I2SD: Reverse Engineering Sequence Diagrams from Enterprise Java Beans with Interceptors

Serguei Roubtsov, Alexander Serebrenik
Eindhoven University of Technology,
Den Dolech 2, P.O. Box 513,
5600 MB Eindhoven, The Netherlands
{s.roubtsov, a.serebrenik}@tue.nl

Aurélien Mazoyer
Polytech’Nice-Sophia,
930 Route des Colles - BP 145,
06903 Sophia Antipolis Cedex France
mazoyer@polytech.unice.fr

Mark van den Brand
Eindhoven University of Technology,
Den Dolech 2, P.O. Box 513,
5600 MB Eindhoven, The Netherlands
m.g.j.v.d.brand@tue.nl

Abstract—An Enterprise JavaBeans (EJB) interceptor is a software mechanism that provides for introducing behavior implemented as separate code into the execution of a Java application. In this way EJB interceptors provide a clear separation of the core functionality of the bean and other concerns, such as logging or performance analysis. Despite the beauty of the idea behind the interceptors, developing, testing and managing dependencies introduced by the interceptors are considered to be daunting tasks. For example, the developers can specify interceptors at multiple locations and by multiple means. However, different locations and specification means influence the order of the interceptor invocation, which is governed by more than fifteen different intertwined rules according to the EJB standard.

To facilitate development of EJB applications we have designed I2SD, Interceptors to Sequence Diagrams, a tool for reverse engineering EJB applications with interceptors to UML sequence diagrams. I2SD provides the developer with a visual feedback and can be used by quality managers to get a broader understanding of the way interceptors are used in their project.

I. INTRODUCTION

Maintaining software is similar to renovating a house; while rebuilding a house one has to understand the location of the pipelines connecting different rooms, software maintenance requires understanding dependencies between different software components. While traditional mechanisms implementing dependencies, such as method calls, are well understood, this is not the case for such mechanisms as interceptors [1]. Interceptors being a restricted form of aspect-oriented-programming (AOP), provide means to dynamically introduce behavior implemented as separate code into the execution of an application. Rather than implementing, for instance, such typical cross-cutting concerns as logging, access control or exception handling as the part of the system core functionality, one can implement them as separate modules and use interceptors to introduce them into the execution of the core application. Many currently available Java frameworks [2], [3] exploit interceptors to extend Enterprise JavaBeans™ (EJB) with AOP features.

Both the EJB standard [4] and the currently available Java frameworks [2], [3] provide multiple ways of specifying interceptors. Business method interceptors are invoked when a certain method is called, while life cycle callback interceptors are invoked when a certain event occurs such as an object creation. The developer can decide to specify the interceptors using XML files, known as deployment descriptors, and/or Java annotations; with respect to a bean class and or to a method; in a separate class, as part of the bean class itself, in a superclass or in an injected bean. For instance, if the developer is interested in logging all invocations of methods of certain class, she should specify a business method interceptor on the class level, using either a deployment descriptor or a Java annotation. In the presence of multiple interceptors the system behavior depends on whether business method interceptors, life cycle callback interceptors or both kinds of interceptors are involved, in what way they are specified, at which level and location. Continuing the running example, assume that in addition to logging, the developer intends to measure the time spent on executing a certain method. If time measurement is implemented as a method level interceptor, by default the logging interceptor will be called first, and then the time measurement will be performed. Should the developer desire to include logging time in the time being measured, she should overrule this default strategy by explicitly specifying the invocation order in the deployment descriptor. For business method interceptors the EJB standard specifies nine different rules governing the order of interceptor invocation, for life cycle callback interceptors—seven. Hence, while EJB interceptors provide the developers with a high degree of flexibility, the associated complexity of dependencies introduced by means of interceptors make managing EJB applications more difficult [5]. Developing such applications is associated with longer periods and higher costs [6], and the developers have been reported to struggle with configuring and debugging such applications [6].

We aim at supporting the aforementioned development process by facilitating understanding, and therefore, maintenance of EJB applications with interceptors. We choose to reverse engineer UML sequence diagrams from EJB applications with interceptors. UML sequence diagrams are a part of the industrial de facto standard and are supported by multiplicity of
development tools. They have also been shown to be beneficial for program comprehension [7]. Since interceptor invocation is essentially sequential and data independent, it is specifically well suited for being portrayed by UML sequence diagrams.

The main contribution of this paper consists, therefore, in presenting a tool for reverse engineering UML sequence diagrams from EJB applications with interceptors. The tool is called I2SD, Interceptors to Sequence Diagrams and is available from http://www.lagusno.com/tools/. Building upon the algorithm for business method interceptors presented in our previous work [8], I2SD targets two kinds of users: software developers and quality assessors. To assist the developers, I2SD should readily provide feedback during the software development. Since numerous software developers spend their workday in an integrated development environment (IDE) [9], I2SD should be integrated in an IDE. Moreover, I2SD should be applicable to incomplete programs, as programs under development are often incomplete. To meet these requirements we opt for a static analysis technique and integrate I2SD in NetBeans. To support the quality assessors, I2SD should be able to run as a stand-alone application and produce plain text descriptions of sequence diagrams, providing for further diagram processing, e.g., metrics calculation [10]. Hence, we also designed a stand-alone version of I2SD.

Remainder of the paper is organized as follows. After a brief discussion of EJB interceptors in Section II, we discuss the design of I2SD in Section III. Section IV discusses application of I2SD in two use cases. We review the related work in Section V, and finally conclude in Section VI.

II. EJB INTERCEPTORS

In this section we present a brief overview of EJB 3.0 interceptors following the EJB 3.0 standard [4]. The standard distinguishes between life cycle callback interceptors, interceptors invoked when objects are, e.g., created or destroyed, and business method interceptors invoked with a business method invocation.

A. EJB 3.0 Callback Methods

Similarly to traditional objects’ life cycle of a Java bean instance starts with the bean instance being created and ends with the bean instance being destroyed. Creation and destruction events can be intercepted allowing the developer, e.g., to allocate and release resources when beans instances are constructed and destroyed. To achieve this she can add a @PostConstruct annotation to methods that allocate resources and a @PreDestroy annotation to methods that release them. She can also specify <post-construct> and <pre-destroy> tags in the deployment descriptor XML file.

Furthermore, one usually distinguishes between stateful beans and stateless beans [11]. Stateful beans record the so-called conversational state, “remembering” the results of previous exchanges of information between the client and the bean. Stateless beans do not record the conversational state. Absence of state information in stateless beans improves the system performance as it becomes possible to reuse bean instances for different clients using pooling [11]. Stateful beans, however, cannot be reused as they need to save client-specific conversational state. This also means that stateful session bean instances with multiple concurrent clients can have a significant memory footprint. In order to alleviate this problem, the EJB container removes idle bean instances from time to time from the memory, serializes them and places them in a temporal storage. This process is known as passivation. Should a passivated bean instance be required by a client, it has first to be activated, i.e., reloaded to the memory. Hence, in addition to creation and destruction events occurring in life cycles of both stateful and stateless beans, life cycle of stateful bean also has passivation and activation events, that can be intercepted as well. Thus, similarly to @PostConstruct and @PreDestroy annotations (or, equivalently <post-construct> and <pre-destroy> tags), stateful beans can be annotated with @PostActivate and @PrePassivate (or, equivalently <post-activate> and <pre-passivate> tags).

Callback methods may be associated with multiple annotations: e.g., a method annotated with @PostConstruct and @PostActivate will be invoked whenever the bean instance is created or activated. A given class may, however, have not more than one life cycle callback method for the same life cycle event.

B. EJB 3.0 Interceptors

Business method interceptors are invoked prior to the beginning of a business method execution, and may resume after its completion, e.g., to inspect the business method return value or exceptions thrown. Annotation @Interceptors allows the developer to indicate which classes should be consulted to determine the interceptors for a given method, or all methods of a given class. Each class implementing a business method interceptor should have exactly one method annotated with @AroundInvoke: this method is the interceptor entry point, it will be invoked when the interceptor should be invoked. Due to this reason business method interceptors are also known as AroundInvoke-interceptors [12]. The entry point can be also specified in the deployment descriptor using the <around-invoke> tag.

Similarly, with life cycle callback interceptors developers can isolate functionality into a class and invoke it when a life cycle event is triggered. Inside these classes, methods that should be invoked are identified by means of annotations or tags discussed in Section II-A.

C. Invocation Order of Interceptors

The EJB 3.0 specification [4] provides the developer with multiple means of specifying both life cycle event callback interceptors and business method interceptors. The developer can decide to specify the business method interceptors using deployment descriptor XML file ejb-jar.xml and/or Java annotations; in a separate class, as part of the bean class itself, in a superclass or in an injected bean; at the default level, the bean class level or the method level. As specified in the invocation rules of EJB 3.0 [4], should multiple interceptors be present,
all these specification options influence the order of their invocation. Furthermore, EJB 3.0 provides mechanisms to exclude invocation of some interceptors: e.g., if a business method is annotated with @ExcludeClassInterceptors, interceptors defined on the class level and applicable to all methods of the class of the business method should not be called.

For business method interceptors the EJB 3.0 standard provides nine rules governing the order of interceptors invocation, for life cycle callback interceptors—seven. Moreover, life cycle event callback interceptors and business method interceptors can be defined on the same class.

Presence of multiple ways to specify interceptors and multiple rules determining the order of invocation challenges, therefore, developers’ comprehension of EJB 3.0-based systems. Without adequate tool support, to understand the order of interceptor invocation, the developer has to scrutinize her code and manually check which of the invocation rules apply.

### III. Tool Design

To assist comprehension of EJB 3.0-based software systems we have developed a tool, called I2SD, for reverse engineering EJB applications with interceptors to UML sequence diagrams. UML sequence diagrams visualize the sequence of method invocations, including interceptors, and have also been shown to be beneficial for program comprehension [7]. Moreover, since interceptor invocation is essentially sequential and data independent, it is specifically well suited for being portrayed by UML sequence diagrams.

The architecture of I2SD is shown on Figure 1: I2SD is implemented as a pipe-and-filter architecture [13]. The Java parser has been obtained using JavaCC [14], a popular parser generation tool. Given a language grammar JavaCC generates a Java program that can recognize matches to the grammar. To parse Java code we have extended the Java grammar used in our visual software analytics toolset SQuaVisiT [15], to include interceptor-related annotations. The reasons to implement the Java parser as a separate component rather than as a part of the central reverse engineering step are facilitation of co-evolution with the Java language and reuse of individual components of I2SD. From the co-evolution perspective we observe that the parser is the only part of I2SD that has to be adapted when new language features are being added to Java, as, e.g., expected in Java 7 under Project Coin. Moreover, once the abstract syntax tree has been stored as an XML file, the same XML parser can be used both for this file and for the deployment descriptor ejb-jar.xml. For XML parsing we have opted for JDOM [16].

The core part of I2SD is the reverse engineering step. We postpone a more detailed discussion of the reverse engineering algorithm for business method interceptors in Section III-A, we proceed with discussing life cycle callback interceptors in Section III-B. The reverse engineering step can either produce a sequence diagram or a warning, indicating that the interceptors chain may be broken (Section III-A). The sequence diagram is stored in the UMS format suitable for visualization generation by UML Speed\(^1\). Furthermore, since UMS is a plain text format, it makes the sequence diagrams generated amenable for further analyses, e.g., such as discussed in [10].

The final step consists in visualizing the sequence diagram using UML Speed. The sequence diagrams are stored as an image in the SVG graphical format. The SVG format [17] has been designed with web-graphics in mind, and, therefore, it allows linking the image to classes and methods mentioned in the sequence diagram, facilitating system comprehension by visual inspection of the diagrams.

I2SD has been implemented as a plugin for version 6.5 of NetBeans. Our main reason for choosing NetBeans is that the EJB support in NetBeans is better than in Eclipse [18]. Moreover, NetBeans has been reported as the best IDE for development of Java EE applications within an educational environment [19]. We also plan to integrate I2SD in Eclipse.

#### A. Business method interceptors

EJB 3.0-based programs combine three forms of method invocation: traditional ("method A calls method B"), object-oriented ("method A calls method B of class C but method B' of the subclass C' is actually executed") and interceptor-based. The business method interceptors reverse engineering algorithm assumes that a class name C and a business method \(m\) in C are given, and produces a sequence diagram, including the interceptors invoked when \(m\) is being called.

In our previous work [8] we have assumed presence of a reverse engineering technique capable of inferring sequence

---

\(^1\)http://umlspeed.sourceforge.net/
diagrams for programs with traditional and object-oriented method invocations, and augmented this technique with an algorithm for programs combining all three invocation forms. While this approach results in the most complete picture of the invocations, it also might result in an overly complex diagram requiring close inspection, and therefore, hindering software development in an IDE rather than facilitating it. Therefore, as opposed to [8], we exclude the analysis of the business method body and focus solely on the interceptors invoked. To alert the developer i2SD generates a warning if an @AroundInvoke method calls another business method of another bean as invocation of this method can involve additional interceptors. In this case, the developer can invoke i2SD again, focusing on one of the methods called.

In addition to the situation above, warnings are generated if InvocationContext.proceed() is potentially unreachable. If InvocationContext.proceed() is unreachable from one of the methods invoked in the interceptor chain, then the EJB container will deadlock. I2SD checks two conditions that can cause InvocationContext.proceed() to become unreachable: if InvocationContext.proceed() occurs within a decision statement or a loop, or when there is no direct call to InvocationContext.proceed(). The latter situation is highlighted in Figure 4.

For the sake of completeness we include the slightly adapted version of the algorithm from [8] as Figures 2 and 3.

\textbf{traverseInheritance}(class D)
1) \textbf{L} = \emptyset, \textbf{S} = \emptyset, \textbf{c} = \textbf{D}.
2) \textbf{DO}
   a) \textbf{IF} (c contains an @AroundInvoke method) \textbf{OR} (there exists <interceptor>
      <interceptor-class>c</interceptor-class>
      <around-invoked>
      <method-name>... </method-name>
      </around-invoked>
   </interceptor>)
   i) Let \textbf{n} be the AroundInvoke method
   ii) \textbf{IF} \textbf{n} not in \textbf{L} push (c, \textbf{n}) on S.
   b) Add all methods of c to the methods list \textbf{L}.
   c) \textbf{c} = superclass(c).
   \textbf{WHILE} (not-empty(S))
   a) \textbf{(c, n)} = Pop(S);
   b) Enqueue (InvocationContext, c, n) to \textbf{Q};
   c) Analyse \textbf{n}:
      i) \textbf{IF} there is no call of InvocationContext.proceed() in \textbf{n}, report a warning.
      ii) \textbf{IF} a call of InvocationContext.proceed() occurs inside a decision statement or a loop, report a warning.
      iii) \textbf{FOR} each method invocation c.m\textsuperscript{'} OR c.m\textsuperscript{'} where m != proceed()
         Notify the user that invocation of \textbf{m} might involve interceptors and recommend creating a new sequence diagram.
      iv) Add (c, InvocationContext.proceed()) to \textbf{Q}.

\textbf{Fig. 2. traverseInheritance}: Interceptors from the superclasses.

\textbf{main}(business method \textbf{m} of a bean class \textbf{C})
1) \textbf{IF} (<interceptor-order> defined on \textbf{C})
   \textbf{FOR} class \textbf{IN}
   <interceptor-order>
   <interceptor-class>... </interceptor-class>
   </interceptor-order>
   \textbf{DO traverseInheritance}
   \textbf{GOTO} 5
2) \textbf{IF} (<interceptor-order> defined on \textbf{m})
   \textbf{FOR} class \textbf{IN}
   <interceptor-order>
   <interceptor-class>... </interceptor-class>
   </interceptor-order>
   \textbf{DO traverseInheritance}
   \textbf{GOTO} 7
3) \textbf{IF} (<exclude-default-interceptors> \textbf{OR} @ExcludeDefaultInterceptors defined on \textbf{m})
   \textbf{ELSE FOR} class \textbf{IN}
   <interceptor-binding>
   <ejb-name>C </ejb-name>
   <interceptor-class>... </interceptor-class>
   </interceptor-binding>
   \textbf{DO traverseInheritance}
4) \textbf{IF} <exclude-class-interceptors> or @ExcludeClassInterceptors defined on \textbf{m},
   \textbf{GOTO} 5
5) \textbf{ELSE FOR} class \textbf{IN} (@Interceptors(...) on \textbf{C}) \textbf{OR} IN
   <interceptor-binding>
   <ejb-name>C </ejb-name>
   <interceptor-class>... </interceptor-class>
   <method>
   <method-name> m.name </method-name>
   </method>
   </interceptor-binding>
   \textbf{DO traverseInheritance}
6) \textbf{FOR} class \textbf{IN}
   <interceptor-binding>
   <ejb-name>C </ejb-name>
   <interceptor-class>... </interceptor-class>
   <method>
   <method-name> m.name </method-name>
   <method-params>
   <method-param>m.params</method-param>
   <method-param>m.params</method-param>
   </method-params>
   </method>
   </interceptor-binding>
   \textbf{DO traverseInheritance}
7) \textbf{FOR} \textbf{C} perform \textbf{traverseInheritance}.
8) Enqueue (InvocationContext, c, m) to \textbf{Q}.
9) Analyse the body of \textbf{m}:
   \textbf{FOR} each method invocation C.m\textsuperscript{'} OR C.m\textsuperscript{'}
   Notify the user that invocation of \textbf{m} might involve interceptors and recommend creating a new sequence diagram.
10) Store \textbf{Q} in the UMS format required for visualization.

\textbf{Fig. 3. main}: Determining the invocation order.
B. Life cycle callback interceptors

Construction of sequence diagrams in presence of life cycle callbacks requires (1) identification of an occurrence of a life cycle event and (2) construction of sequences of method invocations caused by the occurrence of the life cycle event. Life cycle events are managed by the EJB container: while the bean itself can request to be destroyed using the \texttt{@Remove} annotation, the container can also decide to destroy beans based on time-out considerations. Similarly, when the memory reserved by the container to store active stateful beans becomes full, it will decide to passivate the least recently used bean \cite{4}. Hence, destruction and passivation events may occur independently from the source code, based on the settings of the EJB container. Moreover, since a bean can be activated only after it has been passivated and occurrence of a passivation event depends on the settings of the container, occurrence of an activation event also depends on the settings of the container. Finally, the bean instance is created not only when a business method of a stateless session bean is invoked for the first time, but also when the same method has been invoked for the second time and the bean instance has been destroyed between the method invocations. Hence, destruction can happen solely based on the container settings, the same is also true for (some of the) creation events. Dependency on the container settings compromises portability, in the same way dependency on the object request broker implementation compromised portability of the CORBA interceptors \cite{20}. Therefore, since the exact prediction of life cycle events goes beyond the abilities of static source code analysis, our algorithm assumes the class $C$ and the event $e$ as inputs, and generates a warning stating that additional, potentially undesirable, life cycle events can occur during the execution of method calls caused by $e$, depending on the Java source code, XML deployment descriptor and settings of the EJB container.

The reverse engineering algorithm for life cycle interceptors follows the big lines of the reverse engineering algorithm for the business method interceptors. However, since life cycle interceptors are invoked when a life cycle event takes place rather than when a business method is called, all checks related to $m$ are dropped. Moreover, as the event $e$ can be one of \texttt{PostConstruct}, \texttt{PreDestroy}, \texttt{PostActivate} or \texttt{PrePassivate}, four different annotations and the corresponding deployment description tags should be considered instead of \texttt{@AroundInvoke}. Finally, we have to differentiate between the interceptors defined in the bean itself and those defined in other classes: interceptors defined on the bean itself do not need to invoke \texttt{InvocationContext.proceed()}. 

IV. USE CASES

In this section we present two use cases showing the applications of I2SD. In the first use case discussed in Section IV-A we focus on a developer that uses I2SD to gain understanding of the software. In Section IV-B I2SD is applied as part of the quality assessment. Quality assessment is based on comparing the system being assessed with comparable systems. To assist in the latter task Section IV-C compares interceptor use in two benchmark systems.

A. I2SD for software development

To illustrate how I2SD can support a software developer we consider a modification of an existing system. As the running
example we consider a product management system inspired by one of the NetBeans samples. Figure 5 shows a code snippet from one of the files in the NetBeans IDE.

Developer Alice intends to optimize performance of the business method `productInfo`. Given a product identifier, `productInfo` first retrieves the information about the corresponding product from the database, creates the corresponding object (POJO [4]) and then consults the data stored in that object to provide additional information about the product manufacturer, using methods `find` and `Manufacturer.getName`, respectively.

Alice starts by measuring the execution time of `productInfo`: she applies the `PerformanceInterceptor` to the business method. Knowing that the method level interceptors are called in the same order they are listed in the `@Interceptor` annotation, Alice adds `PerformanceInterceptor` after all other interceptors that have already been defined for `productInfo`, i.e., after `ProductIdValidationInterceptor` (see Figure 5).

Running the program Alice observes that the execution time of `productInfo` constitutes 2016 milliseconds, which Alice attributes to the need to retrieve data from the database. Since `productInfo` consults the database twice, Alice decides to measure the execution of each one of the database operations separately. To this end she calls `System.currentTimeMillis` before `find`, immediately after `find` and after `Manufacturer.getName`, and calculates the time elapsed between the calls. She discovers that `find` takes 546 milliseconds, while the time needed for `Manufacturer.getName` is negligible and reported as 0 milliseconds. Where did the remaining $2016 - 546 = 1470$ milliseconds go?

`I2SD` can help Alice to resolve the mystery. By selecting `productInfo` in the IDE (Figure 5) she can create the sequence diagram of the interceptors involved when the `productInfo` is called. The output window in the bottom of Figure 5 shows a link to the sequence diagram produced by `I2SD` (Figure 6). Looking at this diagram Alice can observe that, in fact, `PerformanceInterceptor` measures time spent on

1. user access validation implemented in `EJBObject.validateAccess`,

Fig. 5. `I2SD` integrated in the NetBeans development environment.

Fig. 6. Sequence diagram created by `I2SD` for `ProductFacade.productInfo`. 

```
package ProductManagement; 

import java.util.List; 

public class ProductFacade 
    extends EJBObject implements ProductFacadeRemote 
{ 
    @PersistenceContext 
    private EntityManager em; 

    // Interceptors 
    @Interceptors( 
        [ProductInfoInterceptor.class, 
        PerformanceInterceptor.class]) 

    public String getProductInfo(int id) 
    { 
        Product product = find(new Integer(id)); 
        if (product != null) 
            return "Product ID: " + product.getId() + " Name: " + product.getName(); 
        else 
            return null; 
    } 
```
(2) control transfer by means of InvocationContext.proceed,
(3) logging implemented in ProductFacade.logMethods,
(4) another control transfer by means of InvocationContext.proceed, and finally,
(5) the business method ProductFacade.productInfo itself. Hence, the measurements above indicate that $t_1 + t_2 + t_3 + t_4 + t_5 = 2016$ and $t_5 = 546$, where $t_i$ is the execution time of the step $i$ in milliseconds. Assuming $t_2$ and $t_4$ to be negligible, Alice should check how the remaining 1470 milliseconds are spent on $t_1$ (EJBOBJECT.validateAccess) and $t_3$ (ProductFacade.logMethods). To this end she adds appropriate System.currentTimeMillis calls and discovers that the lion’s share of the execution time has been spent on $t_1$ (EJBOBJECT.validateAccess).

The performance issue encountered by Alice can be attributed to a common problem in using interceptors, i.e., combination of interceptors with inheritance that can easily lead to a very convoluted behavior [21].

B. I2SD for quality assessment

In the second series of experiments we consider the quality assessor’s perspective. Quality assessor Bob decides to use software metrics to get insights in system quality and maintainability. As suggested in [10], he considers depth of the scenario an important characteristics of the architecture complexity. He wants to investigate how deep are the interceptor-related scenarios in his system. Formally, the depth of a scenario is defined as the number of calls in the scenario [10]. We adapt this definition and consider only methods that can be reverse engineered by I2SD, i.e., interceptor invocations, calls to InvocationContext.proceed(), calls to AroundInvoke-methods within the bean itself, business method invocations and methods triggered by life cycle events. For example, the depth of the interceptor-related scenario in Figure 6 is 11.

To calculate scenario depths for different business methods of his system Bob runs I2SD as a batch job that creates a separate UMS file for each class and business method. Next these UMS files are analyzed to count the number of method calls per UMS file and, subsequently, to determine the interceptor-related scenarios’ depths. Finally, he can compare the values obtained with similar values obtained for comparable systems. To assist Bob in the latter task, Section IV-C presents a similar investigation for a number of benchmark systems.

C. I2SD for benchmarking interceptors’ use

To benchmark the use of interceptors we have studied two middle-sized Java systems. To select these systems we have first searched for @AroundInvoke using Google Code Search, eliminated tutorial examples and files related to implementation of EJB containers, and finally chosen two systems with a comparable number of Java files.

a) DataPortal: The first system we consider is the DataPortal, a visual front-end to one or more ICAT repositories, containing scientific data generated by facilities such as synchrotrons, satellites and telescopes. Version 3.2.2.1 of the system contains 635 files, 275 out of them are Java files.

No deployment descriptor is present in the system and there is only one file, DataPortal.java with an @Interceptors annotation, namely @Interceptors({ArgumentValidator}). Class DataPortal inherits from SessionEJBOBJECT, which inherits from EJBOBJECT. While SessionEJBOBJECT does not have AroundInvoke-methods, EJBOBJECT has one, named logMethods, that should be invoked first when any business method of DataPortal is called. Moreover, the @Interceptors({ArgumentValidator}) annotation is specified at the class level in DataPortal.java, meaning that the corresponding interceptors should be invoked for any business method of this class, unless class level interceptors are explicitly excluded. Two out of 24 methods defined in DataPortal.java, init and isFinished, exclude class level interceptors with @ExcludeClassInterceptors. Keeping in mind that interceptor invocations should be followed by a call to InvocationContext.proceed() and that the last call in the sequence diagram generated by I2SD is the call to the business method itself, we can observe that init and isFinished produce scenarios of depth 3, while all other methods of DataPortal produce scenarios of depth 5. Figure 7 shows one such scenario of depth 5, namely, the sequence diagram created for DataPortal.getDataReferences.

The only life cycle annotation in the system is @PostConstruct present in SessionBean and in EJBOBJECT. However, SessionBean inherits from SessionEJBOBJECT, which inherits from EJBOBJECT, and, therefore, when an instance of SessionBean is created both the @PostConstruct method of SessionBean and the @PostConstruct method of EJBOBJECT should be called (Figure 8). Hence, the depth of the scenario produced by SessionBean and the instance construction event is 2. By a similar argument is the depth of the scenario produced by EJBOBJECT and the instance construction event is 1. Moreover, for all classes, directly or indirectly inheriting from EJBOBJECT (with exception of SessionBean) the depth of the scenario corresponding to the bean instance construction is 1. The system contains 16 such classes.

b) WasabiBeans: WasabiBeans, abbreviating Web Application Services and Business Integration, is a JavaEE-based framework to support the establishment of cooperative work and learning environments. WasabiBeans has been developed at University of Paderborn, Germany. The most recent version of WasabiBeans counts 324 Java files.

Similarly to the DataPortal case, no deployment descriptors were found in the system. Two classes contained AroundInvoke-methods, i.e., should be considered as interceptors: DebugInterceptor, that is not mentioned in the remaining Java files, and JCRSessionInterceptor. The latter interceptor annotates six beans, including ObjectService. None of these beans inherits from another bean. However, nine additional beans inherit from ObjectService, bringing the total number of beans that can lead to invocation of

2http://dataportal.googlecode.com/

3http://code.google.com/p/wasabibean/
JCRSessionInterceptor to 15. In all beans the JCRSessionInterceptor is specified at the class level. Since none of the business methods of these classes excludes class-level interceptors, all business methods of the 15 beans give rise to interceptor-related scenarios of depth 3.

Life cycle annotations in the system are @PostConstruct and @PreDestroy. Both annotations are used in the aforementioned six beans, annotated with JCRSessionInterceptor, as well as in four additional beans. None of these 6+4=10 beans inherits from another bean and the @PostConstruct (@@PreDestroy) annotation appears only once in each file. Therefore, only one method will be invoked when a bean instance is being created or destroyed, namely, the @PostConstruct (@PreDestroy) method of the bean itself.

c) Comparing DataPortal and WasabiBeans: We observe that in both systems the use of interceptors has been quite limited: in 18 files out of 275 in DataPortal and in 20 files out of 324 in WasabiBeans. Moreover, neither of the systems specified interceptors in the deployment descriptor.

The limited adoption of interceptors is not surprising as they express dependencies that are known to hinder development of EJB applications [5], [6], [22], and, hence, are avoided by developers. Furthermore, we observe that in both systems life cycle events propagate further through the system than the business method interceptors: in both cases, all beans involved in business method interception are also involved in life cycle interception, but not other way around.

The way interceptors are used differs strongly from one system to another. The DataPortal developers opted for a more complex interplay between inheritance and interceptors involving a limited number of classes, resulting in deeper scenarios for these classes: 5 for business method interceptors and 2 for life cycle interceptors. The WasabiBeans developers have preferred to use a simpler structure reflected in more shallow scenarios, but have applied the business method interceptors technology on a larger scale. Hence, when future development and maintenance of DataPortal demand a more profound knowledge of EJB 3.0 from the developer responsible for a limited number of classes, development and maintenance of WasabiBeans require only a basic knowledge of the technique but (potentially) from a larger group of developers.

V. RELATED WORK

Presence of complicated dependencies between the implementation components and the EJB container make development, testing and management of EJB applications to a
challenging task [5], [6], [22]. To facilitate these tasks a monitoring system [6] and a profiler [22] have been proposed. Both solutions assume, however, that the EJB application is complete and can be executed. We pursue a complementary approach and aim at supporting the ongoing development process, i.e., I2SD is capable of analyzing incomplete, and hence, non executable programs.

Reverse engineering code to UML sequence diagrams is a well-studied research problem: both static and dynamic approaches have been proposed. Static approaches, such as [23]–[26], do not attempt to execute the system under investigation, and infer sequence diagrams from the source code. Dynamic approaches derive sequence diagrams from observing the system’s run-time behaviour [27], [28]. Furthermore, this research has lead to a number of reverse engineering tools (see [29] for a recent survey of the area). Dynamic approaches are known to produce more precise results: e.g., due to presence of dynamic binding. Applicability of the dynamic approaches is, however, restricted by the fact that the system analyzed should be executable, while static approaches are capable of analyzing incomplete systems. As explained above, one of our goals consisted in supporting the software developers during the development process, we had to consider incomplete or not necessarily executable systems, and, hence, we opted for a static approach. When compared with the existing static approaches to reverse engineering to UML sequence diagrams [23], [24], [26], I2SD focuses on dependencies injected in system code, that were not considered by most of the existing approaches. The only work where such programs are considered as a subject of the reverse engineering effort is our previous work [8]. Building on and extending [8] this paper presents I2SD, going beyond the analysis of business method interceptors and focusing on the tool-related aspects as opposed to purely algorithmic ones.

Reverse engineering sequence diagrams can be seen as related to detection of EJB patterns and anti-patterns [30], [31]. Patterns and anti-patterns pertaining to invocation of interceptors can be defined on the level of the corresponding sequence diagram. I2SD can be then used to infer the sequence diagram and check for presence of such anti-patterns. Moreover, EJB anti-patterns can be detected [32] based on EJB Framework Specific Modeling Language [33], which can further be configured to identify interceptors.

Prior to their emergence in Java EJB applications, interceptors were available in CORBA [1]. In CORBA, different interceptor instances can be registered within a object request broker component. Once a request is intercepted, all the registered interceptor instances will be invoked by the object request broker. The invocation order of interceptors might, however, be dependent on the specifics of the object request broker implementation in the same way the invocation order of life cycle callback interceptors depends on the settings of the EJB container. Thus, while some implementations allow interceptors to define the invocation order, this would introduce dependencies between the interceptors, and, hence, compromise their portability [20]. This portability argument holds to lesser extent for business method interceptors: the EJB container has to obey the rules governing the invocation order fixed in the EJB 3 standard [4].

As explained above, interceptors are in a way similar to AOP. In the AOP community sequence diagrams are used in the forward engineering for choice of join points [34]. Cross-cutting concerns have been modelled using UML sequence diagrams [35] and used to derive flow graphs and flow trees to support test generation [36]. While reverse engineering sequence diagrams of systems with aspects or cross-cutting concerns did not seem to have so far attracted attention of the AOP research community, related notions of call graph and control-flow graph have been studied in context of static analysis and program maintenance. Sereni and de Moor [37] adapted the notion of a call graph for aspect-oriented programs. Unlike a sequence diagram the call graph, however, contains only information about which methods (advices) can be called but not about their order of invocation. Moreover, the approach of [37] did not support the “around” advice, essential to implement business method interceptors. These shortcomings have been addressed in [38], where an inter-procedural aspect control flow graph has been proposed. This work is complementary to ours: while control flow graphs necessarily provide more detailed information than sequence diagrams, our technique takes into consideration intricate interplay between inheritance and multiple kinds of interceptors.

Finally, while I2SD applies reverse engineering techniques to programs with interceptors, in a number of papers the opposite approach has been taken, i.e., programs with interceptors have been used as means to implement reverse engineering techniques [39], [40].

VI. CONCLUSION

In this paper we have introduced I2SD, a reverse engineering tool for Enterprise JavaBeans with interceptors. While development, testing and management of EJB applications are experienced as difficult, I2SD can support both developers and quality managers by providing them with appropriate information: developers can benefit from visual representation of the interceptor invocations by means of familiar UML sequence diagrams, while quality managers can obtain brief summaries giving a general overview of the project use of the interceptor technology. I2SD can be used either via NetBeans or as a stand-alone tool.

I2SD has been implemented as a highly modular pipe-and-filter architecture [13]. This architectural decision facilitates evolution of I2SD and reuse of its individual components [41].

As future work we consider a number of possible directions. First, I2SD will be extended to incorporate more recent extensions to the interceptor model such as @InterceptorBinding [42]. Furthermore, I2SD will be connected to our visual software analytics tooolset SQuaVisiT [15]. This connection will make EJB applications immediately amenable for multiple analysis and visualization techniques already integrated in SQuaVisiT. Finally, we intend to conduct a number of user studies involving I2SD: in the first series of studies we will
ask the participants to use i2SD to perform a number of development tasks akin to the task performed by Alice in Section IV-A, while in the second series of studies we will ask the participants to perform a number of analysis tasks akin to the ones carried out by Bob in Section IV-B using the i2SD + SQuaVisiT combination.

REFERENCES

[40] JSR-299: Contexts and Dependency Injection for the Java EE platform, 2009.