Detecting Dependencies in Enterprise JavaBeans with SQuAVisiT

Alexandru Sutii, Serguei Roubtsov, Alexander Serebrenik
Technische Universiteit Eindhoven, The Netherlands
a.sutii@student.tue.nl, {s.roubtsov, a.serebrenik}@tue.nl

***Abstract*—We present recent extensions to SQuAVisiT, Software Quality Assessment and Visualization Toolset. While SQuAVisiT has been designed with traditional software and traditional caller-callee dependencies in mind, recent popularity of Enterprise JavaBeans (EJB) required extensions that enable analysis of additional forms of dependencies: EJB dependency injections, object-relational (persistence) mappings and Web service mappings. In this paper we discuss the implementation of these extensions in SQuAVisiT and the application of SQuAVisiT to an open-source software system.**

I. Introduction

SQuAVisiT, Software Quality Assessment and Visualization Toolset [1], is a generic framework allowing for quality analysis and visualization of software systems. Rather than reimplementing different analysis and visualization techniques we have opted for a generic plugin architecture facilitating reuse of existing tools such as code duplication detector CCFinder [2] and dependency visualizer SolidSX [3]. SQuAVisiT has been successfully applied in a number of industrial case studies [4], [5].

While SQuAVisiT has been developed with traditional software and traditional caller-callee dependencies in mind, in recent years we have seen a growing demand for application of SQuAVisiT to non-traditional software such as Simulink models [6] or Enterprise JavaBeans (EJB)-based applications. In this paper we focus on the SQuAVisiT extensions necessary to support the latter applications. *EJB* [7] is one of the APIs of Java Platform, Enterprise Edition (Java EE), a Java computing platform. EJB is a managed, server-side component architecture for developing business applications. A typical EJB application contains a collection of *beans* that are managed by a container (or an *Application Server*).

As opposed to traditional caller-callee dependency mechanism, beans' interaction at runtime rely upon a dependency injection mechanism hidden inside the container. The "latent" character of the dependency injection mechanism makes tracing dependencies between artifacts in an EJB application a challenging task [8], [9]. To support developers and maintainers of EJB-based applications we extend SQuAVisiT with the ability to extract EJB-specific dependencies, *i.e.*, EJB dependency injections, object-relational (persistence) mappings and Web service mappings.

II. Tool Design

An overview of the architecture of our tool is illustrated in Figure 1. The components that are not contained in the *EjbDependenciesAnalyzer* box existed in SQuAVisiT earlier.

The EJB 3.1 specification provides the developer with two methods of specifying how the container should manage the beans: source code annotations and deployment descriptors. *Java Parser* parses the source code of the EJB application and generates an XML file containing classes and interfaces found. For each class, the XML file lists its methods and fields; for each method—the methods it invokes; for classes, methods and fields—annotations, if present.

For each dependency type, we created a separate component: *Web Analyzer*, *Persistence Analyzer* and *Injection Analyzer*. Note that dependency injection can be specified by means of both annotations and deployment descriptor; whereas, persistence or Web service mappings can only be defined using annotations.

The *Injection Locator* and *Injection Analyzer* components work together in order to extract dependencies between Java classes. The *Injection Locator* provides an API that can be used to find out what class will be injected for a certain variable. It parses both the annotations and the deployment descriptor and creates a mapping from ejb-names to beans. Next, it computes a mapping from each interface to all the classes that implement it. Finally, the deployment descriptor is parsed to obtain custom injections from the *ejb-ref* and *ejb-local-ref* nodes. When determining what bean will be injected in a reference, *Injection Locator* checks whether there exists an entry in the deployment descriptor for this reference. If this is not the case, it checks what bean implements the interface of the reference and returns that bean.

The *Injection Analyzer* browses through the source code and checks where EJB annotations are used. For each annotation it adds a dependency between the class and the annotation it occurs in and the class that will be injected.

In addition to dependencies the three *Analyzer* components identify the hierarchical structure of the system packages and

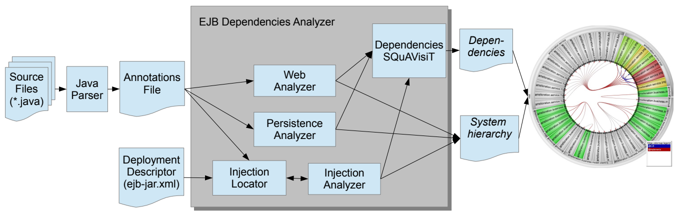

Fig. 1. Tool Architecture

classes. *DependenciesSQuAVisiT* component provides an API for storing the extracted dependencies. When the extraction finishes, this component outputs the *Dependencies* file that can be used, *e.g.*, by the dependency visualizer SolidSX.

III. Case study

Our concern was not just how to extract the dependencies from the source code, but also how to further analyze them in a way a human can easily understand. Therefore, we have used the radial diagram of SolidSX [3], the follow-up tool of ExTraVis, that has been shown to be beneficial for program comprehension [10]. In the radial diagram, the hierarchy of artifacts is displayed as concentric rings, whereas the curved lines represent dependencies between two artifacts (Figure 2).

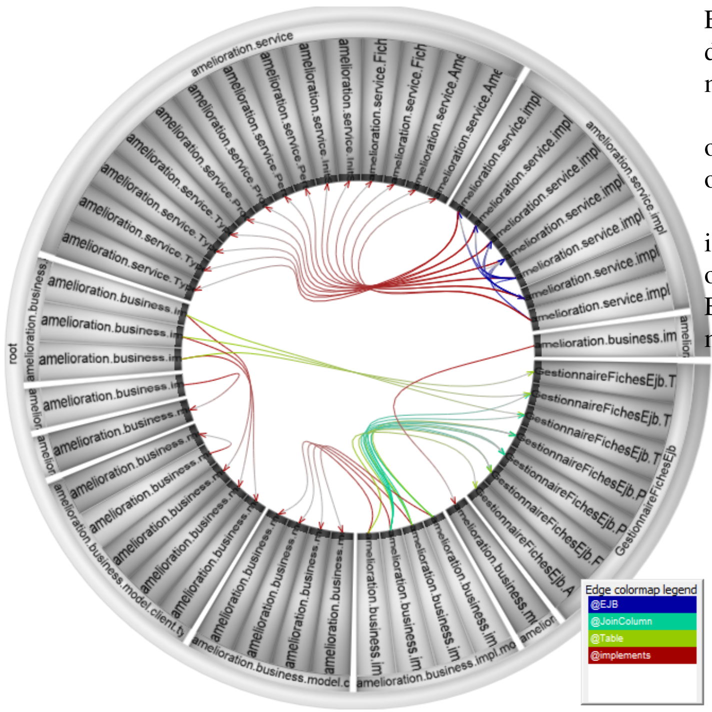

Fig. 2. Traditional *implements*-dependencies between classes and interfaces (Java) are red, injected *@EJB* dependencies (EJB) are blue and persistency dependencies *@JoinColumn* and *@Table* (EJB) are green.

Extended SQuAVisiT has been successfully applied to a number of open-source systems. Figure 2 shows multiple kinds of dependencies found in *Gestionnaire Fiches* [11]. By inspecting Figure 2 a number observations can be made.

First, *amelioration.service.impl* is the only package using dependency injections: its classes are beans managed by the EJB container with no explicit dependencies on each other. Source code inspection confirmed that when one of these beans needs another bean, it does not define a variable of the type of the needed bean. Instead, it uses the interface the needed bean needs to implement while the EJB container makes sure to inject the correct bean.

Second, *amelioration.business.impl.model.type* and *amelioration.business.impl.model* define persistence mappings. Like other beans in this applications, the beans from these packages implement interfaces, *i.e.*, the user does not have to be aware that the implementing beans use persistence mappings.

Third, classes in *amelioration.business.impl.model* have many *@JoinColumn* dependencies on tables created by the classes from *amelioration.business.impl.model.type*, but not *vice versa*. This indicates that there exists an association relation from the former classes to the latter.

Finally, *@implements* shows a clear separation between packages containing Java interfaces and packages containing implementation of these interfaces.

IV. Conclusion

This paper presents recent extensions to SQuAVisiT, Software Quality Assessment and Visualization Toolset, enabling SQuAVisiT to analyse and visualize dependencies found in EJB-based applications. Three EJB-specific additional types of dependencies are extracted: injected dependencies, persistence mappings and Web service mappings.

We have applied the extended SQuAVisiT to a number of small and medium-size open-source EJB applications, and obtained meaningful maintainability insights.

As a future work we consider application of the tool in industrial case studies, integration of SQuAVisiT with I2SD, our tool for reverse engineering Sequence Diagrams from Enterprise JavaBeans with interceptors [12], and definition of metrics based on the new types of dependencies [13], [14].

References

[1] M. G. J. van den Brand, S. Roubtsov, and A. Serebrenik, "SQuAVisiT: A flexible tool for visual software analytics," in *CSMR*. IEEE, 2009, pp. 331–332.
[2] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Software Eng.*, vol. 28, no. 7, pp. 654–670, 2002.
[3] SolidSource website, http://www.solidsourceit.com/products.
[4] D. Bosnacki, A. Mathijssen, and Y. S. Usenko, "Behavioural analysis of an I^{2}C Linux driver," in *FMICS*, 2009, pp. 205–206.
[5] S. Roubtsov, A. Serebrenik, and M. G. J. van den Brand, "D$_n$-based design quality comparison of industrial Java applications," in *CEE-SECR*, 2009, pp. 95–101.
[6] Y. Dajsuren, M. G. J. van den Brand, A. Serebrenik, and S. Roubtsov, "Simulink models are also software: modularity assessment," in *QoSA*. ACM, 2013, pp. 99–106.
[7] K. Saks, "JSR 318: Enterprise JavaBeansTM, Version 3.1, EJB Core Contracts and Requirements," JCP, Tech. Rep., 2009.
[8] C. Richardson, "Untangling Enterprise Java," *Queue*, vol. 4, no. 5, pp. 36–44, Jun. 2006.
[9] S. A. Roubtsov, A. Serebrenik, and M. G. J. van den Brand, "Detecting modularity "smells" in dependencies injected with Java annotations," in *CSMR*, 2010, pp. 244–247.
[10] B. Cornelissen, A. Zaidman, and A. van Deursen, "A controlled experiment for program comprehension through trace visualization," *IEEE Trans. Software Eng.*, vol. 37, no. 3, pp. 341–355, 2011.
[11] C. Souti, "Gestionnaire Fiches," https://code.google.com/p/gestionnaire-fiches-ejb/, Accessed on April 16, 2013.
[12] S. Roubtsov, A. Serebrenik, A. Mazoyer, M. G. J. van den Brand, and E. Roubtsova, "I2SD: reverse engineering sequence diagrams from Enterprise JavaBeans with interceptors," *IET Software*, vol. 7, pp. 150–166, June 2013.
[13] K. Mordal, N. Anquetil, J. Laval, A. Serebrenik, B. Vasilescu, and S. Ducasse, "Software quality metrics aggregation in industry," *Journal of Software: Evolution and Process*, 2012.
[14] B. Vasilescu, A. Serebrenik, and M. G. J. van den Brand, "You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics," in *ICSM*. IEEE, 2011, pp. 313–322.