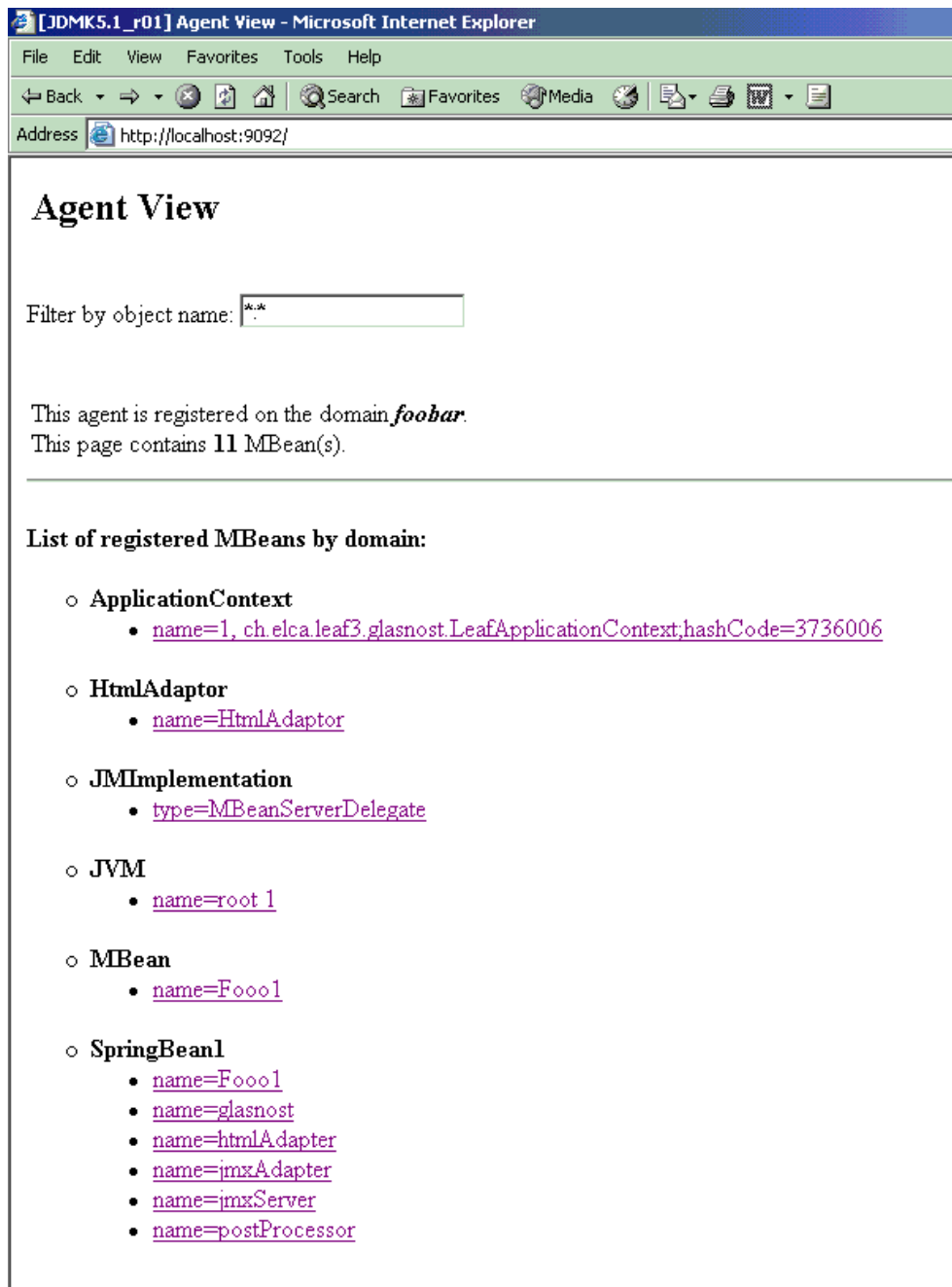
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <import resource="classpath:optional/htmlAdapter.xml"/>
    <bean id="mBeanExporter" class="org.springframework.jmx.export.MBeanExporter"
        depends-on="mBeanServer">
        <property name="beans">
            <map>
                <entry key="MBean:name=Fool">
                    <ref bean="Fool"/>
                </entry>
            </map>
        </property>
        <property name="server">
            <ref bean="mBeanServer"/>
        </property>
    </bean>
    <bean id="Fool" class="ch.elca.el4j.test.Fool">
        <property name="fullName">
            <value>foo</value>
        </property>
    </bean>
</beans>
```

In the bean `mBeanExporter` you can register which beans you want to expose as MBeans, i.e. you want to be able to modify. In this example, bean `Fool` will be exposed. The property `server` of `mBeanExporter` has to be set to the `mBeanServer` bean (which is loaded via `classpath*:mandatory/jmx.xml`). You can define as many `mBeanExporters` as you want, but do not forget to give each `mBeanExporter` bean in the same `ApplicationContext` a different name.

Important: The following Naming Convention has to be preserved regarding the property beans: The key entry has to be "MBean:name="+ `beanName`. Each `SpringBean` contains a link to its `MBean` if there is one.

By directing your browser to `http://localhost:9092`, you get the following view:



As you can see, the **module-jmx** created a proxy bean called "SpringBeanX:name=beanName" where X is a static counter. In the domain "MBean", you can find all the MBeans that you have registered via `jmxAdapter`, which is now called `mBeanExporter`.

9.4.6 Example with more than one ApplicationContext

If more than one `ApplicationContext` is loaded, then you have two possibilities:

- If you want to register another `ApplicationContext` at the same `mBeanServer`, then you have to choose the same `defaultDomain` as the other `Application Context` since the domain of the `MBean`

Server actually defines the MBean Server.

- If you want to register another `ApplicationContext` at a different `mBeanServer`, then you have to follow these two steps:
 - ◆ Override the `defaultDomain` property of the `mBeanServer` bean by the `org.springframework.beans.factory.config.PropertyOverrideConfigurer` for example with the entry `mBeanServer.defaultDomain=foobar2`.
 - ◆ The connector tho this MBean Server has to use a non-used port, e.g. 9093.

9.5 References

Further information regarding JMX can be found under
<http://java.sun.com/products/JavaManagement/index.jsp>.

10 Documentation for module light statistics

10.1 Purpose

The module *lightStatistics* allows setting up performance measurements very easily.

10.2 Important concepts

This module uses a simplified version of the Spring performance interceptor to gather execution times.

10.2.1 Monitoring strategies

- **JMX**: Allows querying performance measurements via JMX
 - ♦ **HTML adapter**: Provides a simple web based JMX interface available by default at <http://localhost:9092>.
 - ♦ **JMX connector**: activates the JMX connector to allow JMX conformant viewer to query data.
- **JAMon admin jsp**: The JAMon admin jsp is deployed along with the web application (in fact, the jsp file is always provided but only usable within a web application server).

10.3 How to use

10.3.1 Configuration

Using either the JAMon admin jsp within a web application container or the JMX HTML adapter, you don't have to do anything except adding the dependency to this module to your project definition. By default, all beans are advised by the measurement interceptor. The set can be limited to a particular name pattern using Spring's [configuration override facilities](#).

10.3.2 Demo

A demo module named *module-light_statistics-demos* is provided. It shows how to use the JMX HTML adapter in a stand-alone application.

10.3.3 How to set up the module-light_statistics for the ref-db sample application

This example shows how to use the performance monitor in the ref-db sample application.

10.3.3.1 binary-modules.xml

Add the following two lines to the `binary-modules.xml` file that is in the refdb's root directory.

```
<attribute name="binrelease.version.module-light_statistics" value="1.0"/>
<attribute name="binrelease.version.module-jmx" value="1.0"/>
```

10.3.3.2 project.xml

Add the following dependency to the `refdb-web` module definition:

```
<dependency module="module-light_statistics">
  <mapping target="jmx"/>
</dependency>
```

If you just want to use the admin jsp file without the JMX support, then replace the mapping target `jmx` with `web` (the jsp file is always provided, but runs in a web application environment only). Doing so, the lines you

have to insert look like this:

```
<dependency module="module-light_statistics">
  <mapping target="web"/>
</dependency>
```

Remark: the difference between the two is that a dependency to a different execution unit is used (this a feature of EL4Ant, please refer there for more documentation).

10.3.3.3 Limit the set of intercepted beans

Spring allows overriding configurations in properties files. Limiting the set of intercepted beans makes use of this feature. The key in the properties file is `lightStatisticsMonitorProxy.beanNames`. More details about how to use spring's configuration features can be found [under the PropertyConfiguration topic](#).

Important: In order to get the ref-db web application run with this module you have to limit the set of advised beans (there are some beans that are not advisable)! e.g. use this in your `override.properties` file:

```
lightStatisticsMonitorProxy.beanNames=reference*
```

and use the following bean definition:

```
<bean id="propsOverride" class="org.springframework.beans.factory.config.PropertyOverrideConfigurer">
  <property name="locations">
    <value>classpath:mandatory/override.properties</value>
  </property>
</bean>
```

Notice: both files, the `override.properties` and the bean definition can be added to the ref-db web's `mandatory` folder to be loaded automatically.

10.4 FAQ

- I got exceptions that Spring can not inject some dependencies. Without the `module-light_statistics`, everything runs nicely.
 - ♦ Maybe there are some classes that cannot be advised. Use the `lightStatistics-override.properties` to specify the beans to advise, as described [here](#).

10.5 References

JAMon web site <http://www.jamonapi.com/>

11 Exception handling guidelines

For an introduction to exception handling in Java, please refer to chapter 2 of LEAF 2 exception handling guidelines.

11.1 Topics

- When to define what type of exception, normal vs. abnormal results
 - ◆ What results are signalled with exceptions?
 - ◆ Use checked or unchecked exceptions?
 - ◆ When to define own exceptions, when to reuse the existing ones?
- Implementing exceptions
- Where and how to handle exceptions
 - ◆ Who handles what exceptions?
 - ◆ How to handle exceptions?
 - ◆ How to trace exceptions?
 - ◆ How to throw an exception as a consequence of another exception?
- Related useful concepts and hints
- Antipatterns
- References

11.2 When to define what type of exceptions? Normal vs. abnormal results

Throwing exceptions is expensive (in some examples up to 800 times slower than returning a "normal" value!). Therefore exceptions should be used for exceptional cases only (i.e., for cases that do not occur frequently).

A method invocation on an interface can have 2 fundamentally different type of results:

- **Normal results:** the result matches the level of abstraction of the interface. Examples: If one tries to make a withdraw on a bank account, possible results are: ok, that the account is overdrawn or locked. These are normal and expected events, on the same level of abstraction than the interface. Normal errors that are expected (i.e. a subset of *normal results*) are often also called *business exceptions*.
- **Abnormal results:** these results are not on the level of abstraction of the interface. They reveal implementation details and/or are for very unlikely events. The caller can typically not do much in response to an abnormal result. Such results are typically best handled on a higher level (often global for an application). Abnormal results are also appropriate to signal that the method was used improperly (e.g. when a precondition has been violated). Abnormal results are also used for situations that can't be handled during runtime. Examples: `OutOfMemoryError`, `PreconditionRTException`, `SQLException` indicating that the connection to the database is lost, `RemoteException`, ...

We will see later that we typically use checked exceptions for normal results and unchecked exceptions for abnormal results.

11.2.1 Further examples

Because this is a very important distinction, here are some more examples. Whether a result is normal or abnormal depends on its *context*:

- Method `Account.withdraw()`
 - ◆ normal results: ok, overdrawn or locked
 - ◆ abnormal results: `RemoteException` (of RMI), `SQLException` **Rationale:** They have nothing to do with the withdraw method, they are an implementation detail.

- Method `DatabaseAccessLayer.connectDb()`
 - ◆ normal results: ok, not ok (not ok may be the same thing as the `SQLException` of the previous example: in this context it is normal)
 - ◆ abnormal results: `RemoteException` (of RMI) **Rationale:** RMI has nothing to do with databases accesses, it's an orthogonal issue.
- Consider an order system of an online-shop. Every 1'000'000th customer gets a gift. Such a result is sufficiently rare that we could say it is abnormal. (So something abnormal does not need to be a mistake!) We could therefore throw a runtime exception for this abnormal case.

11.2.2 How to handle normal and abnormal cases

For **normal results** that are expected special cases (including expected errors) we use **checked exceptions** or special **return values**. One should be conservative with checked exceptions. Avoid many newly defined checked exceptions. This leads to many catch blocks in the code (this makes the code longer and harder to read). Try also to avoid having a method throwing too many checked exceptions. Such a method can be very cumbersome to use. (As a bad example, please have a look at the Java API for reflection (package `java.lang.reflect`)). Signaling special cases via return values is sometimes appropriate when the event occurs often (due to the implied performance overhead of exceptions).

We use **unchecked exceptions** to inform about **abnormal results**. As with checked exceptions, try again to avoid too many new exceptions. Names of unchecked exceptions should have a `RuntimeException` suffix.

Remark: The `RemoteException` of RMI violates these guideline, as it should be an unchecked exception. (Many people consider this a design mistake of RMI.)

11.3 Implementing exceptions classes

You can use the classes `BaseExceptions` and `BaseRuntimeExceptions` as base classes for new exceptions. These classes provide base support for exception internationalization.

Do not use the string message of an exception to differentiate among different exception situations. For example, one should *not* use in a project just one exception class (e.g. the predefined `BaseException`) with different String messages to differ between situations. This bad practice makes it hard to react differently in function of what happened (as it would require parsing the exception message), it would also not allow adding particular attributes to the exception class, and would not document what type of exceptions can be thrown in a method signature. Finally, it would make exception message internationalization harder (because one would need to parse the exception message first). Sometimes it is desirable not to write one exception class per exception situation (e.g. there may just be too many exception classes). In such cases one can use a common base exception class and use an error code to differentiate between the exceptions.

We recommend not to make a difference between exceptions of EL4J code and exceptions of applications using EL4J. (This means that the same rules apply and that there is no separate exception hierarchy for the two contexts.)

Remember that one should avoid adding too many new exceptions. You can reuse (i.e. use in your method signatures) exceptions of the JDK. Frequently useful candidates are `IllegalArgumentException` or `IndexOutOfBoundsException`.

11.4 Handling exceptions

11.4.1 Where to handle exceptions?

Normal results of invocations should be handled by the code making the invocation. Optionally it may make sense to propagate the exception to the caller of the invoking class (in other words: up the calling stack).

Abnormal results (those returned via unchecked exceptions) are typically passed up the calling stack and handled on a higher level (not directly where the invocation was made). Handle an abnormal result only if you can really do something against the problem or if you are on the top-level of a component that is responsible to handle all abnormal cases. A pattern that separates the handling of abnormal situations in a nice way is the [SafetyFacade](#).

11.4.2 How to trace exceptions?

One should not trace normal results (including exceptions that signal normal results!) of method invocations. (Unless there is some external requirement for this.)

Abnormal situations should be traced where they are caught.

Please refer to [TracingInfrastructure](#) for more detail on general tracing. Typically one uses `error` or `fatal` priority levels when tracing abnormal situations.

11.4.3 Rethrowing a new exception as the consequence of a caught exception

Try to avoid making too many such exception translations (i.e. in a catch statement *translate* an exception by throwing another exception for it). If you do it anyway, you should wrap the caught exception in the newly thrown exception (in order not to lose information). The `Exception` class of JDK 1.4 provides support for this.

11.5 Related useful concepts and hints

11.5.1 Add attributes to the exception class

As Java exceptions are classes, it is possible to add attributes to exception classes. This can be useful e.g. to include information needed to fix the abnormal situation or to provide more information about the exceptional situation.

Such attributes are particularly useful when the exception is treated programmatically (e.g. to do something in function of the value of such attributes). Having these attributes explicitly as attributes and not just embedded in the error message avoids that the error message needs to be parsed. In addition, it helps to internationalize exceptions. See also the example of the `BaseException` class that illustrates how this can be used with JDK `MessageFormats`. The string message of exceptions should contain all attributes that are useful for someone trying to figure out what went on. (We don't print automatically all attributes of exceptions.)

11.5.2 Mentioning unchecked exceptions in the Javadoc

Sometimes it is useful to mention unchecked exceptions that can be thrown by a method (even though this is not required by Java). This can be made in the code (one has the right to add an unchecked exception to the `throws` definition of a method) or in the Javadoc. This makes the user of the method aware of the unchecked exception that may be potentially be thrown by the method.

11.5.3 Checking for pre-conditions in code

Assumptions a programmer of code makes about how the code is used, are called pre-conditions. Violated pre-conditions are abnormal situations. Therefore one should use unchecked exceptions to indicate pre-conditions. Pre-conditions are checked in the beginning of the body of method implementations. One should keep such checks in the code even if the code is deployed in a production environment. Such pre-condition checks are particularly useful when a component is used after its creation or in another context. Rationale: such pre-conditions check that the assumptions of the programmer are valid.

There is a [Reject](#) class (in the core module) that helps to support this usage. This usage is also recommended

in the [assertion guidelines](#) of sun. (We refer to the text: "By convention, preconditions on public methods are enforced by explicit checks that throw particular, specified exceptions.") We propose to check such preconditions on public methods via the `Reject` class.

Please refer also the the [AssertionUsageGuidelines](#) for more details about other types of design by contract support.

11.5.4 Exception–safe code

Exception–safety is a property of well–implemented code. There is weak and strong exception safety.

For a method `m()` that is **weakly exception safe**, the following conditions hold when it throws an exception:

1. `m()` does not complete its operation.
2. `m()` releases all the resources it allocated before the exception was thrown.
3. If `m()` changes a data structure, then the structure must remain in a consistent state.

In summary, if a weakly exception safe method `m()` updates the system state, then the state must remain reasonable.

Strongly exception safe methods additionally verify the following condition:

- If a method `m()` terminates by propagating an exception, then it has made no change to the state of the program.

Both exception safety properties are desirable. However as the implementation of strongly exception safe methods can be quite tricky, we only require methods in EL4J to be weakly exception safe. Please refer to § 3.5.1 of the LEAF 2 exception handling guidelines for more details.

11.5.5 Handling SQL exceptions

To handle SQL exceptions, we strongly recommend the helper classes of the spring framework. This support is sometimes referred to as [Spring's generic DataAccessException hierachy](#). To profit from this hierachy, use the Spring simplification templates for integration of iBatis or Hibernate. This allows profiting from this hierarchy almost for free. EL4J provides an improvement to this exception mapping, please refer to the file [sql-error-codes.xml](#) of the core module and the package [ch.elca.el4j.services.persistence.generic.sqlexceptiontranslator](#).

11.5.6 Exceptions and transactions

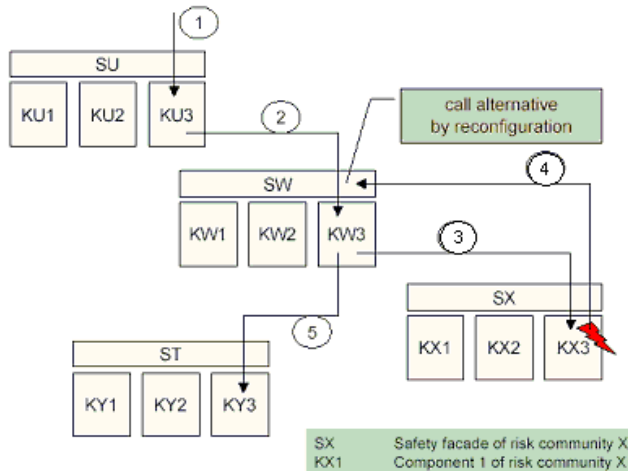
Transactions should often be rolled back after an exception occurs. Please refer to [ModuleTransactionAttributes](#) for a description on how to do this automatically.

11.5.7 SafetyFacade pattern

The goal of the safety facade is that it handles all the abnormal situations, such that for a user of a component, the business operation either succeeds or completely fails. The safety facade has made all attempts to fix or retry and has informed the required parties if necessary. A safety facade takes the responsibility of treating abnormal situation away from normal business code. This is particularly interesting as the code can typically not do much against it. The safety facade wraps a group of component implementations (e.g. via dynamic proxies) and provides a "better" quality of service (i.e. either the components work or they fail completely) for the users of the component implementations.

The following picture illustrates this. Four groups of components ("risk communities") are each assembled with their safety facade. The safety facade treats all abnormal situations. The numbers indicate a sample use of a safety facade: KU3 calls KW3 and KW3 calls KX3. KX3 indicates an abnormal situation (throws an unchecked exception). The safety facade SW therefore reconfigures the system such that KW3 retries once again with the component KY3.

Emergency Handling and Safety Facades (2)



The module `ModuleExceptionHandler` implements a safety facade. This idea is described in "Moderne Software Architekturen", §5.4 .

11.6 Antipatterns

Exceptions are considered to be an important tool of modern programming languages, but they become a nuisance for programmers. We list some of the typical problems (antipatterns) encountered in projects ranging from 1 to more than 100 man-years:

- There are a large number of different exceptions classes. It is neither clear when exceptions should be thrown nor how they should be handled.
- A huge number of exception classes create undesired dependencies between the caller and the callee.
- The code gets messy because of nested try-catch blocks.
- Many catch blocks are either empty, contain little value-adding code (output to the console, useless mappings of one exception class into another) or – at best – some logging, but no true exception handling.
- Exceptions are misused to return ordinary values.

Please keep these antipatterns in mind! They are mostly avoided if you use the previous rules and common sense.

11.7 References

- LEAF 2 exception handling guidelines:
http://leaffy.elca.ch/leaf/Documentation_Mirror/guidelines/LEAFExceptionHandlerGuidelines.doc
 - ♦ The usage of exceptions has changed in the EL4J. Chapter 2 remains valid.
- Errors and Exceptions – Rights and Responsibilities, Johannes Siedersleben, ECCOP 2003, paper:
http://www.sdm.de/web4archiv/objects/download/pdf/vonline_siedersleben_ecoop03.pdf, slides:
<http://homepages.cs.ncl.ac.uk/alexander.romanovsky/home.formal/Johannes-talk.pdf>
- Moderne Software Architekturen, Siedersleben, 2004, Chapter 5 (in German)

- Rules for Developing Robust Programs with Java, Article about exception handling in Java
<http://www.idi.ntnu.no/grupper/su/fordypningsprosjekt-2003/fordypning2003-Nguyen-og-Sveen.pdf>
 - ◆ Easy to read, many interesting patterns about exception handling.

12 Acknowledgments

There are many persons having contributed to EL4J (in alphabetical order):

- Raphael Boog
- Andi Bur
- Christian Gasser
- Jacques–Olivier Haenni
- Martin Meier
- Philipp Oser
- Andreas Pfenninger
- Jean–François Poilpret
- Yves Martin
- Nicola Schiper
- Marc Schmid
- Christoph Schwitter
- Martin Zeltner

Thank you!

13 References

- EL4J Website, <http://el4j.sourceforge.net/>
- EL4Ant Website, <http://el4ant.sourceforge.net/>
- Professional Java Development with the Spring Framework; Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg, Colin Sampaleanu; wrox; 2005
- LEAF 2 Datasheet (the earlier, proprietary J2EE framework with similar goals than EL4J), http://www.elca.ch/Solutions/Technology_Frameworks/LEAF/index.php