Incremental improvements for the Spring Framework

I am working as an architect for a middle-sized software development company, where we have been actively using J2EE extension frameworks for the last 6 years. Our experience is that such frameworks greatly simplify development, improve homogeneity across applications, and make them more agile and maintainable over time.

Our first J2EE framework was proprietary (there was nothing like Spring around when we established it). We used it to build more than 20 applications and were happy with the results of adding an enhancement layer on top of J2EE. More than a year ago, we have switched to an open and light J2EE framework based on the popular Spring Java Framework (http://www.springframework.org/).
Under the name of EL4J, this framework has already been used in 10+ projects within our company. We have now open-sourced it and are ready to contribute significant parts of it to the Spring framework.

EL4J complements Spring in 2 main areas: code module support and remoting extensions. The next 2 sections discuss them in more detail, the following section discusses various other extensions.

(I) Splitting applications in modules

Modularity belongs to the basic architectural best practices. Our module support helps to split your work in smaller sub-parts in order to reduce complexity, to simplify separate development and to decrease code size by using only what is needed. It simplifies using the many emerging Spring extensions: their packaging is unified and the default usage of new features is much easier.

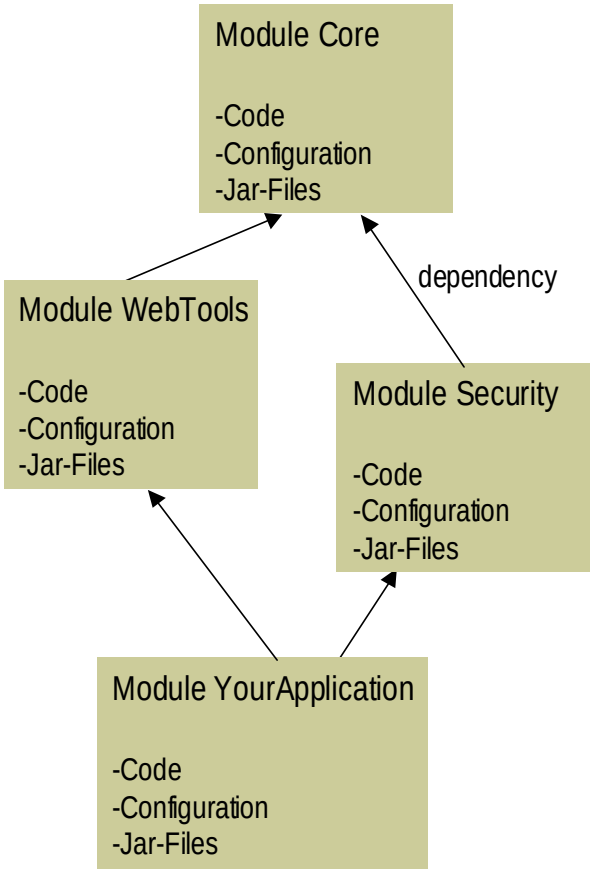

Figure 1: Four sample modules (each with code, configuration and jar-files) with their dependencies. The module YourApplication uses the WebTools and the Security modules (and implicitly the Core module).

For example, to use our JMX module (it publishes all available Spring Beans and their configuration in an Application Context under JMX), one simply adds a `<dependency module="module-jmx"/>` to the module description of the application: all default configurations and CLASSPATH settings are set up automatically.

In EL4J, a module is just a directory and a short module description. Each module can contain (1) code, (2) configuration data and (3) dependencies to other modules and jar files. Dependencies between modules are transitive. If A has a dependency on B and B has a dependency on C, implicitly A has also a dependency on C.

In addition to the gained modularity, modules have these additional benefits:

- **Default configuration for modules**: with spring, configuration can sometimes become complicated. We provide a flexible way for default spring configurations for 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 3 elements:

- a module abstraction at the level of the build system (EL4J currently provides its own build system, but we are looking into support for Maven, too)
- a convention on how to organize configuration information within each module
- the ModuleApplicationContext, which is a thin wrapper for a standard Spring application context
-

The feature to split code in modules is directly based onEL4Ant (http://el4ant.sf.net), a light, Ant-based build system that is very extensible. It generates a standard build.xml file from a description of a project. The typical Ant targets can be launched for all modules or only a single module.

The **convention to organize configuration** helps to indicate what configuration should be automatically loaded when a module is active (this can easily be overridden). One can also define different configuration scenarios among which one must be chosen. A sample scenario could be the choice of whether a module runs in a client or a server (e.g. for remoting or security), or what data access technology to use (e.g. iBatis or Hibernate).
The configuration file organization of a module are saved under a folder '/conf'. This '/conf' folder is divided into different subfolders:

- '/mandatory': Here are all the XML and the property files which should by default be 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. Exactly one XML-file (=scenario) of each directory (= facet) must be chosen. All possible combinations of the scenarios should work.

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

EL4J's ModuleApplicationContext is a light wrapper around the existing application contexts of Spring. Its use is optional, but it provides valuable features:

- it finds all configuration files present in the modules, even if some J2EE-container (e.g. WLS) present them differently to Spring
- it solves issues with the order of loading configuration files in some J2EE-containers
- it allows exluding configuration files that should not be loaded

- it allows publishing all its Spring beans with their configuration (publication is possible e.g. to JMX).

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

Here is how this configuration mechanism can be used:

```
String[] configurationFiles =
 {"classpath*:mandatory/*.xml",  //by default, load all mandatory files
  "scenarios/authentication/stateless.xml", // choose stateless scenario
  "scenarios/logincontext/db.xml",          //choose database login ctxt
  "scenarios/securityscope/local.xml"};     //choose local security

ApplicationContext m_ac =
 new ModuleApplicationContext
  (configurationFiles,                     // list of included config files
   "classpath*:mandatory/jmx-appender.xml");//list of excluded config files
```

This code loads the files from the mandatory directory of all modules the current module depends on (as EL4Ant puts these modules automatically in the CLASSPATH, the expression "classpath*:mandatory/*.xml" finds all those files). In addition, it selects the appropriate scenarios from the security module. It excludes the `jmx-appender.xml` configuration file from the default configuration of a module.
These three mechanisms do not solve all cases, but they significantly reduce the complexity of common configurations.

(II) Enhanced remoting

Remoting is the mechanism that allows invoking methods over process and machine boundaries. Like Spring, we believe that one should be able to rapidly remote POJOs via configuration only. However, we would like to avoid hand-coding EJB session beans to delegate calls to Spring Beans. In our opinion, it should be **just a configuration choice whether to remote via EJBs or via another protocol**.

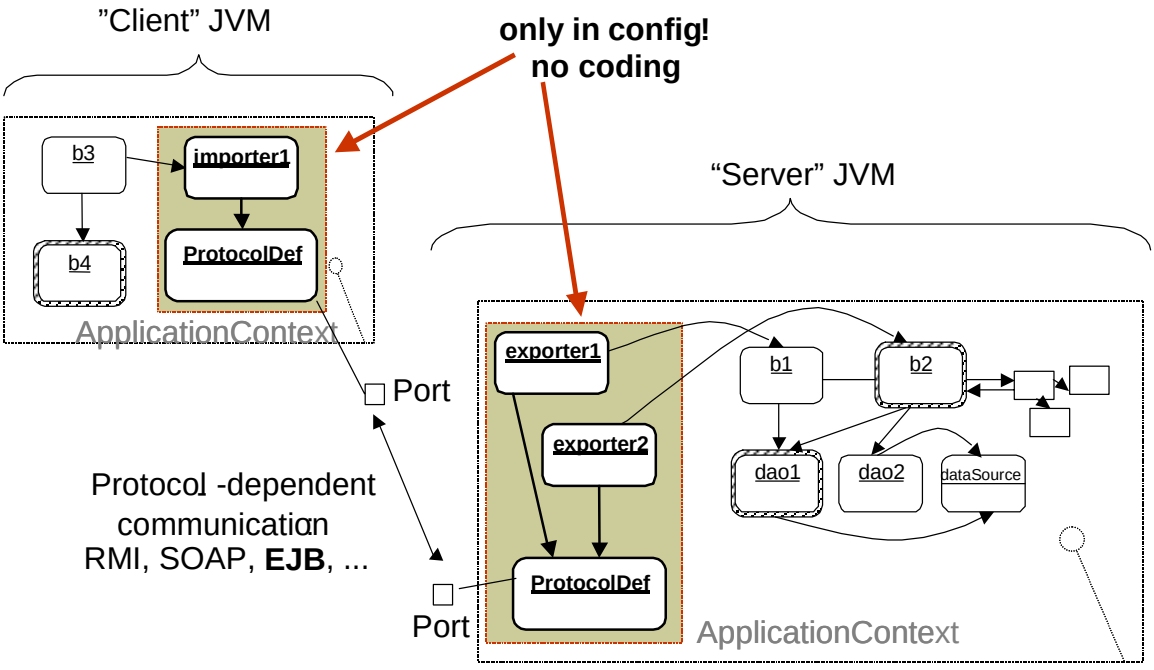

Figure 2: Remoting does not require any particular coding. Configuration suffices.

EL4J's remoting is based on the remoting of Spring. It can in addition generate EJB session bean façades to delegate work to POJOs. We generate a XDoclet-annotated EJB bean and generate the EJB artefacts with XDoclet. In case the interface is not RMI-compliant (for example due to lacking RemoteExceptions), we adapt it accordingly. The user can keep working with his interface of the POJO.

The support for POJO-remoting with SOAP is also improved, as one configures just the Java-XML mappings for the complex types. No other deployment descriptors are needed.

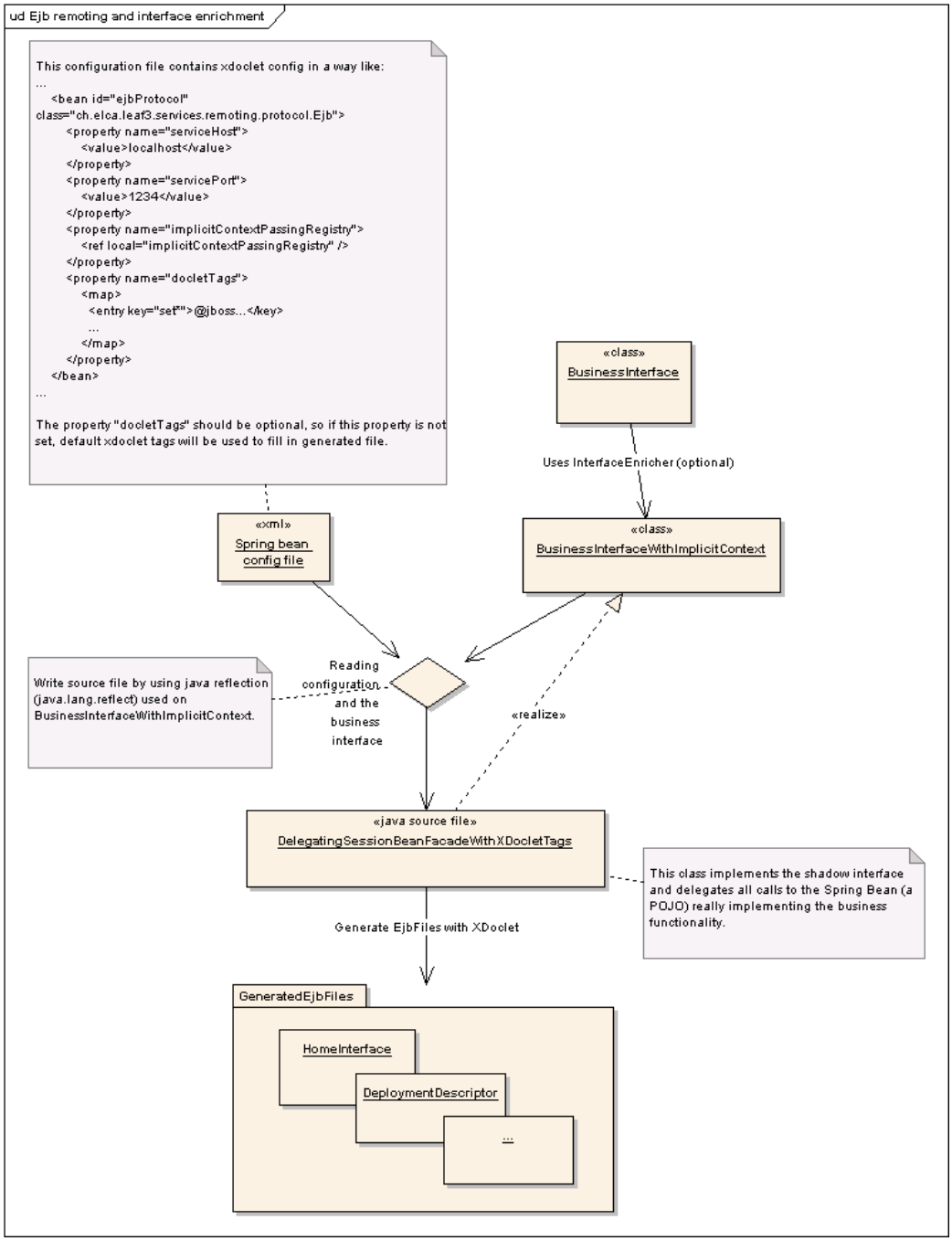

Figure 3: How to automatically remote POJOs as EJB beans (Notation: Mixed UML and workflow).

Optionally the EL4J-remoting can provide **implicit context passing** when making remote calls: this allows adding additional technical information with each remote invocation. What is the interest of this? You could for instance add the security principal to all requests, add an ID to indicate on behalf of what company one executes a transaction, or add other client-context with the call. And contrary to many other approaches for this, you still keep a clean Java interface: no need to tunnel your request through an `Object invoke(Method,Object[])`!

It's easy to implement implicit context passing 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 (RMI, EJB or RMI-IIOP do not support it). Our solution is to add the implicit context in the form of a Map as the last argument of each method. Behind the existing interface, we add transparently a *shadow interface* that has the additional parameter added. The addition of this shadow interface is

normally hidden during runtime. It uses a mechanism similar to CGLIB and it could even be made during build time to avoid concerns with interfaces generated during runtime. Implicit context is only included in the calling direction of calls, not in the response. (Rationale: implicit context returned to the caller can typically be returned explicitly, and it would be difficult to define reasonable semantics for asynchronous invocations.)

(III) A selection of other features of EL4J

Other features include:

- A **daemon manager** can be used to execute multiple daemons inside one JVM. A daemon is a supervised, typically long-running thread. The daemon manager is a light container that can be easily published as an operating system service.
- The JMX module provides an **automatic (implicit) view of the currently loaded spring beans and** their **configurations**, which helps developers to understand applications. It becomes even more interesting as Spring (and EL4J) allow splitting configuration information in many files, making it sometimes hard to figure out what configuration actually applies.

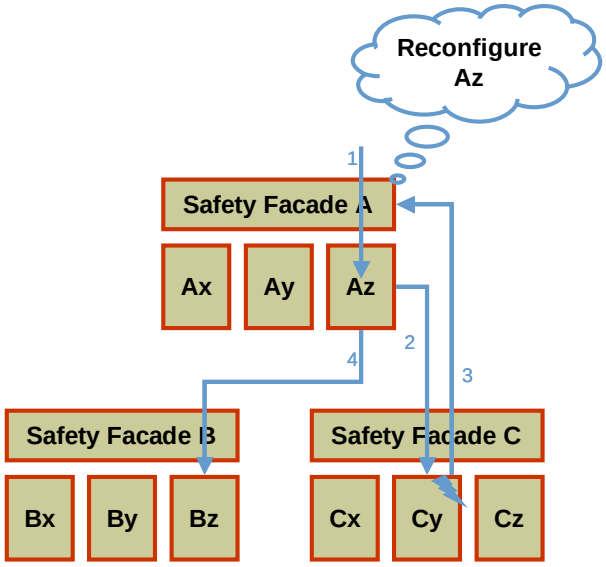

Figure 4: A safety facade can exchange references to components after a technical exception occurs.

- The configurable **exception handlers** allow separating the technical (unchecked) 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.

Conclusion

We continue to develop EL4J further. Features planned for the near future include more demos (Web-UI, .NET interoperability), additional documentation, GUI support, Maven integration, and a generic user management.

References:

- EL4J website http://el4j.sourceforge.net/
- Reference documentation http://el4j.sourceforge.net/el4j/ReferenceDoc.pdf

Philipp H. Oser

Philipp H. Oser has studied Computer Science at ETH Lausanne and at Carnegie Mellon University. He has worked for 5 years at ELCA (http://www.elca.ch/) and his work included architecting two J2EE frameworks: first a proprietary one (LEAF Java) and now EL4J. In parallel, Philipp is a lecturer at the University of Applied Sciences North-West Switzerland.