www.ietdl.org

Published in IET Software
Received on 3rd April 2012
Revised on 17th July 2012
Accepted on 31st August 2012
doi: 10.1049/iet-sen.2012.0056

Special Issue: 11th IEEE International Working Conference on Source Code Analysis and Manipulation

ISSN 1751-8806

I2SD: reverse engineering Sequence Diagrams from Enterprise Java Beans with interceptors

Serguei Roubtsov[1], *Alexander Serebrenik*[1], *Aurelién Mazoyer*[2], *Mark G.J. van den Brand*[1], *Ella Roubtsova*[3]

[1]*Eindhoven University of Technology, POB 513, 5600 MB, Eindhoven, The Netherlands*
[2]*France Labs CICA 2229, route des Crêtes 06560, Valbonne, France*
[3]*Open University of the Netherlands POB 2960, 6401DL Heerlen, The Netherlands*
E-mail: a.serebrenik@tue.nl

Abstract: An Enterprise JavaBeans (EJB) interceptor is a software mechanism that provides for introducing behaviour 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 i nterceptors, 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 15 different intertwined rules defined in 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 unified modeling language (UML) sequence diagrams. I2SD provides the developer with a visual feedback and can be used by quality managers to obtain insights in the ways interceptors are used in their project.

1 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 [1–4]. Interceptors, being a restricted form of aspect-oriented programming (AOP), provide means to dynamically introduce behaviour 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 [5, 6] exploit interceptors to extend Enterprise JavaBeans™ (EJB) with AOP features.

Both the currently available Java frameworks [5, 6] and the EJB standard [2–4] provide multiple ways of specifying interceptors. ‘Business method interceptors’ are invoked when a certain method is called, ‘life-cycle callback interceptors’ are invoked when a certain event occurs such as an object creation, and ‘timeout method interceptors’ are invoked by the EJB Timeout service. 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 a 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 behaviour 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 [7]. Developing such applications is associated with longer periods and higher costs [8], and the developers have been reported to struggle with configuring and debugging such applications [8].

We aim at supporting the aforementioned development process by facilitating understanding, and therefore, maintenance of EJB applications with interceptors. We choose to derive 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 [9]. 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.laquso.com/tools/. Building upon the algorithm for business method interceptors presented in our previous work [10], 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) [11], 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, for example, metrics calculation [12]. Hence, we also designed a stand-alone version of I2SD.

The remainder of the paper is organised as follows. After a brief discussion of EJB interceptors in Section 2, we discuss the design of I2SD in Section 3. Section 4 discusses application of I2SD in three use cases related to software development and quality assessment. To assist in the latter task, Section 5 presents an empirical study of interceptors' use in practice. We review the related work in Section 6, and finally conclude in Section 7.

2 EJB interceptors

In this section we present a brief overview of EJB interceptors following the EJB 3.0 [2] and the EJB 3.1 standards [3, 4] [The next version of the EJB standard, EJB 3.2, also known as JSR-345, is being prepared at the time of writing but is yet to be made public.]. The EJB 3.0 standard distinguishes between life-cycle callback interceptors, interceptors invoked when objects are,for example, created or destroyed, and business method interceptors invoked with a business method invocation. The EJB 3.1 standard adds timeout method interceptors that intercept timeout methods invoked by the EJB Timeout service.

2.1 EJB 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, for example, to allocate and release resources when beans instances are constructed and destroyed. To achieve this a @PostConstruct annotation can be added to methods that allocate resources and a @PreDestroy annotation to methods that release them. It is also possible to specify <post-construct> and <pre-destroy> tags in the deployment descriptor XML file.

Furthermore, one usually distinguishes between 'stateful beans and stateless beans' [13]. 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 system performance as it becomes possible to reuse bean instances for different clients using pooling [13]. 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, serialises 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, that is, 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: for example, a method annotated with @PostConstruct and @PostActivate will be invoked whenever the bean instance is created or activated. A given class may, however, have no more than one life-cycle callback method for the same life-cycle event.

2.2 EJB timeout methods

Many business work flows depend on time: certain activities should happen on a certain date, should be repeated every month or should not take longer than the specified amount of time. To support this behaviour EJB 3.1 introduced the timer service [3]. Using this, service developers can create timers such that when a timer expires, the container calls a timeout method of the bean's implementation class.

EJB distinguishes between timers created programmatically and automatically. For programmatically created timers, timer creation is explicitly implemented in the source code. The corresponding timeout method may be a method that is annotated with the @Timeout annotation or the <timeout-method> in the deployment descriptor, or the bean may implement the javax.ejb.TimedObject interface. This interface has a single method called ejbTimeout. Every bean can have only one programmatic timer.

For automatically created timers, timer creation occurs when a specified moment of time arrives. For instance, the following annotation requires the container to call a timeout method on Mondays: @Schedule(dayOfWeek = 'Mon'). In general, a timeout method for an automatically created timer may be a method that is annotated with annotations @Schedule or @Schedules or tags <schedule> or

< schedules >. Beans may have multiple automatic timers, corresponding, for example, to timeouts that should occur with different frequencies during the working days and during weekends.

Both for programmatically and automatically created timers, the timeout methods should return void, accept a javax.ejb.Timer object as the only parameter and may not throw application exceptions.

2.3 EJB interceptors

'Business method interceptors', known as 'method interceptors' in [4], are invoked prior to the beginning of a business method execution, and may resume after its completion, for example, 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 or associated with an < around-invoke > tag: this method is the interceptor entry point, it will be invoked when the interceptor should be invoked. Owing to this reason business method interceptors are also known as AroundInvoke-interceptors [14]. The entry point can be also specified in the deployment descriptor using the < around-invoke > tag.

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 2.1. In a similar way, developers can intercept calls to the timeout methods by means of 'timeout method interceptors': methods with the @AroundTimeout annotation or < around-timeout > tag will be invoked before timeout methods. Similarly to around-invoke methods, each class implementing a timeout method interceptor should have exactly one method annotated with @AroundTimeout or associated with an < around-timeout > tag.

Fig. 1 shows how the three kinds of interceptors can be defined within one class.

2.4 Invocation order of interceptors

The EJB specifications [2–4] provide the developer with multiple means of specifying interceptors. The developer can decide to specify the 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, the level of all methods with the same name within a class or the method level. As specified in the invocation rules of EJB [2, 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: for example, 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 each kind of interceptors the EJB standards provide a set of rules governing the order of interceptors invocation. Moreover, different kinds of interceptors can be defined on the same class. To illustrate complexity of the rules determining the invocation order, in Fig. 2 we quote the business method interceptors rules [4].

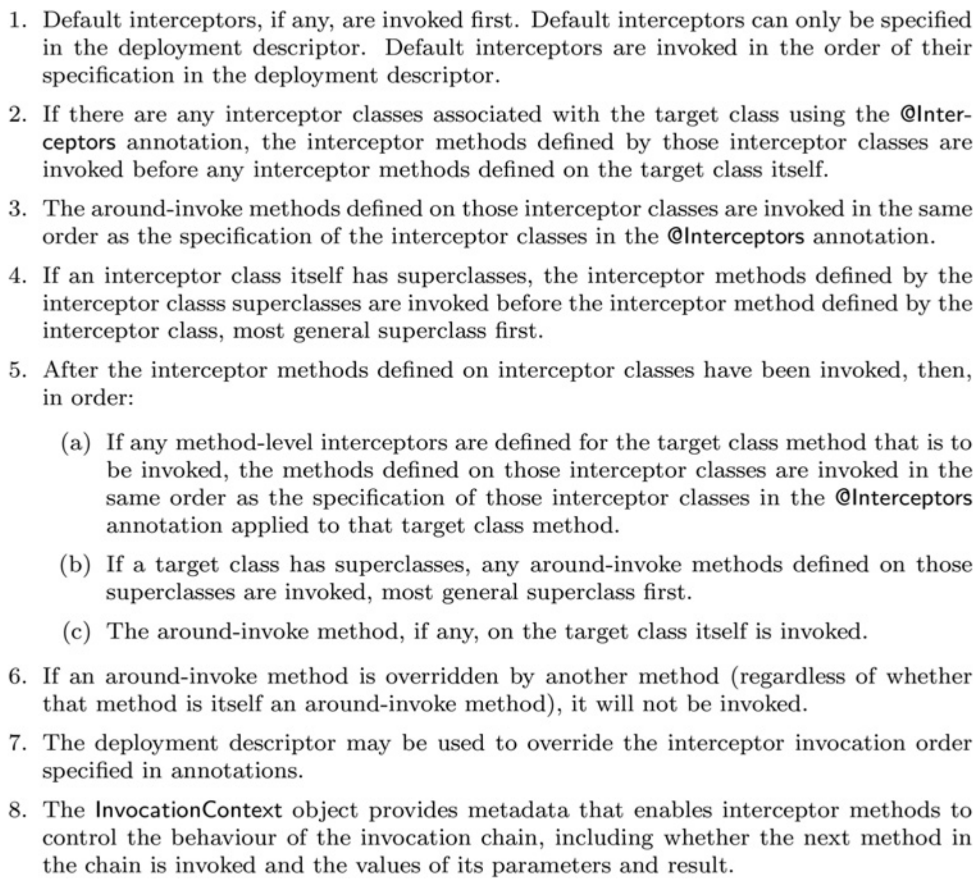
Presence of multiple ways to specify interceptors and multiple rules determine the order of invocation challenges, therefore, developers' comprehension of software systems based on recent version of Enterprise JavaBeans. Without adequate tool support, to understand the order of interceptor invocation, the developer has to scrutinise her code and manually check which of the invocation rules apply.

3 Tool design

To assist comprehension of EJB-based software systems we have developed a tool, called I2SD, for reverse engineering EJB applications with interceptors to UML sequence diagrams. UML sequence diagrams visualise the sequence

```
public class ClassInterceptor {
        @PostConstruct
        private void interceptorPostConstruct(InvocationContext ic) {
                try {
                 ic.proceed();
                } catch (Exception ex) { /* ... */ }
        }
        @AroundInvoke
        protected Object interceptorAroundInvoke(InvocationContext ic) {
                return ic.proceed();
        }
        @AroundTimeout
        protected Object interceptorAroundTimeout(InvocationContext ic) {
                return ic.proceed();
        }
}
```

Fig. 1 *Example of a life-cycle callback interceptor, a business method interceptor and a timeout method interceptor (cf. [57])*



1. Default interceptors, if any, are invoked first. Default interceptors can only be specified in the deployment descriptor. Default interceptors are invoked in the order of their specification in the deployment descriptor.
2. If there are any interceptor classes associated with the target class using the @Interceptors annotation, the interceptor methods defined by those interceptor classes are invoked before any interceptor methods defined on the target class itself.
3. The around-invoke methods defined on those interceptor classes are invoked in the same order as the specification of the interceptor classes in the @Interceptors annotation.
4. If an interceptor class itself has superclasses, the interceptor methods defined by the interceptor classs superclasses are invoked before the interceptor method defined by the interceptor class, most general superclass first.
5. After the interceptor methods defined on interceptor classes have been invoked, then, in order:
   (a) If any method-level interceptors are defined for the target class method that is to be invoked, the methods defined on those interceptor classes are invoked in the same order as the specification of those interceptor classes in the @Interceptors annotation applied to that target class method.
   (b) If a target class has superclasses, any around-invoke methods defined on those superclasses are invoked, most general superclass first.
   (c) The around-invoke method, if any, on the target class itself is invoked.
6. If an around-invoke method is overridden by another method (regardless of whether that method is itself an around-invoke method), it will not be invoked.
7. The deployment descriptor may be used to override the interceptor invocation order specified in annotations.
8. The InvocationContext object provides metadata that enables interceptor methods to control the behaviour of the invocation chain, including whether the next method in the chain is invoked and the values of its parameters and result.

Fig. 2 *Rules for invocation of multiple business method interceptors [4]*

of method invocations, including interceptors, and have also been shown to be beneficial for program comprehension [9]. 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 Fig. 3: I2SD is implemented as a pipe-and-filter architecture [15]. The Java parser has been obtained using JavaCC [16], a popular parser generation tool. Given a language grammar JavaCC generates a Java program that can recognise matches to the grammar. To parse Java code we have extended the Java grammar used in our visual software analytics toolset SQuAVisiT [17], 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, for example, 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 [18].

The core part of I2SD is the reverse engineering step. The reverse engineering algorithm has been based on the EJB specifications [2, 3, 4]. However, the standards are textual descriptions rather than formal specifications and as such, might be subject to misinterpretations and ambiguities. To resolve potential misinterpretations and ambiguities we have tested the algorithm against the actual interceptor order, provided by an existing EJB container. Specifically, we have opted for the GlassFish application server [19], a free application server, shipped by Oracle in a bundle with

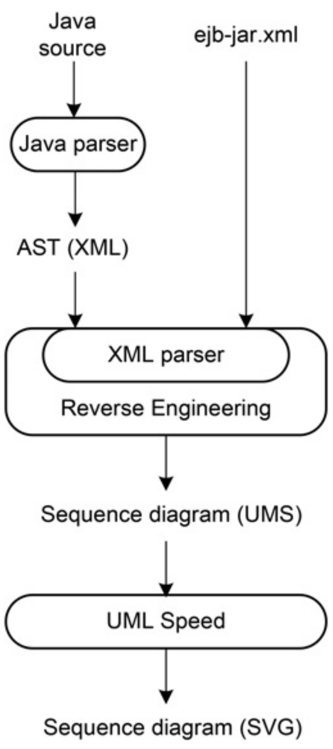


Fig. 3 *I2SD is implemented as a pipe-and-filter architecture*

NetBeans Java EE edition. Since the EJB 3.0 specification is also owned by Oracle, we expected the implementation to adhere to the specification. The test cases were developed using **Product Manager**, the example application distributed with GlassFish. If the sequence diagram reverse engineered by I2SD did not match the behaviour of GlassFish, the reverse engineering algorithm has been corrected to match the behaviour of GlassFish. Examples of discrepancies between the standard and GlassFish application server pertain to specification of total interceptor ordering in the deployment descriptor, relative priorities of <interceptor-order> element and the exclude- elements and structure of the <method> element [10].

We postpone a more detailed discussion of the reverse engineering algorithm for business method interceptors till Section 3.1, life-cycle callback interceptors till Section 3.2 and timeout method interceptors till Section 3.3. The reverse engineering step can either produce a sequence diagram or a warning, indicating that the interceptor's chain may be broken (Section 3.1). The sequence diagram is stored in the UMS format suitable for visualisation generation by UML Speed [http://umlspeed.sourceforge.net/]. Furthermore, since UMS is a plain text format, it makes the sequence diagrams generated amenable for further analyses, for example, such as discussed in [12].

The final step consists in visualising the sequence diagram using UML speed. The sequence diagrams are stored as an image in the SVG graphical format. The SVG format [20] 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 plug-in for version 7.0.1 of NetBeans. Our main reason for choosing NetBeans is that the EJB support in NetBeans is better than in Eclipse [21]. We also plan to integrate I2SD in Eclipse.

3.1 Business method interceptors

EJB-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_1 of the subclass C_1 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.

The reverse engineering algorithm consists of two parts: the main algorithm following the order of the rules in Fig. 2 (Section 3.1.1), and an auxiliary algorithm traversing the inheritance hierarchy (Section 3.1.2).

3.1.1 Main algorithm: The ***main*** reverse engineering algorithm ensures that interceptors are invoked according to the rules in Fig. 2. The algorithm assumes business method ***m*** of a bean class ***C*** to be provided as the input. Auxiliary algorithm ***traverseInheritance*** called by ***main*** is discussed Section 3.1.2.

The precondition of ***main*** is that ***m*** is a *business method* in ***C***, that is, that ***m*** satisfies the following requirements of [3]:

- The method names can be arbitrary, but they must not start with 'ejb' to avoid conflicts with the callback methods used by the EJB architecture.
- The business method must be declared as public.
- The method must not be declared as final or static.
- If the method corresponds to a business method on the session bean's remote business interface or remote interface, the argument and return value types for a method must be legal types for Java Remote Method Invocation interface over the Internet Inter-Orb Protocol (RMI/IIOP) [22]. Legal types for RMI/IIOP are defined in Chapter 4 of [23].
- If the method is a web service method or corresponds to a method on the session bean's web service endpoint, the argument and return value types for a method must be legal types for Java API for XML Web Services or Java API for XML-based RPC (JAX-WS/JAX-RPC [24]). For instance, Java primitive types such as int or boolean are legal both for RMI/IIOP and for JAX-WS/JAX-RPC.
- The throws clause may define arbitrary application exceptions.

Moreover, experiments with GlassFish [19] have revealed that no interceptors are invoked when ***m*** is an around-invoke method. Therefore we assume as an additional precondition that ***m*** is not an around-invoke method.

Since the interceptors are invoked sequentially, we opt for a global queue ***Q*** of all the method invocations as the main data structure used. The queue ***Q*** is initially empty. Whenever ***main*** or ***traverseInheritance*** decide that a method should be invoked, a triple is enqueued to ***Q***, containing names of the caller class, of the callee class, and of the method called. The order of the elements in the queue corresponds to the order of invocation.

For the sake of readability we present the algorithm ***main*** in multiple figures (Figs. 4–9), intertwined with explanation.

Recall that the interceptor invocation order defined in the deployment descriptor, if present, overrides the interceptor invocation order specified in annotations (Rule 7 in Fig. 2). Therefore, the first and the second step of the algorithm (Fig. 4) verify whether an interceptor invocation order is explicitly specified in the deployment descriptor for ***C*** or for ***m***. If this is the case, then for each one of the interceptors mentioned in the deployment descriptor, one has first to check whether their superclasses contain @AroundInvoke methods that have to be invoked first.

If the interceptors' invocation order has not been explicitly modified in the deployment descriptor, then the default interceptors should be applied first if present (Rule 1 in Fig. 2). If default interceptors are present, they are defined in the deployment descriptor and are applicable to a set of target classes. Application of default interceptors can be excluded with the @ExcludeDefaultInterceptors annotation or <exclude-default-interceptors> tag (Fig. 5).

Fig. 6 shows the next step dedicated to the class-level interceptors (Rule 2 in Fig. 2). We stress that the @Interceptors annotation introduces a sequence of interceptor classes, and therefore '**FOR** class **IN** (@Interceptors(...) on ***C***)' also ensures that the interceptor classes are considered in the same order as they are specified (Rule 3 in Fig. 2). The call to ***traverseInheritance*** ensures that if an interceptor class itself has superclasses, then the interceptor methods defined by the superclasses are invoked before the interceptor method defined by the interceptor class, most general superclass first (Rule 4 in Fig. 2).

Method-level interceptors are called after the class-level interceptors (Rule 5a in Fig. 2). Similarly to class-level interceptors, the algorithm steps in Fig. 7 ensure that the interceptors are invoked in the order of specification of their

```
1. IF (<interceptor-order> defined on C)
      FOR class IN
         <interceptor-order>
            <interceptor-class>...</interceptor-class>
         </interceptor-order>
      DO traverseInheritance
      GOTO 6
2. IF (<interceptor-order> defined on m)
      FOR class IN
         <interceptor-order>
            <interceptor-class>...</interceptor-class>
         </interceptor-order>
      DO traverseInheritance
      GOTO 8
```

Fig. 4 *__Main__ algorithm starts by verifying whether the invocation order is explicitly defined in the deployment descriptor*

```
3. IF (<exclude-default-interceptors> OR @ExcludeDefaultInterceptors de-
   fined on m)
   OR (<exclude-default-interceptors> OR @ExcludeDefaultInterceptors de-
   fined on C)
      GOTO 4
   ELSE FOR class IN
      <interceptor-binding>
         <ejb-name>*</ejb-name>
         <interceptor-class>...</interceptor-class>
      </interceptor-binding>
     DO traverseInheritance
```

Fig. 5 *Default interceptors are called before class- or method-level ones*

```
4. IF <exclude-class-interceptors> OR @ExcludeClassInterceptors defined on
   m,
      GOTO 6
5. FOR class IN (@Interceptors(...) on C) OR IN
      <interceptor-binding>
         <ejb-name> C </ejb-name>
         <interceptor-class>...</interceptor-class>
      </interceptor-binding>
     DO traverseInheritance
```

Fig. 6 *Class-level interceptors are invoked of the order of the* @Interceptors *annotation*

```
6. FOR class IN (@Interceptors(...) on m) DO traverseInheritance
7. FOR class IN
       <interceptor-binding>
           <ejb-name> C </ejb-name>
           <interceptor-class>...</interceptor-class>
           <method>
               <method-name> m.name </method-name>
           </method>
       </interceptor-binding>
   OR 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[1]</method-param>
                 <method-param>...</method-param>
                 <method-param>m.params[n]</method-param>
               </method-params>
           </method>
       </interceptor-binding>
   DO traverseInheritance
```

Fig. 7 *Method-level interceptors are invoked in the order of the specification of their classes in the* @Interceptors *annotation*

```
8. For C perform traverseInheritance.
```

Fig. 8 *Interceptor methods defined on* ***C*** *and its superclasses are the last interceptors invoked*

```
 9. Enqueue (InvocationContext,C,m) to Q.
10. Analyse the body of m:
        FOR each method invocation C'.m' OR C.m'
            Notify the user that invocation of m' might involve interceptors
            and recommend creating a new sequence diagram.
11. Store Q in the UMS format required for visualization.
```

Fig. 9 *Finally* ***m*** *is invoked and, if required, warnings are generated*

classes in the @Interceptors annotation. Instead of one step in Fig. 6, for the sake of readability in Fig. 7 we separate the analysis in two steps. Step 6 pertains to analysis of interceptors specified by means of annotations, whereas Step 7 – to interceptors specified in the deployment descriptor.

Finally, interceptor methods defined on *C* and its superclasses are invoked (Rule 5 in Fig. 2).

After all interceptors that should have been invoked, have been invoked, the business method *m* itself should be called (Fig. 9). In our previous work [10] we have assumed presence of a reverse engineering technique capable of inferring sequence 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 overtly complex diagram requiring close inspection, and therefore, hindering software development in an IDE rather than facilitating it. Therefore as opposed to [10], in the algorithm we exclude the analysis of the business method 'body' and focus solely on the interceptors invoked. To alert, the developer the algorithm 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.

3.1.2 Traversing inheritance hierarchy: The ***traverseInheritance*** algorithm presented in Figs. 10–12 ensures that interceptor methods of superclasses are invoked before the interceptor methods of the subclasses, most general superclass first. The algorithm assumes class ***D*** to be provided as the input and updates the global queue ***Q***.

Furthermore, ***traverseInheritance***(class ***D***) makes use of a list ***L*** and a stack **S**. The list ***L*** contains all methods of all classes on the inheritance path from ***D*** to the most general user-defined class from which ***D*** inherits. The stack **S** consists of pairs (***c***, ***n***), where ***n*** is an @AroundInvoke method of a class ***c*** on the inheritance path. Both the list ***L*** and the stack **S** are initially empty.

Recall that rules 4, 5b and 5c in Fig. 2 require the around-invoke method on the target class to be called after the around-invoke methods of the superclasses. Since Step

```
1. L = ∅, S = ∅, c = D.
```

Fig. 10 *Initialisation of the local variables*

```
2. DO
   (a) IF (c contains an @AroundInvoke method) OR (there exists
           <interceptor>
               <interceptor-class>c</interceptor-class>
               <around-invoke>
                   <method-name>...</method-name>
               </around-invoke>
           </interceptor>)
         i. Let n be the AroundInvoke method
        ii. IF n not in L
                push (c, n) on S.
   (b) Add all methods of c to the methods list L.
   (c) c = superclass(c).
   WHILE (user-defined(c))
```

Fig. 11 *Upwards traversal of the invocation chain*

```
3. WHILE(not-empty(S))
   (a) (c, n) = Pop(S);
   (b) Enqueue (InvocationContext, c, n) to Q;
   (c) Analyse n:
         i. IF there is no call of InvocationContext.proceed() in n, report a
            warning.
        ii. IF a call of InvocationContext.proceed() occurs inside a decision
            statement or a loop, report a warning.
       iii. FOR each method invocation c'.m' OR c.m' where m' != pro-
            ceed()
                Notify the user that invocation of m' might involve intercep-
                tors and recommend creating a new sequence diagram.
        iv. Add (c,InvocationContext,proceed()) to Q.
```

Fig. 12 *Reversal of the stack* ***S*** *and update of the invocation queue* ***Q***

3 of ***traverseInheritance*** reverses **S** to build ***Q***, we start by considering the input class itself such that the corresponding invocation becomes the last element of ***Q***, that is, 'the currently analysed class' ***c*** is initialised as the input ***D***.

Starting from ***D*** the algorithm traverses the inheritance chain from a subclass to a superclass, collecting all around-invoke methods along the chain. However, rule 6 in Fig. 2 states that an around-invoke method overridden by another method will not be invoked. Therefore before pushing the pair (***c***, ***n***) on **S** we have to check whether ***n*** is overridden. The construction of ***L*** implies that checking whether ***n*** is overridden is tantamount to checking membership of ***n*** in ***L*** (Step 2(a)ii in Fig. 11).

The last step of ***traverseInheritance***, presented in Fig. 12, reverses the stack **S** and enqueues invocations to ***Q***.

Recall that the invocation order can be controlled using the InvocationContext (Rule 8 in Fig. 2). Specifically, if InvocationContext.proceed() is unreachable from one of the methods invoked in the interceptor chain, then the EJB container will deadlock. Hence, warnings are generated if InvocationContext.proceed() is potentially unreachable. I2SD checks two conditions that can cause InvocationContext.proceed() to become unreachable: if InvocationContext.proceed() occurs within a decision statement or a loop (Step 3(c)i in Fig. 12), or when there is no direct call to InvocationContext.proceed() (there still may be a call to another method that in its turn calls InvocationContext.proceed(); Step 3(c)ii in Fig. 12).

Furthermore, similarly to the ***main*** algorithm of Section 3.1.1 we generate a warning if methods called might involve interceptors. As above, the developer can invoke I2SD again, focusing on one of the methods called.

3.1.3 Correctness of the algorithm: We conclude this section with a brief discussion of the algorithm's correctness. To show partial correctness we have indicated for each step of the algorithm which invocation rules prescribed by the EJB standards [2–4] does the step implement. Algorithm ***traverseInheritance***(class ***D***) terminates because of finiteness of the inheritance path from Object to ***D***. Each one of the loops in ***main*** terminates because of finiteness of the Java code and of the deployment descriptor.

3.2 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 @Remove annotation, the container can also decide to destroy beans based on timeout considerations [These time-out considerations are not related to the EJB Timer Service or timeout method interceptors discussed in Section 3.3.]. 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 [2]. 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 the same method has been invoked for the second time and the bean instance has been destroyed between the method invocations. Hence, since 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 [25]. 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 PostConstruct, PreDestroy, PostActivate or PrePassivate, four different annotations and the corresponding deployment description tags should be considered instead of @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 InvocationContext.proceed().

3.3 Timeout method interceptors

Rules governing the invocation of timeout method interceptors are almost identical to those for business method interceptors in Fig. 2. The only differences are that instead of around-invoke methods the rules consider around-timeout methods, that is, methods with the @AroundTimeout annotation or <around-timeout> tag in the deployment descriptor.

The only difference between business method interceptors and timeout method interceptors is related to class-level interceptors. In the case of business method interceptors, the class-level interceptor applies to all methods of the target class. In the case of timeout method interceptors, however, the class-level interceptor applies only to timeout methods of the target class. Recall from Section 2.2 that timeout methods are methods annotated with @Timeout. This means that the algorithms discussed in Section 3.1 can be used to reverse engineer sequence diagram for timeout method interceptors; however, I2SD offers the opportunity to generate such a diagram if either a timeout method or a class containing at least one such method is selected by the user.

4 Use cases

In this section, we present three use cases showing the applications of I2SD. In the first use case discussed in Section 4.1, we focus on a developer that uses I2SD to gain understanding of the software. In Section 4.2, 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 5 compares interceptor use in two benchmark systems. Finally, in Section 4.3, we discuss the application of I2SD to an incomplete system.

4.1 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. Fig. 13 shows a code snippet from one of the files in the NetBeans IDE.

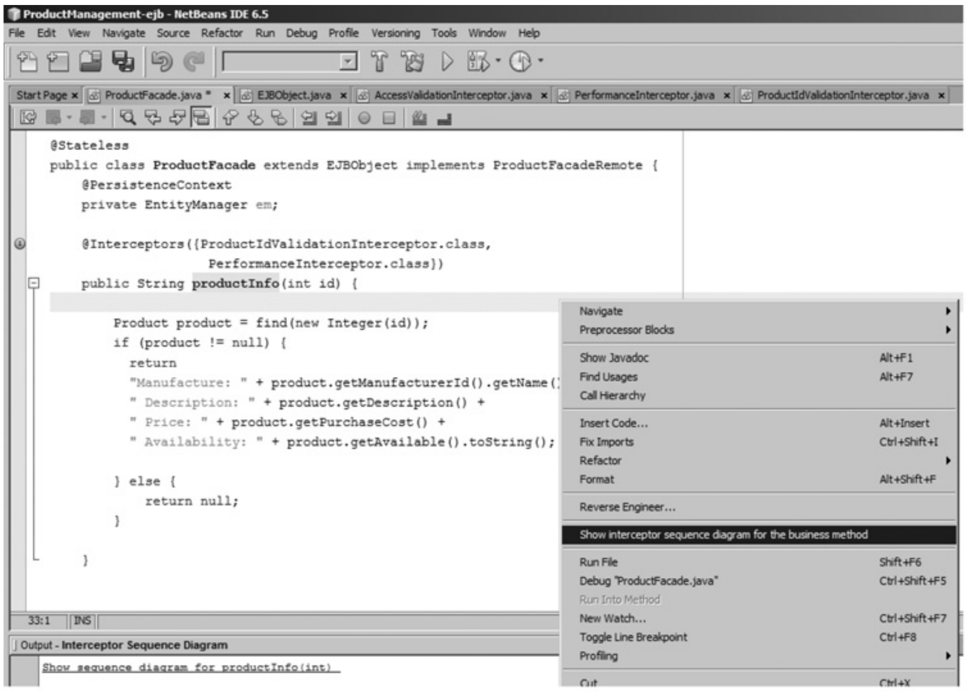
Developer Alice intends to optimise performance of the business method productInfo. Given a product identifier, productInfo first retrieves the information about the corresponding product from the database, creates the corresponding object (POJO [2]) and then consults the data stored in that object to 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 as they are listed in the @Interceptor annotation, Alice adds PerformanceInterceptor after all other interceptors that have already been defined for productInfo, that is, after ProductIdValidationInterceptor (see Fig. 13).

Running the program Alice observes that the execution time of productInfo constitutes 2016 ms, 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 ms, while the time needed for Manufacturer.getName is negligible and reported as 0 milliseconds. Where did the remaining 2016–546 = 1470 ms go?

I2SD can help Alice to resolve the mystery. By selecting productInfo in the IDE (Fig. 13) she can create the sequence diagram of the interceptors involved when the productInfo is called. The output window in the bottom of Fig. 13 shows a link to the sequence diagram produced by I2SD (Fig. 14). Looking at this diagram Alice can observe that, in fact, PerformanceInterceptor measures time spent on

1. user access validation implemented in EJBObject. validateAccess,


ProductManagement-ejb - NetBeans IDE 6.5
File Edit View Navigate Source Refactor Run Debug Profile Versioning Tools Window Help
Start Page
ProductFacade.java *
EJBObject.java
AccessValidationInterceptor.java
PerformanceInterceptor.java
ProductIdValidationInterceptor.java
Navigate
Preprocessor Blocks
Show Javadoc Alt+F1
Find Usages Alt+F7
Call Hierarchy
Insert Code... Alt+Insert
Fix Imports Ctrl+Shift+I
Refactor
Format Alt+Shift+F
Reverse Engineer...
Show interceptor sequence diagram for the business method
Run File Shift+F6
Debug "ProductFacade.java" Ctrl+Shift+F5
Run Into Method
New Watch... Ctrl+Shift+F7
Toggle Line Breakpoint Ctrl+F8
Profiling
Cut Ctrl+X
33:1 INS
Output - Interceptor Sequence Diagram

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

    @Interceptors({ProductIdValidationInterceptor.class,
                   PerformanceInterceptor.class})
    public String productInfo(int id) {

        Product product = find(new Integer(id));
        if (product != null) {
          return
          "Manufacture: " + product.getManufacturerId().getName(
          " Description: " + product.getDescription() +
          " Price: " + product.getPurchaseCost() +
          " Availability: " + product.getAvailable().toString();

        } else {
            return null;
        }

    }
```

```
Show sequence diagram for productInfo(int)
```

Fig. 13 *I2SD integrated in the NetBeans development environment*

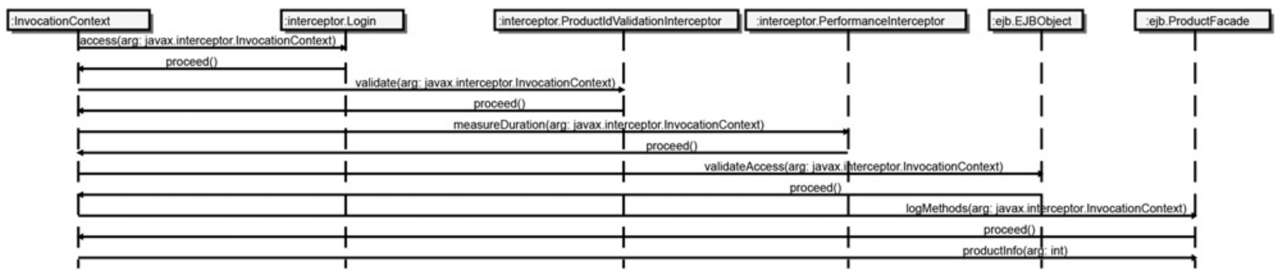

:InvocationContext
:interceptor.Login
:interceptor.ProductIdValidationInterceptor
:interceptor.PerformanceInterceptor
:ejb.EJBObject
:ejb.ProductFacade
access(arg: javax.interceptor.InvocationContext)
proceed()
validate(arg: javax.interceptor.InvocationContext)
proceed()
measureDuration(arg: javax.interceptor.InvocationContext)
proceed()
validateAccess(arg: javax.interceptor.InvocationContext)
proceed()
logMethods(arg: javax.interceptor.InvocationContext)
proceed()
productInfo(arg: int)

Fig. 14 *Sequence diagram created by I2SD for ProductFacade.productInfo*

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, that is, combination of interceptors with inheritance that can easily lead to a very convoluted behaviour [26].

4.2 I2SD for quality assessment

In the second example, 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 [12], he considers depth of the scenario as an important characteristic of the architecture complexity. He wants to investigate how deep the interceptor-related scenarios in his system are. Formally, the depth of a scenario is defined as the number of calls in the scenario [12]. We adapt this definition and consider only methods that can be reverse engineered by I2SD, that is, 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 Fig. 14 is 11.

To calculate the 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 analysed 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 5 presents a similar investigation for a number of benchmark systems.

4.3 I2SD for incomplete systems

We have mentioned in the introduction that I2SD should be applicable to incomplete programs. To illustrate this point, we consider the following example. Let Charlie be a novice programmer that recently joined an EJB3-based project and started implementing his first bean. Charlie's bean is incomplete and contains for the moment only the class name. Charlie believes that when existing beans are being constructed certain resources are always allocated. However, he does not know which resources precisely are allocated, and whether these resources are sufficient for his bean.

Charlie starts with reconstructing the sequence diagram for the @PostConstruct life-cycle callback interceptors of his own bean. Despite the fact that his code is incomplete, default interceptors as well as interceptors on the superclasses of Charlie's bean can be invoked and allocate the resources he needs. Hence, analysing the diagram produced by I2SD, Charlie can decide whether additional resources need to be allocated.

5 I2SD for benchmarking interceptors' use

To assist quality assessor Bob (Section 4.2) in evaluating his project, we need a frame of reference, that is, we need to understand how interceptors are used in practice. While performing an extensive empirical investigation was not the main goal of our work, in this section we show how I2SD can assist in conducting such an evaluation.

We have start by performing a series of searches on Google code search. Specifically, we looked for presence of interceptor-related annotations in Java files and interceptor-related tags in files called ejb-jar.xml. We continued this study by a more in-depth investigation of the interceptors' scenarios' depth induced by business methods in 108 open source software systems.

5.1 Presence of annotations and tags

In January 2012 we have conducted a series of searches on Google code search to determine the frequency of use of different interceptor-related constructs. Statistics obtained by means of these searches are summarised in Fig. 15. Inspecting this figure we observe that the most frequently used annotations are @PostConstruct (3117 hits) and @PreDestroy (690). While these annotations can be used to introduce life-cycle callback interceptors, this is not necessarily the case (cf. Section 2). The third most popular annotation is @AroundInvoke (320 hits) that is related to business method interceptors. Finally, the least popular annotation is @AroundTimeout (14 hits). This, however, should not be surprising: as opposed to life-cycle callback interceptors and business method interceptors introduced in [2], timeout method interceptors have been introduced in [4], that is, three years later.

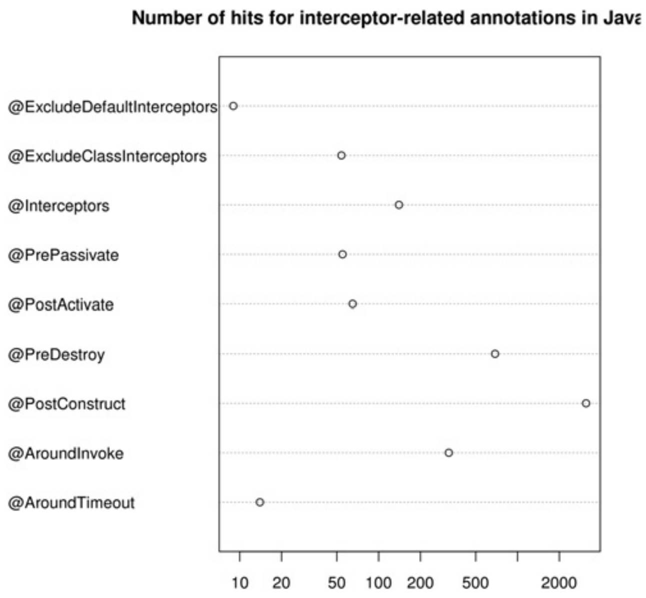


Fig. 15 *Most popular annotations @PostConstruct and @PreDestroy are not necessarily used to introduce interceptors. Least popular one @AroundTimeout introduces the most recent type of interceptors*

The numbers of hits associated with the deployment descriptor XML files were clearly lower that those associated with annotations: the most popular tag <around-invoke> corresponds to 12 hits, while Google code search failed to find files containing the <post-construct> tag. These figures suggest that developers prefer to specify behaviour within Java code as opposed to separate configuration files, and therefore, the results agree with our earlier observation that maintenance of centralised configuration files might become prohibitive [26].

Summarising the preceding discussion, we remark that such annotations as @PostConstruct and @PreDestroy are not necessarily related to interceptors, and the use of timeout method interceptors in the 'Google code search' code base was very limited. Therefore in Section 5.2, we focus on an empirical study of business method interceptors.

5.2 Business method interceptors in practice

To obtain insights in how business method interceptors are used in practice, we have downloaded repositories of all projects found via 'Google code search' that contain files with @AroundInvoke or @Interceptors annotations. In this way, we have obtained 108 repositories: the number of repositories does not equal to the number of @AroundInvoke-hits together with the number of @Interceptors-hits, since multiple annotations, and therefore, hits can be present in the same repository. Next, we have observed that repositories usually contain multiple interrelated versions of the same software system, for example, in the /trunk and /tags folders. Moreover, some repositories contain multiple /trunk and /tags folders associated with different subsystems. To ensure statistical soundness of the results to come, we have decided to consider only one software version per repository, namely the 'development version' consisting of the files in /trunk folders.

Recall that the precondition of the reverse engineering algorithm described in Section 3.1.1 is that the input method is a 'business method', that is, satisfies the requirements of [3] and is not an around-invoke method. The current implementation of I2SD does not verify these conditions completely: we check only that the input method is public and that it is not an around-invoke method. In total, systems in the 108 repositories contained 323 014 public and non-around-invoke methods: the smallest system included 11 methods, the largest one – 60 681. A more precise implementation of the conditions defining when a method is a business method [3] is considered as a part of the future work.

Out of 323 014 methods that could possibly have been intercepted, only 5057 ($\simeq$1.57%) methods were intercepted. This result seems to support the finding of our previous study [27] stating that the use of interceptors is usually limited to a relatively small part of the software system. Closer inspection of the data reveals, however, a slightly different picture:

- 19 systems did not include intercepted methods at all. This might seem puzzling since we have downloaded only projects containing files with @AroundInvoke or @Interceptors annotations. However, in 17 out of 19 cases presence of interceptors does not necessarily indicate their use. For instance, the carebearmail.googlecode.com project defines one interceptor but does not define intercepted methods. Similarly, materiasucb.googlecode.com defines one interceptor and contains a class referring to the interceptor through @Interceptors annotation. However, the referring class does not contain methods. The two remaining systems, redams.googlecode.com and rockframework.googlecode.com make use of Apache Struts interceptors rather than EJB interceptors. While Apache Struts interceptors are essentially similar to the interceptors considered in this paper, the Apache Struts deployment descriptors are not called ejb-jar.xml, and therefore, are not recognised by I2SD.
- 36 systems have less than 1% of intercepted methods, and 31 additional systems have between 1% and 5% of the methods. For these projects, we can indeed claim that the use of interceptors is limited as suggested in [27].
- 18 further systems have between 5 and 25% of the intercepted methods.
- Finally, four systems have almost all methods being intercepted. In these systems, a default interceptor is specified in the deployment descriptor, and therefore, calling any business method should involve invocation of the default interceptor, unless this invocation has been explicitly excluded using <exclude-default-interceptors> or @ExcludeDefaultInterceptors.

The overall distribution of the percentages of methods being intercepted can be considered log-normal (after elimination of the zeros and the logarithmic transformation, Shapiro-Wilk's $W = 0.9871$ and p-value $= 0.5323$, that is, normality hypothesis cannot be rejected).

Using I2SD we have reverse engineered sequence diagrams for the methods above and calculated depths of scenarios, that is, numbers of calls in the sequence diagram [12]. Every sequence diagram consists of a series of interceptor invocations, each followed by a call to InvocationContext.proceed(), and the final call of the business method itself. Therefore depth of scenario is always an odd number:

- 617 methods (12%) induce scenarios of depth 1, that is, scenarios that do not involve interceptor invocation. Scenarios of depth 1 are possible if no interceptors are indicated for the business method, or if some of the interceptors are explicitly excluded with <exclude-class-interceptors> or @ExcludeClassInterceptors. We have also observed that in some situation scenario depth can be underestimated by I2SD. For instance, method getUserName from com.alesj.blade.login.LoginAction in the bladecut.googlecode.com repository contains annotation @Interceptors(SeamInterceptor.class). The file SeamInterceptor.java is part of the Seam framework [6] and, hence, its source code is not included in the bladecut.googlecode.com, and therefore, is not included in the analysis. Hence, I2SD calculates the depth of the scenario for getUserName as 1.
- The lion's share of the intercepted methods, 4064 out of 5057 or 80%, induce a scenario of depth 3, corresponding to invocation of one interceptor.
- 317 methods or 6% induce a scenario of depth 5, corresponding to two interceptors.
- The remaining 59 methods induce scenarios deeper than 5 with the deepest scenarios of depth 13 (six interceptors). The only packages with methods inducing scenario of depths 9, 11 and 13 are subpackages of org.apache.openejb.test in svn.apache.org.

Inspired by the previous observation that the test-packages in svn.apache.org include deep scenarios, we have decided to check whether, in general, methods in test packages induce deeper scenarios than in non-test packages. As test packages we consider packages containing segments 'test' or 'tests' in the fully qualified names. Formally, we state the following hypotheses:

- H_0: Depths of scenarios for methods in test-packages are following the same distribution as those for methods in non-test-packages;
- H_a: Depths of scenarios for methods in test-packages are deeper than those for methods in non-test-packages.

Since the distribution of the scenario depths for methods in test packages is not normal (Shapiro–Wilk's test statistic $W = 0.7832$, p-value $<2.2 \times 10^{-16}$) to test the hypotheses we apply the Mann–Whitney test, a non-parametric counterpart of the classic t-test for two samples. The test statistics equals 624 857.5 and the p-value equals 6.36×10^{-5}, that is, we can confidently reject H_0 and claim that depths of scenarios for methods in test-packages are deeper than those for methods in non-test-packages. Furthermore, H_0 can be rejected even if the aforementioned testing subpackages of org.apache.openejb.test are excluded from consideration (p-value calculated by the Mann–Whitney test equals 1.48×10^{-7}). Rejecting H_0 suggests that at the time of data collection (January 2012) the depths of interceptor scenarios in the production code of open-source software systems was limited.

Additional support of the limited depth of interceptor scenarios can be found in the number of systems with methods inducing scenarios of a given depth. Fig. 16 illustrates that the number of systems with methods inducing scenarios of a given depth rapidly decreases with the increase of scenarios depth: 54 systems have methods inducing scenarios of depth 3, and only 15 – of depth 5.

Threats to validity As any empirical investigation, our study is subject to a number of threats to validity. One commonly distinguishes three kinds of experiment validity: construct validity, internal validity and external validity [28].

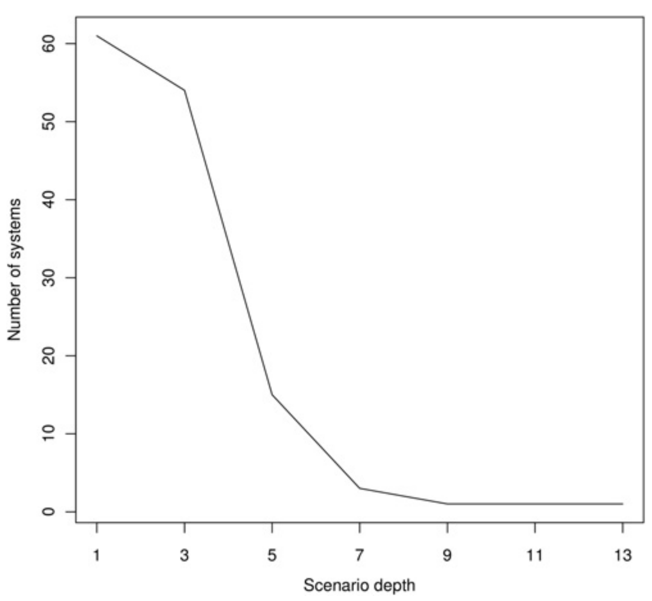


Fig. 16 *In most systems the depths of interceptor scenarios is limited*

'Construct validity' can be threatened by calculation of the scenarios depths performed by I2SD. As explained above, I2SD tends to underestimate depths of scenarios that involve non-EJB interceptors, for example, originating from the Seam framework. Moreover, since the current implementation of I2SD approximates the notion of a business method of [3] by checking whether the input method is public and it is not an around-invoke method, the data obtained includes values for methods that should not be considered as business methods, and, hence, should be excluded.

'Internal validity' pertains to soundness of procedures used to derive conclusions within the experimental settings. To ensure internal validity we have paid special attention to the choice of appropriate statistical procedures.

Finally, 'external validity' pertains to the ability to generalise the conclusions beyond the experimental settings. In our case, this would indicate the ability to generalise our conclusion beyond the systems included in the benchmarks collection. The systems we have included are open-source and are predominantly hosted at googlecode.com. To ensure external validity we intend to replicate this study on a larger and more diverse code base. Moreover, to ensure that our conclusion applies to commercial software, the code base has to include commercial software systems as well.

5.3 DataPortal and WasabiBeans

To illustrate how interceptors are used in practice, we discuss two systems from the collection above. We have opted for two systems with a comparable number of Java files, DataPortal and WasabiBeans. In addition to business method interceptors, in this subsection we also discuss life-cycle callback interceptors.

DataPortal The first system we consider is the DataPortal [http://dataportal.googlecode.com/], 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 and only one file, DataPortal.java, has 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. Fig. 17 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

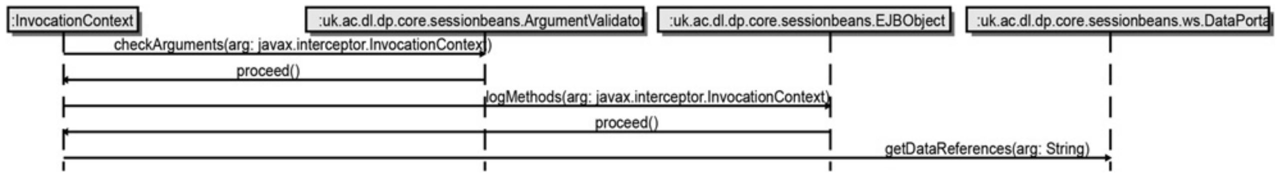

:InvocationContext
:uk.ac.dl.dp.core.sessionbeans.ArgumentValidator
:uk.ac.dl.dp.core.sessionbeans.EJBObject
:uk.ac.dl.dp.core.sessionbeans.ws.DataPorta
checkArguments(arg: javax.interceptor.InvocationContext)
proceed()
logMethods(arg: javax.interceptor.InvocationContext)
proceed()
getDataReferences(arg: String)

Fig. 17 *Sequence diagram created by I2SD for DataPortal.getDataReferences*

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 (Fig. 18). 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.

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 [http://code.google.com/p/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, that is, 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 JCRSessionInterceptor to 15. In all beans, the JCRSessionInterceptor interceptor 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. Inheritance and interceptor use in WasabiBeans are summarised in Fig. 19.

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.

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.

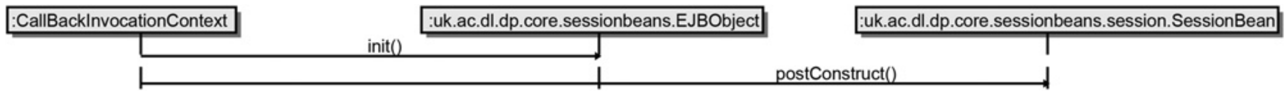

:CallBackInvocationContext
:uk.ac.dl.dp.core.sessionbeans.EJBObject
:uk.ac.dl.dp.core.sessionbeans.session.SessionBean
init()
postConstruct()

Fig. 18 *Sequence diagram for creation of an instance of SessionBean*

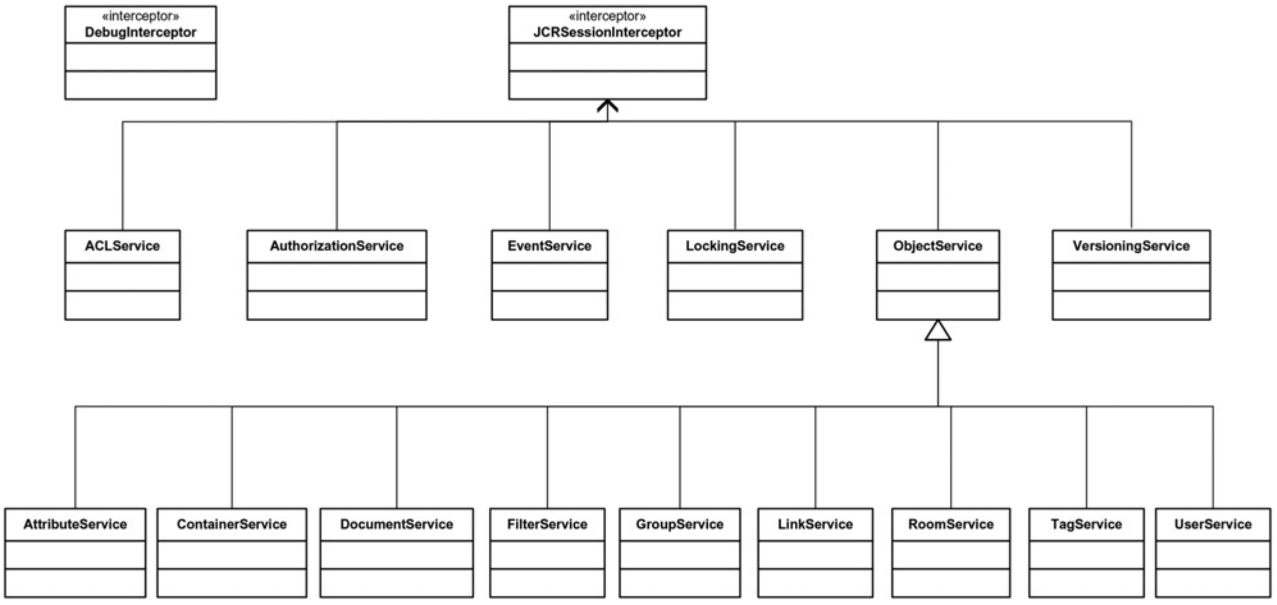


Fig. 19 *Inheritance and interceptor use in WasabiBeans*

The limited adoption of interceptors is not surprising as they express dependencies that are known to hinder development of EJB applications [7, 8, 29], 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: five for business method interceptors and two 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 interceptor's 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.

6 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 [7, 8, 29, 30]. To facilitate these tasks a monitoring system [8] and a profiler [29] 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, that is, I2SD is capable of analysing incomplete, and hence, non-executable programs.

The current paper builds on and extends our previous work on dependency injection [26], and specifically on EJB interceptors [10, 27]. Specifically, we extend [10, 27] by discussing EJB timeout methods (Section 3.3) and applying I2SD for empirical evaluation of the interceptors use (Sections 5.1 and 5.2).

In general, 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 [31–34], 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 [35–40]. Furthermore, this research has lead to a number of reverse engineering tools (see [41] for a recent survey of the area). Dynamic approaches are known to produce more precise results: for example, owing to presence of dynamic binding. Applicability of the dynamic approaches is, however, restricted by the fact that the system analysed should be executable, while static approaches are capable of analysing 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 [31, 32, 34], 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 [10]. Building on and extending [10] 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 [42, 43]. 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 [44] based on EJB Framework Specific Modeling Language [45], 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 an 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 [25]. 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 EJB3 standard [2].

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 [46]. Cross-cutting concerns have been modelled using UML sequence diagrams [47] and used to derive flow graphs and flow trees to support test generation [48]. 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 [49] 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 Sereni and de Moor [49] did not support the 'around' advice, essential to implement business method interceptors. These shortcomings have been addressed in [50], 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, that is, programs with interceptors have been used as means to implement reverse engineering techniques [51, 52].

7 Conclusions

In this paper, we have introduced I2SD, a reverse engineering tool for EJB 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. We stress that the current prototype implementation focuses on providing the desired functionality rather than on performance. While inferring a sequence diagram for a given business method or a life-cycle event happens almost instantaneously, performance of the batch processing of thousands of queries should be improved.

I2SD has been implemented as a highly modular pipe-and-filter architecture [15]. This architectural decision facilitates evolution of I2SD and reuse of its individual components [53]. We have discussed two use cases showing the applications of I2SD during software development (Sections 4.1 and 4.3) and as part of the quality assessment (Section 4.2). To support the application of I2SD for quality assessment we have conducted an empirical evaluation of the interceptors use in practice (Section 5), and observed that the depths of interceptor scenarios in the production code of open-source software systems is limited.

As future work we consider a number of possible directions. The first direction pertains to interpretation of the EJB specifications by I2SD. From the language perspective, I2SD will be extended to incorporate more recent extensions to the interceptor model such as @InterceptorBinding [54]. We also intend to implement a more precise check of the requirements a method has to satisfy to be considered a business method [3]. The second direction is related to the way user can control I2SD. Sequence diagrams produced by the current version of I2SD are focus on interceptors only and do not consider method bodies. We intend to extend the algorithm and provide the user with the ability to indicate the desired nesting level, or method bodies that should be included or excluded in the analysis. This would also allow Bob (cf. Section 4.2) to obtain additional metrics related to the depth of the scenario, for example, the number of interceptors at a given nesting level. Finally, using modern aggregation techniques [55, 56] Bob can obtain a general picture of the interceptor usage from metrics values obtained for individual scenarios. The third direction is related to integration of I2SD with other software systems. I2SD will be connected to our visual software analytics toolset SQuAVisiT [17]. This connection will make EJB applications immediately amenable for multiple analysis and visualisation techniques already integrated in SQuAVisiT. Next, in addition to the NetBeans plugin described in Section 3, we plan to integrate I2SD in Eclipse. Since the core part of the implementation is a standard Java module, only the GUI and the plug-in mechanism should be reimplemented to achieve Eclipse integration. 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 4.1, whereas 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 4.2 using the I2SD + SQuAVisiT combination. After implementing the aforementioned tool extensions we will revisit the empirical evaluation of Section 5.2.

8 Acknowledgments

The first author cordially acknowledges Laboratory for Quality Software (LaQuSo) for making this research possible. This research has been conducted during the third author's stay at Eindhoven University of Technology. During this period he has been supported by the ERASMUS scholarship.

9 References

1 Narasimhan, P., Moser, L.E., Melliar-Smith, P.M.: 'Using interceptors to enhance CORBA', *IEEE Comput.*, 1999, **32**, (7), pp. 62–68

2 Sun Microsystems: Sun Microsystems. JSR-220 Enterprise JavaBeans 3.0 (Final Release), 2006

3 EJB 3.1 Expert Group: EJB 3.1 Expert Group. JSR-318 Enterprise JavaBeans, Version 3.1 (Final Release), 2009

4 EJB 3.1 Expert Group: EJB 3.1 Expert Group. Interceptors 1.1, 2009

5 Vanbrabant, R.: 'Google Guice: agile lightweight dependency injection framework' (APress, Berkeley, CA, USA, 2008)

6 Red Hat: Red Hat. Seam – Contextual Components. A Framework for Java EE 5, 2007

7 Bellur, U.: 'A methodology & tool for determining inter-component dependencies dynamically in J2EE environments'. Proc. Third Int. Conf. Autonomic and Autonomous Systems, Washington, 2007, DC, USA, pp. 14:1–14:8

8 Kehe, W., Zhuo, W., Xing, Z., Gang, M.: 'Design and implementation of the monitoring system for ejb applications based on interceptors'. Thirdrd Int. Conf. Advanced Computer Theory and Engineering (ICACTE), 2010, vol. 4, pp. V4-5–V4-9

9 Tilley, S.R., Huang, S.: 'A qualitative assessment of the efficacy of UML diagrams as a form of graphical documentation in aiding program understanding', SIGDOC, 2003, pp. 184–191

10 Serebrenik, A., Roubtsov, S.A., Roubtsova, E.E., van den Brand, M.G. J.: 'Reverse engineering sequence diagrams for Enterprise JavaBeans with business method interceptors', WCRE, 2009, pp. 269–273

11 Murphy, G.C., Kersten, M., Findlater, L.: 'How are Java software developers using the Elipse IDE?', *IEEE Softw.*, 2006, **23**, (4), pp. 76–83

12 Muskens, J., Chaudron, M.R., Westgeest, R.: 'Software architecture analysis tool: software architecture metrics collection'. Proc. Third PROGRESS Workshop on Embedded Systems, 2002, pp. 128–139

13 Panda, D., Rahman, R., Lane, D.: 'EJB 3 in action' (Manning Publications Co., Greenwich, CT, USA, 2007)

14 Goncalves, A.: 'Beginning Java EE p Platform with GlassFish 3: from novice to professional' (Apress, Berkely, CA, USA, 2009, 1st edn.)

15 Allen, R.B., Garlan, D.: 'A formal approach to software architectures'. Proc. of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture – Information Processing 1992, Amsterdam, The Netherlands, pp. 134–141

16 Copeland, T.: 'Generating parsers with JavaCC' (Centennial Books, Alexandria, VA, USA, 2009, 2nd edn.)

17 van den Brand, M.G.J., Roubtsov, S.A., Serebrenik, A.: 'SQuAVisiT: a flexible tool for visual software analytics', CSMR, 2009, pp. 331–332

18 Harold, E.R.: 'Processing XML with Java' (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002), URL http://www.cafeconleche.org/books/xmljava/

19 Sun Microsystems: Sun Java System Application Server 9.1 Reference Manual, 2007

20 Jackson, D.: 'Scalable vector graphics (SVG): the world wide web consortium's recommendation for high quality web graphics'. ACM SIGGRAPH 2002 Conf. Abstracts and Applications, SIGGRAPH'02, New York, NY, USA, pp. 319–319

21 Savard, M.: 'Development of OASIS v2'. Tech. Rep. CR 2008–332, Defence Research and Development Canada, October 2008. URL http://pubs.drdc.gc.ca/PDFS/unc79/p530476.pdf

22 Oracle: Java RMI over IIOP. Technology Documentation Home Page, 2010. URL http://docs.oracle.com/javase/1.4.2/docs/guide/rmi-iiop/index.html

23 Object Management Group: Java to IDL Language Mapping, 2008. Version 1.4

24 Yawn, M.: 'J2EE and Jax: developing web applications and web services' (Prentice-Hall Professional, Upper Saddle River, NJ, USA, 2003)

25 Baldoni, R., Marchetti, C., Verde, L.: 'CORBA request portable interceptors: analysis and applications', *Concurrency Comput., Pract. Exp.*, 2003, **15**, (6), pp. 551–579

26 Roubtsov, S.A., Serebrenik, A., van den Brand, M.G.J.: 'Detecting modularity 'Smells' in dependencies injected with Java annotations'. Software Maintenance and Reengineering, European Conf., 2010, Los Alamitos, CA, USA, pp. 244–247

27 Roubtsov, S.A., Serebrenik, A., Mazoyer, A., van den Brand, M.G.J.: 'I2SD: reverse engineering sequence diagrams from Enterprise Java Beans with interceptors'. SCAM, 2011, pp. 155–164

28 Perry, D.E., Porter, A.A., Votta, L.G.: 'Empirical studies of software engineering: a roadmap'. Proc. of the Conf. The Future of Software Engineering, ICSE'00, New York, NY, USA, 2000, pp. 345–355

29 Klaczewski, P., Wytrebowicz, J.: 'j2eeprof—a tool for testing multitier applications', Software Engineering Techniques: Design for Quality, SET 2006, 17–20 October, 2006, Warsaw, Poland, *IFIP*, vol. 227, pp. 199–210

30 Arthur, J., Azadegan, S.: 'Spring framework for rapid open source J2EE Web application development: a case study'. Sixth Int. Conf. Software Engineering, Artificial Intelligence, Networking and Parallel/ Distributed Computing, 2005 and First ACIS Int. Workshop on Self-Assembling Wireless Networks, SNPD/SAWN 2005, pp. 90–95

31 Rountev, A., Connell, B.H.: 'Object naming analysis for reverse-engineered sequence diagrams', Int. Conf. Software Engineering, 2005, pp. 254–263

32 Rountev, A., Volgin, O., Reddoch, M.: 'Static control-flow analysis for reverse engineering of UML sequence diagrams', PASTE, 2005, pp. 96–102

33 Tonella, P., Potrich, A.: 'Reverse engineering of the interaction diagrams from C++ code'. Int. Conf. Software Maintenance, 2003, pp. 159–168

34 Korshunova, E., Petković, M., van den Brand, M.G.J., Mousavi, M.R.: 'CPP2XMI: reverse engineering of UML class, sequence, and activity diagrams from C++ source code'. WCRE, 2004, pp. 297–298

35 Briand, L.C., Labiche, Y., Leduc, J.: 'Toward the reverse engineering of UML sequence diagrams for distributed Java software', *IEEE Trans. Softw. Eng.*, 2006, **32**, (9), pp. 642–663

36 Briand, L.C., Labiche, Y., Miao, Y.: 'Towards the reverse engineering of UML sequence diagrams', WCRE, 2003, pp. 57–66

37 Delamare, R., Baudry, B., Le Traon, Y.: 'Reverse-engineering of UML 2.0 sequence diagrams from execution traces'. Workshop on Object-Oriented Reengineering at ECOOP 06, Nantes, France

38 Guéhéneuc, Y.G., Ziadi, T.: 'Automated reverse-engineering of UML v2.0 dynamic models', Proc. Sixth ECOOP Workshop on Object-Oriented Reengineering, 2005, Glasgow, UK

39 Oechsle, R., Schmitt, T.: 'JAVAVIS: automatic program visualization with object and sequence diagrams using the Java debug interface (JDI)', Software visualization, 2002 (*LNCS*, **2269**), pp. 176–190

40 Richner, T., Ducasse, S.: 'Using dynamic information for the iterative recovery of collaborations and roles'. ICSM, 2002, pp. 34–43

41 Bennett, C., Myers, D., Storey, M.A.D., *et al.*: 'A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams', *Journal of Softw. Maint.*, 2008, **20**, (4), pp. 291–315

42 Crawford, W., Kaplan, J.: 'J2EE design patterns' (O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003)

43 Dudney, B., Krozak, J., Wittkopf, K., Asbury, S., Osborne, D.: 'J2EE antipatterns' (John Wiley & Sons, New York, NY, USA, 2002, 1st edn.)

44 Stephan, M.: 'Detection of Java EE EJB antipattern instances using framework-specific models'. Master's thesis, University of Waterloo, Waterloo, 04/2009 2009

45 Antkiewicz, M., Czarnecki, K., Stephan, M.: 'Engineering of framework-specific modeling languages', *Softw. Eng. IEEE Trans.*, 2009, **35**, (6), pp. 795–824

46 Stein, D., Hanenberg, S., Unland, R.: 'Join point designation diagrams: a graphical representation of join point selections', *Int. J. Softw. Eng. Knowl. Eng.*, 2006, **16**, (3), pp. 317–346

47 Deubler, M., Meisinger, M., Rittmann, S., Krüger, I.: 'Modeling crosscutting services with UML sequence diagrams', MoDELS, 2005 (*LNCS*, **3713**), pp. 522–536

48 Xu, W., Xu, D.: 'A model-based approach to test generation for aspect-oriented programs'. First Workshop on Testing Aspect-Oriented Programs, 2005, Chicago, IL, USA, pp. 1–6

49 Sereni, D., de Moor, O.: 'Static analysis of aspects'. Proc. of the Second Int. Conf. Aspect-Oriented Software Development, 2003, New York, NY, USA, pp. 30–39

50 Bernardi, M.L., Di Lucca, G.A.: 'An interprocedural aspect control flow graph to support the maintenance of aspect oriented systems'. Int. Conf. Software Maintenance, 2007, pp. 435–444

51 Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., Yan, H.: 'Discovering architectures from running systems'. *IEEE Trans. Softw. Eng.*, 2006, **32**, (7), pp. 454–466

52 Taïani, F., Killijian, M.O., Fabre, J.C.: 'COSMOPEN: dynamic reverse engineering on a budget. How cheap observation techniques can be used to reconstruct complex multi-level behaviour', *Softw. Pract. Exp.*, 2009, **39**, (18), pp. 1467–1514

53 Garlan, D., Shaw, M.: 'An introduction to software architecture', in Ambriola, V., Tortora, G., (Eds.): 'Advances in software engineering and knowledge engineering' (World Scientific Publishing Company), 1994, pp. 1–39

54 JSR-299 Expert Group: JSR-299: Contexts and Dependency Injection for the Java EE platform, 2009

55 Serebrenik, A., van den Brand, M.G.J.: 'Theil index for aggregation of software metrics values'. Int. Conf. Softw. Maint, 2010, pp. 1–9

56 Vasilescu, B., Serebrenik, A., van den Brand, M.G.J.: 'You can't control the unfamiliar: a study on the relations between aggregation techniques for software metrics'. Int. Conf. Software Maintenance, 2011, pp. 313–322

57 IBM: EJB 3.x interceptors, 2012. http://www14.software.ibm.com/webapp/wsbroker/redirect?version=matt&product=was-nd-dist&topic=rejb_3interceptors