Dependency Injection Refined by Extra-functional Properties

Kamil Ježek, Lukáš Holý, Premek Brada
Department of Computer Science and Engineering
University of West Bohemia
Pilsen, Czech Republic
{kjezek,lholy,brada}@kiv.zcu.cz

***Abstract*—The Dependency Injection has been widely implemented in a lot of frameworks to decouple software parts. However, current frameworks use simple matching algorithms to determine candidates to be injected. For instance, Spring does type-based matching, optionally enriched with a qualifier string. This is limiting mainly in dynamic systems where a lot of candidates may exist. As a result, the matching fails and developer's interaction is needed. In this work, we propose to enrich Dependency Injection with extra-functional properties serving as additional parameters of the matching algorithm to reliably select the most suitable candidate.**

I. Introduction

Dependency Injection (DI) has been adopted as a design pattern to deal with the complexity of software. Managing the complexity, software products are granulated into smaller modules. The typical design is that a module operates with an interface while an implementation is injected using DI.

A lot of industrial frameworks are based primarily on DI. In the field of Java, Spring[1] or PicoContainer[2] provide means to configure the injection via DI. Particularly, Spring configures Spring Beans either in XML descriptive files or by Java Annotations in the code.

There are variants of how to implement DI. The dependencies may be defined statically or dynamically. The static definitions are defined e.g. directly in source code or as part of a configuration. For instance, Spring supports either the setter or constructor static injection. On the other hand, the dynamic injection creates the dependencies automatically by matching candidate types. For instance, Spring provides an autowiring capability that determines whether a dependency candidate type is compatible with the type to be injected to.

A limitation of currently used dynamic DI implementations is its weak ability to deal with more injection candidates. Since systems are large and DI is used to integrate also third-party libraries, a developer interaction is required to resolve the candidate multiplicity. For instance, if Spring's autowiring finds more than one candidate, it simply throws a runtime exception and the system does not start.

The work was partially supported by the UWB grant SGS-2010-028 Advanced Computer and Information Systems and by the Czech Science Foundation project 103/11/1489 Methods of development and verification of component-based applications using natural language specifications.

[1]www.springsource.org/

[2]http://picocontainer.codehaus.org/

In this work, an approach to enrich DI with additional extra-functional properties (EFP) [1] is proposed. The core idea is that these properties are used as additional hints to select the most suitable candidate.

II. The Dependency Candidate Selection

Our approach to refine DI uses a framework called EFFCC[3] presented in our previous work [2]. In a nutshell, EFFCC is the implementation of general EFPs. It consists of a remotely accessible EFPs storage, tools to assign the EFPs and an embeddable evaluator.

In this work, we propose to use the EFPs pre-stored in the repository and annotate software elements first. Secondly, the evaluator is embedded in a concrete system to evaluate whether the EFPs matches. Finally, the matching result reduces the DI candidates list removing unmatched ones, keeping only the most suitable ones. This approach should remove all candidates that are type compatible but provide insufficient extra-functional characteristics, such as low performance, high memory consumption, high response time etc. Therefore, the developers interaction to manually solve candidate clashes should be rapidly decreased.

In addition, EFPs can provide a smooth selection of the candidates leading to the selection of the most suitable one. For instance, a module may require a maximal response time while several candidates may provide a time below this limit. Then, the fastest one can be selected. A benefit is a better scalability than the binary resolution of the compatible/incompatible candidate types.

A. Implementation in The Spring IoC Container

This work proposes an idea of how DI in the Spring IoC container may be enriched with EFPs. The annotation based approach has been used, however, the ideas are applicable also to the XML based configuration. Nonetheless, the idea is also applicable to other existing component frameworks using the DI pattern.

Examples presented below display components from CoCoME [3]. In the examples, the `InventoryData` component uses `InventoryDatabase` to read the database data via the `Jdbc` interface.

[3]Extra-Functional Property Featured Compatibility Checks, http://www.assembla.com/spaces/show/efps

The first example defines the `InventoryData` component that requires the `Jdbc` interface (autowired as a field by Spring). A set of annotations says that a domain of *TradingSystem* is used and a component with $response\_time \geq slow$ is required. The meaning of $slow$ is valid in a context of the *Inventory* sub-domain.

This approach allows to segment values into separated domains with their definitions differing for particular sub-domains. Furthermore, the domain specific values are assigned semantic rich names (e.g. $slow$). It allows to create versatile components with specialised characteristics depending on a concrete area of usage. A detailed discussion about this topic we have already provided in [4].

```
@Service
public class InventoryData
 implements StoreQueryIf {

/** Required JDBC. */
@Require(gr = "TradingSystem",
 conditions = {
  @Condition(lr = "Inventory", condition =
    "(reposnse_time>=slow)") })
@Autowired  // Spring dependency
private Jdbc jdbc;
...
}
```

The second example presents the implementation of the `Jdbc` interface providing EFPs. The used annotations are obviously counterparts of the annotations from the previous example.

```
@Provide(gr = "TradingSystem",
 efps = @Efp(name = "response_time",
  lr = @LrValue(name = "Inventory",
   valueName = "slow"))
@Repository
public class InventoryDatabase
  implements Jdbc {  ... }
```

Having such annotated components, the evaluation of the annotations may be invoked before the dependencies are set. Values from the same domains and sub-domains are matched and only matched ones succeed in DI. In the Spring implementation, we propose to use so called Bean Post-Processors to start the evaluation. The Post-Processors are invoked for each Bean instantiation and its dependency injection. Hence, it is a suitable point in which our mechanisms to match EFPs may be processed.

III. DI Candidates Visualisation

Despite the approach presented so far automatically selects the most suitable DI candidate, the developers may be still burdened with the amount of dependencies they must understand. As a solution we have proposed an interactive visualisation tool that provides a feature withdrawing a particular component of a complex component graph to the, so called, Separated Components Area [5]. This area allows to detach a component with a big number of dependencies from the graph to visually

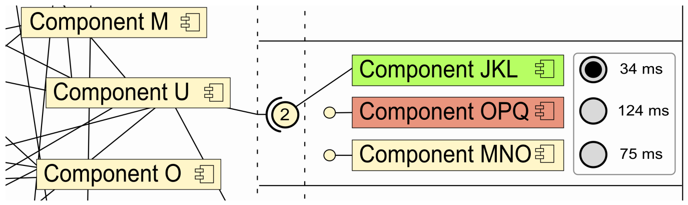

Fig. 1: DI Candidates Grouping

simplify the graph. In this work, we propose to extend this area to visualise all candidates of one DI, colouring component bodies by their characteristics (e.g. red is the slowest one while green is the fastest one), possibly allowing developers to adjust an automatically selected component.

Figure 1 displays three components that are all DI candidates differing in EFPs. Although the automatic matching would select the fastest candidate, a user can select another one observing propagation of the change in the graph.

IV. Conclusion

This work has proposed an automatic selection of dependency injection candidates according to explicit extra-functional properties attached to the candidates. In addition, a visualisation has been proposed to help developers to understand software when it is complex.

On the one hand, similar goals have already been targeted. An example is the selection of the most suitable service in Service Oriented Architecture according to the Quality of Service specifications. On the other hand, the extra-functional properties have not been used to refine the candidate selections in the dependency injection. Since the dependency injection is the fundamental building block of a lot of industrial frameworks, we believe our approach is innovative and worth implementing in developer's tools. The implementation is our future work.

References

[1] M. Glinz, "On non-functional requirements," in *Requirements Engineering Conference*. Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 21–26.

[2] K. Ježek and P. Brada, "Correct matching of components with extra-functional properties - a framework applicable to a variety of component models," in *Evaluation of Novel Approaches to Software Engineering (ENASE)*. SciTePress, 2011, ISBN: 978-989-8425-65-2.

[3] S. Herold, H. Klus, Y. Welsch, A. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, Benjamin, Hummel, M. Meisinger, and C. Pfaller, "Common component modelling example (CoCoME)," Book Chapter, available at http://agrausch.informatik.uni-kl.de/CoCoME/downloads (2010).

[4] K. Jezek, P. Brada, and P. Stepan, "Towards context independent extra-functional properties descriptor for components," in *Proceedings of the 7th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA 2010), Electronic Notes in Theoretical Computer Science (ENTCS) Volume 264, page 55-71, ISSN: 1571-0661*. Elsevier Science Publishers B. V., 10th August 2010, pp. 55–71.

[5] L. Holý, K. Ježek, J. Snajberk, and P. Brada, "Lowering visual clutter in large component diagrams," in *16th International Conference Information Visualisation*, 2012.