

EL4J Reference Documentation Version 1.1.0

Incremental Improvements for Spring

Rapport	Version	Date	Author(s)	State	Visa
123456	1.32	31 Oct 2006 – 14:46	ABU, MZE, POS, RBO, AMA	Valid	POS

Table of Contents

1 Introduction.....	1
2 Unique Features of EL4J.....	2
3 Maven 2 and installing modules.....	4
3.1 Maven 2.....	4
3.2 Installing modules.....	4
3.3 Steps to install a module with Maven 2.....	4
4 Documentation for module core.....	6
4.1 Purpose.....	6
4.2 Support for Maven 2 modules on the level of Spring.....	6
4.2.1 Module abstraction of Maven 2.....	6
4.2.2 ModuleApplicationContext.....	7
4.2.3 Convention on how to organize configuration.....	7
4.2.3.1 Examples.....	8
4.2.3.2 Usage of configuration using this convention.....	10
4.3 Java 5 annotations for Transactions.....	10
4.4 Convenience Attributes for Transactions – DEPRECATED.....	11
4.4.1 Purpose.....	11
4.4.2 Introduction.....	11
4.4.3 Configuration.....	13
4.4.3.1 Overview.....	13
4.4.3.2 How to define an attribute.....	13
4.4.3.3 How to organize the transaction attributes.....	14
4.4.3.4 Transaction propagation behaviors.....	15
4.4.3.5 Transaction attribute classes.....	15
4.4.3.6 Adding spring beans to enable transactional behaviour.....	16
4.4.3.7 Programmatical transaction demarcation (start transaction, commit, rollback in code).....	18
4.4.3.8 setRollbackOnly is not equals to setReadOnly.....	18
4.4.4 Internal design.....	18
4.5 Meta data support.....	18
4.5.1 Purpose.....	18
4.5.2 Configuration meta data type Java Annotation.....	19
4.5.3 Configuration meta data type Common Attributes.....	19
4.5.4 Implementation of the Interceptor.....	20
4.5.5 Integration of existing which uses the displaced module Attribute Convenience.....	21
4.5.6 Working of the inheritance.....	22
4.5.7 Internal design.....	24
4.5.7.1 Classes TODO.....	24
4.5.7.2 Collection procedure.....	24
4.6 Search service.....	24
4.7 Additional Features.....	26
4.7.1 Configuration merging via property files.....	26
4.7.2 Bean locator.....	27
4.7.3 Bean type auto proxy creator.....	27
4.7.4 Exclusive bean name auto proxy creator.....	27
4.7.5 Abstract parent classes for the Typesafe Enumerations Pattern.....	27
4.7.6 Reject (Precondition checking).....	27
4.7.7 JNDI Property Configurers.....	28
4.7.8 Generic repository.....	28
4.7.9 DTO helpers.....	28
4.7.10 Primary key.....	28
4.7.11 SQL exception translation.....	28
4.8 Packages that implement the core module.....	28

Table of Contents

5 Documentation for module web.....	30
5.1 Purpose.....	30
5.2 Features.....	30
5.3 How to use.....	30
5.3.1 General configuration of the web module.....	30
5.4 Reference documentation for the Module-aware application contexts.....	30
5.4.1 Concept.....	30
5.4.1.1 ModuleDispatcherServlet.....	31
5.4.1.2 ModuleContextLoader.....	31
5.4.2 Build system integration.....	32
5.4.2.1 Adding files manually.....	32
5.4.3 Limitations.....	32
5.4.4 MANIFEST.MF configuration section format.....	33
5.4.5 Implementation Alternative: Idea.....	33
5.4.6 Resources.....	34
6 Documentation for module remoting.....	35
6.1 Purpose.....	35
6.2 Introduction.....	35
6.3 How to use.....	35
6.3.1 Remoting modules.....	35
6.3.2 Configuration.....	36
6.3.2.1 Recommended configuration file organisation.....	36
6.3.2.2 Configuration summary.....	37
6.3.2.3 How to use the Rmi protocol.....	39
6.3.2.4 How to use the Hessian protocol.....	39
6.3.2.5 How to use the Burlap protocol.....	42
6.3.2.6 How to use the HttpInvoker protocol.....	42
6.3.2.7 How to use the Soap protocol (XFire).....	42
6.3.2.8 How to use the Soap protocol (Axis).....	44
6.3.2.9 How to use the EJB protocol.....	49
6.3.2.10 How to use the Load Balancing composite protocol.....	49
6.3.2.11 Introduction to implicit context passing.....	52
6.3.3 Benchmark.....	53
6.3.4 Remoting semantics/ Quality of service of the remoting.....	54
6.3.4.1 Cardinality between client using the remoting and servants providing implementations.....	54
6.3.4.2 What happens when there is a timeout or another problem during remoting.....	54
6.4 Internal design.....	55
6.4.1 Sequences.....	55
6.4.1.1 Sequence diagramm from client side.....	55
6.4.1.2 Sequence diagramm from server side.....	57
6.4.2 Creating a new interface during runtime.....	59
6.4.3 Internal handling of the RMI protocol (in spring and EL4J).....	59
6.4.4 To be done.....	59
6.5 Related frameworks.....	60
6.5.1 extrmi.....	60
6.5.2 Javaworld 2005 idea.....	60
7 Documentation for module EJB remoting.....	61
7.1 Purpose.....	61
7.2 Important concepts.....	61
7.3 How to use.....	61
7.3.1 Configuration.....	61
7.3.1.1 How to use the EJB protocol.....	61
7.3.1.2 How to use the build system plugin.....	63

Table of Contents

7 Documentation for module EJB remoting	63
7.3.1.3 How to use the EJB remoting module without the EL4Ant build system.....	63
7.4 References.....	64
7.5 Internal design.....	64
7.5.1 EJB generation.....	64
7.5.2 Adding support for another container.....	65
8 Documentation for module security.....	67
8.1 Purpose.....	67
8.2 Features.....	67
8.3 How to use.....	67
9 Documentation for module exception handling.....	68
9.1 Purpose.....	68
9.2 Important concepts.....	68
9.3 How to use.....	68
9.3.1 Configuration.....	68
9.3.1.1 Exception handlers.....	69
9.3.1.2 Example 1: Safety Facade for one Bean.....	69
9.3.1.3 Example 2: Context Exception Handler.....	70
9.3.1.4 Example 3: RoundRobinSwappableTargetExceptionHandler.....	70
9.3.1.5 Example 4: Using several exception handlers, each configured by a separate exception configuration.....	71
9.4 References.....	72
9.5 Internal design.....	73
9.5.1 Context Exception Handler.....	73
10 Documentation for module JMX.....	74
10.1 Purpose.....	74
10.2 Introduction to Java management eXtensions (JMX).....	74
10.3 Feature overview.....	74
10.4 Usage.....	75
10.4.1 Spring/JDK versioning issue.....	75
10.4.1.1 Spring versions 1.1 <--> 1.2.....	75
10.4.1.2 JDK versions 1.4.2 <--> 1.5.....	75
10.4.2 Basic Configuration (implicit publication).....	75
10.4.3 Connector.....	76
10.4.3.1 HtmlAdapter.....	76
10.4.3.2 JmxConnector.....	76
10.4.4 Example with one ApplicationContext.....	77
10.4.5 Configuration (explicit publication).....	77
10.4.6 Example with more than one ApplicationContext.....	78
10.5 Implemented Features.....	79
10.5.1 JVM-Monitor.....	79
10.5.2 Log4jConfig.....	80
10.5.3 Spring Beans.....	82
10.5.4 JDK 1.5 Standard MBeans.....	82
10.6 Patch.....	82
10.7 References.....	83
11 Documentation for module light statistics.....	84
11.1 Purpose.....	84
11.2 Important concepts.....	84
11.2.1 Monitoring strategies.....	84
11.3 How to use.....	84
11.3.1 Configuration.....	84

Table of Contents

11 Documentation for module light statistics	85
11.3.2 Demo.....	85
11.3.3 How to set up the module-light_statistics for the ref-db sample application.....	85
11.3.3.1 binary-modules.xml.....	85
11.3.3.2 project.xml.....	85
11.3.3.3 Limit the set of intercepted beans.....	85
11.4 FAQ.....	86
11.5 References.....	86
12 Documentation for module Spring Rich Client Platform (Spring RCP).....	87
12.1 Purpose.....	87
12.2 Important concepts.....	87
12.2.1 Overview.....	87
12.2.2 Application in general.....	89
12.2.3 Windows.....	89
12.2.4 Pages.....	90
12.2.5 Components.....	91
12.2.6 Executors.....	93
12.2.7 Application services.....	93
12.2.7.1 Message source accessor.....	94
12.2.7.2 Image and icon sources.....	94
12.2.7.3 Rule source.....	94
12.2.7.4 Other services.....	94
12.3 How to use.....	94
12.3.1 Launch the application.....	94
12.3.2 General configuration.....	95
12.3.3 Property merger and overrider.....	95
12.3.4 Rule source.....	96
12.3.5 Window commands.....	96
12.3.6 Message and image property files.....	97
12.3.7 Pages.....	98
12.3.8 Views.....	100
12.3.8.1 Bean table view.....	100
12.3.8.2 Search view.....	102
12.3.8.3 Combining a search view and a bean table view.....	103
12.3.9 Executors.....	104
12.3.9.1 Overview.....	104
12.3.9.2 AbstractBeanExecutor.....	104
12.3.9.3 SelectAllBeanExecutor.....	104
12.3.9.4 AbstractDisplayableBeanExecutor.....	105
12.3.9.5 AbstractConfirmBeanExecutor.....	105
12.3.9.6 AbstractFinishBeanExecutor.....	105
12.3.9.7 AbstractEditorBeanExecutor.....	105
12.3.9.8 AbstractWizardBeanExecutor.....	106
12.3.9.9 AbstractBeanPropertiesExecutor.....	106
12.3.9.10 AbstractBeanNewExecutor.....	107
12.3.9.11 AbstractBeanDeleteExecutor.....	108
12.3.9.12 Conclusion.....	109
12.4 References.....	109
13 Documentation for module IBatis.....	110
13.1 Purpose.....	110
13.2 How to use.....	110
13.2.1 Dao layer.....	110
13.2.2 Type handler callbacks.....	110

Table of Contents

14 Documentation for module Hibernate.....	112
14.1 Purpose.....	112
14.2 How to use.....	112
14.2.1 Criteria transformation.....	112
14.2.2 Generic Hibernate repository.....	112
14.2.3 Hibernate validation support.....	112
15 Documentation for module XmlMerge.....	114
15.1 Purpose.....	114
15.2 Introduction.....	114
15.3 Module contents.....	114
15.4 Important concepts.....	115
15.4.1 Original and Patch.....	115
15.4.2 Processing model.....	115
15.4.3 Core Concepts as Java Interfaces.....	116
15.4.3.1 Operations.....	116
15.4.3.2 Configuration with Factories.....	117
15.5 Built-in implementations.....	117
15.5.1 Operations.....	117
15.5.1.1 Matchers.....	117
15.5.1.2 Mapper.....	117
15.5.1.3 Actions.....	117
15.5.2 Aliases for Built-In Operations.....	118
15.5.3 XmlMerge Implementation.....	118
15.5.4 Operation Factories.....	118
15.6 Configuring your Merge.....	118
15.6.1 Programming the Configuration.....	118
15.6.2 Configuring with XPath and Properties.....	119
15.6.3 Configuring with Inline Attributes in Patch Document.....	120
15.7 Writing your own Operations.....	121
15.8 How to use.....	122
15.8.1 Command-line Tool.....	122
15.8.2 Ant Task.....	123
15.8.3 Spring Resource.....	123
15.8.4 Web demo.....	124
15.9 References.....	124
16 Documentation for module Web Test.....	125
16.1 Purpose.....	125
16.2 Overview of our webtests.....	125
16.3 Example.....	125
17 Documentation for module TcpForwarder.....	126
17.1 Purpose.....	126
17.2 Important concepts.....	126
17.3 How to use.....	126
17.3.1 Command line user interface to switch TCP connections on or off.....	126
17.3.1.1 Parameters (tbd in code).....	126
17.3.1.2 Commands.....	126
17.3.1.3 Notes.....	127
17.3.2 Programmatically halting and resuming network connectivity.....	127
17.3.2.1 Code configuration.....	127
17.3.2.2 Switch on / off connections.....	127
17.4 Demonstration code.....	127

Table of Contents

18 Generic DAOs in EL4J.....	128
18.1 Basic introduction.....	128
18.2 Sometimes we can omit or bypass the service layer.....	128
18.3 Benefits of the approach.....	129
18.4 References.....	130
19 Exception handling guidelines.....	131
19.1 Topics.....	131
19.2 When to define what type of exceptions? Normal vs. abnormal results.....	131
19.2.1 Further examples.....	131
19.2.2 How to handle normal and abnormal cases.....	132
19.3 Implementing exceptions classes.....	132
19.4 Handling exceptions.....	132
19.4.1 Where to handle exceptions?.....	132
19.4.2 How to trace exceptions?.....	133
19.4.3 Rethrowing a new exception as the consequence of a caught exception.....	133
19.5 Related useful concepts and hints.....	133
19.5.1 Add attributes to the exception class.....	133
19.5.2 Mentioning unchecked exceptions in the Javadoc.....	133
19.5.3 Checking for pre-conditions in code.....	133
19.5.4 Exception-safe code.....	134
19.5.5 Handling SQL exceptions.....	134
19.5.6 Exceptions and transactions.....	134
19.5.7 SafetyFacade pattern.....	134
19.6 Antipatterns.....	135
19.7 References.....	135
20 Acknowledgments.....	136
21 References.....	137

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:

- 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 for long-running daemons
- Support to see the active beans and their configuration in JMX
- A light exception handling framework that implements a safety facade
- Improvements for Spring RCP

Used libraries and tools

- Most libraries that are included in the spring framework
- Maven 2

For another short introduction to EL4J we refer to the EL4J datasheet available on our [company webpage](#).

EL4J is a package for Java developers – ready to start working. It is an explicit goal of EL4J that you should not loose time and be able to get working right away. From version 1.1 it is published under the LGPL (<http://www.gnu.org/licenses/lgpl.txt>) at sourceforge. Please contact info@elca.ch for other licensing.

EL4J is already in use in 16+ projects within ELCA (<http://www.elca.ch>).

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

2 Unique Features of EL4J

This document lists the distinctive features of EL4J. A frequent question about EL4J is what it provides additionally to the frameworks it includes. One benefit of EL4J is certainly the selection, integration, and pre-configuration of leading components. More benefit comes from the *new* features that EL4J provides.

The following list shows the distinctive features of EL4J (this list is not exhaustive, please check also the module documentation and the javadoc):

- Application templates to get quickly started: for GUI, Web and ULC. The goal is to have a running sample application within 10 minutes! In this running application you have a proven structure and sample solutions for typical development issues.
- Support for modules with code, default configuration and dependencies. This feature is based on the build system (Maven 2), the basic spring abstractions, some EL4J support and conventions.
 - ◆ More flexible and robust loading of configuration resources
 - ◇ Inclusion and exclusion list to include/ exclude configuration files
 - ◇ Store the list of configuration resources to load in the jar-file manifest
 - ◇ Merging of spring configuration: adding more parameters to an existing list of parameters
 - ◆ Each EL4J module packages functionality with samples, documentation and default configuration.
- Improved remoting
 - ◆ Easier switching between remoting protocols (unification of remoting protocols)
 - ◆ Automatically deploy POJOs as EJB 2.1 beans
 - ◆ Remote POJOs via SOAP (simpler than with basic Spring)
 - ◆ Provide light load-balancing via the more flexible remoting layer
 - ◆ Implicit context passing over process boundaries
- EL4J cockpit
 - ◆ Auto-publication of the list of spring beans with their configuration values, interceptors and other useful info
 - ◆ Get a simple overview of the running threads
 - ◆ Change the log4j configuration dynamically
- Exception handling
 - ◆ Exception handling guidelines
 - ◆ Safety facade
 - ◆ More exception mappings for database accesses (additionally: duplicate values, out of bound values)
- Convenient Maven 2.0 setup
 - ◆ Well thought-through use of Maven. Hierarchical split of configurations. Use of fine-grained projects.
 - ◆ Bugfixes for maven and related tools (we have submitted about 20 patches, some of which are already included in maven)
 - ◆ Own plugin to extend maven: copy tool for combined report generation.
 - ◆ Presentation about how to migrate to mvn and many detailed information and hints
 - ◆ Maven cheat sheet
- GUI: Extension of Spring RCP (an open Swing GUI framework): better support for auto-POJO display, less coding required, more components, ...
- Daemon manager
- License manager
- XML Merger
- Extended file support (fast file observation, directory size information, easier file search capabilities)
- Generic DAO implementation (reduce coding, improve homogenization)
- Easier support for annotation to interceptor mappings (no coding required for basic cases)
- Ajax demo
- TCP forwarder to automatically test TCP connection failures

- Tracking the invocation graph (potentially over process boundaries), measuring performance and generating a sequence diagram for it
- Auto-idempotency interceptor
- Better documentation
 - ◆ Architecture discussions
 - ◆ EL4J Datasheet
 - ◆ Annotation cheat sheets
 - ◆ FAQ & infos on how to solve common problems
 - ◆ Documentation of each feature

The following external components are integrated in EL4J (this list is not exhaustive, please check also the list of included jar-files):

- Spring 2.0 framework
- Maven 2.0, JUnit
- Commons logging, log4j
- Hibernate
- Ibatis
- Acegi security framework
- Spring RCP
- **JWebUnit** and **HtmlUnit**
- Eclipse BIRT
- CGLib
- XFire
- Axis
- Caucho remoting: Hessian & Burlap
- XPF
- DWR
- Struts
- JaMon
- Quartz
- ULC

3 Maven 2 and installing modules

3.1 Maven 2

EL4J uses Maven 2 as its build system. For more information about maven we refer to its [website](#) and our introductory Maven presentation. The specific EL4J Maven plugins are documented via the standard plugin documentation support of Maven (available on our website).

3.2 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.

3.3 Steps to install a module with Maven 2

Installing a module with Maven 2 is very easy. Normally you have a "root-pom" for your own project that points to the "root-pom" of EL4J. Your project "root-pom" could look like the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>ch.elca.el4j</groupId>
    <artifactId>el4j</artifactId>
    <version>1.1.0-SNAPSHOT</version>
  </parent>

  <groupId>ch.elca.myproject</groupId>
  <artifactId>my-project</artifactId>
  <version>0.1-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>My Project</name>
  <description>My Project is an example project.</description>

  <repositories>
    <repository>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
      <releases>
        <enabled>true</enabled>
      </releases>
      <id>el4jReleaseRepositoryExternal</id>
      <name>External release repository of the EL4J project</name>
      <url>http://el4.elca-services.ch/el4j/maven2repository</url>
    </repository>
    <repository>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
      <releases>
        <enabled>false</enabled>
      </releases>
      <id>el4jSnapshotRepositoryExternal</id>
      <name>External snapshot repository of the EL4J project</name>
      <url>http://el4.elca-services.ch/el4j/maven2snapshots</url>
    </repository>
  </repositories>
</project>
```

```

    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
      <releases>
        <enabled>true</enabled>
      </releases>
      <id>el4jReleasePluginRepositoryExternal</id>
      <name>External release repository of the EL4J project</name>
      <url>http://el4.elca-services.ch/el4j/maven2repository</url>
    </pluginRepository>
    <pluginRepository>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
      <releases>
        <enabled>false</enabled>
      </releases>
      <id>el4jSnapshotPluginRepositoryExternal</id>
      <name>External snapshot repository of the EL4J project</name>
      <url>http://el4.elca-services.ch/el4j/maven2snapshots</url>
    </pluginRepository>
  </pluginRepositories>
</project>

```

This "root-pom" uses the EL4J version 1.1.0. Due to the repository definition in this "root-pom" Maven will look at the URL <http://el4.elca-services.ch/el4j/maven2repository> for pom artifact with groupId=ch.elca.el4j, artifactId=el4j, and version=1.1.0. The lookup URL will be <http://el4.elca-services.ch/el4j/maven2repository/ch/elca/el4j/el4j/1.1.0>

If the used EL4J version is a snapshot (i.e. 1.1.1-SNAPSHOT) Maven will look for the newest snapshot build in the URL <http://el4.elca-services.ch/el4j/maven2snapshots> instead of URL <http://el4.elca-services.ch/el4j/maven2repository>

Every dependency defined in your "root-pom" or a child pom of it will now profit from the dependency management of EL4J's "root-pom". Every EL4J module is predefined with version and scope. If one of your pom depends i.e. on **module-core** the dependency config snippet looks like the following:

```

<dependency>
  <groupId>ch.elca.el4j.modules</groupId>
  <artifactId>module-core</artifactId>
</dependency>

```

Maven will automatically use the version of **module-core** that belongs to the taken version of EL4J. To define the used version of **module-core** yourself, just add the version element like in the following:

```

<dependency>
  <groupId>ch.elca.el4j.modules</groupId>
  <artifactId>module-core</artifactId>
  <version>[1.6-20061124.131529-4]</version>
</dependency>

```

By surrounding the version number with square brackets Maven, will insist using just the given version of **module-core**, otherwise it looks for a new version of **module-core** every day.

4 Documentation for module core

4.1 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.

4.2 Support for Maven 2 modules on the level of Spring

The *module* support of the core module is provided in combination with the Maven 2 build system. Maven 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:

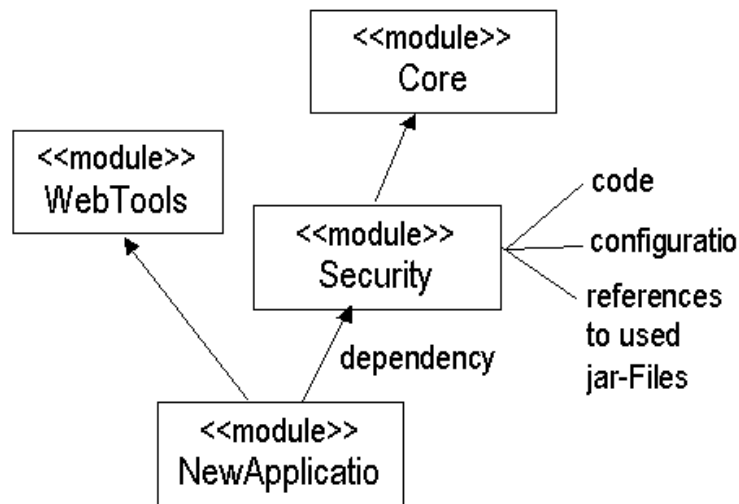
- the module abstraction of Maven 2
- the `ModuleApplicationContext` (a wrapper for the standard Spring application context)
- a convention on how to organize configuration information within each module

These three parts are described in the next sections.

4.2.1 Module abstraction of Maven 2

Maven 2 (<http://maven.apache.org/>) is a build system that gives you higher-level build abstractions than Ant. With Maven 2 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). Maven 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 [Maven 2](#).

4.2.2 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 Maven. (A Maven plugin 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](#).

4.2.3 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 `/resources`. This `/resources` 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. The testcases of a test module are also placed in the scenarios folder.

- ◆ '/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.
- '/etc': This folder contains various files that do not suit to another configuration folder, e.g. templates one can provide which can be helpful to efficiently develop applications or to understand the module or images used by the web modules.

By loading all files in '//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.

4.2.3.1 Examples

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

4.2.3.1.1 Example 1

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

- ch.elca.el4j.core.services.security:
 - ◆ '/resources/mandatory/':
 - ◊ security-attributes.xml
 - ◆ '/resources/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
 - ◆ '/resources/optional/':
 - ◆ '/resources/etc/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 **ModuleRemoting** module, the remote protocol is defined in that scope.

4.2.3.1.2 Example 2

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

- ch.elca.el4j.core.services.remoting:

- ◆ '/resources/mandatory/':
 - ◆ '/resources/scenarios/':
 - ◇ 'scope/':
 - client-config.xml
 - server-config.xml
 - ◇ 'protocol/':
 - rmi-protocol-config.xml
 - hessian-protocol-config.xml
 - burlap-protocol-config.xml
 - ◆ '/resources/optional/':
 - ◆ '/resources/etc/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 '/resources/etc/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.

4.2.3.1.3 Example 3

This example illustrates the `ModuleJmx`:

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

4.2.3.1.4 Example 4

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

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

TBD: is the following still correct as we use no longer the EL4Ant execution units?

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**.

4.2.3.2 Usage of configuration using this convention

This section presents how the security module (as defined above) could be used in an application. Note that Maven 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 ModuleApplicationContext(configurationFiles, new
    String[] {}, false);
```

4.3 Java 5 annotations for Transactions

By declaring annotations (see [Java language specification](#)) on methods the application context of Spring is able to detect which method to intercept and what kind of transaction to start or not. There are two annotations specially made for transaction declaration.

- ***org.springframework.transaction.annotation.Transactional***
 - ◆ Javadoc: <http://static.springframework.org/spring/docs/2.0.x/api/org.springframework.transaction.annotation.Transactional.html>
 - ◆ Annotation `Transactional` should be used only on methods in implementation classes.
- ***ch.elca.el4j.core.transaction.annotations.RollbackConstraint***
 - ◆ Contains only some elements from annotation `Transactional`. Used to declare rollback behavior if exception is thrown on method where this annotation is declared. Needs to be declared in combination with a `Transactional` annotation to enable transactional behavior.

Here an example of a declaration on an interface:

```
public interface KeywordDao {
    @RollbackConstraint(rollbackFor = { DataAccessException.class,
        DataIntegrityViolationException.class,
        OptimisticLockingFailureException.class })
    Keyword saveOrUpdate(Keyword keyword) throws DataAccessException,
        DataIntegrityViolationException, OptimisticLockingFailureException;
}
```

And here on an implementation class:

```
public class KeywordDaoImpl implements KeywordDao {
    @Transactional(propagation = Propagation.REQUIRED)
    Keyword saveOrUpdate(Keyword keyword) throws DataAccessException,
        DataIntegrityViolationException, OptimisticLockingFailureException {
        ...
        return xy;
    }
}
```

If the impl class above is now defined as bean in Spring application context you just have to add the predefined Spring config file
`classpath*:optional/interception/transactionJava5Annotations.xml` in application context's config locations ([view the config file](#)).

If classes `java.lang.RuntimeException` and `java.lang.Error` are not defined in ***no-rollback*** elements of annotations they will be automatically added to element `rollbackFor`.

Rollback behavior can be defined on annotation `Transactional` too but be aware that only the most specific annotation will be taken. It is not possible to merge multiple `Transactional` annotations. The same matches to annotation `RollbackConstraint`. All rollback constraints defined in annotation `RollbackConstraint` will be automatically added to annotation `Transactional`.

4.4 Convenience Attributes for Transactions – DEPRECATED

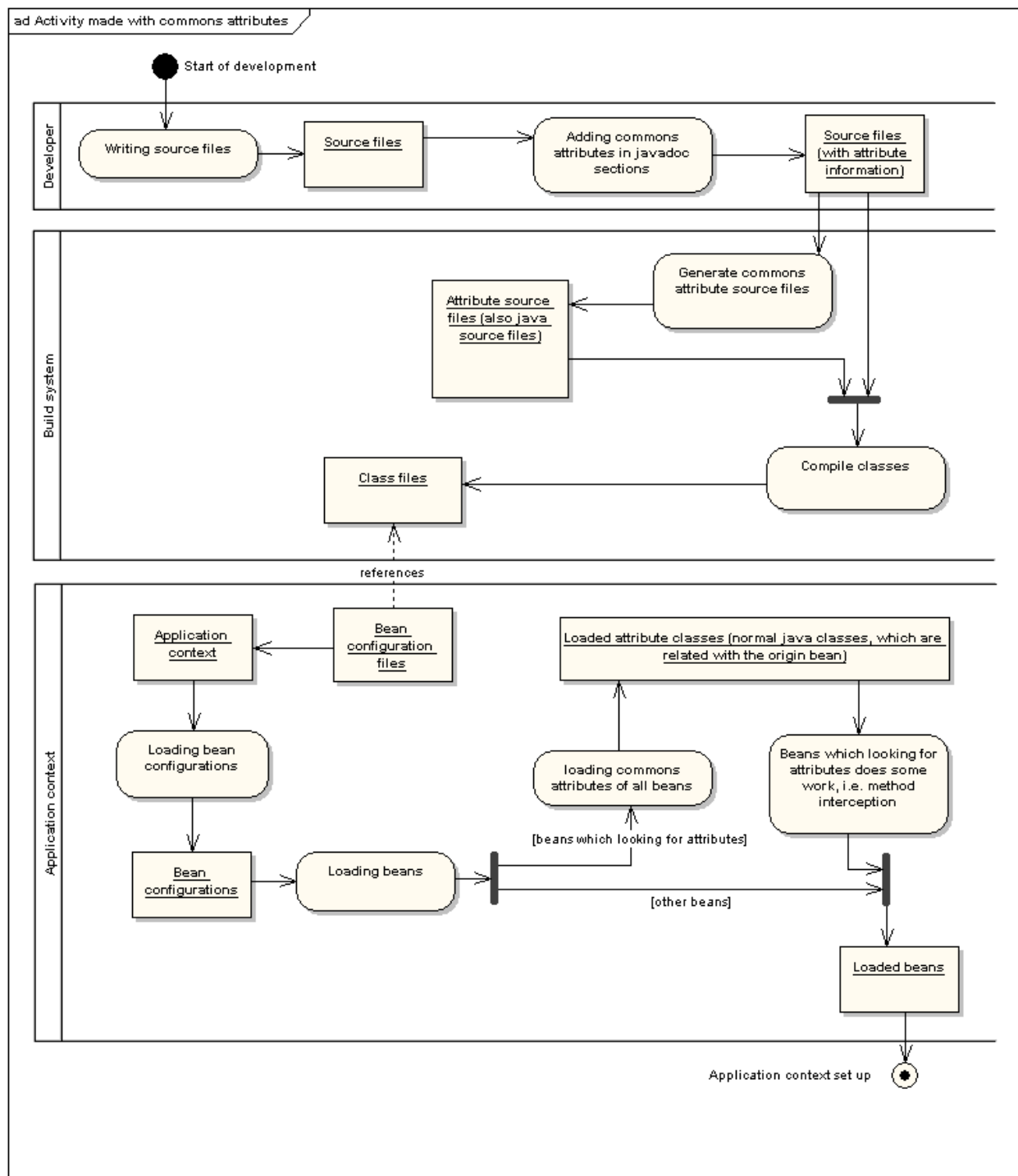
Using Commons Attributes in Java5 is deprecated. Please see the section above for a how to for transactions declared with Java5 annotations.

4.4.1 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.

4.4.2 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.

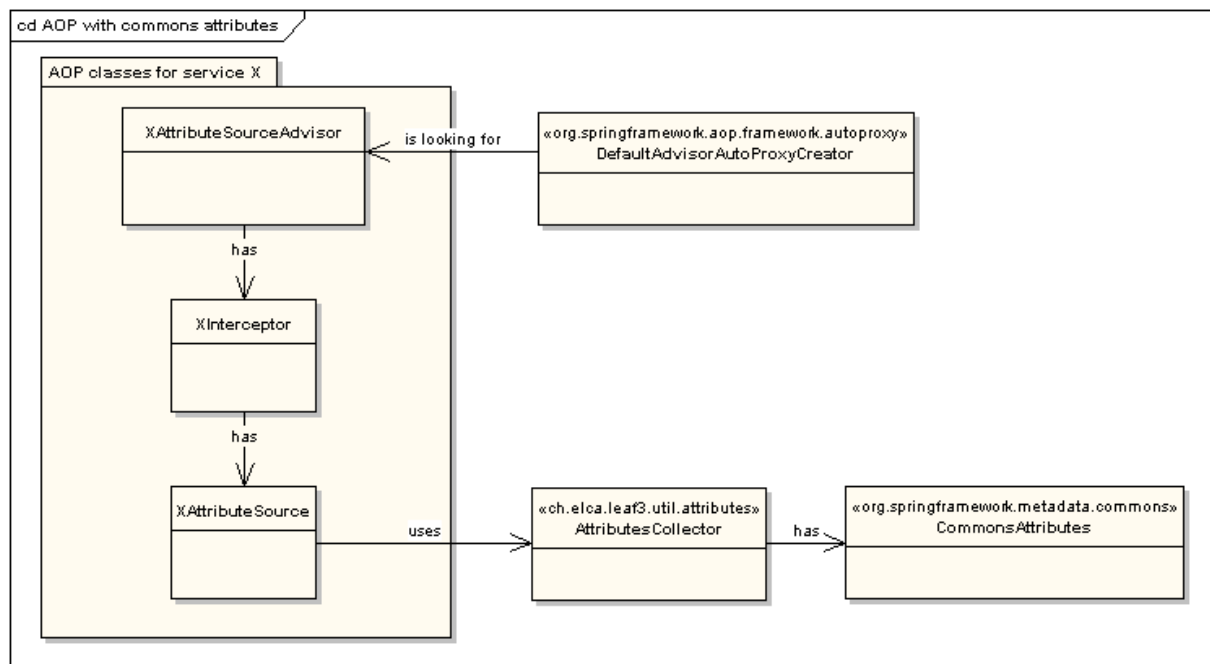
By using common attribute classes to access attributes of your classes you can do with attributes what you want. In many cases, attributes are used to enable method interceptors.

4.4.3 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!

4.4.3.1 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 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.

4.4.3.2 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](#).

4.4.3.3 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.

ATTENTION: If classes where Commons Attributes are defined will be proxied by **OTHER** autoproxy creators, not all Commons Attributes will be found anymore. If the proxy is made by using JDK proxies only attributes which are defined on interfaces can be found. Attributes defined on implementation classes are lost! If the target class will be proxied by using CGLIB **NO** attributes will be found anymore.

SOLUTION: Do not use other autoproxy creators, than the one for transaction manager on beans where Commons Attributes are defined. Apply these interceptors by wrapping each target bean in configuration with a `ProxyFactoryBean`.

4.4.3.4 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.

4.4.3.5 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.***

4.4.3.6 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="transactionAutoproxy"
        class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator">
        <property name="proxyTargetClass">
            <!--
                false:
                Jdk proxies will be used.
                Only interface methods will be intercepted.
            -->
        </property>
    </bean>
</beans>
```

```

        true:
            Cglib will be used to intercept methods.
            Intercepted beans must have a default constructor and
            must not be final.
        -->
        <value>false</value>
    </property>
</bean>

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

4.4.3.7 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.*).

4.4.3.8 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()`.

4.4.4 Internal design

These classes just extend the corresponding attribute classes from spring.

4.5 Meta data support

Pay attention: Module Description is not yet finished.

4.5.1 Purpose

Meta data support is the follower of the module *Attribute Convenience*. Meta data is used to describe data; for example in Java can meta data define that a method is deprecated with the annotation `@Deprecated`.

EL4J supports the use of meta data for AOP aspects. So defines a meta data a pointcut for an advice. At the moment are supported attributes and java annotations. The meta data collection structure is constructed, that in future other meta data types can be supported by defined a new collector implementation.

Benefits of using this module:

- Java Annotations are usable with the concept of interceptors, which is known from the displaced module *Attribute Convenience*.
- Spring AOP supports the use of Java Annotations only on methods. EL4J supports also the use of annotations on classes.
- The module *meta data support* supports inheritance of meta data.
- In spring, 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 module to alleviate this.

4.5.2 Configuration meta data type Java Annotation

The use of Java Annotations is the default case. Therefore just the `GenericMetadataAdvisor` has to be configured.

TODO / Inheritance Configuration

Here's a sample 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">

    <!-- Defines 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="genericMetadataAdvisor"
        class="ch.elca.el4j.util.metadata.GenericMetadataAdvisor">
        <property name="interceptor">
            <ref local="exampleInterceptor"/>
        </property>
        <property name="interceptingMetadata">
            <list>
                <value>ch.elca.el4j.tests.util.metadata.annotations.helper.ExampleAnnotationOne</value>
            </list>
        </property>
    </bean>

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

    <!-- Define the bean which owns a method that should be intercepted. -->
    <bean id="foo"
        class="ch.elca.el4j.tests.util.metadata.attributes.helper.FooImpl"/>

</beans>
```

- The `autoproxy` bean looks for Advisors.
- The `genericMetadataAdvisor` bean extends the `org.springframework.aop.support.DefaultPointcutAdvisor`.
 - ◆ The Advice, e.g. a `MethodInterceptor` can be injected via the property `interceptor`. It is necessary to define one, otherwise, an exception is thrown.
 - ◆ The property `interceptingMetadata` takes a list of meta data. The defined interceptor will be invoked if one of these meta data is defined at a method/class. If the parameter is not set, all meta data defined at a method/class are collected.
- The `exampleInterceptor` bean extends `org.aopalliance.intercept.MethodInterceptor`. It also implements `ch.elca.el4j.util.metadata.MetadataCollectorAware` which sets the `metadataCollector` of this `Interceptor` since the `Interceptor` needs to access the meta data.
- 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`.

4.5.3 Configuration meta data type Common Attributes

Here's a sample 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">

<!-- Defines 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="genericMetadataAdvisor"
      class="ch.elca.el4j.util.metadata.GenericMetadataAdvisor">
  <property name="interceptor">
    <ref local="exampleInterceptor"/>
  </property>
  <property name="interceptingMetadata">
    <list>
      <value>ch.elca.el4j.tests.util.metadata.attributes.helper.ExampleAttributeOne</value>
    </list>
  </property>
  <property name="metadataCollector">
    <ref local="attributesCollector" />
  </property>
</bean>

<!-- Define the collector for common attributes -->
<bean id="attributesCollector"
      class="ch.elca.el4j.util.metadata.attributes.DefaultGenericAttributeCollector" />

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

<!-- Define the bean which owns a method that should be intercepted. -->
<bean id="foo"
      class="ch.elca.el4j.tests.util.metadata.attributes.helper.FooImpl"/>

</beans>
```

Different to the configuration to collect Java Annotations are:

- A bean of `ch.elca.el4j.util.metadata.attributes.DefaultGenericAttributeCollector` has to be defined. The `genericMetadataAdvisor` takes it in the property `metadataCollector`.

4.5.4 Implementation of the Interceptor

The implementation of the interceptor is not different between the Common Attributes and Java Annotations. To have access to the meta data collector, the interceptor (specified in the configuration) has to implement the interface `ch.elca.el4j.util.metadata.MetadataCollectorAware`. The collector defines the following methods to get the meta data on the specified target.

TODO | List all methods

These methods returns a *Collection* of the found meta data or *null* if no meta data was found.

```
/**
 * If a method containing the meta data ExampleMetadata and the parameters
 * are from typ int, check that there value is not higher as defined in ExampleMetadata.
 * If the argument/s is/are higher, the method will proceeded with the value defined in
 * ExampleMetadata.
 */
public Object invoke(MethodInvocation methodInvocation) throws Throwable {

    int[] param = null;
```

```

// Get meta data from the intercepted method
Collection metaData = m_metaDataCollector.getMethodOperatingMetaData(methodInvocation);

// Proceed meta data for the method specified (cf. Javadoc)
if (metaData != null & metaData.size() > 0) {

    // Set the new arguments of the intercepted methods if
    // they are of type int.
    try {
        param = methodInvocation.getArguments();

        for (Iterator iter = collection.iterator(); iter.hasNext();) {
            Object element = (Object) iter.next();

            if (element instanceof ExampleMetaData) {
                int value = element.value();
                for (int i=0; i < param.length; i++) {
                    if (param[i] > value) { param[i] = value; }
                }
            }
        }
    } catch (Exception e) {
        //Do nothing with the arguments; just proceed the method
    }
}

// Proceed intercepted method and return its result
Object retVal = null;
try {
    // Execute the intercepted method
    retVal = methodInvocation.proceed();
} catch (Throwable ex) {
    throw ex;
}

return retVal;
}

/**
 * {@inheritDoc}
 */
public void setMetaDataSource(GenericMetaDataSource metaDataSource) {
    m_metaDataCollector = metaDataSource;
}

```

4.5.5 Integration of existing which uses the displaced module *Attribute Convenience*

If already code exists which using the displaced module *AttributeConvenience* `ch.elca.el4j.util.attributes`, the following changes are necessary:

Configuration (cf. chapter Configuration meta data type Common Attributes for a sample configuration)

- The `GenericAttributeAdvisor` is no longer available. Use instead `ch.elca.el4j.util.metadata.GenericMetaDataAdapter`.
 - ◆ The interceptor is no longer settable with the constructor. Use the property `interceptor` to set the interceptor.
 - ◆ The property `interceptingAttributes` has been changed to `interceptingMetaDataAdapter`. Change this property name. The property takes still the same data type.
- By default, the `DefaultGenericAnnotationCollector` is used to collect the meta data. Therefore a bean of a

`ch.elca.el4j.util.metadata.attributes.DefaultGenericAttributeCollector` has to be configured. This bean has to be set in the property `metaDataCollector` of the `genericMetaDataAdvisor`.

Interceptor

- The interceptor has to implement the `ch.elca.el4j.util.metadata.MetadataCollectorAware` instead of the `AttributeSourceAware`.
- The `GenericMetadataCollector` set by the `MetadataCollectorAware` supports no longer the Method `getAttribute(Method, Class):Object` to get the meta data. Use an appropriate method from the new interface, for example the method `getMethodOperatingMetaData(MethodInvocation):Collection`.

4.5.6 Working of the inheritance

Sometimes it is useful to inherit a meta data to the childs. In other cases, inheritance is not wished because the clearance of the code decreases. Therefore in el4j it is configurable if, and how deep meta data will be inherited to the childs. The inheritance can be configured in the following steps.

`includePackages` meta data on packages will be inherited to all classes, interfaces and its methods in the corresponding package and all its subpackages. → Not yet implemented

`includeInterfaces = true`; meta data on interfaces will be inherited to all classes which implements the interface. The inheritance goes on to all subclasses of these classes. Example: Class A implements Interface One. Class B extends Class A. So inherit Class B the meta data from Interface One.

`includeSuperclasses = false`; the superclasses inherit its meta data to all its childs and its methods.

`includeClass = true`; the class inherits its meta data to its methods.

If nothing will be configured, the following default configuration is used: `includePackages = false`; `includeSuperclasses = false`; `includeInterfaces = true`; `includeClass = true`;

**Note:* All inheritance will only made, if the meta data type allows it (e.g. java annotations can be specific to use only on specific targets, for example only on methods).*

**Hint:* If inheritance is used, document it clearly! Otherwise the clearance of the code can decrease strongly.*

Overriding

Child meta data overrides parent meta data.

Example A class uses the annotation `@ExampleAnnotationOne("Class")` and its method uses `@ExampleAnnotationOne("Method")`. In this case a method interceptor got the value Method to proceed.

If all options of the inheritance are used, the following points has to be mentioned:

- Interface meta data are stronger than superclass meta data (cf. [ExampleAnnotationTwo](#)[?] in the example)...
- Method meta data are stronger than class, interface and superclass meta data. Also when the class, interface or superclass is in the hierarchie nearer than the meta data definition on the method (cf. [ExampleAnnotationTen](#)[?] in the example).

Example

→ Full configuration (everything true)

```

@ExampleAnnotationOne()
public interface Base {

    @ExampleAnnotationTen()
    public void inheritFromMethod(int input);

}

@ExampleAnnotationTwo()
public interface Foo extends Base{

    public void inheritFromClass(int input);

}

@ExampleAnnotationThree()
@ExampleAnnotationTen("Not stronger than ExampleAnnotationTen
on inheritFromMethod(int) in interface Base")
public interface FooBase {

    @ExampleAnnotationTwelve()
    public void overwriteAnnotations(int input);

}

@ExampleAnnotationSix()
@ExampleAnnotationTwo("Not stronger than ExampleAnnotationTwo
on interface Foo")
public abstract class AbstractFoo implements Foo {

@ExampleAnnotationFourteen()
public abstract void inheritFromMethod(int input);

}

@ExampleAnnotationEight()
public class FooImpl extends AbstractFoo implements FooBase {

@ExampleAnnotationSixteen()
public void inheritFromMethod(int input) {...}

public void inheritFromClass(int input) {...}

@ExampleAnnotationEight("Overwritten")
@ExampleAnnotationTwelve("Overwritten")
public void overwriteAnnotations(int input) {...}

}

```

method public void inheritFromClass(int input) inherit following annotations: – @ExampleAnnotationEight() – @ExampleAnnotationThree() – @ExampleAnnotationTen("Not stronger than [ExampleAnnotationTen](#)[?] on inheritFromMethod(int) in interface Base") – @ExampleAnnotationTwo() Same annotation type on Superclass [AbstractFoo](#)[?] is not inherited because the definition on the interface Foo is stronger. The definition is stronger, also if superclass [AbstractFoo](#)[?] is nearer in the hierarchy. – @ExampleAnnotationOne() – @ExampleAnnotationSix()

method public void inheritFromMethod(int input) inherit following annotations: – @ExampleAnnotationSixteen() – @ExampleAnnotationTen() Same annotation type on Interface [FooBase](#)[?] is not inherited because the definition on the method in interface Base is stronger. The definition is stronger, also if interface [FooBase](#)[?] is nearer in the hierarchy. – @ExampleAnnotationEight() – @ExampleAnnotationThree() – @ExampleAnnotationTen("Not stronger than [ExampleAnnotationTen](#)[?] on

inheritFromMethod(int) in interface Base”) – @ExampleAnnotationTwo() Same annotation type on Interface Superclass `AbstractFoo`? is not inherited because the definition on the interface Foo is stronger. The definition is stronger, also if superclass `AbstractFoo`? is nearer in the hierarchie. – @ExampleAnnotationOne() – @ExampleAnnotationSix()

method public void overwriteAnnotations(int input) inherit the same as inheritFromClass(int input) except: – @ExampleAnnotationEight(“Overwritten”) Annotation on Class is overwritten by the method. – @ExampleAnnotationTwelve(“Overwritten”) Annotation defined by the interface `FooBase`? is overwritten by the method in class `Foolmpl`?

4.5.7 Internal design

4.5.7.1 Classes *TODO*

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

- `GenericAttributeAdvisor`: All advisors are collected by the `autoproxy` 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.

4.5.7.2 Collection procedure

The `AbstractGenericMetaDataCollector` makes the following steps to collect the meta data on **method x** in **class a**.

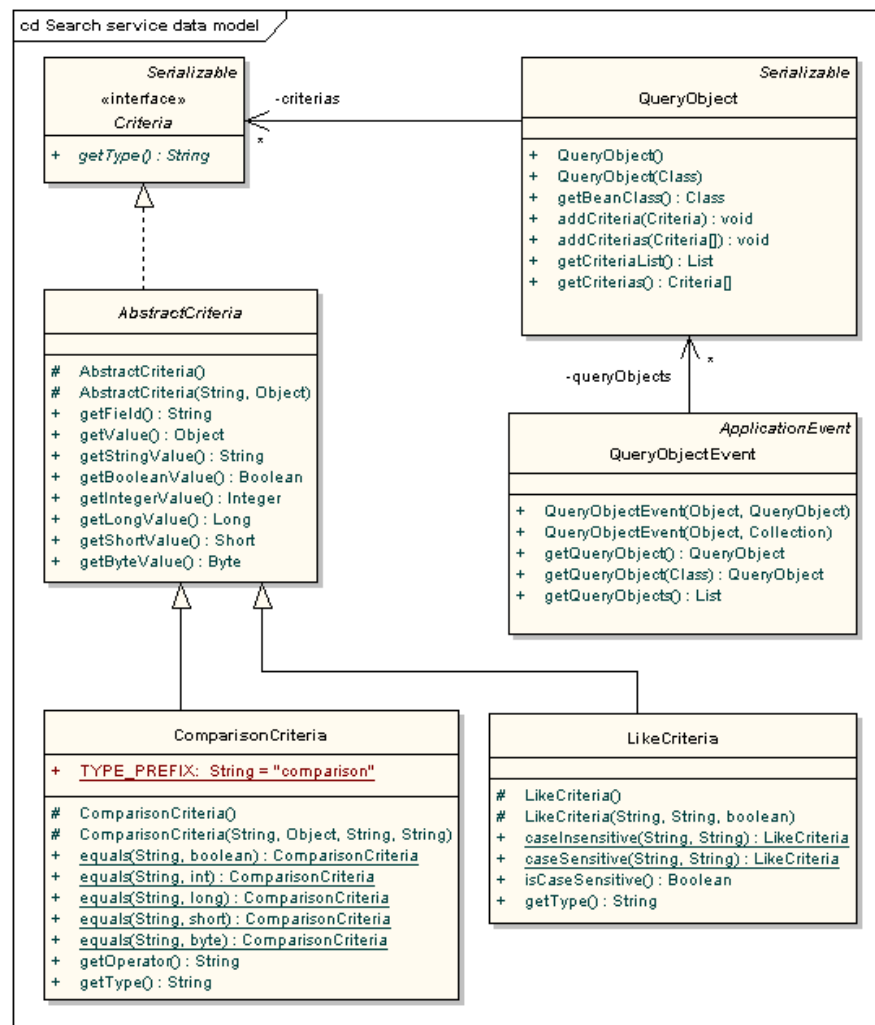
1. Gets meta data on **method x** in **class a**.
2. If option `includeInterfaces = true`: Gets meta data on **method x** in all interfaces which are implemented by **class a**.
3. If `includeInterfaces = true`: Gets meta data on **method x** in all interfaces which are implemented by superclasses of **class a**.
4. If `includeSuperclasses = true`: Gets meta data on **method x** in all superclasses of **class a**.
5. If `includeClasses = true`: Gets meta data on **class a**.
6. If `includeInterfaces = true`: Gets meta data on all interfaces which are implemented by **class a**.
7. If `includeInterfaces = true`: Gets meta data on all interfaces which are implemented by superclasses of **class a**.
8. If `includeSuperclasses = true`: Gets meta data on all superclasses of **class a**.
9. If `includePackages = true`: Gets meta data on package from **class a** and its parent packages.

The `AbstractGenericMetaDataCollector` makes the steps 6 to 9 to get the meta data on **class a**.

4.6 Search service

In the search service we have implemented the **Query Object** pattern of Martin Fowler. See <http://www.martinfowler.com/eaCatalog/queryObject.html> for a short introduction.

The idea is to create a query object on client-side and send this query object trough to the dao layer. There should be no need to change the query object in between these layers. With this approach you can add search conditions on client-side without modifying service interfaces.



In center we have the query object class. A query object normally belongs to one java bean, where the java bean is a dto like the reference dto of Reference–Database–Application (see [here](#)). In this dto we have nearby other properties property name, description and incomplete. Properties name and description are strings and property incomplete is a boolean.

A query object can have multiple criterias. Currently we have three criteria classes. The like criteria is made to do searches on strings with the SQL like syntax. The second criteria is the comparison criteria, used to compare values. Currently only equals compares are implemented. The third criteria is the include criteria, which is used to test if a given value is included in a given set.

```

ReferenceService service = ...
QueryObject query = new QueryObject(ReferenceDto.class);
query.addCriteria(LikeCriteria.caseInsensitive("name", "%JAVA%"));
query.addCriteria(LikeCriteria.caseInsensitive("description", "%WEB%"));
query.addCriteria(ComparisonCriteria.equals("incomplete", true));
query.addCriteria(new IncludeCriteria("keywords", kJava.getKeyAsObject()));
List list = service.searchReferences(query);
...

```

The code above shows the use of these three criteria objects combined with the reference dto. In this code we execute a search on reference dto's fields name, description, incomplete and keywords. The expected result is to receive all reference dtos with string java (case-insensitive) somewhere in property name, with string web (case-insensitive) somewhere in property description, where property incomplete is set to true and where the kjava keyword is included in the reference dto's keywords set. To get all references we could send an empty query object (without any criterias) to the reference service.

How the query object could be handled with IBatis you can have a look at [dao class](#) and [IBatis config files](#) of the Reference–Database–Application.

To see how the query object can be handled with Hibernate, you can have a look at the [dao class](#) of the Reference–Database–Application and the [CriteriaTransformer](#) class of the Hibernate module.

The query object event is used to wrap query objects. This event can be used with Spring's application event publisher. Most application contexts are such an application event publisher. Each singleton bean that implements interface `org.springframework.context.ApplicationListener` will receive this event. Prototype beans must be handled separately. For an example you can have a look at handling of views (prototype beans) in *module–springrcp*.

4.7 Additional Features

4.7.1 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://el4j.sourceforge.net/javadoc/framework-modules/ch/elca/el4j/core/config/ListPropertyMergeConfigurer.html>

Further the list property merge configurer has the possibility to add the new values before or after the existing values. By default the new values (from property file) will be appended. To prepend the new values you have to set following property in configurer bean:

```
<property name="insertNewItemsBefore" value="true"/>
```

4.7.2 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://el4j.sourceforge.net/javadoc/framework-modules/ch/elca/el4j/core/beans/BeanLocator.html>

4.7.3 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.

4.7.4 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>
```

4.7.5 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://el4j.sourceforge.net/javadoc/framework-modules/ch/elca/el4j/util/codingsupport/AbstractDefaultEnum.html>
- <http://el4j.sourceforge.net/javadoc/framework-modules/ch/elca/el4j/util/codingsupport/AbstractComparableEnum.html>

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

4.7.6 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://el4j.sourceforge.net/javadoc/framework-modules/ch/elca/el4j/util/codingsupport/Reject.html>

4.7.7 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`).

4.7.8 Generic repository

The `ch.elca.el4j.services.persistence.generic.dao.GenericRepository` interface serves as generic access to storage repositories. It is the interface for the DDD–Book's Repository pattern. The repository pattern is similar to the DAO pattern, but a bit more generic. This interface can be implemented in a generic way and can be extended in case a user needs more specific methods. It is based on an idea from the [Hibernate website](#). A more detailed description, illustrating how this interface can be used, can be found [here](#).

4.7.9 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.

4.7.10 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].

4.7.11 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`.

4.8 Packages that implement the core module

- `ch.elca.el4j.core.**`
- `ch.elca.el4j.services.persistence.generic.**`
- `ch.elca.el4j.services.monitoring.notification.CoreNotificationHelper`
- `ch.elca.el4j.services.search.**`
- `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](#).

5 Documentation for module web

5.1 Purpose

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

5.2 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 `OldWebApplicationTemplate`.
- Xml Tag Transformer. Escapes xml tags in order to display them properly on web pages.

5.3 How to use

5.3.1 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).

5.4 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.

5.4.1 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.

5.4.1.1 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>
```

5.4.1.2 ModuleContextLoader

There is also the possibility to declaratively configure and start up the `ModuleWebApplicationContext` via servlet context parameters in the `web.xml` file. The `ch.elca.el4j.web.context.ModuleContextLoader` class provides this capability. This context loader is created by the `ch.elca.el4j.web.context.ModuleContextLoaderListener` class.

You have to configure the `ModuleContextLoaderListener` in the `web.xml` file as follows:

```
<listener>
  <listener-class>
    ch.elca.el4j.web.context.ModuleContextLoaderListener
  </listener-class>
</listener>
```

When the web application starts up, this listener will execute the `ModuleContextLoader`, which initializes and starts a `ModuleWebApplicationContext` based on one or more servlet context parameters that are merged. You can specify the following context parameters:

- **inclusiveLocations** (mandatory): specifies the configuration locations which will be included in the application context
- **exclusiveLocations** (optional, defaults to null): specifies the configuration locations which will be excluded from the application context
- **overrideBeanDefinitions** (optional, defaults to false): indicates whether bean definition overriding is allowed in the application context
- **mergeResources** (optional, defaults to true): indicates whether the resources retrieved by the configuration files section of the manifest files should be merged with resources found by searching in the file system

A typical configuration example could look like this:

```
<context-param>
  <param-name>inclusiveLocations</param-name>
  <param-value>
    classpath*:mandatory/*.xml,
    classpath*:scenarios/db/raw/*.xml,
    classpath*:scenarios/dataaccess/hibernate/*.xml,
    classpath*:optional/interception/transactionJava5Annotations.xml
  </param-value>
</context-param>

<context-param>
  <param-name>exclusiveLocations</param-name>
  <param-value>
    classpath*:scenarios/dataaccess/hibernate/keyword-core-repository-hibernate-config.xml
  </param-value>
</context-param>
```

5.4.2 Build system integration

We provide a hook task for the Maven 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**.

TBD: correct the above link to the resource plugin to the new maven concept for this. Check also if the content below is still valid.

5.4.2.1 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.

TBD: how is this done with the Maven plugin?

Please check the javadoc of the **ModuleApplicationContexts**: there are some new features to control the loading of classpath resources.

5.4.3 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.
- 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.

5.4.4 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
```

5.4.5 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!/
```

5.4.6 Resources

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

6 Documentation for module remoting

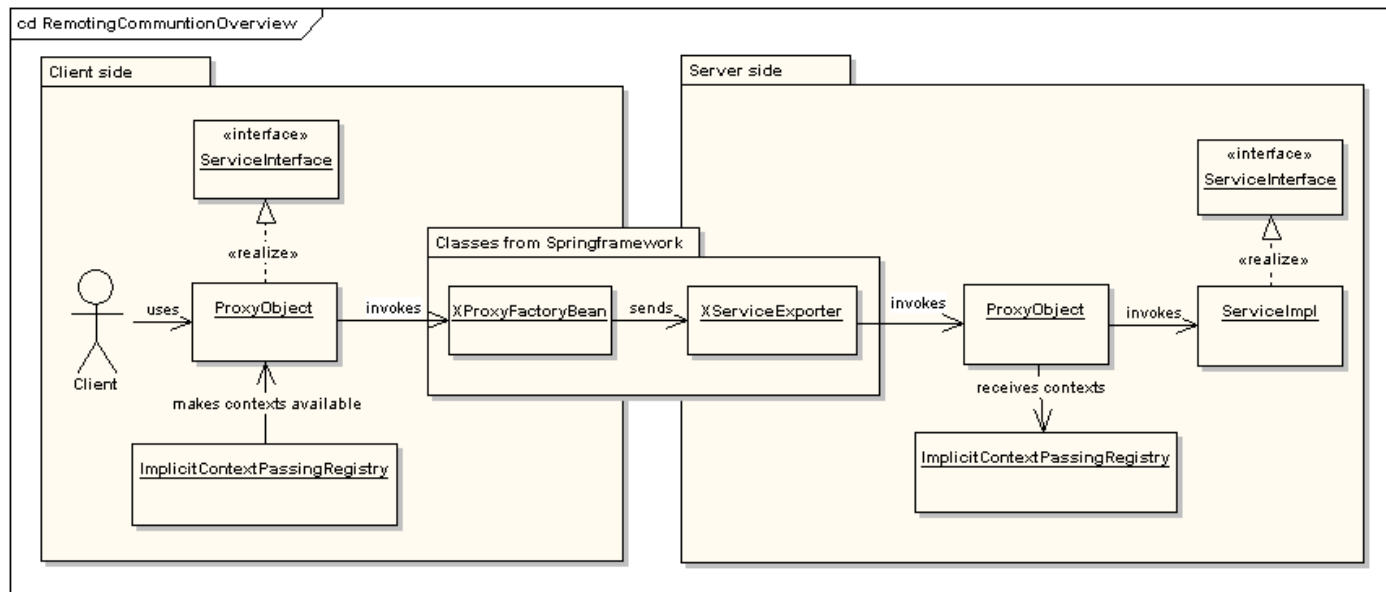
6.1 Purpose

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

6.2 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!

6.3 How to use

6.3.1 Remoting modules

Currently there are four modules for remoting:

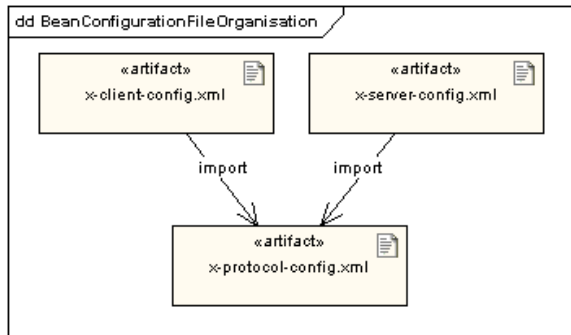
- The core remoting module with name **module-remoting_core** contains only protocol RMI.
- For Hessian and Burlap you have to use **module-remoting_caucho**.
- The Soap protocol can be found in **module-remoting_soap**.
- For the EJB remoting protocol you have to use **module-remoting_ejb**.

In addition, there exists a composite protocol which supports load balancing:

- The load balancing protocol can be found in **module-remoting_core**.

6.3.2 Configuration

6.3.2.1 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`.

6.3.2.1.1 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.

6.3.2.1.2 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.

6.3.2.1.3 x-server-config.xml

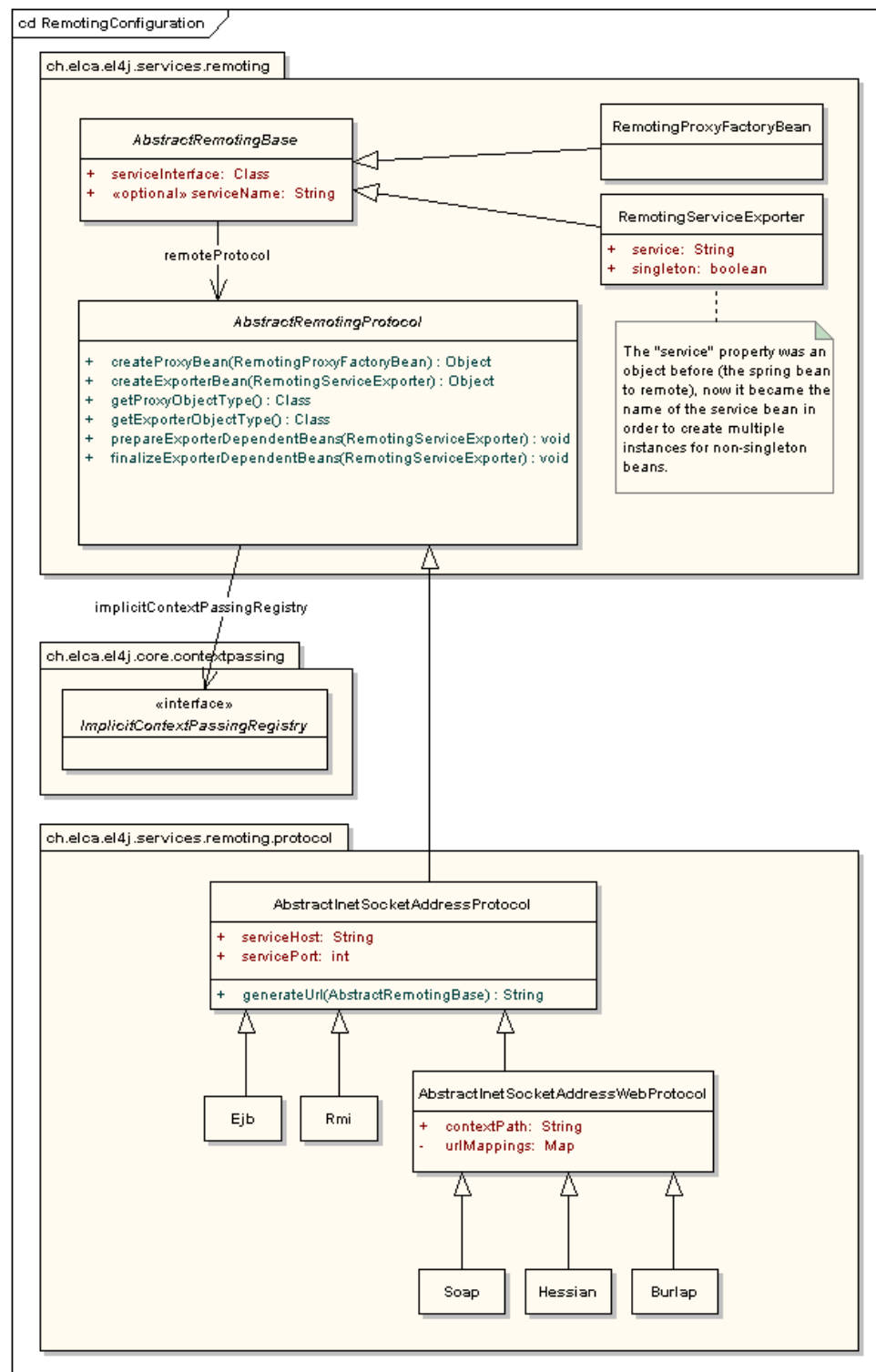
This file contains the following for the protocol `rmi`:

```
<?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">
      <idref bean="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.

6.3.2.2 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.

6.3.2.3 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.

6.3.2.4 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.

6.3.2.4.1 Create a web-deployable module

With Maven 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:

TBD: adapt the following to Maven

```
<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-caucho` in this module.

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

6.3.2.4.2 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`.

6.3.2.4.3 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.

6.3.2.4.4 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.

6.3.2.4.5 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.

6.3.2.4.6 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.

6.3.2.5 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`.

6.3.2.6 How to use the HttpInvoker protocol

The usage of the `HttpInvoker` protocol is exactly the same as for the `Hessian` and `Burlap` protocols. Just read the `Hessian` subchapter and replace the word `Hessian` with `HttpInvoker`. One additional change is required: Replace the full qualified path name of the `HttpInvokerServiceExporter` to `"org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter"`.

6.3.2.7 How to use the soap protocol (XFire)

The usage of the `XFire` protocol is similar to the `Hessian` protocol. The `XFire` protocol requires two additional properties to be set. These are the properties `"xfire"` and `"serviceFactory"`. The property `"xfire"` is always assigned a reference to the bean `"xfire"`. By setting the `"serviceFactory"` property one can choose, which service factory should be used for creating the service.

An common configuration of the protocol could look as follows:

```
<beans>
  <import resource="classpath:org/codehaus/xfire/spring/xfire.xml"/>
```

```

<bean id="xFireProtocol" class="ch.elca.el4j.services.remoting.protocol.XFire">
  <property name="serviceHost">
    <value>${j2ee-web.host}</value>
  </property>
  <property name="servicePort">
    <value>${j2ee-web.port}</value>
  </property>
  <property name="contextPath">
    <value>module-remoting-demos-web</value>
  </property>
  <property name="implicitContextPassingRegistry">
    <ref local="implicitContextPassingRegistry" />
  </property>
  <property name="xfire">
    <ref bean="xfire" />
  </property>
  <property name="serviceFactory">
    <ref bean="xfire.serviceFactory" />
  </property>
</bean>
</beans>

```

On the server side, a web.xml and xfire-servlet.xml file should be setup. Example web.xml file:

```

<?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 http:
version="2.4">
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      classpath:org/codehaus/xfire/spring/xfire.xml
    </param-value>
  </context-param>
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
  <servlet>
    <servlet-name>xfire</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>xfire</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>

```

Example xfire-servlet.xml file:

```

<?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*:mandatory/*.xml"/>
  <import resource="classpath:common/protocols-config.xml"/>
  <import resource="classpath:server/web/benchmark-xfire-server-config.xml"/>
</beans>

```

Important Remarks:

- The "org.codehaus.xfire.jaxb2.JaxbServiceFactory" provided by codehaus inherits from the [AnnotationServiceFactory](#)[?]. Because of this service classes (interfaces + implementation) are required to have [WebAnnotations](#)[?] on them. For this reason a new JAXB service factory was implemented, which does not require annotated classes: "ch.elca.el4j.services.remoting.protocol.xfire.JaxbServiceFactoryWithoutWebAnnotations".
- The current xfire-remoting implementation has problems using JAXB.

- The current xfire-remoting implementation cannot transport exceptions from the server side to the client properly (they are converted into [XFireFaults](#)[?]).

6.3.2.8 How to use the soap protocol (Axis)

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.

6.3.2.8.1 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_soap*** we use [Axis](#).

6.3.2.8.2 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.

6.3.2.8.3 Soap style and usage

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

6.3.2.8.4 Server side setup

6.3.2.8.4.1 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.

6.3.2.8.4.2 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.

6.3.2.8.4.3 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="soapExceptionHandler">
    <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.BaseRTEException</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 `soapExceptionHandler` 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.SoapExceptionHandler`.

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.

- **soapExceptionHandler**

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.

6.3.2.8.4.4 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.

6.3.2.8.5 Client side setup

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

6.3.2.8.5.1 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.

6.3.2.8.6 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>

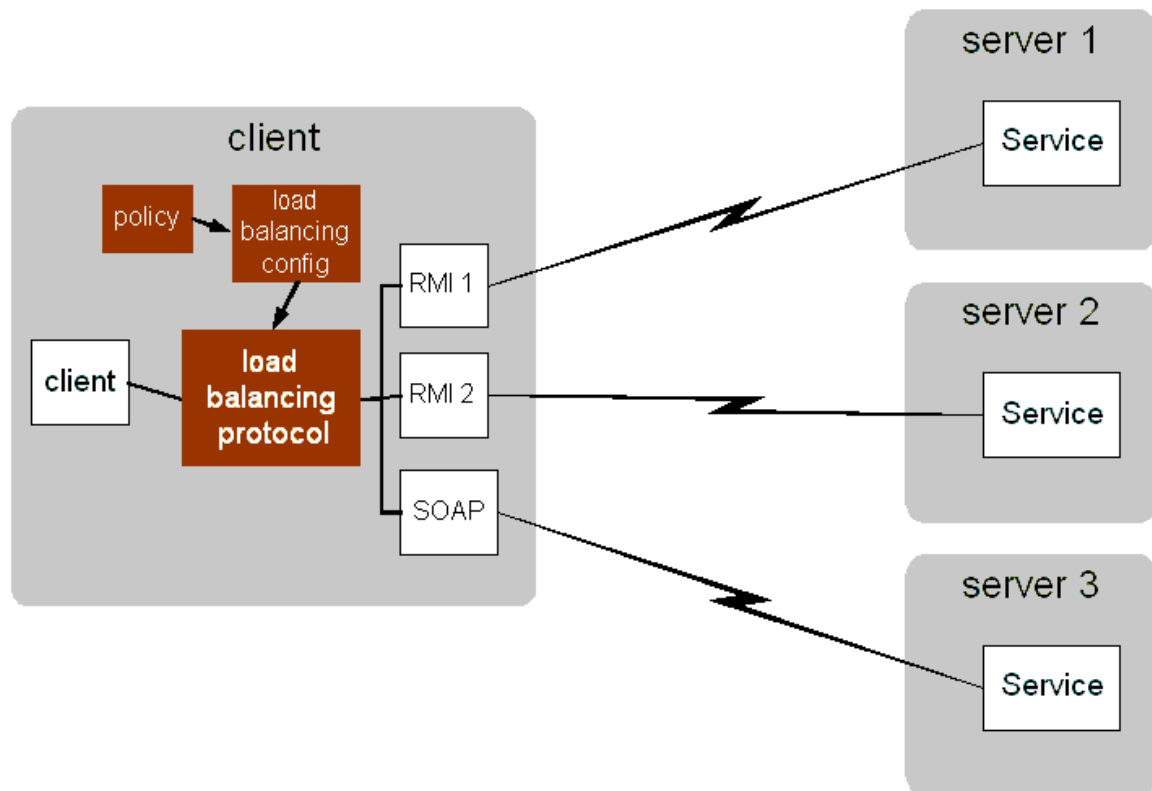
6.3.2.9 How to use the EJB protocol

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

6.3.2.10 How to use the Load Balancing composite protocol

The load balancing protocol is a so-called composite protocol. It applies the Composite Design Pattern and thus allows the user to compose several of the atomic protocols (i.e., a non-composite protocol such as RMI) into this composite protocol. To the outside, it behaves like an atomic protocol. Note that the load balancing protocol is only used on the client side of a (remote) invocation and requires no modifications to existing remoting protocols.

The following figure shows an overview of the load balancing protocol usage. In this example, load balancing composes three (atomic) protocols, however, any number of protocols are supported. Components in red (or in dark color) are part of load balancing, the others are part of other remoting protocols or business objects.



The bean class *LoadBalancingConfiguration* groups the configuration parameters that are supported by load balancing. As an example, consider the following configuration, which defines the client side of the invocation. It balances load between 3 RMI-servers.

```

<?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="businessObj"
    class="ch.elca.el4j.services.remoting.RemotingProxyFactoryBean">
    <property name="remoteProtocol">
      <ref bean="loadBalancingProtocol" />
    </property>
    <property name="serviceInterface">
      <value>
        myServiceInterface
      </value>
    </property>
  </bean>

  <bean id="loadBalancingProtocol" class="ch.elca.el4j.services.remoting.protocol.loadbalancing.protocol"
    <property name="protocolSpecificConfiguration">
      <ref bean="loadBalancingProtocolConfiguration" />
    </property>
  </bean>

  <bean id="loadBalancingProtocolConfiguration"
    class="ch.elca.el4j.services.remoting.protocol.loadbalancing.protocol.LoadBalancingProtocolConfiguration">
    <property name="protocols">
      <list>
        <ref bean="rmiProtocol1"/>
        <ref bean="rmiProtocol2"/>
        <ref bean="rmiProtocol3"/>
      </list>
    </property>
    <property name="policy">
  
```

```

        <ref bean="randomPolicy" />
    </property>
</bean>

<bean id="rmiProtocol1" class="ch.elca.el4j.services.remoting.protocol.Rmi">
    <property name="serviceHost">
        <value>localhost</value>
    </property>
    <property name="servicePort">
        <value>8092</value>
    </property>
</bean>

<bean id="rmiProtocol2" class="ch.elca.el4j.services.remoting.protocol.Rmi">
    <property name="serviceHost">
        <value>localhost</value>
    </property>
    <property name="servicePort">
        <value>8094</value>
    </property>
</bean>

<bean id="rmiProtocol3" class="ch.elca.el4j.services.remoting.protocol.Rmi">
    <property name="serviceHost">
        <value>localhost</value>
    </property>
    <property name="servicePort">
        <value>8099</value>
    </property>
</bean>
</beans>

```

Although nested load balancing protocols are possible, their usage is discouraged.

6.3.2.10.1 Handling Connection Failures

The load balancing protocol attempts to establish an initial connection to a particular server. If this connection attempt fails, it will ask for the next server from the policy bean and attempt to connect to this server. It repeats this behavior until it succeeds to connect, or no more servers are available. In the latter case, it throws a `ch.elca.el4j.services.remoting.protocol.loadbalancing.NoProtocolAvailableRTException`.

Once a protocol has been initialized and a connection established, it behaves as it would without load balancing. Thus, connection failures are notified to the user.

Retries after a failure: this load-balancing meta-protocol does *not* do automatic retries after a connection failure. In case you would like to have retries, have a look e.g. at the auto-idempotency module that has a retry-interceptor for this purpose. The semantics of the load-balancing meta-protocol is similar to the one of the jboss clustering support.

6.3.2.10.2 Policies

The load balancing protocol comes with a set of predefined policies. These policies govern the sequence in which protocol instances are invoked. Before every method invocation, the load balancing protocol retrieves the next protocol instance to invoke from the installed policy instance.

To minimize overhead, the load balancing protocol caches protocol instances and reuses these cached instances rather than recreating them every time.

Assume that p_i denotes policy instance p_i and that load balancing composes the protocol set $\{p_1, p_2, p_3\}$. For instance, p_i could denote the protocol RMI connecting to server running on `xyz.elca.ch:7000`. The following policies are currently supported:

- *random*: each new call goes to a randomly found protocol instance
- *roundrobin*: p_1 -> p_2 -> p_3 -> p_1 -> p_2 -> ...
- *redirectuponfailure*: p_1 -> p_1 -> p_1 ----"p1 fails"----> p_2 -> p_2 -> ...

The *random* policy removes protocols when a connection failure occurs. The *roundrobin* and *redirectuponfailure* policies do not exclude protocols that cause a connection failure, but switch to the next protocol. This behavior is well suited to handle transient network failures. With a transient failure, the server is still up and running and there is no reason not to reconnect to this server again at a later point in time. With the *random* policy, such servers are excluded. Indeed, with random policy, the load balancing protocol may (with low probability) repeatedly try to connect to the same, temporarily unavailable, server. Thus, these "failed" servers need to be excluded. Consequently, an unstable network may lead to the case in which servers are no longer considered although they may be up and running. It is the application developers responsibility to pick a policy suitable to his/her application, or to plugin his/her own policy.

The installation of an appropriate policy can be done using attribute `policy`. The default policy is *random*.

Defining a customary policy

If the need arises applications can install their own policies to work with load balancing. All policies must extend class

```
ch.elca.el4j.services.remoting.protocol.loadbalancing.protocol.policy.AbstractPolicy.
```

The policy implementation receives a notification every time a failure occurs with a particular atomic protocol.

6.3.2.10.3 Limitations

Currently, the load balancing plugin has only been tested with the RMI protocol. Although the tests with other protocols have not yet been performed, there is no reason it should not work with other protocols. Indeed, the load balancing protocol makes no assumption on the protocols other than the ones used also by the instantiating factory.

6.3.2.10.4 Further reading

Please see the load balancing test cases in module-remoting-tests-apps for further examples of how to use the load balancing protocol. Also, please refer to the documentation of the corresponding atomic protocols to learn how to use these.

6.3.2.11 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.

6.3.3 Benchmark

The module ***module-remoting-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 3.0 GHz processor and 1 GB of memory:

```
Please wait, benchmarks are running...
Benchmark 1 of 9 with name 'rmiWithoutContextCalculator' is running... done.
Benchmark 2 of 9 with name 'rmiWithContextCalculator' is running... done.
Benchmark 3 of 9 with name 'hessianWithoutContextCalculator' is running... done.
Benchmark 4 of 9 with name 'hessianWithContextCalculator' is running... done.
Benchmark 5 of 9 with name 'burlapWithoutContextCalculator' is running... done.
Benchmark 6 of 9 with name 'burlapWithContextCalculator' is running... done.
Benchmark 7 of 9 with name 'soapWithContextCalculator' is running... done.
Benchmark 8 of 9 with name 'httpInvokerWithoutContextCalculator' is running... done.
Benchmark 9 of 9 with name 'httpInvokerWithContextCalculator' is running... done.
```

Name of test	*Method 1 [ms]*	*Method 2 [ms]*	*Method 3 [ms]*
rmiWithoutContextCalculator	1.57	1.72	5.94
rmiWithContextCalculator	1.1	1.87	3.91

hessianWithoutContextCalculator	0.63	11.57	18.78	
hessianWithContextCalculator	0.63	13.14	18.3	
burlapWithoutContextCalculator	0.62	1.1	17.68	
burlapWithContextCalculator	0.78	1.09	17.52	
soapWithContextCalculator	9.7	17.2	27.54	
httpInvokerWithoutContextCalculator	2.66	3.13	4.69	
httpInvokerWithContextCalculator	2.04	2.66	5.63	

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.

6.3.4 Remoting semantics/ Quality of service of the remoting

6.3.4.1 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

6.3.4.2 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!

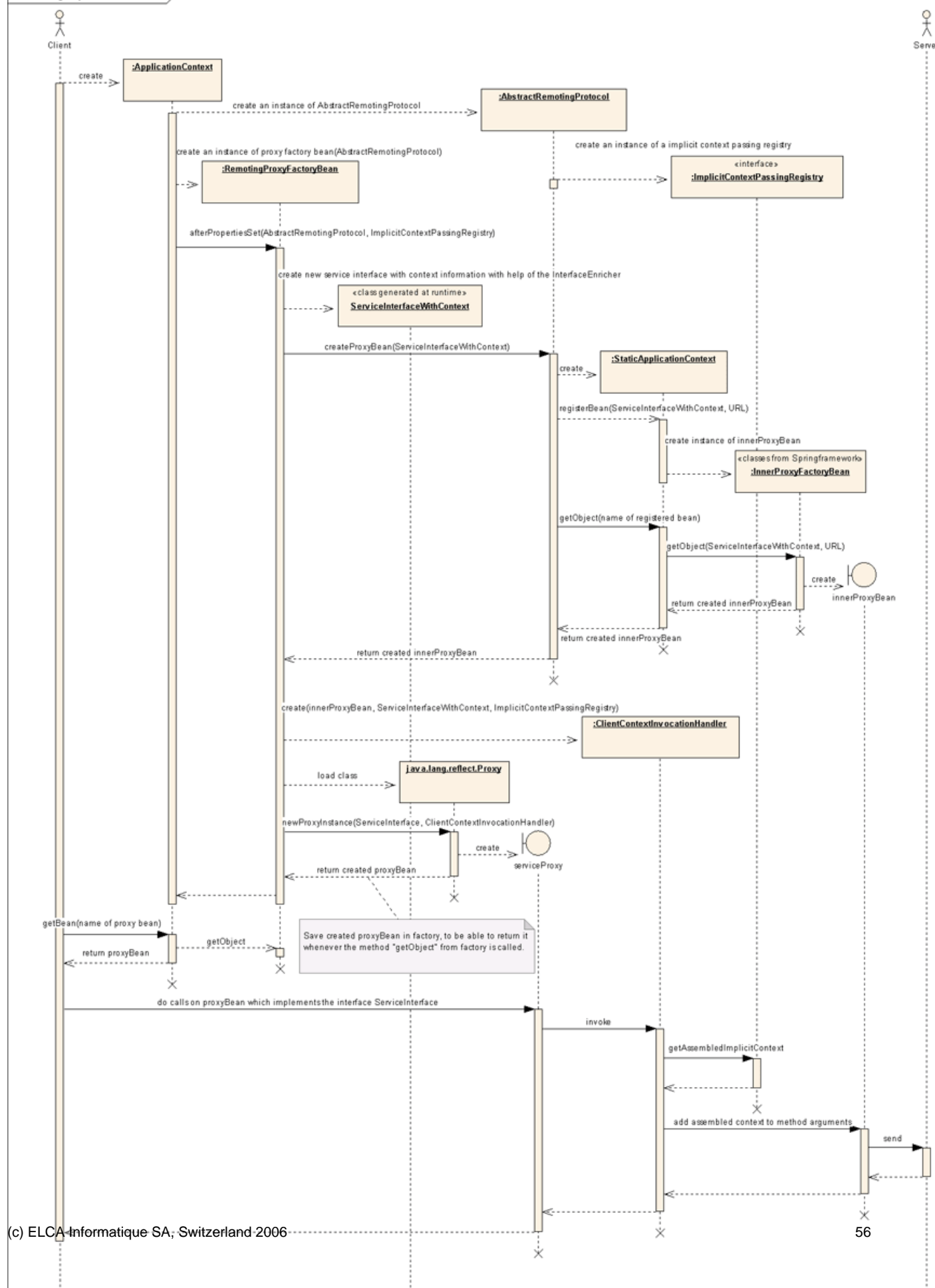
[RemoteServiceBehaviour_10.doc](#).

6.4 Internal design

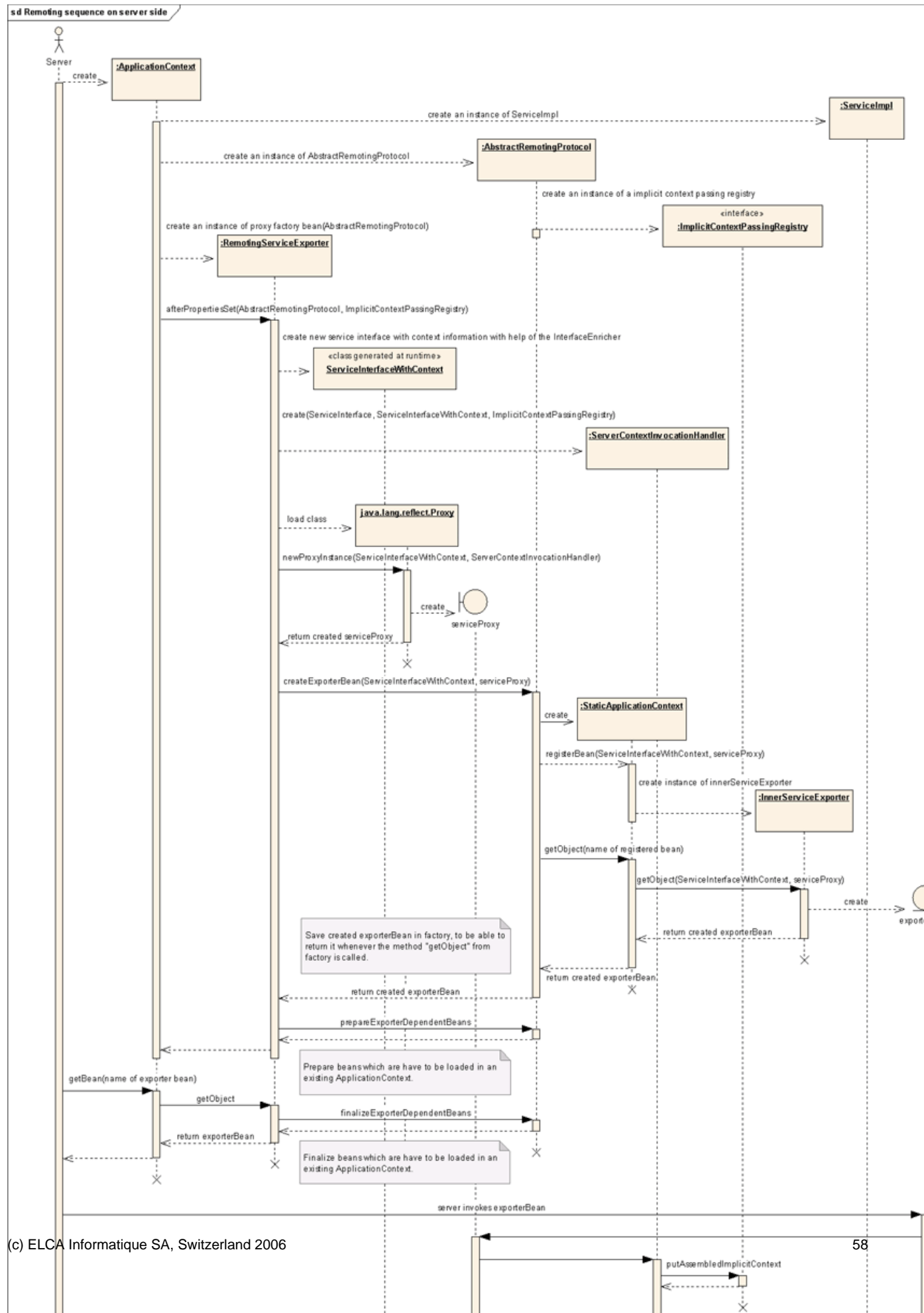
6.4.1 Sequences

6.4.1.1 Sequence diagramm from client side

sd Remoting sequence on client side



6.4.1.2 Sequence diagramm from server side



6.4.2 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.

6.4.3 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)

6.4.4 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.

6.5 Related frameworks

6.5.1 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

6.5.2 Javaworld 2005 idea

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

7 Documentation for module EJB remoting

7.1 Purpose

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

7.2 Important concepts

Remark: The EJB support is currently not working in EL4J 1.1 (due to the new build system). We will fix it as soon as possible. The documentation still is EL4Ant-specific.

EL4J provides its own remoting services infrastructure (**ModuleRemoting**) 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.

7.3 How to use

7.3.1 Configuration

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

7.3.1.1 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.

7.3.1.1.1 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.

7.3.1.1.2 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).

7.3.1.1.3 Method Mappings

EJB method name	property name	additional info / constraints
<code>ejbActivate()</code>	<code>activate</code>	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.
<code>ejbPassivate()</code>	<code>passivate</code>	
<code>ejbRemove()</code>	<code>remove</code>	
<code>setSessionContext(SessionContext ctx)</code>	<code>sessionContext</code>	A method that takes a <code>javax.ejb.SessionContext</code> as parameter
<code>create(Object[] args)</code>	<code>create</code>	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>
<code>afterBegin()</code>	<code>afterBegin</code>	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.
<code>beforeCompletion()</code>	<code>beforeCompletion</code>	
<code>afterCompletion(boolean commit)</code>	<code>afterCompletion</code>	A method that takes a <code>boolean</code> argument. All Exceptions are kept on server side.

7.3.1.1.4 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.

7.3.1.1.5 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)

7.3.1.1.5.1 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>
  </map>
</property>
```

```

    </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>

```

7.3.1.2 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"/> <!-- configur

```

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"/>

```

7.3.1.2.1 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

7.3.1.3 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).

7.3.1.3.1 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.

7.4 References

- `ModuleRemoting`
 - 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/>
-

7.5 Internal design

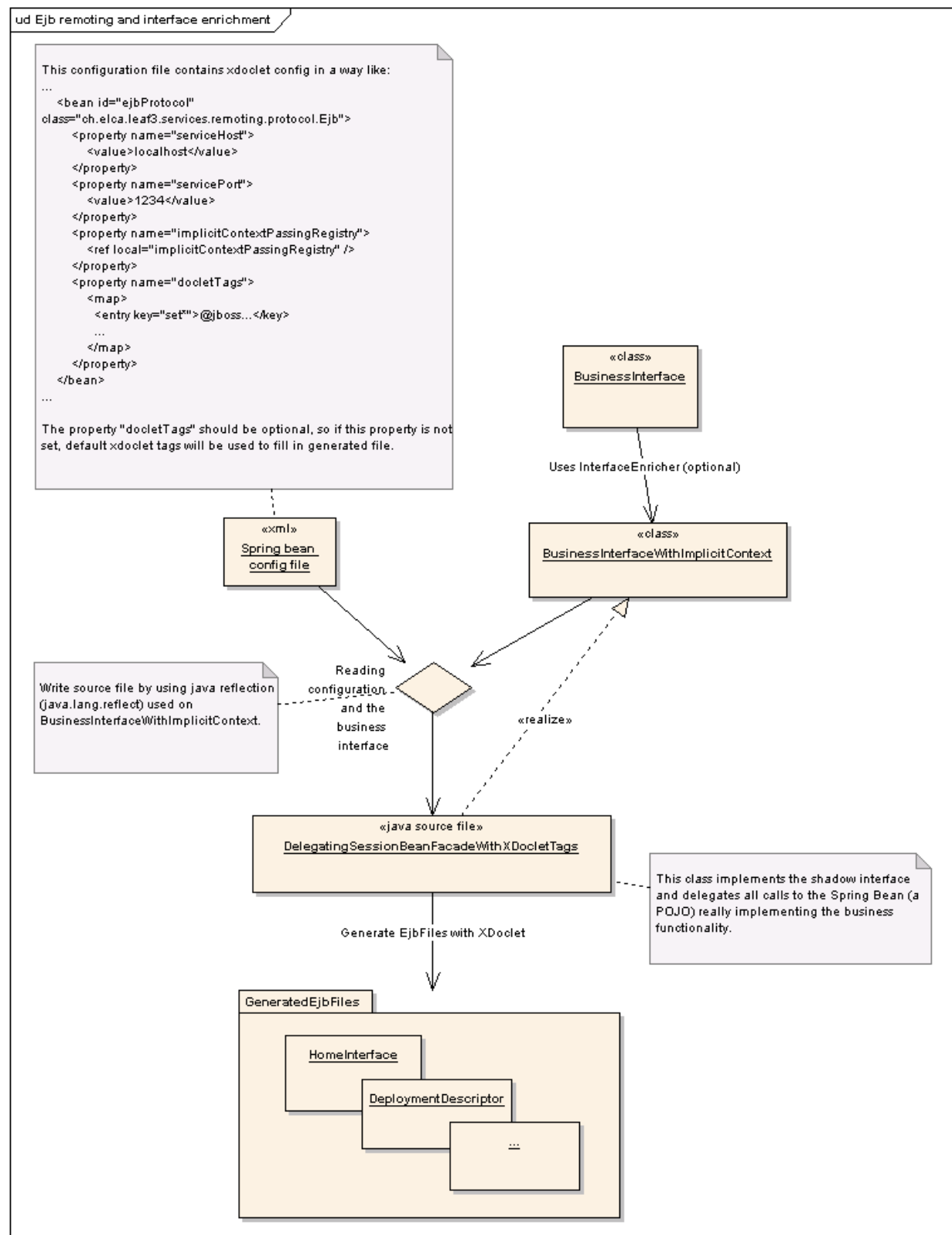
7.5.1 EJB generation

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 its 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.



7.5.2 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 XDocletting 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.

8 Documentation for module security

8.1 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.

8.2 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`.

8.3 How to use

Please refer to the demo application for now.

9 Documentation for module exception handling

9.1 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.

9.2 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.

9.3 How to use

9.3.1 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).

9.3.1.1 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).

9.3.1.2 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>
```

9.3.1.3 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
```

9.3.1.4 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.RoundRobinSwappableTa">
              <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>

```

9.3.1.5 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>

        <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>
```

9.4 References

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

9.5 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
<i>defaultBehaviourConsume</i>	<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
<i>forwardSignatureExceptions</i>	<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
<i>handleRTSignatureExceptions</i>	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

9.5.1 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.

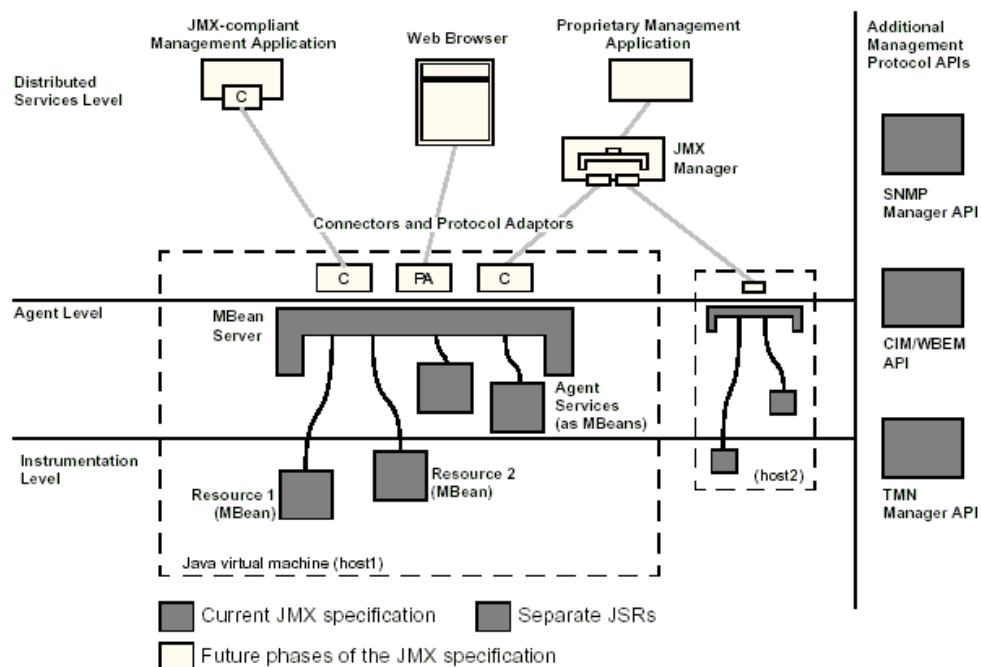
10 Documentation for module JMX

10.1 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. For the impatient: [JmxModuleForTheImpatient](#)

10.2 Introduction to Java management eXtensions (JMX)

The following picture shows the components of JMX:



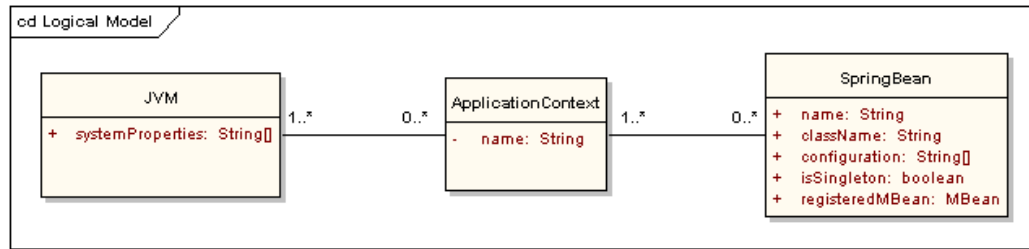
The central part of JMX is the *MBean Server*. *Managed Beans*, special Java objects that a developer wants to have controlled during runtime, are registered at the MBean Server. These Managed Beans or Mbeans are typically proxies for other components in the JVM one wants to monitor. These MBeans can 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.

10.3 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.

10.4 Usage

10.4.1 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.

10.4.1.1 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.

10.4.1.2 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.

10.4.2 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.

10.4.3 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`.

10.4.3.1 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>

```

10.4.3.2 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).

10.4.4 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();
        }
    }
}
```

10.4.5 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:

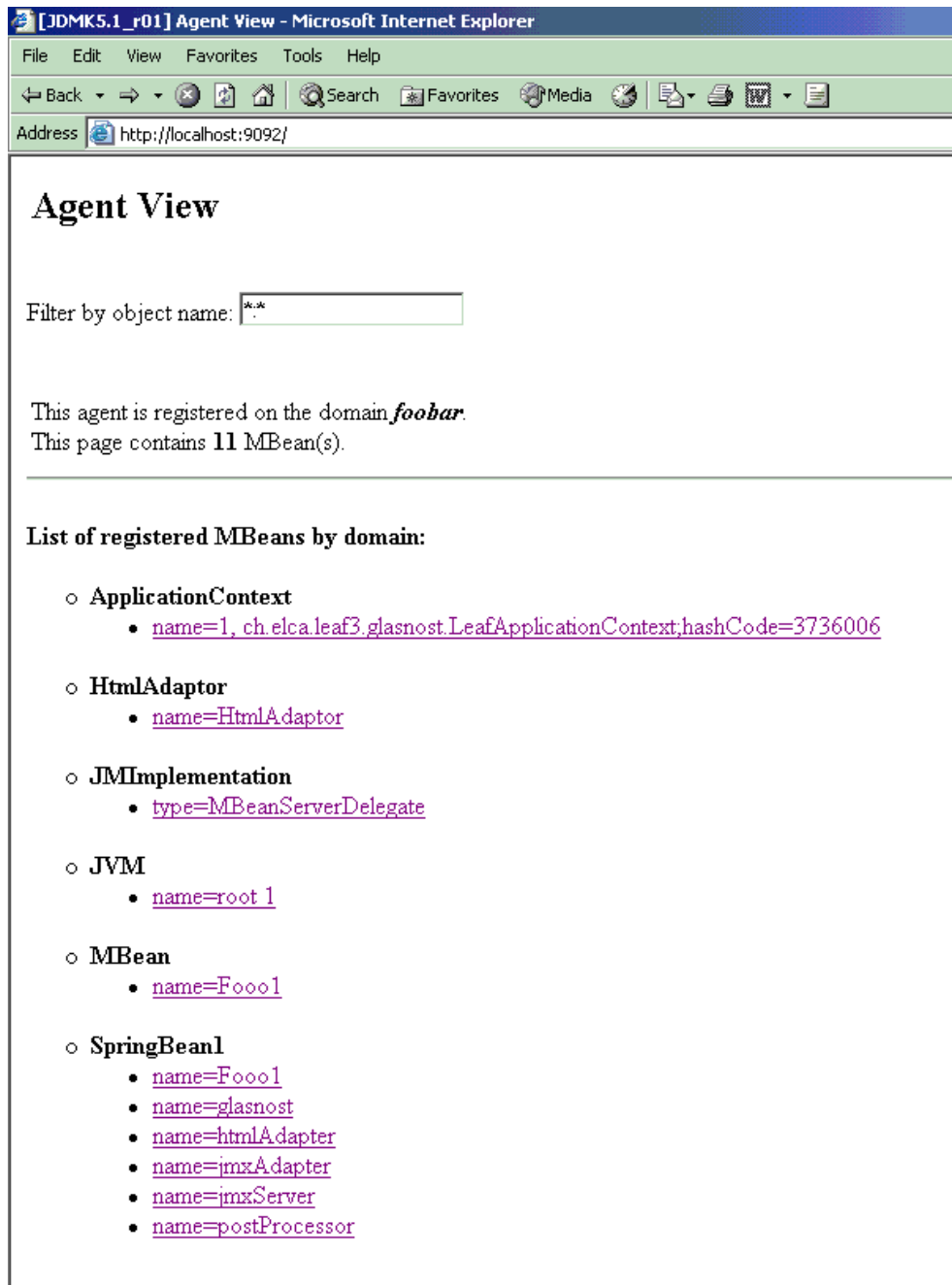
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<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 `jmxAdaptor`, which is now called `mBeanExporter`.

10.4.6 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.

10.5 Implemented Features

There are a lot of MBeans already implemented and published. Here only a few examples are shown. The best way to find out more about the implemented beans is to browse yourself through them!

10.5.1 JVM-Monitor

The JVM-Monitor MBean is published under the domain 'JVM'. It contains important information about the current JVM, such as which application context is loaded, values of the system properties and properties of the currently running threads (see screen shots below).

Array View

- ◆ **MBean Name:** JVM:name=jvmRootMonitor 1
- ◆ **MBean Attribute:** SystemProperties
- ◆ **Array of:** java.lang.String

[Back to MBean View](#)

Element at	Access	
0	RO	java.runtime.name = Java(TM) 2 Runtime Environment, Standard Edition
1	RO	sun.boot.library.path = C:\Program Files\Java\jdk1.5.0_07\jre\bin
2	RO	java.vm.version = 1.5.0_07-b03
3	RO	java.vm.vendor = Sun Microsystems Inc.
4	RO	java.vendor.url = http://java.sun.com/
5	RO	path.separator = ;
6	RO	java.vm.name = Java HotSpot(TM) Client VM
7	RO	file.encoding.pkg = sun.io
8	RO	user.country = CH
9	RO	sun.os.patch.level = Service Pack 2
10	RO	java.vm.specification.name = Java Virtual Machine Specification
11	RO	user.dir = D:\Projects\EL4J\external\framework\demos\light_statistics
12	RO	java.runtime.version = 1.5.0_07-b03
13	RO	java.awt.graphicsenv = sun.awt.Win32GraphicsEnvironment

showThreadTable Successful

The operation [showThreadTable] was successfully invoked for the MBean [JVM:name=jvmRootMonitor 1].
The operation returned with the value:

Thread Id	Name	isDeamon	State	Thread Group	Priority	
10	Thread-2	false	RUNNABLE	main	6	java.lang.Thread.dumpThreads(Method)sun.reflect.NativeMethod
1	main	false	TIMED_WAITING	main	5	java.lang.Thread.sleep(Native M
2	Reference Handler	true	WAITING	system	10	java.lang.Object.wait(Native Me
3	Finalizer	true	WAITING	system	8	java.lang.Object.wait(Native Me
4	Signal Dispatcher	true	RUNNABLE	system	9	
8	HtmlAdapter:name=HtmlAdapter1	false	RUNNABLE	main	6	java.net.PlainSocketImpl.socket

[Back to MBean View](#)

10.5.2 Log4jConfig

The Log4jConfig MBean is published under the domain 'JVM'. It shows information about the loaded loggers. Furthermore it allows change of the level of loggers. Furthermore it can also generate XML configuration code of logger level changes made during a session (see screen shots below).

List of MBean attributes:

Name	Type	Access	
<u>Name</u>	java.lang.String	RO	log4j
<u>RootLoggerLevel</u>	java.lang.String	RW	WARN

List of MBean operations:

Description of showLogLevelCache

java.lang.String

Description of showAppenders

[Lorg.apache.log4j.Appender; (java.lang.String)p1

Description of showLogLevel

org.apache.log4j.Level (java.lang.String)p1

Description of changeLogLevel

void (java.lang.String)p1
 (java.lang.String)p2

Description of showAvailableAppendersList

[Ljava.lang.String;

showLogLevelCache Successful

The operation [showLogLevelCache] was successfully invoked for the MBean [JVM:name=log4jConfig].
 The operation returned with the value:

```
<logger name="ch.elca.el4j.demos.statistics.light">
  <level value="DEBUG"/>
</logger>

<root>
  <level value="WARN"/>
</root>
```

[Back to MBean View](#)

Some available features:

- The 'changeLogLevel(category , level)' method allows to change the log level of a certain category logger. To see the log level of a category, the method 'showLogLevel(category)' can be used.
- The 'RootLoggerLevel' property allows to change the level of the root logger.
- The 'showLogLevelCache' method returns an XML string, which represents all the logger level changes made through the methods 'changeLogLevel' or property 'RootLoggerLevel' to the logger levels. The output string is suitable for copy–pasting into a Log4j.xml configuration file (see output in screen shot above).
- Normally appenders to loggers are specified in the Log4j.xml file. But sometimes its quite handy to be able to add/remove appenders for certain loggers, without having to shutdown the application. To enable this functionality, four methods are implemented: The 'AvailableAppendersList' property shows a list of appenders (appenderName and reference to appenderObject), which are available for attachment to a logger. The 'addAppender(category , appenderName)' method allows to attach an appender (which is listed in the 'AvailableAppendersList' property), to a logger category. The 'removeAppender(category , appenderName)' disattaches an appender from a logger category. For seeing, which appenders are connected to a logger the method 'showAppenders(category)' can be used. The appenders which are available ('AvailableAppendersList' property), are loaded from a bean with the name 'log4jJmxLoader' during the initialization of the application. The 'log4jJmxLoader' bean and appenders can be instantiated as follows:

```
<bean id="log4jJmxLoader" class="ch.elca.el4j.services.monitoring.jmx.Log4jJmxLoader">

    <property name="appenders">
        <map>
            <entry>
                <key>
                    <value>nullAppender</value>
                </key>
                <ref bean="nullApp" />
            </entry>
        </map>
    </property>
</bean>

<bean id="nullApp" class="org.apache.log4j.varia.NullAppender" />
```

The 'Log4jJmxLoader' class has a hashmap property 'appenders'. The key of this hashmap is the name of the appender bean (the appenderName as shown in the 'AvailableAppendersList' property). The hashmap value is an appender bean.

10.5.3 Spring Beans

Spring beans are published under the domain 'SpringBean'. Among other properties, it can be found out if the spring bean is proxied, which interceptors it has, the application context, etc.

10.5.4 JDK 1.5 Standard MBeans

If JMX is running under a JRE version 1.5 or higher, automatically the JDK 1.5 MBeans are published under the domain 'java.lang'. These beans give information about garbage collection, memory management, threads, operating system, etc.

10.6 Patch

The original 'jmxtools-1.4.2.jar' jar–file from Sun contained code, which instantiated two threads, which were not started as daemon threads. The problem with this approach was, that these two threads remained active, also after the main application thread was finished. Therefore these two threads hindered the JVM from being shut down. This 'bug' was localized in the 'com.sun.jdmk.comm' package(classes 'CommunicatorServer' and 'HtmlRequestHandler'). The 'HtmlRequestHandler' class was patch directly, by setting the thread to daemon, before starting it. The 'CommunicatorServer' class was patched in its subclass

'HtmlAdaptorServer', because the source code of 'CommunicatorServer' was not available to us. The patched jar-file was named 'jmxtools-1.4.2_deamon_patch.jar'.

10.7 References

- Further information regarding JMX can be found under <http://java.sun.com/products/JavaManagement/index.jsp>.
- `JmxModuleForTheImpatient` shows how easily you can use this in your applications.

11 Documentation for module light statistics

11.1 Purpose

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

11.2 Important concepts

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

11.2.1 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).

11.3 How to use

11.3.1 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](#).

11.3.2 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.

11.3.3 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.

TBD: the following needs to be adapted to how this is done with maven:

11.3.3.1 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"/>
```

11.3.3.2 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>
```

11.3.3.3 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.

11.4 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](#).

11.5 References

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

12 Documentation for module Spring Rich Client Platform (Spring RCP)

12.1 Purpose

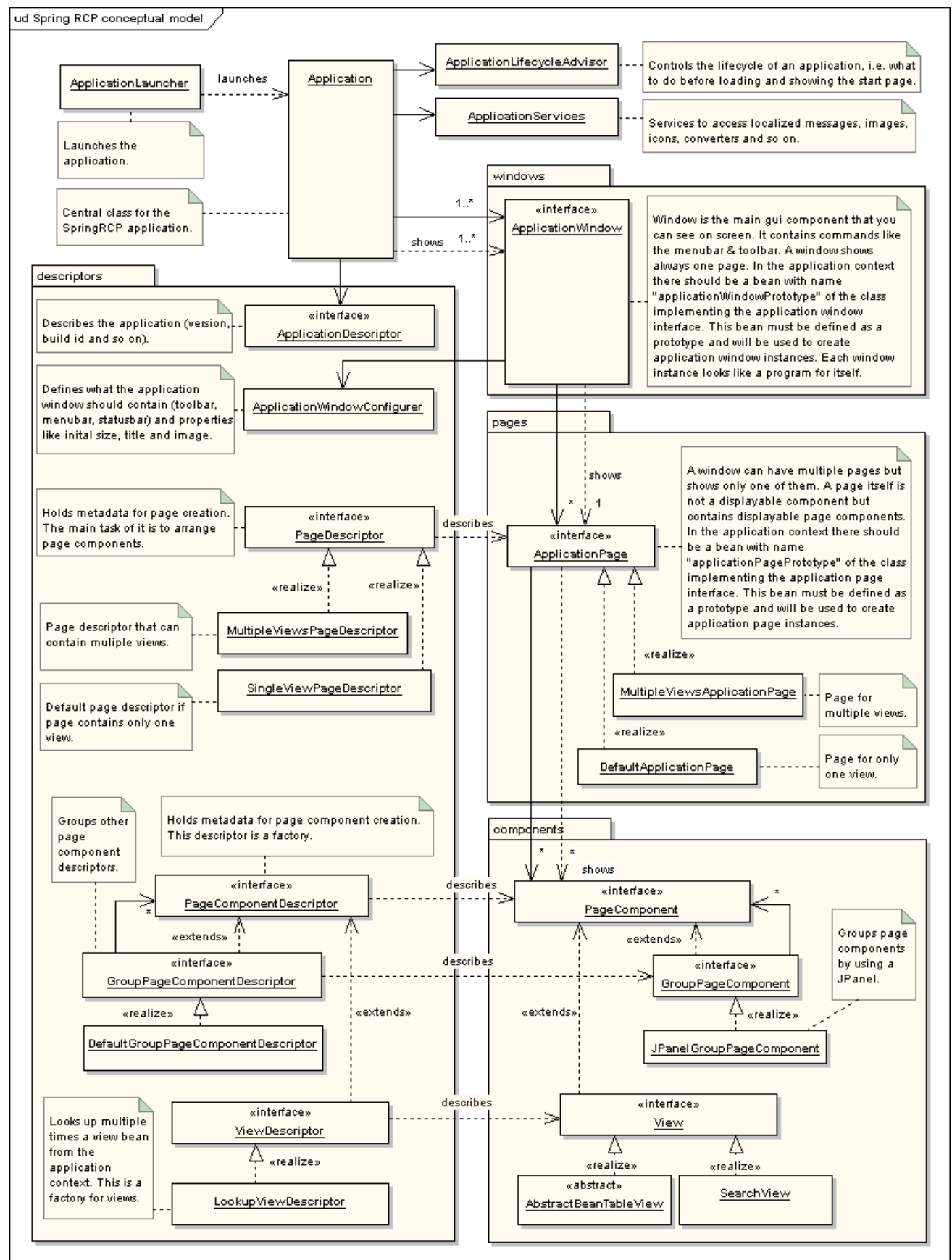
Spring RCP packaging and extensions. Used to *easily create professional Swing GUIs* with master-detail-views *based on java beans*.

12.2 Important concepts

The goal of the Spring Rich Client Platform (Spring RCP, <http://sourceforge.net/projects/spring-rich-c>) is to simplify the implementation of professional, enterprise-ready rich client applications. The idea of this module is to package and further simplify the Spring RCP usage. Our support includes more convenient support for displaying and editing java beans.

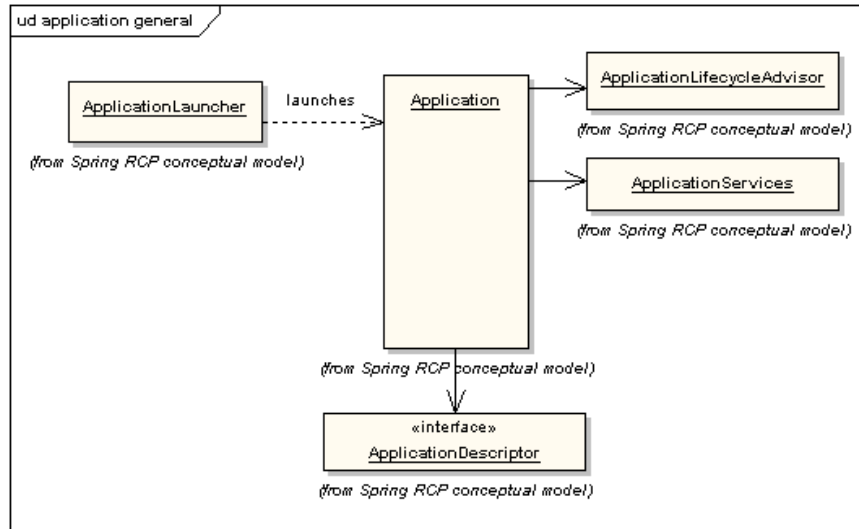
By using this module, you can develop your rich client by writing spring configuration files and some small java classes. Our Spring RCP demo application is represented by the *refdb-gui* module, which will be described later in the how to sections. We first have a look at the conceptual model of Spring RCP.

12.2.1 Overview



The huge diagram above shows you the main concepts of Spring RCP. The following sections describe the different parts of this diagram. We start our discussion with the upper left part of the diagram.

12.2.2 Application in general



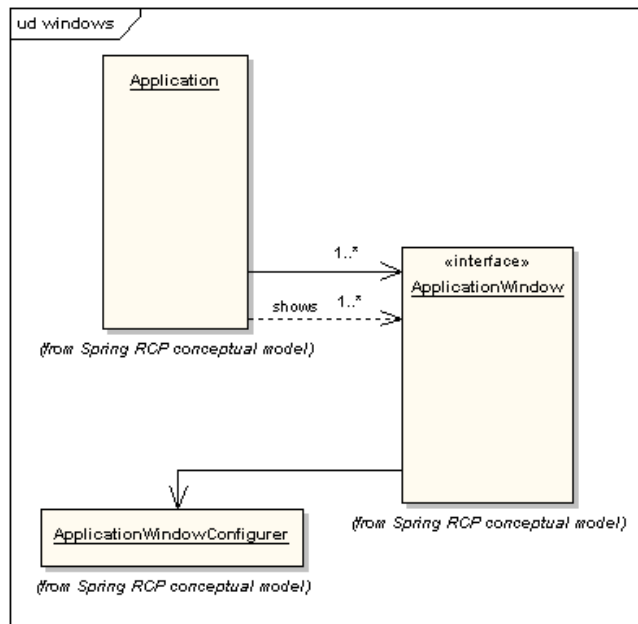
In the center we have the application class

(`org.springframework.richclient.application.Application`). This class is a singleton via which you access every part of the Spring RCP application. This singleton is created by the application launcher (`ch.elca.el4j.services.gui.richclient.ApplicationLauncher`). The application launcher receives two spring config locations. The first one is only used while loading the second one. The first one is named **startup context**. This config contains only one file with one bean named `splashScreen`. The configured image of this bean will be displayed during loading the second spring config location named **application context**. The application launcher will look for a bean with the name `application`. This bean must be of type `Application`. The application has a reference to an application lifecycle advisor, an application service class and an application descriptor. The descriptor only contains some data to describe the application. The lifecycle advisor

(`org.springframework.richclient.application.config.ApplicationLifecycleAdvisor`) is used for the application's lifecycle. By subclassing it you can control what should be done before loading the first window, i.e. user authentication or what should be done if the user closes the application. The application service class is used to get access to services like message source, image source, conversion service (something to string and string to something), command configurers or the application context the application was created with. The application services class

(`org.springframework.richclient.application.ApplicationServices`) can be accessed by the static `services()` method of the application class.

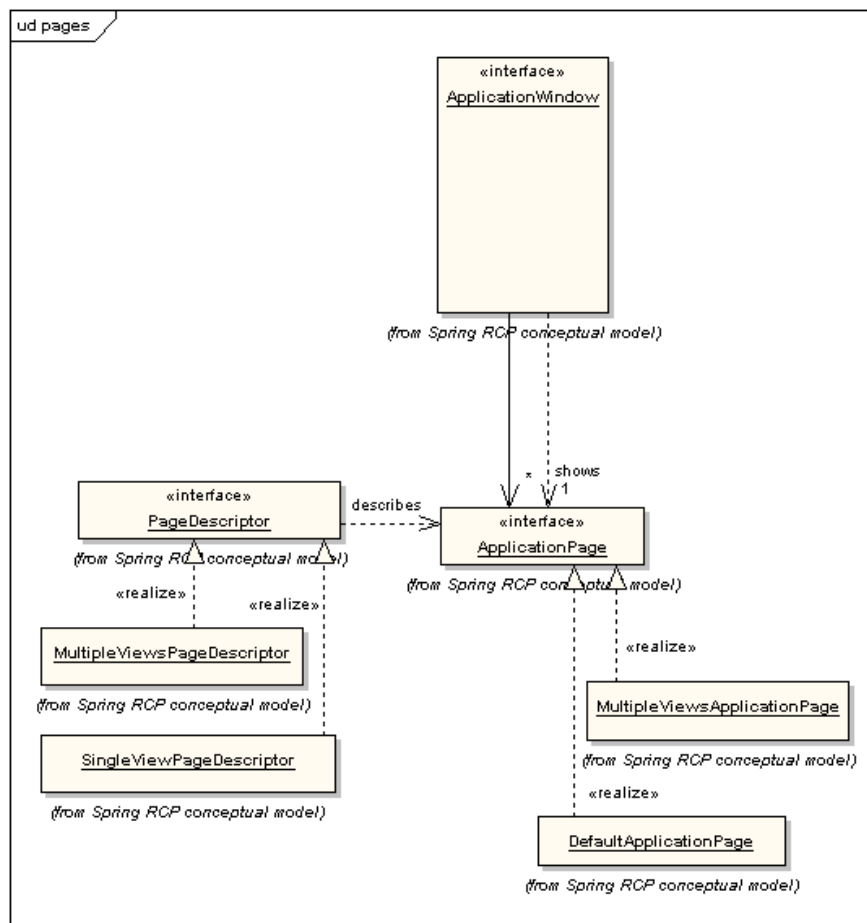
12.2.3 Windows



The application instantiates and shows at least one window

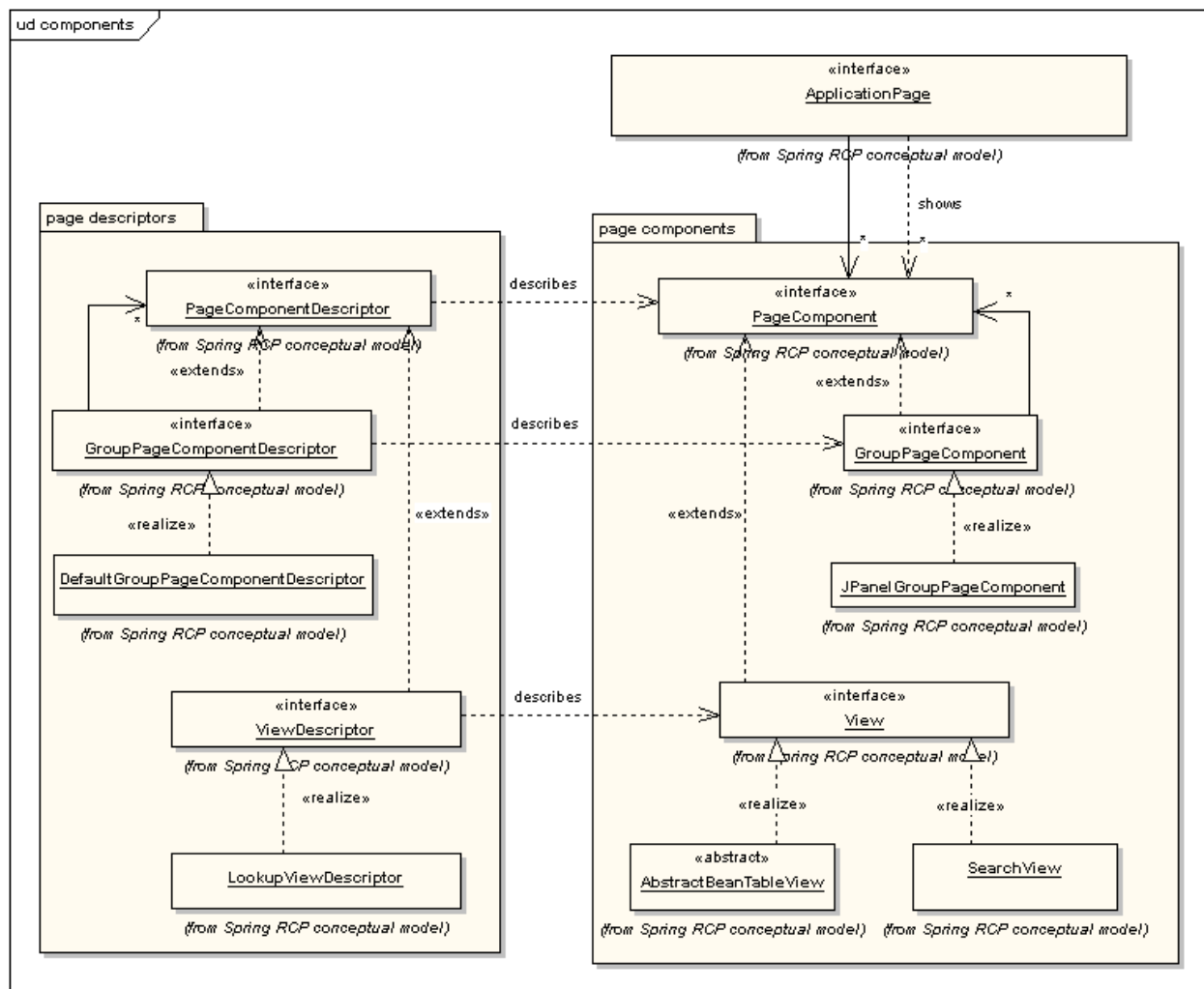
(`org.springframework.richclient.application.ApplicationWindow`). Each window can have a menubar, a toolbar and a statusbar. The components with commands for these *bars* are created by the application lifecycle advisor. These window commands (***windowCommandBarDefinitions*** is the property to set in the application lifecycle bean) are defined in their own spring config file. We will have a look at this in the how to sections. The window configurer can be defined to change the initial size of the window, or to make bars (menubar, toolbar or statusbar) invisible. We provide the class `ch.elca.el4j.services.gui.richclient.windows.MultipleViewsApplicationWindow` as a default implementation for a window.

12.2.4 Pages



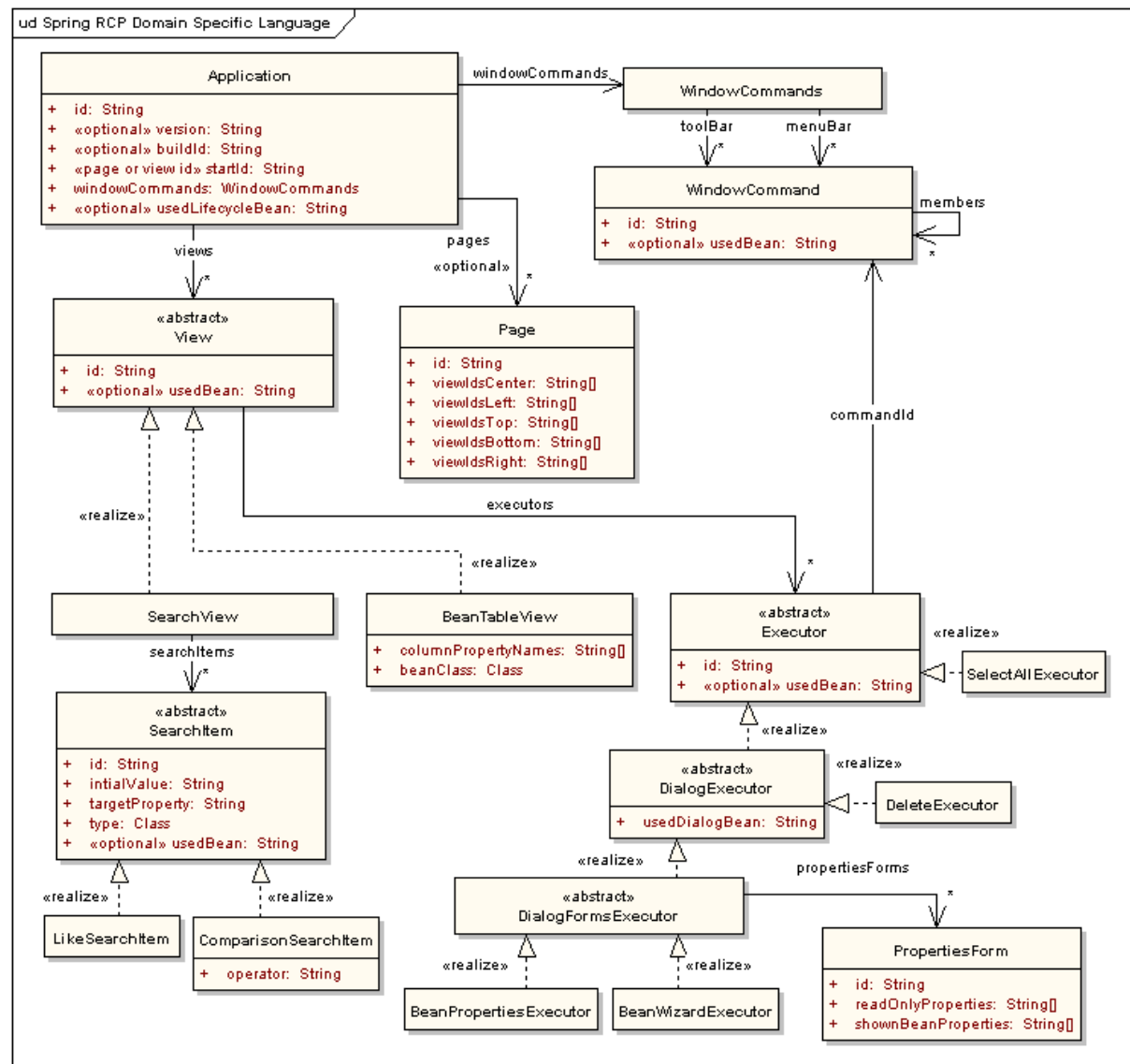
Each window can have multiple pages but a window can display only one page at a time. For those who know Eclipse, a perspective is the equivalent of a page. To create a page we need a page descriptor, because it contains metadata for page creation. In the application lifecycle advisor we have to define the page that should be loaded as default (property `startingPageId`).

12.2.5 Components



We saw that a window has pages. Until now only an empty window would be displayed. To populate the application we need page components (`org.springframework.richclient.application.PageComponent`). Each page can have multiple page components. Similar to pages, the page components must have descriptors, namely page component descriptors (`org.springframework.richclient.application.PageComponentDescriptor`). These page component descriptors are factories for page components. The most important page component is the view (`org.springframework.richclient.application.View`) with its descriptor (`org.springframework.richclient.application.ViewDescriptor`). At the moment, there are two views. The first one is the abstract bean table view (`ch.elca.el4j.services.gui.richclient.views.AbstractBeanTableView`). Its idea is to display java beans in a sortable table and to perform actions like edit, delete and insert on it. Possible actions on this view are defined in the interface `ch.elca.el4j.services.gui.richclient.presenters.BeanPresenter`. The second one is a search view (`ch.elca.el4j.services.gui.richclient.views.SearchView`). The goal of this view is to easily create a mask for searching. These two views are displayable components. Now the question is how to bring them on a page. The multiple views page descriptor from the latter section has a `pageComponentDescriptors` property that needs an array of page component descriptors. We can put our two view descriptors directly in there. In order to control how these views are organized on a page, we have implemented a group page component (`ch.elca.el4j.services.gui.richclient.pagecomponents.GroupPageComponent`). With this page component, you can group other page components. In some cases you have only one view on a page. In this case you can set the `startingPageId` property of the application lifecycle advisor directly to the id of your view descriptor. In background a single view page descriptor will be used.

12.2.6 Executors



As you can see in the overview above, a view can have executors. An executor is a unit that performs actions such as select all, delete, edit and create. Each executor belongs normally to only one view. Each application has window commands. These commands are used in the menu- and toolbar. A command can execute a hard-coded or a view-specific action. For example, the exit command in the menubar will execute always the same exit procedure, independently of the views which are displayed. A properties command to edit java beans is mostly view-specific. An executor will always be bound to its associated command. For example, we bind our properties executor to the properties window command. This command will change the procedure to execute on each change of the active view.

12.2.7 Application services

Each application has several services that can be accessed via the static `services()` method of the application class. The application services class (`org.springframework.richclient.application.ApplicationServices`) always tries to get the requested service from the application context by using a constant bean name. Example: If the conversion service is requested, the application context will be asked for a bean with the name `conversionService`. If the application context does not contain a bean with this name, the application service instance will create the default conversion service.

12.2.7.1 Message source accessor

The message source accessor is used to retrieve localized messages. Every displayed text must be taken from the message source to be able to change the software language. The texts can be defined in a properties file as name–value pairs. The name of a name–value pair will be used to get the associated text (value). The `messageSource` bean is taken by the application context and Spring RCP will request the application context for texts.

12.2.7.2 Image and icon sources

As the message source, the image and icon sources (beans `imageSource` and `iconSource`) will get the path to an image/icon via a properties file.

12.2.7.3 Rule source

The `ruleSource` bean is of type `org.springframework.rules.RuleSource` and is used to check that a java bean property has a correct value.

12.2.7.4 Other services

Service	Purpose
<code>componentFactory</code>	Used to create swing components
<code>conversionService</code>	Used to convert a type, i.e. string to date
<code>applicationObjectConfigurer</code>	Used to set the label, description and icon of an application object/gui component
<code>commandConfigurer</code>	Used to set the label, description and icon of a command/window command
<code>lookAndFeelConfigurer</code>	Used to set the look and feel of the application (windows style, linux style and so on)
<code>formComponentInterceptorFactory</code>	Used to intercept form components, for example in a properties executor to change the background color of a textfield if the content breaks its rules of the rule source
<code>formPropertyFaceDescriptorSource</code>	Provides the label, description and icon for a form property
<code>binderSelectionStrategy</code>	Specifies how properties/types must be displayed in a gui
<code>bindingFactoryProvider</code>	Used to create the binding between a model and a gui component, i.e. a swing component
<code>valueChangeDetector</code>	Used to check if two values are equal
<code>applicationSecurityManager</code>	Security manager for login and logout actions (authentication)
<code>securityControllerManager</code>	Security controller to control method invocations (authorisation)

12.3 How to use

The sections below show how parts of *module-springrcp* are used for a simple application such as the one of module *refdb-gui* (<http://svn.sourceforge.net/viewcvs.cgi/el4j/trunk/el4j/framework/apps/refdb/app/gui>). BTW: In most cases you can simply replace *myproject* with your project name to get a usable config.

12.3.1 Launch the application

The application is launched by using the application launcher (`ch.elca.el4j.services.gui.richclient.ApplicationLauncher`). Here an example how this could look like:

```
/**
 * Start method.
 */
```

```

* @param args
*       Are the command line arguments. These are ignored.
*/
public static void main(String[] args) {
    String startupContext
        = "classpath:scenarios/springrcp/startup/*.xml";
    String[] applicationContexts = {
        "classpath*:mandatory/*.xml",
        "classpath*:scenarios/db/raw/*.xml",
        "classpath*:scenarios/dataaccess/ibatis/*.xml",
        "classpath:optional/interception/transactionCommonsAttributes.xml",
        "classpath:scenarios/springrcp/myproject/application/*.xml"
    };

    try {
        new ApplicationLauncher(startupContext, applicationContexts);
    } catch (Exception e) {
        String message = "MyProject-Application exited "
            + "exceptionally for an unknown reason! See stack trace for "
            + "details.";
        s_logger.fatal(message, e);
        DialogUtils.showErrorMessageDialog(e, null);
        System.exit(1);
    }
}

```

For the startup context have a look at the directory

<http://svn.sourceforge.net/viewcvs.cgi/el4j/trunk/el4j/framework/apps/refdb/app/gui/conf/scenarios/springrcp/refdb/startup/>

The last string of the application context array is the one that contains the Spring RCP application beans.

12.3.2 General configuration

In *module-springrcp* you find predefined spring config parts for your Spring RCP application. In most cases you can include the config files directly in your config file. We recommend to create a config file **general.xml** in directory **scenarios/springrcp/myproject/application** of your gui module's `conf` folder (of your new module `myproject-gui`). Your file could look like this:

```

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

<beans>
    <!-- Application general parts. -->
    <import resource="classpath:scenarios/springrcp/application/applicationGeneral.xml"/>
    <!-- Are the message, image and icon sources. -->
    <import resource="classpath:scenarios/springrcp/application/imageIconMessageSources.xml"/>
    <!-- Are some services for the application. -->
    <import resource="classpath:scenarios/springrcp/application/applicationServices.xml"/>
    <!-- Are helpers and factories to create and manage components. -->
    <import resource="classpath:scenarios/springrcp/application/applicationComponentParts.xml"/>
    <!-- Are binders for gui components with data models. -->
    <import resource="classpath:scenarios/springrcp/application/componentBinders.xml"/>
</beans>

```

12.3.3 Property merger and overrider

Now you have to override or merge the default values with the ones specific to your application. Create the following three property files in the directory **scenarios/springrcp/myproject/application**. Before changing some property values, you have to wait until you have created the corresponding resources.

property-override-configurer.properties:

```

applicationDescriptor.version=1.0
applicationDescriptor.buildId=2006-02-03

```

```
applicationLifecycleAdvisor.windowCommandBarDefinitions
    =classpath:scenarios/springrcp/myproject/application/internal/commands.xml
applicationLifecycleAdvisor.startingPageId=myStartPage
```

property-merge-configurer-after.properties:

```
imageResourcesFactory.locations
    =classpath:scenarios/springrcp/myproject/properties/images.properties
```

property-merge-configurer-before.properties:

```
messageSource.basenames=scenarios.springrcp.myproject.properties.messages
```

Add the following three beans to your ***general.xml*** to override or merge properties.

```
<!-- Property override for general purpose. -->
<bean id="refdbGuiSpringrcpPropertyOverrideConfigurer"
      class="org.springframework.beans.factory.config.PropertyOverrideConfigurer">
    <property name="location"
        value="classpath:scenarios/springrcp/myproject/application
              /property-override-configurer.properties"/>
</bean>

<!-- List property merger. Appends given values to existing values. -->
<bean id="refdbGuiSpringrcpListPropertyMergeConfigurerAfter"
      class="ch.elca.el4j.core.config.ListPropertyMergeConfigurer">
    <property name="location"
        value="classpath:scenarios/springrcp/myproject/application
              /property-merge-configurer-after.properties"/>
    <property name="insertNewItemsAfter" value="true"/>
</bean>

<!-- List property merger. Prepends given values to existing values. -->
<bean id="refdbGuiSpringrcpListPropertyMergeConfigurerBefore"
      class="ch.elca.el4j.core.config.ListPropertyMergeConfigurer">
    <property name="location"
        value="classpath:scenarios/springrcp/myproject/application
              /property-merge-configurer-before.properties"/>
    <property name="insertNewItemsBefore" value="true"/>
</bean>
```

12.3.4 Rule source

Write a rule source to validate the java beans you would like to work on. Have a look at the class [RefdbValidationRulesSource.java](#) to have an example. Create a bean with your rule source class, name it ***ruleSource*** and add it in the file ***general.xml***.

12.3.5 Window commands

Create a new spring config file ***commands.xml*** in the directory ***scenarios/springrcp/myproject/application/internal***. Here is the file from the demo application. By default you must have the beans `windowCommandManager`, `menuBar` and `toolBar` defined. In the window command manager you define the property `sharedCommandIds` that holds a list of command ids that you would like to share in the application. Later, executors can be registered for these shared ids. The menu- and toolbar are beans of type `org.springframework.richclient.command.CommandGroupFactoryBean`. In property members, they can have further command group factory beans. Such a property can also consist of the `separator` string for a visible or the `glue` string for an invisible separator, a shared command id or directly an abstract command (`org.springframework.richclient.command.AbstractCommand`).

12.3.6 Message and image property files

Create in the directory ***scenarios/springrcp/myproject/properties*** the following two property files: ***messages.properties*** and ***images.properties***. The ***messages.properties*** file is used to get text and the ***images.properties*** file is used to get images.

messages.properties (example):

```
applicationDescriptor.title=My project as Spring RCP application
applicationDescriptor.caption=Short description of my project.
applicationDescriptor.description=Long description of my project.

newXyCommand.label=Create new &Xy
newXyCommand.caption=Creates a new Xy bean
```

Every gui component with text must have an entry in the ***messages*** property file. Some messages are already defined in message property files of ***module-springrcp*** or the Spring RCP jars. As you remember, we have prepended the value ***scenarios.springrcp.myproject.properties.messages*** to the string array of property ***basenames***, bean ***messageSource***. Prepending is needed in order not to lose existing values from the array and to be able to override existing name-value pairs in your ***messages.properties*** file. If you like to internationalize your application, you have to copy your ***messages.properties*** to ***messages_xy.properties*** and translate the messages. The ***xy*** is the locale of the target language, like ***de*** for German or ***fr*** for French. BTW, ***messages.properties*** will be taken if no specific properties file is available.

In the example above, you can see that the label and caption for the shared command id ***newXyCommand*** is defined. The label is the displayed text and the ampersand configures this command to be directly selectable via typing the ***x*** key, i.e. if the parent menu is expanded. Caption will be used as "tool-tip-text".

The application object configurer service will test each bean and look if it implements one of the following interfaces. If this is the case and if the corresponding property exists in message source, the setter method of the interface will be invoked with the looked-up property value. In the table below, the considered bean is called ***myGuiComponent***.

Interface	Property name
org.springframework.richclient.core.TitleConfigurable	myGuiComponent.title
org.springframework.richclient.core.LabelConfigurable	myGuiComponent.label
org.springframework.richclient.command.config.CommandLabelConfigurable	myGuiComponent.label
org.springframework.richclient.core.DescriptionConfigurable	myGuiComponent.caption, myGuiComponent.description

images.properties (example):

```
newXyCommand.icon=myproject_xy_new.gif
```

Every gui component which should or must be displayed as an icon/image must have an entry in the ***images*** properties file. Some images are already defined in images property files of ***module-springrcp*** or the Spring RCP jars. As you remember, we have appended the value ***scenarios/springrcp/myproject/properties/images.properties*** to the resource array of property ***locations***, bean ***imageResourcesFactory***. Appending is needed in order not to lose existing values from the array and to be able to override existing name-value pairs in your ***images.properties*** file.

In the example above you can see that the shared command id ***newXyCommand*** gets ***myproject_xy_new.gif*** as icon. By default, the ***myproject_xy_new.gif*** image must be in the ***images*** directory of the classpath, i.e. in the ***images*** directory of your module's ***conf*** directory.

The application object configurer service will test each bean and look if it implements one of the following interfaces. If this is the case and if the corresponding property exists in the image/icon source, the setter method of the interface will be invoked with the looked-up property value. In the table below, the considered bean is called `myGuiComponent`.

Interface	Property name
<code>org.springframework.richclient.image.config.ImageConfigurable</code>	<code>myGuiComponent.image</code>
<code>org.springframework.richclient.image.config.IconConfigurable</code>	<code>myGuiComponent.icon</code>
<code>org.springframework.richclient.command.config.CommandIconConfigurable</code>	<code>myGuiComponent.icon</code> , <code>myGuiComponent.pressedIcon</code> , <code>myGuiComponent.disabledIcon</code> , <code>myGuiComponent.rolloverIcon</code> , <code>myGuiComponent.selectedIcon</code> , <code>myGuiComponent.large.icon</code> , <code>myGuiComponent.large.pressedIcon</code> , <code>myGuiComponent.large.disabledIcon</code> , <code>myGuiComponent.large.rolloverIcon</code> , <code>myGuiComponent.large.selectedIcon</code>

12.3.7 Pages

Create a new spring config file with the name ***pages.xml*** in the directory ***scenarios/springrcp/myproject/application***. In this file, we add page descriptor beans. The default implementation is

`ch.elca.el4j.services.gui.richclient.pages.descriptors.MultipleViewsPageDescriptor`.

The most important properties of this page descriptor are ***pageComponentDescriptors*** and ***layoutManager***.

Property `pageComponentDescriptors` needs an array of page component descriptors. Property `layoutManager` needs a layout manager. The given page component will be arranged by the given layout manager. Here is a small example:

```
<bean id="myStartPage"
  class="ch.elca.el4j.services.gui.richclient.pages.descriptors.MultipleViewsPageDescriptor">
  <property name="pageComponentDescriptors">
    <list>
      <ref bean="xySearchView" />
      <ref bean="xyTableView" />
    </list>
  </property>
  <property name="layoutManager">
    <bean class="java.awt.GridLayout" />
  </property>
</bean>
```

This page arranges the page component `xySearchView` on the left and the page component `xyTableView` on the right. Thanks to grid layout, both page components have the same and maximum size. By default, a page uses a border layout. If a given page component descriptor implements interface `ch.elca.el4j.services.gui.richclient.pagecomponents.descriptors.LayoutDescriptor`, the preferred position argument and index will be taken for component arrangement.

Do not forget to add a label and a caption property in `messages.properties` and an image property in `images.properties` for the page `myStartPage`. These properties could look as follows:

messages.properties (example):

```
myStartPage.label=&Start page@ctrl S
myStartPage.caption=Start page
```

images.properties (example):

```
myStartPage.image=myproject-startpage-image.gif
```

The `@ctrl S` means that this page can be opened by hitting Ctrl-S.

To group page components on a page, you can use a group page component descriptor

(`ch.elca.el4j.services.gui.richclient.pagecomponents.descriptors.GroupPageComponentDescriptor`)

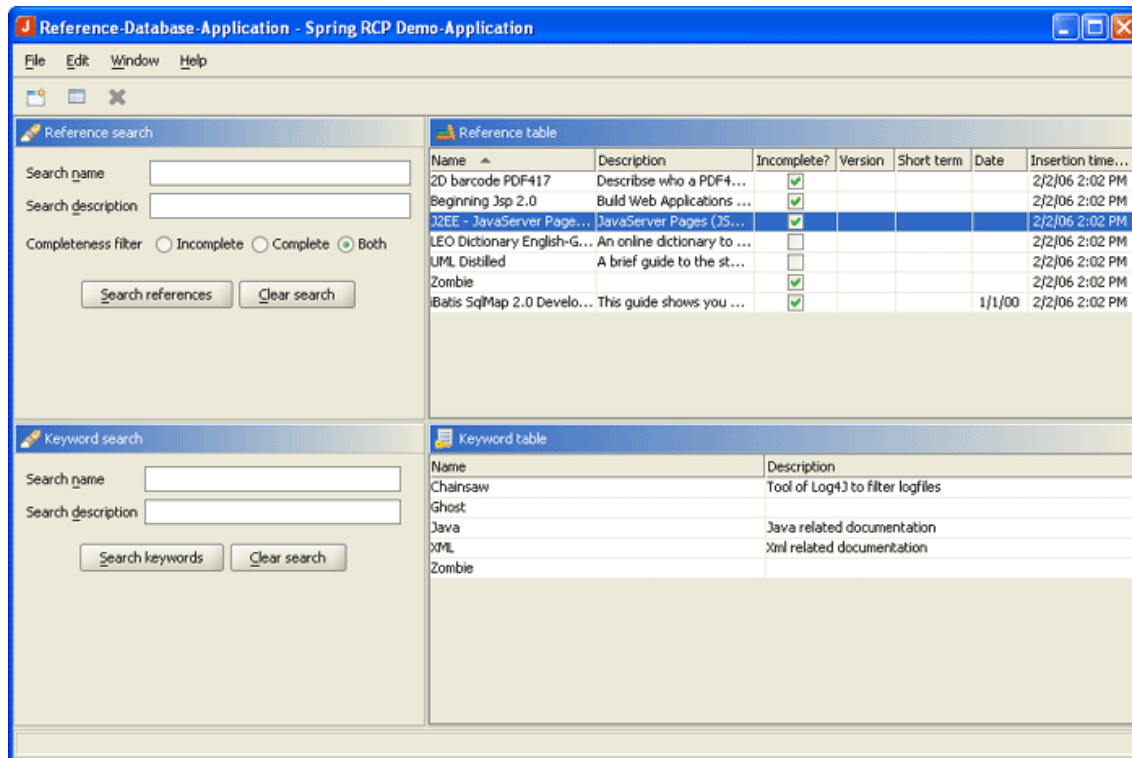
Our default implementation is the

`ch.elca.el4j.services.gui.richclient.pagecomponents.descriptors.impl.DefaultGroupPageComponentDescriptor`

that arranges the page components on a `javax.swing.JPanel` by using the given layout manager. By default, the layout manager is set to grid layout with one column and multiple rows.

```
<bean id="myStartPage"
    class="ch.elca.el4j.services.gui.richclient.pages.descriptors.MultipleViewsPageDescriptor">
    <property name="pageComponentDescriptors">
        <list>
            <bean name="tableGroup"
                class="ch.elca.el4j.services.gui.richclient.pagecomponents
                    .descriptors.impl.DefaultGroupPageComponentDescriptor">
                <property name="preferredPositionArgument" value="Center"/>
                <property name="pageComponentDescriptors">
                    <list>
                        <ref bean="xyTableView" />
                        <ref bean="abTableView" />
                    </list>
                </property>
            </bean>
            <bean name="searchGroup"
                class="ch.elca.el4j.services.gui.richclient.pagecomponents
                    .descriptors.impl.DefaultGroupPageComponentDescriptor">
                <property name="preferredPositionArgument" value="West"/>
                <property name="pageComponentDescriptors">
                    <list>
                        <ref bean="xySearchView" />
                        <ref bean="abSearchView" />
                    </list>
                </property>
            </bean>
        </list>
    </property>
</bean>
```

With the configuration above, the page could look like in following screenshot of Reference-Database-Application (RefDb).



12.3.8 Views

Create a new spring config file with name **views-xy.xml** in the directory **scenarios/springrcp/myproject/application** where *xy* is the name of your application part, i.e. keyword for keyword-views. In this file we add view descriptor beans and view beans.

12.3.8.1 Bean table view

First we add a view descriptor bean for the view `xyTableView` we referenced in the page of the previous section.

```
<bean id="xyTableView"
      class="ch.elca.el4j.services.gui.richclient.views.descriptors.impl.LookupViewDescriptor">
  <property name="viewPrototypeBeanName">
    <idref bean="xyTableViewTarget" />
  </property>
  <property name="preferredGroup" value="tableGroup"/>
  <property name="preferredIndex" value="0"/>
</bean>
```

The view bean with name `xyTableViewTarget` will be defined next. This bean must be defined as a prototype so we can instantiate the view multiple times. BTW, if this view descriptor is not referenced in an opened page, it will be arranged in the group page component with name `tableGroup` as the first page component. If the group does not exist, the view will be placed as the first page component in the root of the page.

The view bean `xyTableViewTarget` must be of type `ch.elca.el4j.services.gui.richclient.views.AbstractView`. In the current case, we would like to have a view where our beans will be displayed in a table. For this, we have prepared the class `ch.elca.el4j.services.gui.richclient.views.AbstractBeanTableView`. Just create a subclass of this abstract bean table view and name it `XyView` (for keywords, it would have the name `KeywordView`). At the beginning you don't have to override any method. Add now a bean of type `XyView` called `xyTableViewTarget`.

```

<bean id="xyTableViewTarget"
      class="ch.elca.myproject.gui.views.XyView"
      singleton="false">
  <property name="beanTableModel">
    <ref bean="xyBeanTableModel" />
  </property>
  <property name="beanExecutors">
    <list>
      <ref bean="xyCreateNewExecutor" />
      <ref bean="xyDeleteExecutor" />
      <ref bean="xySelectAllExecutor" />
      <ref bean="xyPropertiesExecutor" />
    </list>
  </property>
</bean>

```

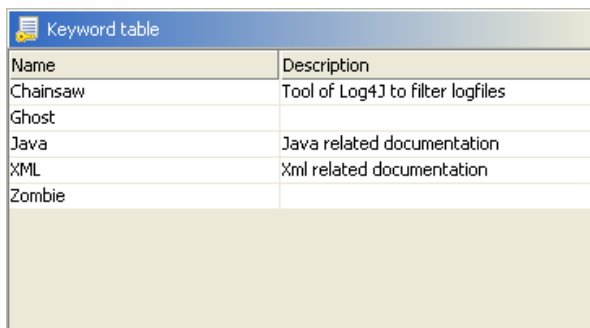
Such a bean table view has two important properties, `beanTableModel` and `beanExecutors`. The bean table model is used to describe which bean properties of the java bean `ch.elca.myproject.dto.Xy` should be displayed. We assume that this java bean has at least the properties `name` and `description`.

```

<bean id="xyBeanTableModel"
      class="ch.elca.el4j.services.gui.richclient.models.BeanTableModel"
      singleton="false">
  <property name="beanClass">
    <value>"ch.elca.myproject.dto.Xy"</value>
  </property>
  <property name="columnPropertyNames">
    <list>
      <value>name</value>
      <value>description</value>
    </list>
  </property>
</bean>

```

For the config above, but with java bean `KeywordDto` of `RefDb`, the bean table view could look like this:



Name	Description
Chainsaw	Tool of Log4J to filter logfiles
Ghost	
Java	Java related documentation
XML	Xml related documentation
Zombie	

Do not forget to add the appropriate properties in `messages.properties` and `images.properties`. You should at least set the label for table view, the "standard" (no `. *`), the label and description for each column property and the image of the table view.

messages.properties (example):

```

xyTableView.label=&Xy table view

name=Name
name.label=&Name
name.description=Name of the object
description=Description
description.label=&Description
description.description=Description of the object

```

If you have got two different java beans (i.e. `XyDto` and `AbDto`) and both have a property `name` you can prepend the uncapitalized class name to define different labels (i.e. `xyDto.name=Name of Xy` and

abDto.name=Name of Ab).

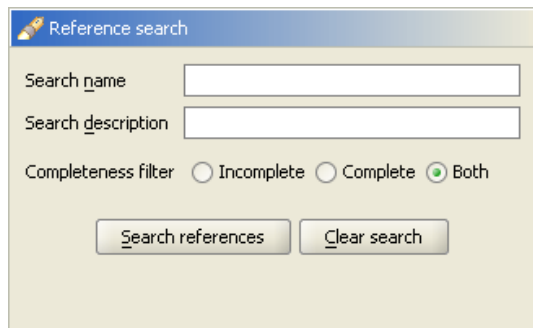
images.properties (example):

```
xyTableView.image=my-xy-table-view.gif
```

As in previous example of `images.properties` the image `my-xy-table-view.gif` must be placed by default in the `images` directory.

12.3.8.2 Search view

A further very useful view implementation is the search view (`ch.elca.el4j.services.gui.richclient.views.SearchView`). You do not have to subclass it.



In the screenshot above, a search for a reference java bean of `RefDb` is displayed. The three properties of this java bean are `name`, `description` and `incomplete`. The `name` and `description` properties are of type `java.lang.String` and the `incomplete` property is a boolean. The search view only needs the `searchItems` property to be set. This property is an array of abstract search items (class `ch.elca.el4j.services.gui.search.AbstractSearchItem`). Each search item points to a property of a java bean. Currently there are two types of search items, a like search item (`ch.elca.el4j.services.gui.search.LikeSearchItem`) and a comparison search item (`ch.elca.el4j.services.gui.search.ComparisonSearchItem`).

```
<bean id="xySearchView"
    class="ch.elca.el4j.services.gui.richclient.views.descriptors.impl.LookupViewDescriptor">
    <property name="viewPrototypeBeanName">
        <idref bean="xySearchViewTarget" />
    </property>
    <property name="preferredGroup" value="searchGroup" />
</bean>

<bean id="xySearchViewTarget"
    class="ch.elca.el4j.services.gui.richclient.views.SearchView"
    singleton="false">
    <property name="searchItems">
        <list>
            <bean class="ch.elca.el4j.services.gui.search.LikeSearchItem">
                <property name="targetProperty" value="name" />
                <property name="targetBeanClass"
                    value="ch.elca.myproject.dto.Xy" />
            </bean>
            <bean class="ch.elca.el4j.services.gui.search.LikeSearchItem">
                <property name="targetProperty" value="description" />
                <property name="targetBeanClass"
                    value="ch.elca.myproject.dto.Xy" />
            </bean>
            <bean class="ch.elca.el4j.services.gui.search.ComparisonSearchItem">
                <property name="targetProperty" value="incomplete" />
                <property name="targetBeanClass"
                    value="ch.elca.myproject.dto.Xy" />
            </bean>
        </list>
    </property>
</bean>
```

```

        </list>
    </property>
</bean>

```

The first bean is the view descriptor for the search view. The second bean is the search view. The like search item is by default used to perform a case-insensitive SQL like search on a string. You can use % as a placeholder for any characters. The comparison search item is by default used to perform an equals search on booleans (two states). The search item type is a `java.lang.Boolean` so it can have one of the three states `true`, `false` or `null` (unknown). In the used spring config `scenarios/springrcp/application/componentBinders.xml`, the `java.lang.Boolean` type is mapped to the `ch.elca.el4j.services.gui.richclient.forms.binding.ThreeStateBooleanBinder` binder to display a `javax.swing.JRadioButton` per state (see screenshot above).

Do not forget to add the appropriate properties in `messages.properties` and `images.properties`. You should at least set a label and an image for the search view and a label for each search item. The name of a search item is its `targetProperty` concatenated with its capitalized search item type (currently `Like` or `Comparison`). Further the three state boolean binder must have the appropriate properties for its states (see `incompleteComparison` in example below).

messages.properties (example):

```

xySearchView.label=&Xy search view

nameLike=Search name
nameLike.label=Search &name
nameLike.description=Is the name to search for. The percent sign is used as placeholder.
descriptionLike=Search description
descriptionLike.label=Search &description
descriptionLike.description=Is the description to search for. The percent sign is used as placeholder.
incompleteComparison=Search for incompletes?
incompleteComparison.label=Completeness filter
incompleteComparison.description=Completeness filter for xys.
incompleteComparison.true.displayName=Incomplete
incompleteComparison.true.description=Search for incomplete xys.
incompleteComparison.false.displayName=Complete
incompleteComparison.false.description=Search for complete xys.
incompleteComparison.unknown.displayName=Both
incompleteComparison.unknown.description=Search for incomplete and complete xys.

```

If you have two different search views (i.e. `xySearchView` and `abSearchView`) and both have a like search item for property name, you can prepend the search view name to define different labels (i.e.

```
xySearchView.nameLike=Search name of Xy and abSearchView.nameLike=Search name of Ab).
```

images.properties (example):

```
xySearchView.image=my-xy-search-view.gif
```

As in the previous examples, the image `my-xy-search-view.gif` must be placed by default into the `images` directory.

12.3.8.3 Combining a search view and a bean table view

If you click on the search button of a search view, a query object event (`ch.elca.el4j.services.search.events.QueryObjectEvent`) will be sent to every registered `org.springframework.context.ApplicationListener`. On each abstract view (`ch.elca.el4j.services.gui.richclient.views.AbstractView`), the `onQueryObjectEvent(QueryObjectEvent event)` method will be invoked. Now you have to override this method in the bean table views to fill their tables. Here is an example illustrating this.

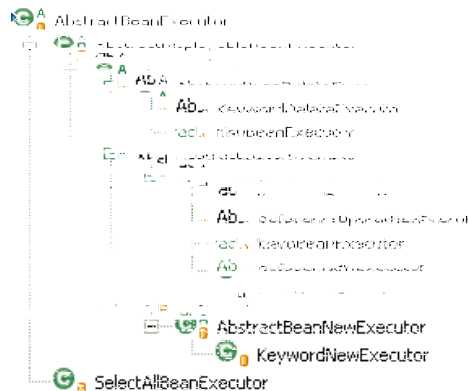
```
/**
 * {@inheritDoc}
 */
protected void onQueryObjectEvent(QueryObjectEvent event) {
    if (isControlCreated() && isQueryObjectComingFromNeighbour(event)) {
        QueryObject queryObject = event.getQueryObject(KeywordDto.class);
        if (queryObject != null) {
            ReferenceService referenceService
                = ServiceBroker.getReferenceService();
            List list = referenceService.searchKeywords(queryObject);
            setBeans(list);
        }
    }
}
```

In this example, the service broker ([see source](#)) is used get the reference service. This code is used in the keyword bean table view ([see source](#)) of RefDb.

12.3.9 Executors

Create a new spring config file with name ***executors-xy.xml*** in the ***scenarios/springrcp/myproject/application*** directory where ***xy*** is the name of your application part, for example ***keyword*** for ***keyword-executors***. In this file, we add executor beans. Each of your executors must extend the ***ch.elca.el4j.services.gui.richclient.executors.AbstractBeanExecutor***. Below now an overview of the existing executors.

12.3.9.1 Overview



Each executor implements interface `org.springframework.richclient.command.ActionCommandExecutor` with the entry point method `execute`.

12.3.9.2 AbstractBeanExecutor

This is the central executor class in **SpringRCP** of EL4J. It has a reference to the bean presenter (`ch.elca.el4j.services.gui.richclient.presenters.BeanPresenter`), which is normally the view where it is used, to change the containing beans. Further it contains the `commandId` property to bind window commands to executors. Property `id` and `schema` are used to refine the fetched messages and images. BTW, if `id` is not set the name of the bean will be taken.

View the source [here](#).

12.3.9.3 SelectAllBeanExecutor

This is the simplest executor in EL4J. It only selects all beans of given bean presenter and does not display anything extra. Further this is the only one that do not have to be extended.

View the source [here](#).

Here a sample spring config:

```
<bean id="xySelectAllExecutor"
      class="ch.elca.el4j.services.gui.richclient.executors.SelectAllBeanExecutor"
      singleton="false" />
```

12.3.9.4 AbstractDisplayableBeanExecutor

This executor has a property `displayable` that needs an object of type `ch.elca.el4j.services.gui.richclient.executors.displayable.ExecutorDisplayable`. This object is a gui element and let the user work with. Depending to the executed task a method from interface `ch.elca.el4j.services.gui.richclient.executors.action.ExecutorAction` will be called. This interface is implemented by the current executor so we come back to the executor for real execution of the task.

View the source [here](#).

12.3.9.5 AbstractConfirmBeanExecutor

This executor handles with a displayable element where a question is asked an you can confirm or cancel. By default `displayable`

`ch.elca.el4j.services.gui.richclient.dialogs.BeanConfirmationDialog` will be used.

View the source [here](#).

12.3.9.6 AbstractFinishBeanExecutor

This executor will work on a bean. This bean is encapsulated in a form model to be able to revert or commit changes to the given bean. To define the editable properties of the given bean property `beanPropertiesForms` it needs instances of type `ch.elca.el4j.services.gui.richclient.forms.BeanPropertiesForm`. In most cases we need only one so for example bean `Xy` with properties `name` and `description` the bean definition could looks like the following:

```
<bean id="xyFormNormal"
      class="ch.elca.el4j.services.gui.richclient.forms.BeanPropertiesForm"
      singleton="false">
  <property name="shownBeanProperties">
    <list>
      <value>name</value>
      <value>description</value>
    </list>
  </property>
</bean>
```

View the source [here](#).

12.3.9.7 AbstractEditorBeanExecutor

This executor contains a reference to a dialog page that includes the gui elements to edit properties of a bean. By property embedded you have the possibility to get by default a displayable as a separate dialog (`ch.elca.el4j.services.gui.richclient.dialogs.BeanTitledPageApplicationDialog`) or as an in page embedded page component (`ch.elca.el4j.services.gui.richclient.views.DialogPageView` with its descriptor `ch.elca.el4j.services.gui.richclient.views.descriptors.impl.DialogPageViewDescriptor`). By default if multiple bean properties forms are given each bean properties form will be displayed in a tab.

View the source [here](#).

12.3.9.8 AbstractWizardBeanExecutor

This executor contains a reference to a wizard that includes the gui elements to fill the properties of the newly created bean. By default if multiple bean properties forms are given additionally a back and a next button will be displayed to go sequential through the bean properties forms.

View the source [here](#).

12.3.9.9 AbstractBeanPropertiesExecutor

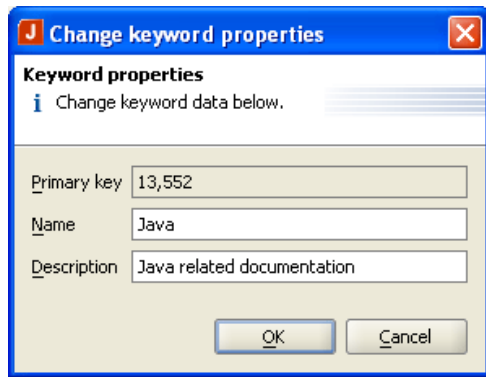
This executor is a convenience class to edit beans of type `ch.elca.el4j.services.persistence.generic.dto.PrimaryKeyObject`. See code from [Reference–Database–Application](#) to see how to use it.

View the source [here](#).

```
<bean id="xyPropertiesExecutor"
      class="ch.elca.myproject.gui.executors.XyPropertiesExecutor"
      singleton="false">
  <property name="id" value="xyProperties"/>
  <property name="beanPropertiesForms">
    <ref local="xyFormAll" />
  </property>
</bean>

<bean id="xyFormAll"
      class="ch.elca.el4j.services.gui.richclient.forms.BeanPropertiesForm"
      singleton="false">
  <property name="shownBeanProperties">
    <!--
      Are the shown bean properties. The order is the same in
      dialog.
    -->
    <list>
      <value>key</value>
      <value>name</value>
      <value>description</value>
    </list>
  </property>
  <property name="readOnlyBeanProperties">
    <!--
      Are shown but read only bean properties.
    -->
    <list>
      <value>key</value>
    </list>
  </property>
</bean>
```

With following spring config above the displayable could look like in following picture.



The appropriate message properties have to be set.

messages.properties (example):

```
xyProperties.title=Change properties of Xy
xyProperties.xyFormAll.title=Xy properties
xyProperties.xyFormAll.description=Change data for Xy below.
```

```
key=Primary key
key.label=&Primary key
key.description=Is the unique key of this object.
name=Name
name.label=&Name
name.description=Name of the object
description=Description
description.label=&Description
description.description=Description of the object
```

The name.* and description.* properties were already used in the bean table model. The key.* property is already defined in messages.properties of **module-springrcp**. Unfortunately it is currently not possible to define a specific property for java bean properties in property executors, but this will be coming soon.

12.3.9.10 AbstractBeanNewExecutor

This executor is a convenience class to create new beans of type `ch.elca.el4j.services.persistence.generic.dto.PrimaryKeyObject`. See code from **Reference-Database-Application** to see how to use it.

View the source [here](#).

```
<bean id="xyCreateNewExecutor"
  class="ch.elca.myproject.gui.executors.XyNewExecutor"
  singleton="false">
  <property name="id" value="xyCreateNew"/>
  <property name="beanPropertiesForms">
    <ref local="xyFormNormal" />
  </property>
</bean>

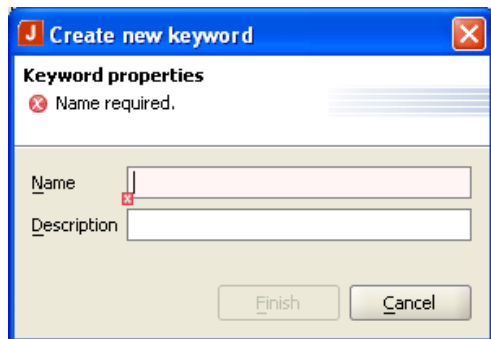
<bean id="xyFormNormal"
  class="ch.elca.el4j.services.gui.richclient.forms.BeanPropertiesForm"
  singleton="false">
  <property name="shownBeanProperties">
    <!--
      Are the shown bean properties. The order is the same in
      dialog.
    -->
    <list>
      <value>name</value>
```

```

        <value>description</value>
    </list>
</property>
</bean>

```

With following spring config above the displayable could look like in following picture.



The appropriate message properties have to be set.

messages.properties (example):

```

xyCreateNew.title=Create new Xy
xyCreateNew.xyFormNormal.title=Xy properties
xyCreateNew.xyFormNormal.description=Fill data for new Xy below.

```

```

name=Name
name.label=&Name
name.description=Name of the object
description=Description
description.label=&Description
description.description=Description of the object

```

The `name.*` and `description.*` properties were already used in the bean table model. Unfortunately, it is currently not possible to define a specific property for java bean properties in wizards but this will be coming soon.

12.3.9.11 AbstractBeanDeleteExecutor

This executor is a convenience class to delete beans of type `ch.elca.el4j.services.persistence.generic.dto.PrimaryKeyObject`. See code from [Reference–Database–Application](#) to see how to use it.

View the source [here](#).

Here a sample spring config:

```

<bean id="xyDeleteExecutor"
      class="ch.elca.myproject.gui.executors.XyDeleteExecutor"
      singleton="false">
    <property name="id" value="xyDelete"/>
</bean>

```

The appropriate message properties have to be set.

messages.properties (example):

```

xyDelete.1.title=Delete Xy
xyDelete.1.message=Are you sure you want to delete the selected Xy?
xyDelete.n.title=Delete Xys

```

```
xyDelete.n.message=Are you sure you want to delete the selected Xys?
```

The message that is taken depends on the number of beans to delete. If only one bean is selected, the 1, otherwise the n message and title will be taken.

12.3.9.12 Conclusion

Do not forget to define beans in spring config as prototype (***singleton="false"***).

12.4 References

- Spring RCP Wiki: <http://opensource.atlassian.com/confluence/spring/display/RCP/Home>
- Spring RCP on SourceForge: <http://www.springframework.org/spring-rcp>
- Demo application module can be found at
<http://svn.sourceforge.net/viewcvs.cgi/el4j/trunk/el4j/framework/apps/refdb/app/gui/>

13 Documentation for module IBatis

13.1 Purpose

Convenience module for IBatis.

13.2 How to use

13.2.1 Dao layer

Just include location ***classpath:scenarios/dataaccess/ibatisSqlMaps.xml*** in spring config location and you have bean ***convenienceSqlMapClientTemplate*** available to have access to IBatis. See **keyword dao** of Reference–Database–Application for example usage.

13.2.2 Type handler callbacks

Type handler callbacks are used to convert database types to java types and vice versa.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE sqlMap
  PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
  "http://ibatis.apache.org/dtd/sql-map-2.dtd">

<sqlMap namespace="refdb-core">
  <typeAlias
    type="ch.elca.el4j.services.persistence.ibatis.callback.BlobToObjectTypeHandlerCallback"
    alias="blobHandler" />

  <resultMap id="file" class="ch.elca.el4j.apps.refdb.dto.FileDto">
    <result property="key" column="KEYID"/>
    <result property="keyToReference" column="KEYTOREFERENCE"/>
    <result property="name" column="NAME"/>
    <result property="mimeType" column="MIMETYPE"/>
    <result property="size" column="CONTENTSIZE"/>
    <result property="content" column="CONTENT" typeHandler="blobHandler"/>
    <result property="optimisticLockingVersion"
      column="OPTIMISTICLOCKINGVERSION"/>
  </resultMap>

  <statement id="getFileByKey" parameterClass="int" resultMap="file">
    select KEYID, KEYTOREFERENCE, NAME, MIMETYPE, CONTENTSIZE, CONTENT,
      OPTIMISTICLOCKINGVERSION from FILES where KEYID=#value#
  </statement>

  <update id="updateFile">
    update FILES set KEYTOREFERENCE=#keyToReference#, NAME=#name#,
      MIMETYPE=#mimeType#, CONTENTSIZE=#size#,
      CONTENT=#content,handler=blobHandler#,
      OPTIMISTICLOCKINGVERSION=OPTIMISTICLOCKINGVERSION+1
    where KEYID=#key#
      and OPTIMISTICLOCKINGVERSION=#optimisticLockingVersion#
  </update>
</sqlMap>
```

In example above we save a serializable java object in a blob and read/deserialize it as/to a java object.

Here the callback type classes:

java type	database type	type callback handler class
-----------	---------------	-----------------------------

java.lang.String	CLOB	com.ibatis.sqlmap.engine.type.ClobTypeHandlerCallback
byte[]	BLOB	com.ibatis.sqlmap.engine.type.BlobTypeHandlerCallback
java.io.Serializable	BLOB	ch.elca.el4j.services.persistence.ibatis.callback.BlobToObjectTypeHandlerCallback

14 Documentation for module Hibernate

14.1 Purpose

Convenience module for Hibernate.

14.2 How to use

Just include `classpath:scenarios/db/hibernateDatabase.xml` and create a new Spring config leaned on `template for session factory bean`.

You can then access Hibernate via the `ConvenienceHibernateTemplate` class. See `keyword dao` of `Reference-Database-Application` for example usage.

14.2.1 Criteria transformation

This module includes a `CriteriaTransformer` class which allows the transformation of EL4J criteria (described in `ModuleCore`) to hibernate criteria. This is useful if you want to use Hibernate to perform search queries which are based on a `QueryObject` object. See `keyword dao` of `Reference-Database-Application` for example usage.

14.2.2 Generic Hibernate repository

This module also contains the `ch.elca.el4j.services.persistence.dao.GenericHibernateRepository` class, a Hibernate-specific implementation of the `ch.elca.el4j.services.persistence.generic.dao.GenericRepository` interface.

14.2.3 Hibernate validation support

This module also contains support for bean validation. Bean validation is performed by specifying invariant constraints on the domain object model using the validation annotations defined by the Hibernate Validator, which is part of the `Hibernate Annotations` project. This is the way we recommend implementing validations on objects. The goal is to define the invariant constraints **only once** (on the domain object model), and to reuse them wherever the object is used.

The `Hibernate Annotations reference documentation` describes how validation constraints are implemented on the domain object model and how domain objects are validated.

Single bean properties can be validated using the pre-defined validation annotations of the Hibernate Annotations project, which check that bean properties respect different constraints (for example the `@NotNull` or the `@Range(min, max)` annotations). In addition to applying these built-in validation constraints, it is possible to perform *custom* bean validation. This can be achieved by adding a custom validation method to the domain class which will be validated and by annotating this method with the `@AssertTrue` annotation. It is the responsibility of this validation method to specify the custom validation constraints for the domain class in which it is defined. In such a method, it is for example possible to verify that different properties of a domain object are consistent between each other. The `@AssertTrue` annotation checks that this validation method evaluates to true, which should only be the case if all the custom constraints defined by that method evaluate to true.

The following example shows the usage of Hibernate's validation support and illustrates how to perform custom bean validation:

```
public class Reference {
```

```

private Date m_documentDate;
private Date m_whenInserted;

@NotNull
public Date getDocumentDate() {
    return m_documentDate;
}

@NotNull
public Date getWhenInserted() {
    return m_whenInserted;
}

/**
 * Checks whether the reference is valid. Should always be true.
 * @return true if the reference is valid, false otherwise
 */
@AssertTrue
public boolean invariant() {
    return (getDocumentDate().getTime() <= getWhenInserted().getTime());
}
}

```

In this example, we define validation constraints on the `Reference` domain object. We use the built-in `@NotNull` annotation to express that both the `documentDate` and the `whenInserted` properties must not be null. Custom validation, which can be used in conjunction with the other validation annotations, is implemented in the bean's `invariant()` method, which ensures that the reference's creation date is an earlier date than its insertion date.

The *validation* of a bean (i.e. the verification of the constraints) is performed as described in section 4.2 of the [Hibernate Annotations reference documentation](#): verification can either take place at the database schema level, or via the Hibernate Validator's built-in Hibernate event listeners, or at the application level. This applies for both the built-in validation annotations and the custom validations defined in a validation method.

The `RefDB` demo application also contains an example illustrating bean validation. The `ch.elca.el4j.apps.refdb.dto.ReferenceDto` domain object is annotated in a similar way as the `Reference` bean in this documentation, and the `ch.elca.el4j.tests.refdb.ReferenceValidationTest` shows how to perform validation at the application level.

15 Documentation for module **XmlMerge**

15.1 Purpose

The XmlMerge module is a pragmatic library to merge XML documents.

15.2 Introduction

The aim of the XmlMerge module is to merge XML documents. Merging means producing a new document out of several source documents. Merging XML documents can be useful in many situations, such as adding modularity to configuration files, deployment descriptors or build files. XMLMerger internally relies on JDOM.

Here is a merge example:

<pre><root> <a> <d id="0"/> <d id="1"/> </root></pre>	+	<pre><root> <a> <c/> <d id="1" newAttr="2"/> </root></pre>	=	<pre><root> <a> <c /> <d id="0" /> <d id="1" newAttr="2" /> </root></pre>
---	---	---	---	--

original

To obtain such a merge, here is the code:

```
public String merge(String original, String patch) {
    Configurer configurer = new PropertyXPathConfigurer("xpath.1=/root/d \n matcher.1=ID");
    return new ConfigurableXmlMerge(configurer).merge(new String[] { original, patch} );
}
```

In the sample above, we configure that for the merging of the part `/root/d` one should use the `ID` matcher.

The design is configurable and extensible in order to fulfill any requirement in the behavior of the merge. The rest of this document explains how to use the module and how to extend it.

Note that the design is focused towards flexibility and extensibility and not performance.

Quick Reading Guide: if you want to get a quick overview read only the following sections:

- How to use
- Original and Patch
- Processing model
- Operations
- Aliases for built-in operations
- Configuring with XPath and Properties

15.3 Module contents

The module contains the following stuff:

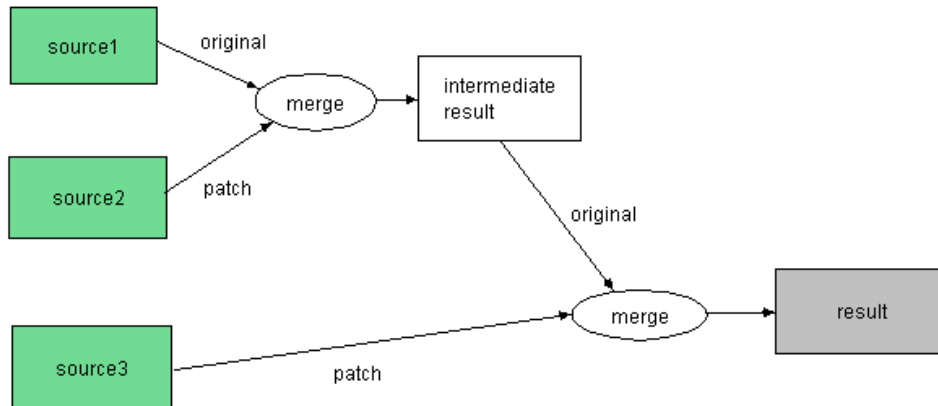
- Interfaces and infrastructure supporting the concepts the module is based on (in fact this forms the API and SPI).
- Default implementations of these interfaces.
- Convenience support for configuring your merge using XPath (outside of the XML documents) or with XML attributes within the XML documents to merge.

- Tool to merge XML files from the command-line.
- Ant task for merging XML files from ant scripts.
- **SpringFramework** resource implementation merging XML files read from other resources.
- Web application to rapidly demonstrate the module.

15.4 Important concepts

15.4.1 Original and Patch

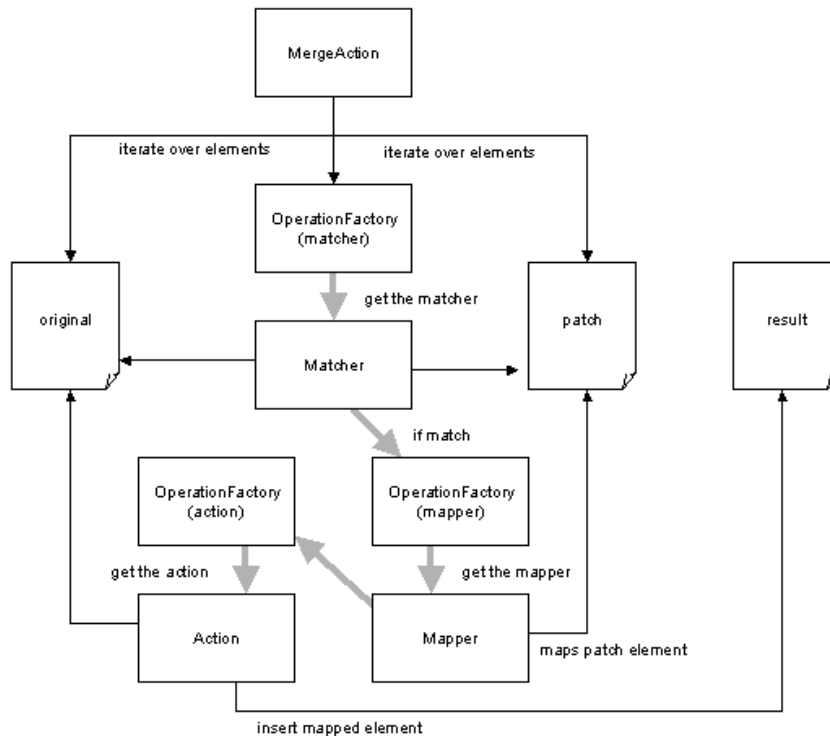
The *sources* are The XML documents (as `java.lang.String`, `java.io.InputStream` or `org.w3c.dom.Document`) that we want to merge.



Several sources can be given, but the merge is always performed two-by-two, using an *original* and a *patch* document. For example, when merging three documents, the *result* of the merge of the two first documents is used as *original* for merging with the third.

15.4.2 Processing model

The natural way of merging documents is to recursively traverse the elements of each *original* and *patch* document and apply the following process to each element.



In the picture above, in the boxes `OperationFactory (matcher)`, `OperationFactory (action)` and `OperationFactory (mapper)` one can plug-in particular implementations. There is a simplification in the picture: *action* works on the parent node, the original node, and the patch element that was already mapped.

15.4.3 Core Concepts as Java Interfaces

See also the *javadocs*.

The XmlMerge infrastructure is based on the following concepts. For each of them, a Java interface is defined in the module.

15.4.3.1 Operations

- **Matcher.** A *matcher* implements a way to compare two XML elements (one of the original document and one of the patch document) and decides if they match.
- **Action.** When two elements match, an *action* is applied on both to produce the *result* element and the *result element* is inserted in the *result* document. An action can also be destructive, i.e. it can insert nothing in the result.
- **Mapper.** Before applying the action, the *patch* element is optionally transformed by a mapper to give it the right form to appear in the *result* document.

NB: If an element of the *original* document does not *match* any *patch* element, it is nevertheless passed to the action with `null` as *patch* element. Respectively, if the *patch* element does not match any *original* element, the action is applied with `null` as *original* element.

15.4.3.2 Configuration with Factories

Used terminology

- **Operation.** Concepts and marker interfaces covering *Matcher*, *Action* and *Mapper* for using factories.
- **OperationFactory.** Provides the corresponding operation for a pair of *original* and *patch* element. The implementation of the factory decides according to its configuration which implementation of the operation must be applied to the pair of elements.
- **MergeAction.** Sub-interface of the **Action** interface. This is the kind of action implementing the traversing of the element's sub-elements and applying the corresponding matcher, mapper and action. Hence, a **MergeAction** is configured by dependency injection with the **OperationFactory** objects providing the mappers, matchers and actions for the sub-elements. Note that a **MergeAction** is also responsible to pass the factories to the merge actions applied to the sub-elements.
- **XmlMerge.** Entry-point to perform the merge, it provides the `merge` methods. One can configure it by injecting the **MergeAction** and **Mapper** applied to the root element.
- **Configurer.** Interface for convenience classes configuring the root merge action and root mapper of an `XmlMerge` instance; thus, it can also configure the operation factories. This way, with only a few lines of code, one can use `XmlMerge`. The **ConfigurableXmlMerge** wrapper class automatically applies a **Configurer** on an `XmlMerge` instance.

15.5 Built-in implementations

15.5.1 Operations

The module provides implementations of operations that are commonly used:

15.5.1.1 Matchers

- **TagMatcher.** The original and patch elements match if the tag name is the same.
- **IdentityMatcher.** The original and patch elements match if the tag name are the same and the *id* attribute value are the same.
- **SkipMatcher.** The original and patch elements never match. Useful to force inserting the elements.

15.5.1.2 Mapper

- **IdentityMapper.** "Do nothing" mapper, it returns an exact copy of the element.
- **NamespaceFilterMapper.** Maps by removing all elements and attributes of a given namespace. Useful with the **AttributeOperationFactory** which allow defining the actions to apply as attributes in the patch document.

15.5.1.3 Actions

- **OrderedMergeAction.** Default merge action. It traverses parallelly the original and patch elements, the matching pairs are determined in the order of traversal. This is generally sufficient for all usage because most of original and patch documents will have elements in the same order.
- **ReplaceAction.** Replaces the original with the patch element or creates the element if not in original.
- **OverrideAction.** Replaces with the patch element only if it exists in the original.
- **CompleteAction.** Copy the patch element only if it does not exist in the original.
- **DeleteAction.** Copy the original element only if it does not exist in the patch. If it exists in the patch, then nothing is added to the result.
- **PreserveAction.** Invariantly copies the original element regardless of the existence of patch element.
- **InsertAction.** Inserts the patch element after elements of the same already existing in the result. Use with the **SkipMatcher** to merge on one level and keep the same relative order of elements.

- ***DtdInsertAction***. Inserts the patch element in the result according to the order specified in the original document's DTD. Use with the ***SkipMatcher*** to merge on one level and make the document valid.

15.5.2 Aliases for Built-In Operations

For convenience in configuration, the built-in operations have short aliases, so that we can refer to them using the aliases instead of the full class names:

- **Matchers**: TAG, ID, SKIP.
- **Mappers**: IDENTITY
- **Actions**: MERGE, REPLACE, OVERRIDE, COMPLETE, PRESERVE, INSERT, DTD.

These constants are defined in the classes ***StandardMatchers***, ***StandardMappers***, ***StandardActions***.

15.5.3 XmlMerge Implementation

The `DefaultXmlMerge` class applies the `OrderedMergeAction`, `TagMatcher` and `IdentityMapper` to all elements.

15.5.4 Operation Factories

Three implementations of the operation factory are provided:

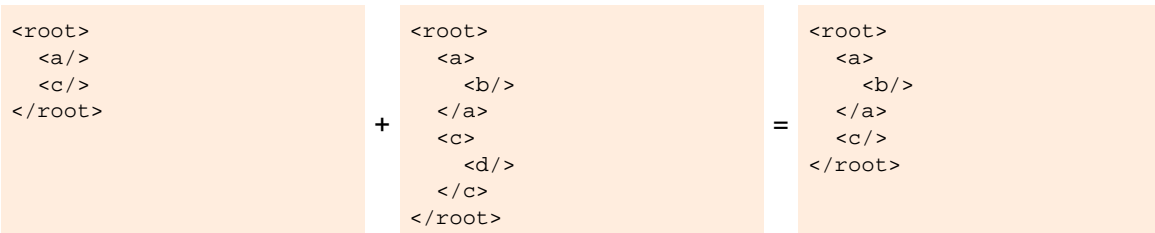
- ***StaticOperationFactory***. Returns the same operation for all element pairs. Used when the same behaviour applies to all elements of the document.
- ***XPathOperationFactory***. Configured with a map of {XPath, Operation}, it returns the operation of the first XPath matching the element path.
- ***AttributesOperationFactory***. Configured with attributes in the patch element.

15.6 Configuring your Merge

You have currently three ways to configure an `XmlMerge` instance:

15.6.1 Programming the Configuration

This is the most powerful but tedious way to configure. You create the instances of the root merge action, root mapper and factories programmatically. Example:



original

```

public void testXPathOperationFactory() throws Exception {

    String[] sources = {
        "<root><a/><c/></root>",
        "<root><a><b/></a><c><d/></c></root>" };

    XmlMerge xmlMerge= new DefaultXmlMerge();

```

```

MergeAction mergeAction = new OrderedMergeAction();

XPathOperationFactory factory = new XPathOperationFactory();
factory.setDefaultOperation(new CompleteAction());

Map map = new LinkedHashMap();
map.put("/root/a", new OrderedMergeAction());

factory.setOperationMap(map);

mergeAction.setActionFactory(factory);

xmlMerge.setRootMergeAction(mergeAction);

String result = xmlMerge.merge(sources);

String expected =
"<?xml version=\"1.0\" encoding=\"UTF-8\"?>" + NL +
"<root>" + NL +
"  <a>" + NL +
"    <b />" + NL +
"  </a>" + NL +
"  <c />" + NL +
"</root>";

assertEquals(expected.trim(), result.trim());
}

```

Note that this kind of configuration can be interesting in conjunction with the [SpringFramework](#), since these components can be configured in Spring configuration files.

15.6.2 Configuring with XPath and Properties

The most usual way is to configure XmlMerge with the **PropertyXPathConfigurer** which uses a Properties object.

The properties define XPath entries and the associated matchers, mappers and actions. Syntax:

```

xpath.pathName=XPath

matcher.pathName=Matcher alias or class name mapper.pathName=Mapper
alias or class name action.pathName=Action alias or class name

```

By default, the **OrderedMergeAction**, **IdentityMapper** and **TagMatcher** is used for all elements.

Example:

test.properties:

```

action.default=COMPLETE

xpath.path1=/root/a
action.path1=MERGE

```

```

<root>
  <a/>
  <c/>
</root>

```

+

```

<root>
  <a>
    <b/>
  </a>
  <c>
    <d/>
  </c>
</root>

```

=

```

<root>
  <a>
    <b/>
  </a>
  <c/>
</root>

```

original

```
public void testPropertyXPathConfigurer() throws Exception {

    String[] sources = {
        "<root><a/><c/></root>",
        "<root><a><b/></a><c><d/></c></root>" };

    Properties props = new Properties();
    props.load(getClass().getResourceAsStream("test.properties"));
    Configurer configurer = new PropertyXPathConfigurer(props);
    XmlMerge xmlMerge= new ConfigurableXmlMerge(configurer);

    String result = xmlMerge.merge(sources);

    String expected =
        "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" + NL +
        "<root>" + NL +
        "  <a>" + NL +
        "    <b />" + NL +
        "  </a>" + NL +
        "  <c />" + NL +
        "</root>";

    assertEquals(expected.trim(), result.trim());
}
```

15.6.3 Configuring with Inline Attributes in Patch Document

Another way, to avoid using external `Properties` and show explicitly the merge behavior in the patch document, is to use the ***AttributeMergeConfigurer***.

You simply add attributes with a special namespace in the patch elements describing the operations to apply. Example:

<pre><root> <a> <d/> <e id='1' /> <e id='2' /> </root></pre>	+	<pre><root xmlns:merge='http://xmlmerge.el4j.elca.ch'> <a merge:action='replace'>hello <c/> <d merge:action='delete' /> <e id='2' newAttr='3' merge:matcher='ID' /> </root></pre>	=	<pre><root> <a>hello <c /> <e id="1" /> <e id="2" newAttr="3" /> </root></pre>
--	---	---	---	--

original

```
public void testAttributeMerge() throws Exception {

    String[] sources = {

        "<root>" +
        "  <a>" +
        "    <b/>" +
        "  </a>" +
        "  <d/>" +
        "  <e id='1' />" +
        "  <e id='2' />" +
        "</root>" +
        "<root xmlns:merge='http://xmlmerge.el4j.elca.ch'>" +
        "  <a merge:action='replace'>hello</a>" +
        "  <c/>" +
        "  <d merge:action='delete' />" +
        "  <e id='2' newAttr='3' merge:matcher='ID' />" +
        "</root>"

    };

    Properties props = new Properties();
    props.load(getClass().getResourceAsStream("test.properties"));
    Configurer configurer = new AttributeMergeConfigurer(props);
    XmlMerge xmlMerge= new ConfigurableXmlMerge(configurer);

    String result = xmlMerge.merge(sources);

    String expected =
        "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" + NL +
        "<root>" + NL +
        "  <a>" + NL +
        "    <b />" + NL +
        "  </a>" + NL +
        "  <c />" + NL +
        "  <e id=\"1\" />" + NL +
        "  <e id=\"2\" newAttr=\"3\" />" + NL +
        "</root>";

    assertEquals(expected.trim(), result.trim());
}
```

```

        "</root>"
    };

    String result = new ConfigurableXmlMerge(new AttributeMergeConfigurer()).merge(sources);

    String expected =
        "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" + NL +
        "<root>" + NL +
        "  <a>hello</a>" + NL +
        "  <c />" + NL +
        "  <e id=\"1\" />" + NL +
        "  <e id=\"2\" newAttr=\"3\" />" + NL +
        "</root>";

    assertEquals(expected.trim(), result.trim());
}

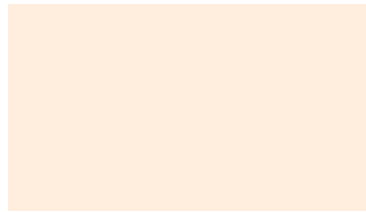
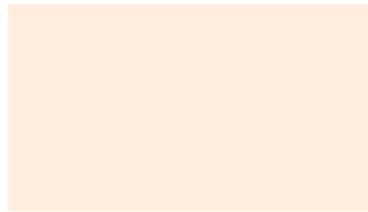
```

15.7 Writing your own Operations

It is easy to customize and extend the behavior of the XmlMerge module by writing new operations.

For example, one may want to merge `web.xml` files. To add a new parameter to an existing servlet, we must match the right servlet entry, thus match using the tag `<servlet-name>`. See below an example of a new **Matcher** implementation, the **ServletNameMatcher**.

<pre> <web-app> <servlet> <servlet-name> hello </servlet-name> <servlet-class> test.HelloServlet </servlet-class> </servlet> <servlet> <servlet-name> bye </servlet-name> <servlet-class> test.ByeServlet </servlet-class> </servlet> <servlet-mapping> <servlet-name> hello </servlet-name> <url-pattern> /hello </url-pattern> </servlet-mapping> <servlet-mapping> <servlet-name> bye </servlet-name> <url-pattern> /bye </url-pattern> </servlet-mapping> </web-app> </pre>	+	<pre> <web-app> <servlet> <servlet-name> bye </servlet-name> <init-param> <param-name> message </param-name> <param-value> Bye bye! </param-value> </init-param> </servlet> </web-app> </pre>	=	<pre> <web-app> <servlet> <servlet-name> hello </servlet-name> <servlet-class> test.HelloServlet </servlet-class> </servlet> <servlet> <servlet-name> bye </servlet-name> <servlet-class> test.ByeServlet </servlet-class> <init-param> <param-name> message </param-name> <param-value> Bye bye! </param-value> </init-param> </servlet> <servlet-mapping> <servlet-name> hello </servlet-name> <url-pattern> /hello </url-pattern> </servlet-mapping> <servlet-mapping> <servlet-name> </pre>
--	---	---	---	--



```

    by
  </servlet-name>
  <url-pattern>
    /by
  </url-pattern>
</servlet-mapping>
</web-app>

```

original

Ensure your ***ServletMatcherClass*** is in the classpath and configure it in the XPath properties:

```

xpath.path1=/web-app/servlet
matcher.path1=com.mycompany.ServletNameMatcher

# Do not touch the existing name
xpath.path2=/web-app/servlet/servlet-name
action.path2=PRESERVE

# Do not touch existing init-params
xpath.path3=/web-app/servlet/init-param
action.path3=INSERT

```

ServletNameMatcher implementation:

```

package com.mycompany;

public class ServletNameMatcher implements Matcher {

    public boolean matches(Element originalElement, Element patchElement) {
        String originalServletName = originalElement.getChildText("servlet-name");
        String patchServletName = patchElement.getChildText("servlet-name");

        return patchServletName != null && originalServletName != null &&
            originalServletName.trim().equals(patchServletName.trim());
    }
}

```

15.8 How to use

This section shows the different possibilities how this module can be used.

15.8.1 Command-line Tool

The module includes a tool to merge XML files from the command-line.

To be able to use the command-line tool, you have to execute the following steps:

- Go to `EL4J_HOME/framework`
- Recursively compile all required targets files: `ant jars.rec.module.module-xml_merge`
- Create an executable distribution of the `xml_merge` module:
`create.distribution.module.eu.module-xml_merge.console`
- The executable distribution can be found in the `module-xml_merge-default` folder under `EL4J_HOME/framework/dist/distribution`. You can copy this folder to any location you want.
- To be able to execute the command-line tool from your desired location, you have to add the location containing the executable distribution your `PATH` environment variable:

- ◆ **Windows:** add `YOUR_LOCATION\module-xml_merge-default` to the right end of your `PATH` environment variable, where `YOUR_LOCATION` denotes the folder into which you have copied the `module-xml_merge-default` folder.
- ◆ **Unix:** launch the following command to set the `PATH` environment variable, where `YOUR_LOCATION` denotes the folder into which you have copied the `module-xml_merge-default` folder: `export PATH=$PATH: "YOUR_LOCATION/module-xml_merge-default"`

The previous steps have to be executed only once. You are now ready to launch the command-line tool from any location by launching the `xmlmerge` script:

```
xmlmerge [-config <config-file>] file1 file2 [file3 ...]
```

In this command, `config-file` denotes an optional XPath property file and `file1`, `file2`, `file3` etc are the xml files to merge. The result is outputted on the standard output.

15.8.2 Ant Task

The module also includes an Ant task for merging XML files from ant scripts.

Here is an example which shows the usage of this Ant task in a `build.xml` file:

```
<target name="test-task">
  <taskdef name="xmlmerge" classname="ch.elca.el4j.xmlmerge.anttask.XmlMergeTask"
    classpath="module-xml_merge.jar;jdom.jar;jaxen.jar;saxpath.jar"/>

  <xmlmerge dest="out.xml" conf="test.properties">
    <fileset dir="test">
      <include name="source*.xml"/>
    </fileset>
  </xmlmerge>
</target>
```

In this task, `dest` denotes the output merged file, and `conf` denotes an optional XPath property file. The indicated fileset selects the files to merge (in this example, the files in the `test` directory whose name begins with `source` will be merged).

The jar files which are needed on the classpath to execute this task are `module-xml_merge.jar`, `jdom.jar`, `jaxen.jar` and `saxpath.jar`. The `module-xml_merge.jar` file can be found in the `EL4J_HOME/framework/dist/lib` folder, and the three other ones can be found in the `EL4J_HOME/framework/lib` folder. If you have created an executable distribution of the `xml_merge` module (see [command-line tool](#)), you can also find these libraries in the `lib` folder of the executable distribution.

15.8.3 Spring Resource

You can also use this module to create an XML Spring Resource on-the-fly by merging XML documents read from other resources. Here is a configuration example:

```
<bean name="merged" class="ch.elca.el4j.xmlmerge.springframework.XmlMergeResource">
  <property name="resources">
    <list>
      <bean class="org.springframework.core.io.ClassPathResource">
        <constructor-arg>
          <value>ch/elca/el4j/xmlmerge/r1.xml</value>
        </constructor-arg>
      </bean>
      <bean class="org.springframework.core.io.ClassPathResource">
        <constructor-arg>
          <value>ch/elca/el4j/xmlmerge/r2.xml</value>
        </constructor-arg>
      </bean>
    </list>
  </property>
</bean>
```

```
        </constructor-arg>
    </bean>
</list>
</property>
<property name="properties">
    <map>
        <entry key="action.default" value="COMPLETE" />
        <entry key="xpath.path1" value="/root/a"/>
        <entry key="action.path1" value="MERGE" />
    </map>
</property>
</bean>
```

This configuration example is also part of the module and can be found in the `conf/template/xmlmerge-config.xml` file.

15.8.4 Web demo

The module also contains a web application to demonstrate how XML documents can be merged.

To be able to launch the web application, you have to execute the following steps:

- Go to `EL4J_HOME/framework`
- Recursively compile all required targets files: `ant jars.rec.module.module-xml_merge`
- Deploy the demo application into Tomcat: `ant deploy.war.module.eu.module-xml_merge.web`
- Open in <http://localhost:8080/xmlmerge/demo> a browser.

15.9 References

* Analysis about general merging of XML (shows that the "perfect XML merge is highly complex and that a pragmatic approach seems reasonable): <http://www.cs.hut.fi/~ctl/3dm/thesis.pdf>

16 Documentation for module Web Test

16.1 Purpose

The module Web Test includes all necessary libraries to test Web Applications using **JWebUnit** or **HtmlUnit**.

16.2 Overview of our webtests

Please refer to the short presentation about web tests in the following ppt presentation:
http://leaffy.elca.ch/java/el4j/Marketing_Mirror/MonthlyNews/NewEL4JFeaturesJune06.ppt

16.3 Example

```
// Import
import net.sourceforge.jwebunit.TestingEngineRegistry;
import net.sourceforge.jwebunit.WebTestCase;

// Test Class
public class JWebUnitExampleTest extends WebTestCase {

    // set up, choose test engine and base URL
    public void setUp() {
        setTestingEngineKey(TestingEngineRegistry.TESTING_ENGINE_HTMLUNIT);
        getTestContext().setBaseUrl("http://www.elca.ch");
    }

    // describe tests
    public void testElcaURL() throws Exception {
        beginAt("/");

        // check title
        assertEquals("ELCA: Technology-Consulting-Innovation: " +
            "job opportunities: careers: Informatiker: software engineer: " +
            "Programmierer: jobs: software design: business strategy: crm: " +
            "web design: data warehouse: edm");

        // check if text is present
        assertTextPresent("ELCA is a leading Swiss IT solutions provider");

        // check if link is present
        assertLinkPresentWithText("Newsletter");
    }
}
```

17 Documentation for module TcpForwarder

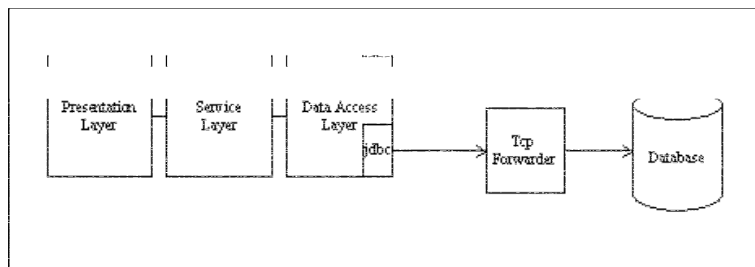
17.1 Purpose

The TcpForwarder module helps to test network failures or connection problems at runtime by controlling TCP connections that can be switched on / off between the source and the destination.

17.2 Important concepts

The TCP forwarder represents a service intended to forward TCP traffic directed to a specific port. The TCP forwarder controls connections between a source and a destination and is able to switch them on / off on demand (either using a simple user interface or programmatically). Therefore, the original application has to switch its destination port, e.g. the port connected to a database, to the input port of the TCP forwarder.

The following picture shows how TCP traffic is forwarded from an application's data access layer to a database:



17.3 How to use

17.3.1 Command line user interface to switch TCP connections on or off

EL4J provides a simple user interface to switch on/off TCP connections called *TCPForwarderTool*.

17.3.1.1 Parameters (tbd in code)

- Input Port: The TCP forwarder's input port – your application has to switch its destination port to this port to be able to use the TCP forwarder
- Destination Port: The TCP forwarder's target port, which has to be your application's original destination port
- Destination URL (optional)

After startup, the input and destination Ports are connected: the TCP forwarder listens on the input port and forwards all traffic to the destination port.

17.3.1.2 Commands

Once started, you can control the TCP forwarder using console commands:

- '1' to unplug the connection between input port and destination port
- '2' to restore the connection between input port and destination port
- '3' to exit

17.3.1.3 Notes

- Don't forget to switch the destination port of your application to the input port of your TCP forwarder.

17.3.2 Programmatically halting and resuming network connectivity

It is also possible to programmatically control network connectivity by using the TCP forwarder's elementary functions directly in code. An example is presented in the automated tests (using JUnit or **WebTests** with **JWebUnit**) of the *tcp_forwarder-tests* module.

17.3.2.1 Code configuration

17.3.2.1.1 Import libraries:

```
import java.net.Inet4Address;
import java.net.InetSocketAddress;
import java.net.SocketAddress;

import ch.elca.el4j.tcpred.TcpInterruptor;

// only needed for JWebUnit tests
import net.sourceforge.jwebunit.TestingEngineRegistry;
import net.sourceforge.jwebunit.WebTestCase;
```

17.3.2.1.2 Set up TCP forwarder:

- Forwarding from INPUT_PORT to DEST_PORT:

```
TcpInterruptor ti = new TcpInterruptor(INPUT_PORT, DEST_PORT);
```

- Forwarding from INPUT_PORT to DEST_URL:DEST_PORT:

```
SocketAddress target = new InetSocketAddress(Inet4Address.getByName(DEST_URL), DEST_PORT);
TcpInterruptor ti = new TcpInterruptor(INPUT_PORT, target);
```

17.3.2.2 Switch on / off connections

- Cut a connection: `ti.unplug()` ;
- Restore a connection: `ti.plug()` ;

17.4 Demonstration code

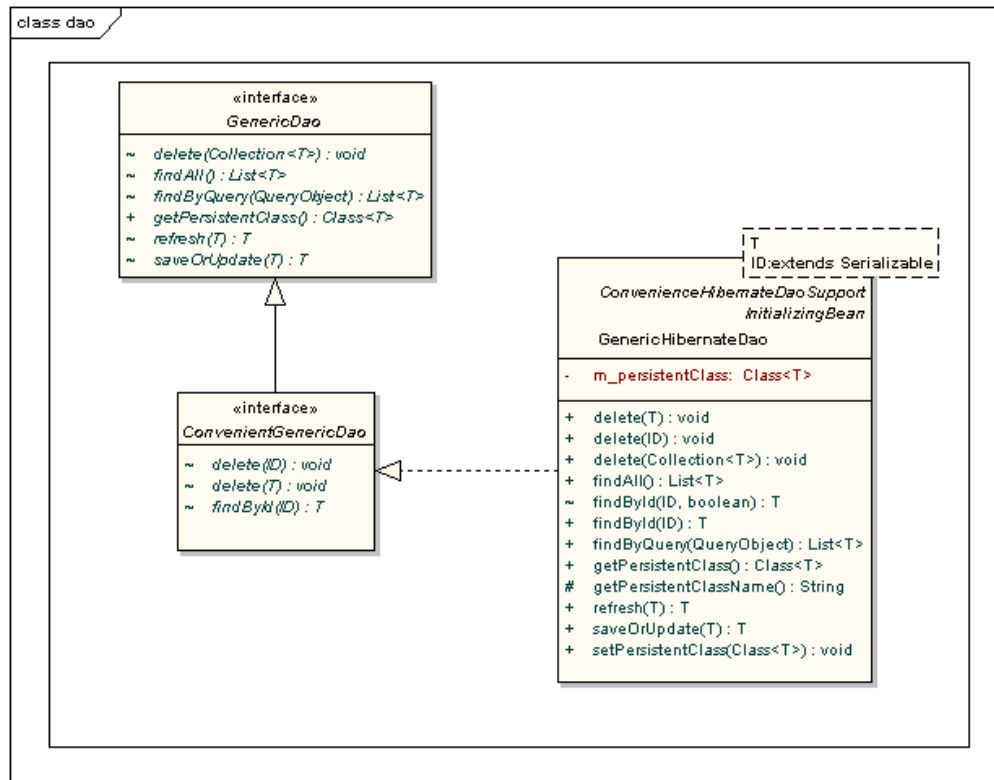
Please refer to the tests for this module here:

http://svn.sourceforge.net/viewvc/el4j/trunk/el4j/framework/tests/tcp_forwarder/java/ch/elca/el4j/tcpred/tests/TestDB.java

18 Generic DAOs in EL4J

18.1 Basic introduction

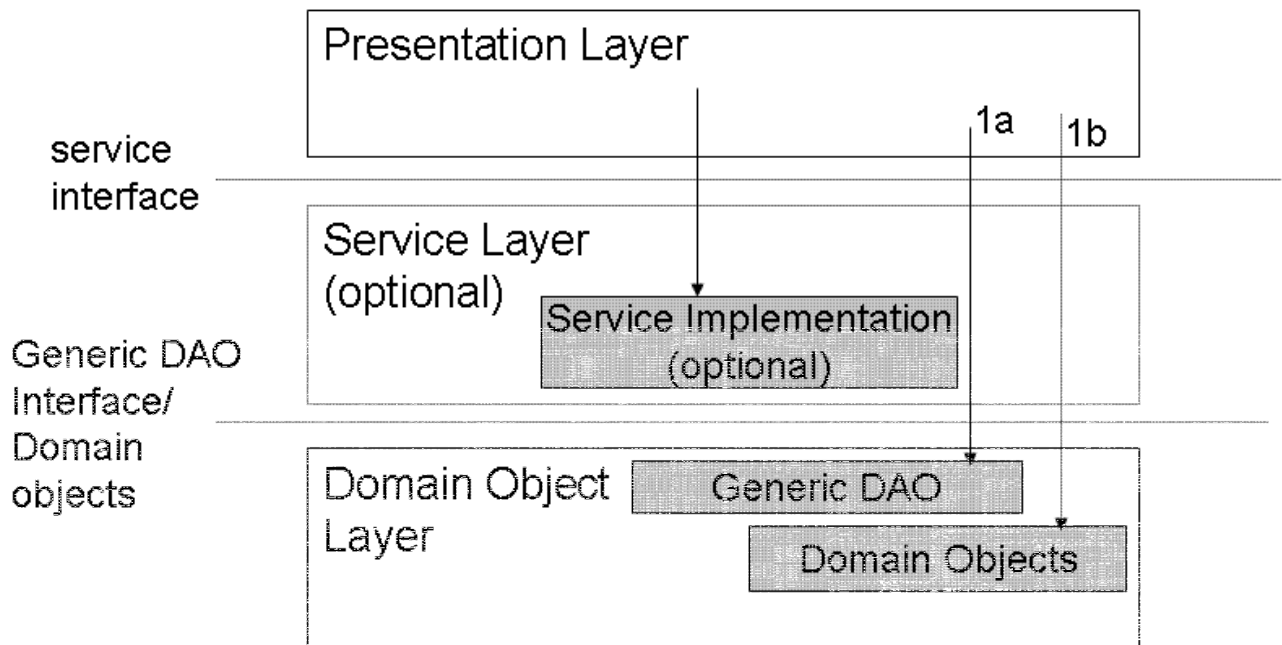
When writing DAOs (Data Access Objects), the basic CRUD (Create, Read, Update, Delete) operations tend to be more or less similar. The goal of the GenericDAO of EL4J is to eliminate these repetitive steps. The following class diagram illustrate this:



A user can use the **ConvenientGenericDao** interface to access a database. It already provides the CRUD operations. The class **GenericHibernateDao** implements this interface for Hibernate (there is also an implementation for iBatis). When custom DAO-methods are required, they can be added in a subclass of the **ConvenientGenericDao** (these custom methods are then implemented by hand). (It is also possible to use the canonical **GenericDao**, but we provide it mainly for internal use.)

18.2 Sometimes we can omit or bypass the service layer

The approach has the following reference architecture (the horizontal, dashed lines indicate the interfaces between layers):



We use the three "traditional" layers one uses typically. But there are some changes:

1. The Service Layer is *optional*. One can bypass the service layer in case the functionality is data-centered (CRUD-operations on a domain object model). The service layer may be useful for complex business functionality or for operations that involve many domain objects. It is also possible that one mainly works on the domain object model and only uses the service layer for a small part of more complex processings.
2. Instead of traditional DAO interfaces to find and store domain objects we use a *GenericDao* object (there is (at the moment) one *GenericDao* class per Entity). The standard find and store interface is generic (no hand-coding involved): There is no need to write the basic interface by hand. The latter also simplifies to bind (=attach) normal data access into the presentation layer. For particular functionality, one can *extend* the generic DAO interface.
3. The domain object model (normal POJOs) is annotated (with JDK 1.5 annotations) in order to add constraints/ additional data about the model only once (on the model). Sample constraints include: design by contract rules, validation information, mapping to database (JPA) or XML (JAXB), ... These annotations are then consumed by various tools that work with the domain object model (UI-validation, O/R-mapping, O/X-mapping, UI-rendering, ...). The domain object model is made up of *Entities* in the DDD terminology. The Entities in the domain object model are not just a "dumb" holder of data: it contains methods for "real" processings of the domain. When working on the domain layer, one either I) finds Entities via the *GenericDao* first and then works on these Entities (1a and 1b) and stores them again or II) one creates Entities (via *new*), works on them and stores them via the *GenericDao*.

Please refer also to the Annotation Cheat Sheets.

18.3 Benefits of the approach

- Less duplication & cleaner code: you concentrate on the essential
- No code generation needed
- For the benefits of the DDD, we refer to the referenced book
- For data-centric applications you avoid to have a mostly delegating service layer
- Providing information such as how to validate objects on the domain model and having the domain model everywhere available helps to avoid duplication of such information.

18.4 References

- Where we have the original idea from <http://www.hibernate.org/328.html>
- A description of a similar implementation
<http://www-128.ibm.com/developerworks/java/library/j-genericdao.html>
- Domain driven design (DDD) book: summary <http://www.domaindrivendesign.org/>

19 Exception handling guidelines

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

19.1 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

19.2 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.

19.2.1 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.

19.2.2 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.)

19.3 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`.

19.4 Handling exceptions

19.4.1 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`.

19.4.2 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.

19.4.3 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 loose information). The Exception class of JDK 1.4 provides support for this.

19.5 Related useful concepts and hints

19.5.1 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.)

19.5.2 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.

19.5.3 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 to the [AssertionUsageGuidelines](#) for more details about other types of design by contract support.

19.5.4 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.

19.5.5 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 hierarchy](#). To profit from this hierarchy, 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`.

19.5.6 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.

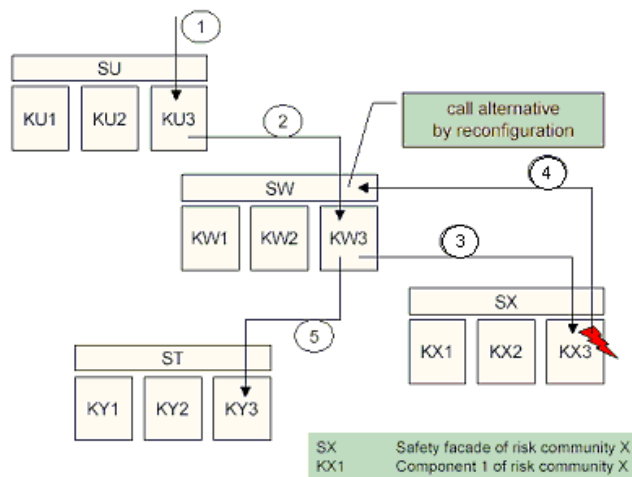
19.5.7 SafetyFacade pattern

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 .

19.6 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.

19.7 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.
- EL4J `HighLevelExceptionHandlerGuidelines`

20 Acknowledgments

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

- Raphael Boog
- Simon Börlin
- Laurent Bovet
- Andi Bur
- Christian Gasser
- Adrian Häfeli
- Jacques–Olivier Haenni
- Marc Lehmann
- Yves Martin
- Alex Mathey
- Martin Meier
- Adrian Moos
- Philipp H. Oser
- Andreas Pfenninger
- Stefan Pleisch
- Jean–François Poilpret
- Nicola Schiper
- Marc Schmid
- Christoph Schwitter
- Florian Süss
- Michael Vorburger
- Rachid Waraich
- Martin Zeltner
- Martin Zingg

Thank you!

21 References

- EL4J Website, <http://el4j.sourceforge.net/>
- Commercial EL4J Website, http://www.elca.ch/Solutions/Technology_Frameworks/EL4J/EL4J.php
- 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