Analyzing Program Dependencies in Java EE Applications

Anas Shatnawi, Hafedh Mili, Ghizlane El Boussaidi, Anis Boubaker, Yann-Gaël Guéhéneuc,
Naouel Moha, Jean Privat, Manel Abdellatif
LATECE Laboratory, Université du Québec à Montréal, Montréal, Canada

***Abstract*—Program dependency artifacts such as call graphs help support a number of software engineering tasks such as software mining, program understanding, debugging, feature location, software maintenance and evolution. Java Enterprise Edition (JEE) applications represent a significant part of the recent legacy applications, and we are interested in modernizing them. This modernization involves, among other things, analyzing dependencies between their various components/tiers. JEE applications tend to be multilanguage, rely on JEE container services, and make extensive use of late binding techniques–all of which makes finding such dependencies difficult. In this paper, we describe some of these difficulties and how we addressed them to build a dependency call graph. We developed our tool called DeJEE (Dependencies in JEE) as an Eclipse plug-in. We applied DeJEE on two open-source JEE applications: Java PetStore and JSP Blog. The results show that DeJEE is able to identify different types of JEE dependencies.**

***Index Terms*—program dependency, code analysis, Java EE application, modernization, container services, server pages.**

I. Introduction

Program dependency artifacts such as call graphs help support a number of software engineering tasks such as software mining, program understanding, debugging, feature location, maintenance and evolution. Since the early 2000's, most new enterprise applications were developed using distributed objects, with Java technologies taking the lead. The early generation of JEE (Java Enterprise Edition) applications suffered from many ills due to a combination of, 1) the inherent limitations of the technology, and 2) an improper use of its features (see e.g. EJB antipatterns [1]). JEE applications represent a significant part of the recent legacy applications. We are interested in analyzing such applications to support their modernization, including migrating them to more loosely coupled architectures such as *Service-Oriented Architectures* (SOA). This involves building several representations of such applications that highlight different dependencies between the application elements.

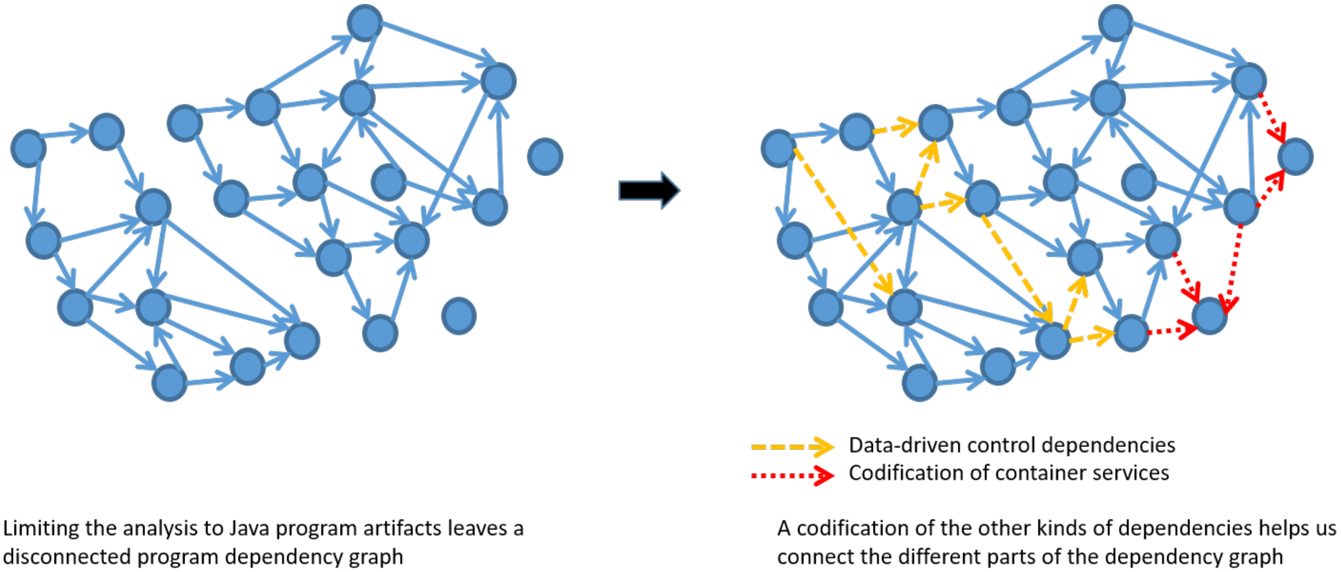
However, JEE applications present a number of characteristics that make analysis difficult:

- they tend to be distributed and multi-tiered –as opposed to monolithic applications– which makes tracing calls across tiers more difficult;
- a number of (calls) dependencies are implicit in frameworks and services offered by JEE servers/containers: they are not 'visible' in the user code;
- they tend to make extensive use of late binding techniques, including language-specific features (e.g. reflexion package) and coding and design idioms/tricks;
- a number of dependencies will be recorded in configuration files that have ad-hoc syntax, and whose semantics are fully embodied in the tools that consume them; and
- they usually embed portions of code written in other programming/data presentation languages other than Java.

All of these considerations mean that to get a complete representation of relationships between program elements, we need to augment the traditional *unilingual* program static analysis techniques with other kinds of analyses, involving other kinds of artifacts, but also, possibly, involving the codification of services offered by containers/application servers. Figure 1 illustrates this.

Many approaches were proposed in the literature to analyze program dependencies in JEE applications with different purposes. Most of these approaches focus on a subset of JEE technologies that is part of a tier. For examples, Naumovich and Centonze analyzed EJBs and JEE access policies [3], Kirkegaard and Moller focused on the analysis of Java Servlets [4], and Zaidman supports the understanding of Ajax-based web applications [6]. While each one of these technologies present significant analysis challenges, in their own right, none of the approaches attempt an integrated view of the technologies.

In this paper, we look at some of the challenges posed by JEE characteristics to build a *dependency call graph*, and illustrate with some details the problems–and solutions– of uncovering invocation relationships between server pages (Servlets, JSPs and JSFs). We also give some insights on our envisioned approach to discover dependencies related to *container services* such as RMI, life-cycle management, and persistence. We rely on the OMG's Knowledge Discovery Meta-model (KDM) [2] to represent different program elements and dependencies in JEE applications. To implement our solution, we developed the DeJEE (**De**pendencies in **JEE**) tool. DeJEE is implemented as an Eclipse plug-in based on the MoDisco [8] tool. We applied DeJEE on two open-source JEE applications; Java PetStore and JSP Blog. We compared dependency call graphs resulting from DeJEE and MoDisco tools. We find that DeJEE increases MoDisco's recall by detecting 70.5% more accurate program elements as well as their related dependencies in average for our case studies. DeJEE does not only identify a dependency call graph, but


Data-driven control dependencies
Codification of container services

Limiting the analysis to Java program artifacts leaves a disconnected program dependency graph

A codification of the other kinds of dependencies helps us connect the different parts of the dependency graph

Fig. 1. Java code analyses have to be complemented with the analysis of, 1) other program artifacts, and 2) services offered by the run-time infrastructure

also it detects dependencies toward program elements that do not exist in the implementation.

The rest of this paper is organized as follows. We first provide a short overview of the anatomy of a JEE application (Section II). In Section III, we identify the underlying problems at analyzing dependencies in JEE applications and propose our solutions. We discuss the implementation and some preliminary results of DeJEE tool on two case studies in Section IV. Section V provides a discussion about some important issues related to the proposed approach. Next, we explore related work in Section VI. Finally, we present our conclusion and future work in Section VII.

II. The Anatomy of JEE Applications

A. *Overview of JEE applications*

JEE applications are managed, developed and deployed based on a multi-tier distributed model. In this context, the application logic of a JEE application is decomposed into a set of components distributed among different tiers based on their functionalities. Fig. 2 shows an example of two JEE applications that are split into four tiers: client, web, business logic and data tiers. The client tier can be client applications that access business logic functionalities running on server-side machines, or HTML pages that use HTTP requests to communicate with web tier components. The web tier implements the presentation logic of the JEE application. The business tier encapsulates the business logic of the JEE application. The data tier stores the enterprise data.

JEE applications are implemented using different technologies that can be written either using normal Java code (e.g. Servlets, JavaBeans, Managed Beans) or scripting languages using XML tags (e.g. JSP, JSF). The different technologies communicate through various mechanisms. For instance, clients may either use HTTP requests or Remote Method Invocation (RMI) to request server's provided services, while Enterprise Beans use Java Database Connectivity (JDBC)–or more generally, the Java Connector Architecture (JCA)–to access the Enterprise data stored on the data tier.

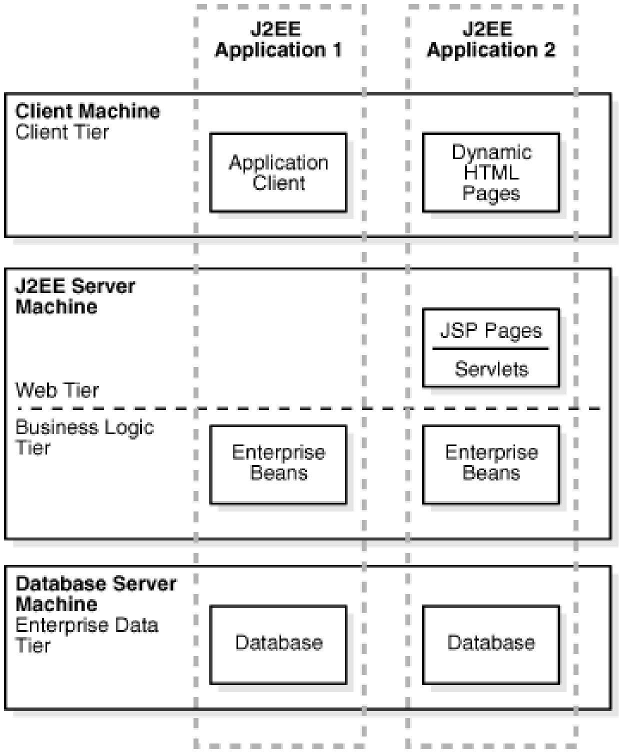


Fig. 2. Architecture of multi-tiered JEE applications

B. Server pages in JEE applications: Servlets, JSPs, and JSFs

In web tier layer, server pages allow to extend web application capabilities to offer clients functionalities through HTTP requests. Server pages can be implemented based on three JEE technologies: Servlets, JSPs, and JSFs. Servlets represent the traditional implementation of server pages and are implemented using normal object-oriented Java code (see *javax.Servlet* and *javax.Servlet.http* packages in Java APIs), while the presentation of the corresponding server pages is realized through string parameters attached to HTTP requests (readable by the client browser).

JSPs (JavaServer Pages) and JSFs (JavaServer Faces) are built on the top of Java Servlets and are used for developing dynamic server-side pages based on scripting languages (based on XML). They can contain a mix of static content (XML/HTML tags) and dynamic content (like dynamic JSP tags). They provide reusable components via *tag libraries* (taglibs). In addition, developers are allowed to include normal Java code fragments and to invoke external Java components such as JavaBeans and Managed Beans. JSF enhances JSP in several ways including its model-view-controller approach and its componentization standards.

III. Analyzing Dependencies in JEE Applications

We identified several problems during the analysis of our two case studies. These are: 1) the various JEE technologies conform to different (exogenous) meta-models; 2) the container services' related dependencies are implicit; 3) the use of multi-language code in single source files; 4) the embedding of dependencies in string literals; 5) the different mechanisms to codify dependencies; and 6) the need to look in different files to interpret dependencies.

In the remaining of this section, we explain and analyze these problems and sketch their solutions.

A. Problem with the variety of meta-models

1) Problem: Different JEE technologies were designed for different objectives. Therefore, they follow different meta-models that have various structures based on the inventors' point of views about what it is the right lexical, syntax and semantics. As servlets are implemented using Java code, they follow an object-oriented meta-model where their program elements are associated to classes, methods and attributes, and their program dependencies are mainly based on method invocations, attribute access and object instantiations. On the other hand, JSPs and JSFs follow an XML-based meta-model such that their program elements are based on XML tags that define their attributes and dependencies. This can be generalized to other JEE technologies, including JEE container services. Therefore, there is a need for an unifying standard representation that is able to represent all of JEE technologies considering their program elements and dependencies.

2) Solution: To solve the problem of the variety of meta-models, we select to use an intermediate language-independent meta-model. Such a meta-model should be able to offer a common interchange format that unifies all of JEE technologies with respect to their program elements and relationships as well. Among existing language-independent meta-models (such as KDM [2], FAMIX [7] and Abstract Syntax Tree), we selected KDM (Knowledge Discovery Meta-model), an Object Management Group (OMG) standard for the following reasons. First, it is an open specification that allows us to extend it through a light-weight extension mechanism [22, p. 39-46]. Second, it is able to represent software artifacts physically and logically in terms of entities and relations at different levels of abstraction based on the container concept. This allows one to perform several software engineering tasks such as software analysis, re-engineering, refactoring, modernization, quality analysis, etc. Third, it can represent the entire legacy software artifacts including source code, design, configuration files, etc. For technical details about the KDM specification please refer to [2], [22].

For KDM tool support, we considered two of the most mature tools: MoDisco [8] and KDM Analytics [9]. However, MoDisco is the only one that offers open-source implementation that can be extended to understand JEE dependencies. Therefore, we used MoDisco as an underlying tool to extract and to manipulate KDM models.

However, MoDisco suffers from two limitations. First, the current version only supports the construction of KDM models from to object-oriented languages like Java and C++. Although it does permit the analysis of non-object oriented languages, such as JSP, JSF or HTML, the concepts are extracted following an alternate (i.e. non-KDM) meta-model. Each concept is represented in terms of string literals. These string literals need to be analyzed for mining dependencies within and across the JSP and JSF pages, in addition to dependencies with other JEE technologies including Bean components and other Java source code.

The second problem with MoDisco is that configuration files are statically represented as XML-based models without interpreting their content with respect to the other software artifacts. As a result, the KDM model produced by MoDisco does not take into account any of the JEE framework specifics. To address these issues, we developed a specific processor to parse JSPs, JSFs and HTML, in order to represent them (and their dependencies) by complementing MoDisco's KDM model. We define JSPs, JSFs and HTML instances in terms of *KDMEntity*, and dependency instances between pages are realized in terms of *KDMRelationship*.

B. Problem with codifying container services

1) Problem: JEE *containers* offer a number of services to host applications such as remote method invocation (RMI), lifecycle management, transactions, persistence, and security. Unlike CORBA services, which developers have to *explicitly* invoke through service APIs, JEE services do not require end-user programming, but are specified *declaratively* through *deployment* descriptors. Many such services rely on *callback methods* to be implemented by application developers, to either provide custom behavior for the service, or to notify the user code that the service has been rendered. This means that *several* program dependencies will *not* be visible in application code. Examples[1] include:

- A call on a `create` (or `find`) method on an EJB Home object on the client side, results into calling an `ejbCreate` (or `ejbFind`) method on the *bean class*, with different return types.
- Some callback methods are *not* meant to be *called* by user code, such as lifecycle callback methods (`ejbPostCreate`, `ejbActivate`, etc.).
- Some methods will be triggered by the occurrence of conditions/events, not all of which are directly traceable to application code actions.

These examples show us that uncovering JEE container specific dependencies is not an easy problem to solve. Even if we the source code of the JEE server–or where able to reverse engineer it–we would face two problems: 1) we would have to delve into reams of code dependencies that would needlessly clutter our dependency graph, and 2) even that would not be enough because of the heavy reliance of the server on Java reflection. Thus, we are better off *codifying* the dependencies inherent in container services *explicitly*.

2) Solution: We codify the dependencies induced by container services as a set of $pattern \rightarrow dependency$ rules, where $pattern$ describes a particular code configuration, and $dependency$ describes a dependency induced by the container, when such pattern occurs. Referring to the first example above:

$(\forall\ type\ T\ s.t.\ isHomeInterface(T))(\forall\ method\ m \ \epsilon\ T\ )$
$(\forall\ method\ mb\ \epsilon\ BeanClass(EJB(T))\ )$
$((\ name(m)\ =\ ”create”\ and\ name(mb)\ =\ ”ejbCreate”$
$paramlist(m)\ =\ paramlist(mb)\ )$
$\rightarrow\ (\ add\ dependency\ (T.m\ calls\ BeanClass(EJB(T)).mb)))$

where `isHomeInterface(.)` returns true for types that extend the `EJBHome` interface, `EJB(.)` is a function that returns the EJB associated with the home interface T, and `BeanClass(.)` is a function that takes an EJB and returns the bean implementation class[2]. Practically, all of these methods will operate on the KDM representation of the code (see Fig. 3).

The rule above is one of the simplest ones. A study of the main services (RMI, persistence, life-cycle management, transaction, and security) identified the following variations:

- Some code properties are implicit in the code (e.g. whether a type is a home interface) whereas others are specified in deployment descriptors and configuration files (e.g. transactional and security properties).
- Beyond the basic call, there are a number of *other* interesting relationships, including, precedence *without* explicit invocation, which have architectural implications.
- Differences between type-wide dependencies, and instance-specific dependencies.
- *Discretionary dependencies* where the JEE standard leaves it to the discretion of the vendor[3].

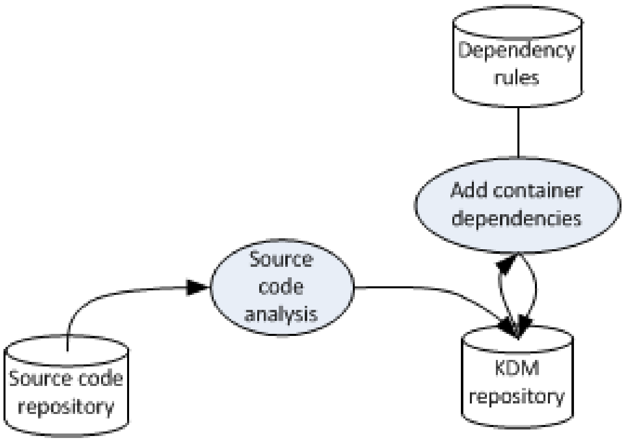


Fig. 3. Adding container dependencies to KDM repository

Some of the dependencies we codified do not concern program flow, per se, but reflect likely run-time behavior that is useful for architectural purposes. The problem of container services dependencies still requires deeper analysis and is still a work in progress. Therefore, our tool implementation (see Section IV) will not cover this problem.

C. Multi-language in one file to realize dependencies

1) Problem: Some JEE technologies allows to mix code from the other JEE technologies in their implementing source code files to realize dependencies. For example, in a Servlet, one might use JSP and JSF code (even HTML) as string literals given as parameters for the response object (Listing 1), while JSPs enable to embed Java code using the *scriptlet* tags. The content of a *scriptlet* may forward requests to another server page (see Listing 2). Therefore, the JEE analyzer should:

1) be able to detect which files having multi-language code;
2) distinguish between the different language codes in a given file;
3) develop a specific processor for each piece of code, depending on its JEE technology while parsing a given source code file; and
4) be able to detect dependencies between the different parts written using different languages (e.g., in JSP, defining a variable using Java code and use it in JSP tag).

Listing 1. Inoking a relative-URL

```
@WebServlet("/relative-URL")
public class MyServlet extends HttpServlet {
public void service(HttpServletRequest request,
    HttpServletResponse response) {
PrintWriter out = response.getWriter();
out.println("</TABLE> <FORM ACTION=\"relative-URL\">
    <BIG><CENTER> <INPUT TYPE=\"SUBMIT\" VALUE=\"Proceed
    to Checkout\"> </CENTER></BIG></FORM>"); } }
```

[1] Our examples use the EJB 1.1. and EJB 2.0 standards, unless otherwise stated

[2] The Java class that implements the `EntityBean` interface, and the methods of the remote interface

[3] For example, whether the `ejbLoad` method is called at the beginning of *every* business method–transactional or not–or "only as needed", etc.

Listing 2. JSP embeds Java code to make dependencies

```
<jsp:scriptlet> RequestDispatcher dispatcher =
getServletContext().getRequestDispatcher("relative-URL");
     dispatcher.forward(request, response); </jsp:scriptlet>
```

2) Solution: We use a common representation model (KDM model) that abstracts away language and paradigm differences. We develop an analyzer that is able to detect multi-language code usage in a given source file. It identifies pieces of code related to different JEE technologies. Then, it parses each piece of code based on its technology to identify program elements and dependencies, and represent them in the central KDM model. We also consider detecting dependencies between program elements belonging to different technologies in order to keep the dependency call graph connected.

D. Dependencies embedded in string literals

1) Problem: String literals are frequently used in JEE applications for several purposes such as codifying dependencies and encapsulating parameters.

a) Codifying dependencies: In Servlets, JSPs and JSFs, string literals are used to refer to relative-URLs that are related to other server pages. Listing 1 shows an example of a Servlet that use string literals that embed a dependency for a relative-URL. String literals are also used to refer to the usage (reference) of *JavaBeans* and/or *Managed Beans* components, and to determine which members (attributes or methods) are accessed in the Beans. Listing 3 shows an example of JSF code that use string literals to refer to the *myBean* Managed Beans component. In this example string literals are used in two cases: the value entered by the end-user will be stored in the *myAttribute* attribute of *myBeans* and the value returned from *myMethod()* determines what is the relative-URL, and thus the request will be forwarded to the server page related to this relative-URL. As we can see, *myMethod()* has a parameter (*Parm*) that is also based on string literals. Moreover, if we consider the second *h:commandButton*, the value *page1?faces-redirect=true* of the *action* attribute is even more complex and needs special analysis to identify which part refers to the name of a relative-URL (page1) and which part denotes to parameter(s) (faces-redirect=true).

In enterprise Java Beans, clients identify interfaces related to enterprise beans using JNDI. This is based on string literals that are given as a parameter to the *lookup()* method of the *Initial Context* instance (e.g., *initialContext.lookup("java:comp/env/ejb/cart");*).

Thus, we need to parse these string literals to mine meaningful relative-URLs and bean's interfaces to account for the dependencies to the corresponding server pages and beans, respectively.

Listing 3. Example of JSF code that use string literals to codify dependencies

```
<h:form>
<h:inputText id="name" value="#{myBean.myAttribute}"/>
<h:commandButton id="submit"
     action="#{myBean.myMethod('Parm')}" />
<h:commandButton action="page1?faces-redirect=true"
     value="Page1" />
</h:form>
```

b) Encapsulating parameters: String literals are utilized to encapsulate parameters in requests based on an *id:value* pattern. In Listing 4, the server page firstly reads three parameters from the sender through the *request* object, does its thing, and then attaches another parameter to the *request*. This works as long as both the sender and the receiver agree on parameter ids (i.e. *gradeX* and *average* in our example). Thus, we need to verify the compatibility of the string literals used by the sender and the receiver, which requires us to correlate the source code of both the sender and the receiver.

2) Solution: We studied Oracle's JEE specification to identify where string literals could be used to codify dependencies. For JSPs and JSFs, we made a list of tags and their attributes that are related to program dependencies to server pages and Bean components. For example, that list includes the *action* attribute of the *form* tag, since its value corresponds to a relative-URL corresponding to a server page. For Servlets, we develop a lexical analyzer that is able to process the string literals used inside the Java implementation of Servlets, and identifies what relative-URLs are invoked in these string literals. This lexical analyzer is built based on XML concepts (i.e. tags and attributes). In our example in Listing 1, it returns *"relative-URL"*.

Listing 4. Example of encapsulating paramerts using string literals

```
<!-- read parameters as a receiver -->
<% int grade1 = request.getParameter("grade1");
int grade2 = request.getParameter("grade2");
int grade3 = request.getParameter("grade3"); %>
<!-- send a parameter as a sender -->
<% double avg= (double) (grade1 + grade2 +grade3)/3.0;
request.setAttribute("average", avg); %>
```

Listing 5. Differnt mechanisms for invoking relative-URLs in JSPs and JSFs

```
// In JSPs:
<jsp:include page="relative-URL" flush="true" />
<%@ include file="relative-URL" %>
<jsp:directive.include file="relative-URL" />
<jsp:forward page="relative-URL"> </jsp:forward>
<c:redirect url="relative-URL">  </c:redirect>
<c:url value="relative-URL" var="completeURL">  </c:url>
<%@ page errorPage="relative-URL" %>
<jsp:directive.page errorPage="relative-URL"/>
// In JSFs:
<h:commandButton id="submit" value="Submit"
     action="/myPage.jsp"/>
<h:commandLink value="LINk" action="/myPage.jsp"/>
<a href="page.xhtml">Message</a>
```

Listing 6. JSP page invokes a relative-URL using a HTML form

```
<form action="relative-URL" method="get">
Name: <input type="text" name="name"><br>
<input type="submit" value="Submit">
</form>
```

E. Different mechanisms to codify dependencies

1) Problem: Different JEE technologies use different mechanisms to codify dependencies. Although Servlets, JSPs and JSFs represent different implementation styles to achieve similar results, they use different mechanisms to invoke each other through their relative-URLs. Servlets invoke relative-URLs based on API calls of the standard-interface *javax.Servlet.RequestDispatcher* such that the relative-URL is given as a string parameter to the *getRequestDispatcher()* method. JSPs and JSFs invoke relative-URLs in five and three different ways, respectively (see Listing 5). Furthermore, these tags use different attributes to parametrize relative-URLs. For instances, <*jsp:include*> uses the *page* attribute, while <*%@ page%*> and <*jsp:directive.page*> use *errorPage* and <*jsp:directive.include*> uses the *file* attribute. Thus, it is required to identify all of these mechanisms to be able to capture and to codify all dependencies during the analysis of JEE applications.

Moreover, JEE technologies have many versions. Newer versions preserve backward compatibility and allow developers to use older tags to codify dependencies. For instance, The scriptlet tag can be of two equal forms; <*% ... %*> and <*jsp:scriptlet*> ... <*/jsp:scriptlet*>. Moreover, Servlets, JSPs and JSFs still use other old technologies to codify dependencies such as the traditional HTML form (see Listing 6). Thus, several codification mechanisms coexist and can be used to perform a single task. This requires to analyze all versions of each JEE technology to identify how each version codifies dependencies.

2) Solution: By analyzing the JEE specification, we identified all the mechanisms that could be used for codifying dependencies. Additionally, we consider analyzing all versions of Servlets, JSPs and JSFs, in addition to HTML during parsing server pages.

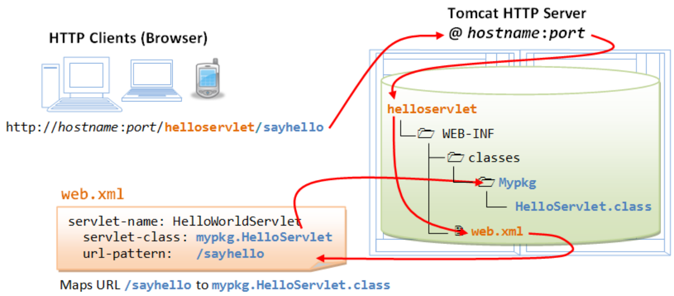
For JSPs and JSFs tags, we build an abstract simple table that abstracts the syntax variants for defining and refining these tags (respectively their attributes). For Servlet API calls, we parse the string parameter to identify relative-URLs. We also consider the mix of many technologies together (e.g., when a JSP uses normal Java code).

F. Need to look into different places to interpret dependencies

1) Problem: In order to identify dependencies in JEE applications, it does not suffice to analyze the source code files but one must also consider configuration files. As an example, Fig. 4 illustrates the several files we need to consider in order to map a relative-URL to the corresponding Servlet. To make things worse, this set of files to be analyzed depends on the specific JEE *revision* we are considering in the analysis. For example, we first need to visit a server page file[4] to identify a group of relative-URLs that have been invoked. These relative-URLs are mapped to server pages either using the *web.xml* file and/or *@WebServlet* annotation[5]. Consequently, it is essential to visit (and parse) *web.xml* (located in the *WEB-INF/* directory of a JEE application) and *@WebServlet* annotations (within the Servlet's source code).

This is also the case with EJBs, for which we need to visit (and parse) both the *ejb.xml* and the Java implementation of EJBs in order to identify the type of a given bean (session or entity), what is the home interface, the type of persistence (bean-managed or container-managed) and so on. For JSFs and Managed Beans, we have different configuration files to be analyzed (the *faces-config.xml*).

Therefore, we need to identify the set of files to be analyzed, their structure and their location, in order to detect each type of dependencies.



```
servlet-name: HelloWorldServlet
  servlet-class: mypkg.HelloServlet
  url-pattern:   /sayhello
```

Fig. 4. Visiting different places to map a relative-URL to the corresponding Servlet

2) Solution: For each dependency, we identified the set of files that need to be parsed to extract this dependency. For example, we need to parse the implementation of Servlets, JSPs and JSFs as well as the *web.xml* configuration file to detect dependencies in server pages. Then, we identified what information needs to be considered form each file. In *web.xml*, we consider five elements to identify mappings between server pages and relative-URLs, namely: <*Servlet*>, <*Servlet-mapping*>, <*Servlet-class*>, <*jsp-file*> and <*url-pattern*>. Then, we developed a parser to extract the needed information from each file. Finally. we built a look-up table that maps each relative-URL to its corresponding server page(s). The same process is applied to the other kind of dependencies like JavaBeans and Managed Beans mapping.

In Table I, we present the list of configuration files that need to be parsed to analyze dependencies in different JEE technologies. We also provide examples of some XML tags that need to be analyzed.

IV. DeJEE Tool

We developed a tool called DeJEE (**De**pendencies in **JEE**) that implement our solution. DeJEE is applied on two case studies to demonstrate the applicability of our approach to identify dependencies in JEE applications.

A. DeJEE implementation

We developed DeJEE as an Eclipse plug-in that extends MoDisco[6] tool. DeJEE supports the representation of multi-language code of JEE technologies in one common KDM model. The resulting KDM model is able to include program elements and their related dependencies existing across and within Java code, Servlets, JSPs, JSFs, JavaBeans and Managed Beans technologies. Based on this common KDM model, we extract a dependency call graph.

[4] Servlets are located in *Java Resources*, JSPs and JSFs are located in *WebContent* resources

[5] Starting from Servlet 3.0 specification

[6] Available at *https://eclipse.org/MoDisco/*

TABLE I
LIST OF CONFIGURATION FILES NEEDED TO ANALYZE DIFFERENT JEE TECHNOLOGIES

JEE Technology	**Coniguration file(s)**	**Example of some important attributes**
JSP	*web.xml*	<Servlet>, <Servlet-name>, <jsp-file>, <Servlet-mapping>, <error-page>, <url-pattern>, <init-param>, <param-name>, <param-value>, <welcome-file-list>
JSF	*faces-config.xml* and *web.xml*	<navigation-rule>, <from-view-id>, <navigation-case>, <from-action>, <from-outcome>, <to-view-id>
Servlets	annotation and *web.xml*	<Servlet>, <Servlet-name>, <Servlet-class>, <Servlet-mapping>, <error-page>, <url-pattern>, <init-param>, <param-name>, <param-value>,<welcome-file-list>
EJBs	annotation, *web.xml*, *ejb.xml*, *ejb-jar.xml*, *orion-ejb-jar.xml*, *toplink-ejb-jar.xml*, *ejb3-toplink-sessions.xml*, *persistence.xml* ...	<enterprise-beans>, <session>, <ejb-name>, <env-entry>, <env-entry-name>, <env-entry-type>, <env-entry-value>,<ejb-ref>, <ejb-ref-name>, <ejb-ref-type>, <home>, <remote>
Managed Beans	annotation and *faces-config.xml*	<managed-bean>, <managed-bean-name>, <managed-bean-class>, <managed-bean-scope>, <managed-property>, <property-name>, <value>
tag libs	*web.xml*	<taglib>, <taglib-uri>, <taglib-location>

B. Case studies

1) Subject: As case studies, we selected two JEE projects publicly available, namely Java PetStore [20] and JSP Blog[21]. Java PetStore is the official Sun Microsystems' showcase example to demonstrate how to develop flexible, scalable, cross-platform JEE applications. We selected Java PetStore due to: 1) the availability of both its source code and documentation, and 2) its coverage of several JEE technologies including JSP pages, Java Servlets, Enterprise JavaBeans (EJB), and Java Message Service (JMS) technologies. The implementation of Java Pet Store consists of 88 JSP pages, 233 normal Java classes and 8 HTML pages.

JSP Blog is a web-blogger that is developed based on JSP technology using the Tomcat server and MySQL as a persistence backend. The implementation of JSP Blog is composed of 10 JSP pages and 1 HTML page. We selected this application as these JSP pages make heavy use of multi-language code (mixing normal Java code in JSP pages), and do not rely on normal Java classes, which will let us expose the limitations of existing tools. Thus, it is considered as a good case study to evaluate how DeJEE works with these multi-language files.

2) Method: We aim to show two aspects: 1) the ability of DeJEE tool to codify problems underlying dependencies an analysis in JEE applications, and 2) How much DeJEE extends MoDisco to identify more program elements and dependencies. To do so, we have executed both MoDisco and DeJEE and noted the differences between each tool's results. At each run, we have parametrized DeJEE to focus independently on each of the aforementioned problems. Then, we verified the relevancy of the dependencies that were only detected by DeJEE.

3) Result:

a) The ability of DeJEE to codify dependencies: For Java PetStore, we identify 40 JSP pages having multi-language code composed of HTML tags, JSP tags and normal Java code[7]. These represent 45.4% of the total number of JSP pages. In JSP Blog, we identify 6 JSP pages mixing normal Java code inside JSP tags that makes 54.5% (6/11) of its implementing files. Dependencies in JSP Blog can be mainly classified into two categories. The first refers to dependencies implemented in Java code based on normal Java dependencies (method invocation, object instance, etc.). The second category refers to dependencies implemented in JSP tags. We identify three tags in this category. These are *<a href="relative-URL">*, *<%@ include file="relative-URL" ... %>* and *<form action = "relative-URL"... >*, which are used 15, 1, and 4 times, respectively. As a result of parsing these dependencies, we identified 41 program dependencies that are realized in the Java code using $scriptlet$, and 20 program dependencies are realized using other JSP tags.

In server pages of Java PetStore and JSP Blog, DeJEE respectively identifies 1541 and 204 string literals in total such that each server page contains 17.51 and 18.54 string literals in average. Dependencies are embedded in 46.7% (720/1541) and 31.8% (65/204) of these string literals for Java PetStore and JSP Blog respectively. The results show that 20.4% and 12.7% (26/204) of string literals embed dependencies to other source code files, while 26.2% and 19.1% (39/204) are used for passing parameters of Java PetStore and JSP Blog respectively.

As results of codifying these problems, DeJEE discovers 329 and 11 instances of *KDMEntity* for representing program elements related to Java code, JSP and HTML files, as well as 315 and 61 instances of *KDMRelationship* for representing program dependencies in server pages, respectively for JavaPetStore and JSP Blog.

b) How much DeJEE extends MoDisco: DeJEE provides a dependency call graph such that *KDMEntity* instances represent the set of nodes and *KDMRelationship* instances represent the set of links. For JSP Blog case study, Fig. 5 shows the resulting dependency call graph from DeJEE tool, from which we exclude external dependencies (e.g., those related to Java APIs). DeJEE goes beyond our expectation and detects dependencies toward program elements that are *missing* from the project code, that are undetected at compile time, but will lead to run-time errors if execution takes it down that path (references to missing JSP pages *delnews.jsp*, *edituser.jsp* and *deluser.jsp*). As MoDisco's KDM model does not support server pages, the resulting KDM model related to JSP Blog has

[7] These results do not include yet dependencies related to container services (see Section III-B) as these are still a work in progress.

0 *KDMEntity* and 0 *KDMRelationship* instances. Therefore, we do not have a dependency call graph.

For Java PetStore, MoDisco produces a KDM model representing only the Java implementation side. Thus, its call graph consists of 233 nodes corresponding to Java classes. On the other hand, DeJEE includes server pages and Java classes in its KDM model. Therefore, DeJEE's call graph contains 329 nodes. As a result, our approach increases MoDisco's recall by detecting 41% ((329-233)/233) and 100% ((100-0)/100) more accurate program elements as well as their related dependencies respectively for Java PetStore and JSP Blog.

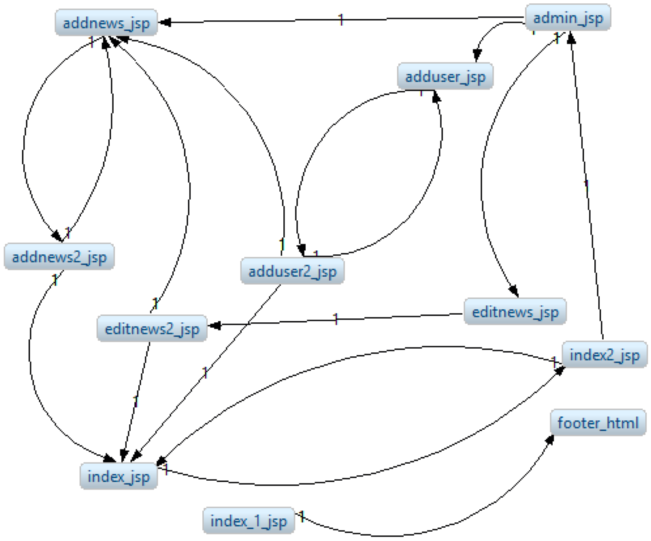


Fig. 5. Dependency call graph of JSP Blog resulting from DeJEE tool

V. Discussion

In this section, we discuss some important issues to be considered regarding our approach. These are: 1) the correctness and the coverage of our approach, 2) the prevalence of the various dependencies, 3) Difficulties that we face during parsing configuration files, 4) Java reflexion with string literals, 5) user-defined tag libraries and 6) dependencies can be common in other web technologies.

A. Correctness and coverage of our approach

We studied Oracle's JEE specification to identify how dependencies can be codified to take into account different versions of the JEE technologies. For example, for JSPs, we identified the different ways that a JSP page can invoke another JSP page. In terms of coverage, that represent 100% of pure specific inter-JSP dependencies.

We manually evaluated the resulting call graph. For each link, we examined the implementation of the corresponding two nodes to verify that the produced dependency is relevant. We found that all links refer to real dependencies with 100% precision. Furthermore, we tested our tool using many examples to validate its ability to capture such dependencies (for example, <*form action="relative-URL"*> meaning that we have 100% of a static reference to the service page related to this relative-URL).

Unfortunately, we cannot measure the recall of our approach due to the lack of full dependency call graph corresponding to the selected case studies where we can measure if we really codified all of the dependencies (i.e., we cannot know if we have false positives).

B. Prevalence of the various dependencies

The dependencies that we codified are frequently used in JEE applications. The prevalence of the various kind of dependencies depend on application type, use of frameworks, and design quality. For example, Servlets, JSPs and JSFs use string literals a lot for several reasons. For JSPs and JSFs, tags are described based on attributes whose values are based on string literals. Servlets use string literals to embed HTML and JSP tags in the *request* object that can be handled on the client-side.In our case studies, the results show that 100% of server-page- dependencies are codified based on string literals that require to be lexically compared. EJBs are referenced in the clients' codes based on string literals. Relative-URLs represent the common protocol used for communication between different server pages regardless of the implementation programming language (similar to method invocations in object-oriented languages). In our case studies, relative-URLs represent 100% of the interdependencies between server pages. For JavaBeans components, JSP Blog does not have any JavaBeans, while server pages of Java PetStore reference to JavaBeans components 26 times. The case of multilanguage in one file is realized in 49.9% ((45.4%+54.5%)/2) of server pages of our case studies.

C. Difficulties in parsing configuration files

For JSP Blog, the *web.xml* file is used to configure the welcome page and to define references to 4 tag libraries (*.tld* configuration files) and one jar file. There are no JSPs to relative-URLs mapping in *web.xml*. The developers directly use the names of JSP pages as references (e.g., *addnews.jsp*).

For Java PetStore, identifying dependencies in configuration files is a more complicated task compared to the case of JSP Blog as the developers did not use direct mapping of server pages. The design of Java PetStore is based on several sub-projects. We identified 3 *web.xml* files that correspond to different sub-projects. Each one uses different mapping style. In *petstoreadmin* subproject, *web.xml* maps all relative-URLs to one Java Servlet (*AdminRequestProcessor*). *AdminRequestProcessor* maps the requests programmatically into four different JSP pages based on a conditional statement evaluated against parameter values embedded in the HTTP request. The mapping is directly to the name of JSP pages. This means that we do not need further processing in this case as their dependencies can be detected by our string literal parser. In the second sub-project, *web.xml* file maps all requests to the *MainServlet.java* Servlet class that does the mapping task. The mapping of relative-URLs to server page is defined in a *Map* instance and, to identify these mappings, we need to resort to data-flow analysis. In the third sub-project, *web.xml* file also maps all requests to another *MainServlet.java* Servlet class. Then, this *MainServlet.java* reads another *xml* files (e.g., *screendefinitions.xml*, *screendefinitions.xml* and *requestmappings.xml*) for doing the mapping using *requestMappingsURL = getServletContext().getResource("/WEB-INF/xml/requestmappings.xml").toString();*. These files do not follow a standard format. Thus, we have to study them for identifying patterns corresponding to their structures. These patterns are then given to our analysis tool to be able to detect relative-URL-to-server-page-mappings.

D. Java reflexion with string literals

Some dependencies might be expressed as string literals using the Java Reflexion mechanism. These literals will then be replaced, at run-time, by the actual class/method towards which the request should be redirected. In order to identify such dependencies, a data-flow analysis is then required to evaluate those strings passed as parameters (for example, the target of a relative-URL). However, if the string is constructed with user input (for example, the concatenation of a variable and string input by the user), then data-flow analysis will not be enough. We have to evaluate how much/frequent a problem that is. This will affect the recall of our approach.

E. User-defined tag libraries

In some cases, developers develop their own tag libraries in terms of template, and then they use these template in their source code. Some of these tags could codify dependencies. As a real example from PetStore JEE applicatiopn[8], the developers override the HTML form in their tag library to be called *PrevFormTag* (can be used as *<j2ee:prevForm action="relative-URL"> ... </j2ee:prevForm>*). Such override dependencies need farther processing to be caught.

F. Dependencies can be common in other web technologies

Among dependencies that we addressed in this paper, we can generalize some of them for other programming languages and technologies. For example, relative-URLs are used as a communication protocol in all type of server pages regardless of the used programming language. Therefore, our approach of capturing dependencies related to relative-URLs can be used for other web applications (JavaScript, PHP ...). The same case is applied for dependencies embedded in string literals. The lexical analyzer can be used to parse other technologies and programming languages by providing it the set of patterns that the string literals are used and composed. Moreover, the resulting KDM model is language-independent one which allows to easily extend our approach for including other programming languages such as JavaScript and PHP.

VI. Related Work

Several approaches were proposed to target the analysis of web applications with different purposes. Most of these approaches focus on a subset of technologies. The approach in [3] analyzes EJBs and JEE access policies to find inconsistencies. It relies on static analysis and it uses graphs to represent the extracted information. However, this approach is limited to retrieving objects, the fields they are manipulating and the methods that load or store these fields. With the purpose of checking a number of desirable properties of web applications, the approach in [4] focused on the analysis of Servlets. To analyze the behavior of an application, grammars are built to approximate the output of Servlets and they are translated into XML graphs. Cloutier et al. [5] proposed WAVI (WebAppViewer), a reverse engineering tool for retrieving and documenting the structure of a web application. The tool uses static analysis to retrieve dependencies and a number of heuristics to specifically resolve function calls in JavaScript. FireDetective [6] is a tool that supports the understanding of Ajax-based web applications. The tool uses dynamic analysis to retrieve execution traces on both client and server sides and it visualizes them.

On the multi-language analysis front, we cite the work of Mayrhauser and Vans who studied various program comprehension models and sketched an integrated model that can be used to explain developers' comprehension of components written in any programming language [10]. Müller and his group developed Rigi, an environment to reverse engineer, explore, visualize, and re-document components in C, C++, or COBOL, but in isolation from one another [11]. Moise and Wong were among the first researchers to extract, represent, and study cross-language dependencies [12] (see also [13] [14]). They used the API provided by each language to identify cross-language calls, e.g., calls to Java Native Interface API in C and Java. Kraft et al. developed a technique to identify cross-language clones using the Microsoft CodeDOM library for .NET languages and a hybrid token/tree-based algorithm for clone detection [15]. They reported clones whose siblings exist in components written in both C# and Visual Basic.NET. German and Hassan described five possible kinds of dependencies between (heterogeneous) components (linking, forking, subclassing, inter-process communication, and plugin) and identified these in 124 open-source software systems [16]. Using the identified component dependencies and their licenses, they proposed 12 patterns of license integration. Mayer and Schroeder proposed a technique based on the MOF Query, View, Transformation Relations specification (QVT/R) to identify dependencies among heterogeneous components, warn of potential missing dependencies, and propagate renamings among heterogeneous components [17]. This approach targets the Java ecosystem and it focuses on discovering artifacts and binding them. However, the bindings are mostly based on artifacts names. Ayers et al. [18] proposed TraceBack to diagnose bugs in multi-language software by collecting data through run-time instrumentation of control-flow blocks. The

[8] https://docs.oracle.com/cd/E17802_01/blueprints/blueprints/code/jps11/src/com/sun/j2ee/blueprints/petstore/taglib/list/PrevFormTag.java.html

data is collected by statically rewriting the binaries and/or instrumenting the intermediate languages to generate a unified trace of the components' execution. However, dynamic analysis is expensive to setup and does not guarantee full relations (missing of usage scenarios). Yazdanshenas and Moonen [19] built homogeneous KDM models of heterogeneous systems, with components in C, C++, and Java and configuration files in XML. They used these models to obtain system dependency graphs and sliced these graphs to show if a given input is used to produce the expected output, typically in sensor/actuator systems and other such component-based systems. However, their approach considered only programming languages following object-oriented meta-model (as does MoDisco [8]), disregarding JSPs, JSFs and other non object-oriented languages.

Though all these approaches contribute largely to support the understanding and analysis of web applications, none of these approaches tackle all the technologies found in JEE application tiers. To the best of our knowledge, none of the existing approaches tackled the problem of codifying container service dependencies.

VII. Conclusion and Future Work

A. Conclusion

Program dependency artifacts such as call graph supports a number of key software engineering tasks including software mining, program understanding, debugging, feature location, maintenance and evolution. Static code analysis can be quite challenging when dealing with single-language monolithic applications–depending on the language (e.g. typed or not) and the task at hand (e.g. understanding versus data flow analysis). Things get far more complicated when dealing with multi-language systems such as JEE applications. Further, JEE applications rely on JEE container services that, in the process of *hiding/abstracting* the complexities of the run-time infrastructure, end up hiding some useful dependencies. These dependencies need to be identified, in order to improve the accuracy of the other software engineering tasks. In this paper, we highlighted the major difficulties in analyzing dependencies across key JEE technologies (Servlet, JSP, JSF, and EJBs), and presented our strategies for addressing them.

To implement our solution, we developed a tool called DeJEE (**De**pendencies in **J**EE) as an Eclipse plug-in. We reused the KDM APIs offered by MoDisco. We applied DeJEE on two open-source JEE applications. The results show that DeJEE is able to detect different type of dependencies that MoDicso does not. Also, DeJEE detects dependencies toward program elements that do not exist in the application implementation.

B. Future Work

We are extending our work in four research directions. These are:

1) Generalization from JEE applications to multi-language applications: The problems encountered with JEE applications are common to modern legacies which *also* tend to be multi-language, and *also* rely on various frameworks, middlewares, and container services, which relieve application developers from the complexities of the run-time environment, but also obfuscate dependencies that would otherwise help explain functional dependencies and run-time behavior. The *principles* underlying our solutions should apply to other recent legacies.

2) The application of the resulting dependency call graph: Thanks to the thoroughness and precision of the dependencies that we are uncovering, we are exploring *other* uses for the program dependency graphs that we are building, including: 1) help with program understanding tasks, 2) change impact analysis, and 3) performance engineering–thanks, in part, to the codification of container services.

3) Extending DeJEE tool to include all dependencies in JEE applications: The current implementation of DeJEE tool does not support the codification of dependencies related to container services. We are currently working on that. Moreover, JEE allow developers to use other programming languages inside the implementation of JEE applications such as JavaScript code. DeJEE does not cover dependencies existing in these other languages and we plan to consider them in our future work.

4) Experimenting with large number of case studies: To generalize the results of our approach for other JEE applications, we need to evaluate it with a large number of JEE applications. Different JEE applications can be implemented using different sets of JEE technologies following different design patterns by developers with various knowledge and experience levels. Therefore, we want to collect a large set of JEE applications as case studies developed for various purposes by different developers. We aim to study two main aspects. First, we will study the impact of design patterns on the correctness and coverage of our approach. To do so, we will classify the case studies based on their design patterns. Then, we will identify what design patterns are covered by our approach, and will address the other design patterns. Second, we want to generalize that our approach is able to identify dependencies in JEE applications, regardless the design patterns or the technologies used for their development.

5) The evaluation of our approach by human experts: We demonstrated the applicability of our approach through the most common ways that developers used for developing JEE applications (Java PetStore that represents the official demonstration offered by Oracle and several representative examples taken form the JEE specification offered by Oracle). However, the resulting dependency call graph should be validated using external human experts. Therefore, we plan to validate our results using the help of human experts. We will try to contact the developers of case studies to evaluate the resulting call graphs. Otherwise, we will select a number of master and PhD students to study the case studies in order to be able to evaluate the resulting call graphs.

References

[1] Bill Dudney, Stephen Asbury, Jospeh K. Krozak, and Kevin Wittkopf. *J2EE Antipatterns*. Wiley, 2003.

[2] Ricardo Pérez-Castillo, Ignacio Garcia-Rodriguez De Guzman, and Mario Piattini, "Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems," *Computer Standards & Interfaces*, 33(6):519–532, 2011.

[3] Gleb Naumovich and Paolina Centonze, "Static analysis of role-based access control in j2ee applications," *ACM SIGSOFT Software Engineering Notes*, 29(5):1–10, 2004.

[4] Christian Kirkegaard and Anders Møller, "Static analysis for java Servlets and jsp," In *International Static Analysis Symposium*, pages 336–352. Springer, 2006.

[5] Jonathan Cloutier, Sỳgla Kpodjedo, and Ghizlane El Boussaidi, "Wavi: A reverse engineering tool for web applications," In *IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–3. IEEE, 2016.

[6] Andy Zaidman, Nick Matthijssen, Margaret-Anne Storey, and Arie Van Deursen, "Understanding ajax applications by connecting client and server-side execution traces," *Empirical Software Engineering*, 18(2):181–218, 2013.

[7] Tichelaar Sander, Stéphane Ducasse and Serge Demeyer, "FAMIX: Exchange experiences with CDIF and XMI," In *Proceedings of the ICSE 2000 Workshop on Standard Exchange Format (WoSEF 2000)*, 2000.

[8] Bruneliere, Hugo, Jordi Cabot, Grégoire Dupé and Frédéric Madiot, "Modisco: A model driven reverse engineering framework," *Information and Software Technology*, 56 (8):1012–1032, 2014.

[9] KDM Analytics, ww.kdmanalytics.com, visited in 2016.

[10] A. von Mayrhauser. "Program comprehension during software maintenance and evolution," In *IEEE Computer*, 28(8):44–55, 1995.

[11] H. M. Kienle and H. A. Mllër. "Rigi–an environment for software reverse engineering, exploration, visualization, and redocumentation". In *Science of Computer Programming*, 75(4):247–263, 2012.

[12] D. L. Moise and K. Wong. "Extracting and representing cross-language dependencies in diverse software systems," In *Proceedings of the 12th Working Conference on Reverse Engineering*. IEEE Computer Society Press, November 2005.

[13] L. Deruelle, N. Melab, M. Bouneffa, and H. Basson. "Analysis and manipulation of distributed multi-language software code," In *Proceedings of the 1st International Workshop on Source Code Analysis and Manipulation*, pages 45–56. IEEE Computer Society Press, November 2001.

[14] P. K. Linos, Z. hong Chen, S. Berrier, and B. O'Rourke. "A tool for understanding multi-language program dependencies," In *Proceedings of the 11th International Workshop on Program Comprehension*, pages 64–72. IEEE Computer Society Press, May 2003.

[15] N. A. Kraft, B. W. Bonds, and R. K. Smith. "Cross-language clone detection," In *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering.*, pages 54–59. Knowledge Systems Institute, July 2008.

[16] D. M. German and A. E. Hassan. "License integration patterns: Addressing license mismatches in component-based development," In *Proceedings of the 31st International Conference on Software Engineering*, pages 188–198. ACM Press, May 2009.

[17] Philip Mayer and Andreas Schroeder, "Automated multi-language artifact binding and rename refactoring between java and dsls used by java frameworks," In *European Conference on Object-Oriented Programming*, pages 437–462. Springer, 2014.

[18] A. Ayers, R. Schooler, C. Metcalf, A. Agarwal, J. Rhee, and E. Witchel. "Traceback: first fault diagnosis by reconstruction of distributed control flow," In *Proceedings of the 26th conference on Programming language design and implementation*. pages 201–212. ACM Press, June 2005.

[19] A. R. Yazdanshenas and L. Moonen. "Crossing the boundaries while analyzing heterogeneous component-based software systems," In *Proceedings of the 27th International Conference on Software Maintenance*, pages 193–202. IEEE CS Press, September 2011.

[20] Sun MicroSystems, Java Pet Store, http://www.oracle.com/technetwork/java/petstore1-3-1-02-139690.html, *last access: Feb. 8th 2017.*

[21] JSP Blog, http://jspblog.sourceforge.net, *last access: Feb. 8th 2017.*

[22] Object Management Group, "Architecture-Driven Modernization: Knowledge Discovery Meta-Model (KDM), v. 1.3".