External dependencies in software development

Aless Hosry, Nicolas Anquetil

External dependencies in software development

Aless Hosry[1] and Nicolas Anquetil[1]

Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
{aless.hosry, nicolas.anquetil}@inria.fr

Abstract. Successful software requires constant modifications. To guarantee the continuous proper functioning of the applications, developers need to understand them well, particularly by having an accurate map of the dependencies between the parts they are modifying. However, some of these dependencies are not easily identified. For example, in an Android application, there are dependencies between the Java source code and XML parts, some of which are materialized by a generated "R" Java class. We call such dependencies external because they are introduced by some agent external to the source code. On top of the categorization of dependencies defined in the literature, we define restrictions on the *External Dependencies* that allow us to verify the source code and identify possible flaws. We created a common approach relying on reusable patterns to search for containers and entities that are part of such dependencies and implemented it in a prototype that we validate on two different projects from GitHub and developed using different frameworks.

Keywords— External dependency, Cross language, Multi-language, Multi-tier

1 Introduction

Repeated modifications are necessary for successful software. This cycle of modifications is known as software evolution. When performed by developers or tools (e.g., refactoring tools) without enough knowledge of the applications, maintenance will lead to decreased software quality. The knowledge required involves identifying all incoming and outgoing dependencies of a software artifact to be modified. We define dependency as the need for one artifact to rely on another artifact in order to fully operate as expected. Developers are more likely to miss *External Dependencies* than explicit dependencies and introduce bugs into software [19].

Yu and Rajlich [20] call hidden dependencies, data dependencies between two modules that are not readily apparent in the source code. As such, these dependencies are considered design faults. We are more interested in dependencies introduced by external tools or agents like Android. GWT[1], J2EE[2], ODBC[3], etc. Contrary to Yu et al. definition, such dependencies are inevitable and therefore not design faults. We call them *External Dependencies*: a dependency between two artifacts that is created through an external agent. Often the two dependent artifacts will be in two different

[1] Google Web Toolkit gwtproject.org
[2] Java 2 Platform, Enterprise Edition oracle.com/java/technologies/appmodel.html
[3] Open Database Connectivity wikipedia.org/wiki/Open_Database_Connectivity

source files, but "external" refers to the fact that the dependency is introduced by an external agent, not to the fact that the artifacts are external. For example, a GUI framework like JavaFX will offer widgets like a Button and callbacks on these widgets (`setOnAction(EventHandler)`). The dependency between a button and its handler is not clear in the source code if one does not know how JavaFX works. It is handled *externally*, even if both artifacts are defined in the same file.

Examples of such "external agents" are frameworks working with a variety of programming languages suited for different objectives such as building user interfaces, handling logic, and querying databases. Other frameworks allow different projects to collaborate, for example in client/server (multi-tier) applications.

We found that there is no clear, unique definition in the literature of what a hidden dependency is. That's why we introduce the notion of *External Dependency*. Also, most of the available tools only consider specific types of *External Dependencies* in specific contexts. The majority focuses on identifying the dependencies, ignoring other perspectives like detecting errors due to missing dependencies or excessive dependencies.

As a result, we will go over the following points that outline our strategy in this paper to detect all possible types of *External Dependencies*, built on top of the earlier work of Hecht et al. [6]:

1. We list in this paper all the types of *External Dependencies* we found in the previous works [20] [6], in addition to the exceptions;
2. We list the commonalities we found that lead to a single approach for *External Dependency* detection;
3. We explain the restrictions for those commonalities following each type and how they lead to the detection of errors;
4. We validate our approach with a tool that works using common search patterns by experimenting it for a first implementation of two applications developed in two different frameworks.

The paper is structured as follows: In section 2 we list all the previous works in the literature. Section 3 lists different categories of *External Dependencies*, then lists the cardinalities found for each one. Our tool *Adonis* is presented in section 4, followed by two experiments and results 5. Finally, we end up with a conclusion 6.

2 Related work

In the literature, many solutions exist for analyzing different types of *External Dependencies*. We first present the different publications found before proposing a classification scheme for them.

2.1 Existing tools

Yu and Rajlich [20] are discussing "hidden dependencies", but they use a definition (classes having a data dependency that is not readily apparent in the source code) different from what we are discussing in this paper.

DeJEE developed by Shatnawi et al. [16], is able to generate automated analysis for J2EE solutions in order to identify a dependency call graph of a given J2EE application using the KDM metamodel; They use static analysis to build a KDM model of the software and match it with XML artifacts through rules with lexical matching.

BabelRef is a tool developed by Nguyen et al. [12]. It detects automatically the dependencies between generated client artifacts (HTML and JavaScript) and generated server artifacts (PHP). It relies on dynamic analysis with a single tree-based structure called the D-model using object matching.

EdgeMiner is a tool developed by Yinzhi et al. [2] to automatically detect callbacks for Android framework applications, whether they are created in Java source code. The analysis is performed statically on the source code and returns a list of callbacks that were identified using object matching, for example, when callbacks are introduced by an implemented Java interface. The authors also provided a different method of implementing callbacks using XML resources in Android applications, although their strategy does not address their detection.

Hecht et al. [6] developed another approach that detects dynamically the dependencies established in J2EE applications using codified rules. This is the only work we found that aims for a generic approach, considering different types of dependencies (callbacks, multi-language, multi-tiers). Still, it is limited to the J2EE framework.

Polychniatis et al. [15] proposed a static method for detecting cross-language links based on matching lexically common tokens between two possibly dependent modules. The detection algorithm is then followed by applying specific filters, such as filtering frequent tokens and omitting one-character tokens.

Grichi et al. [4] made a study on 10 Java Native Interface (JNI) open-source multilanguage systems to identify dependencies between Java and C++ using the Static Multilanguage Dependency Analyzer (S-MLDA) to detect static cross-language links and the Historical Multilanguage Dependency Analyzer (H-MLDA) based on software co-changes to identify links that could not be detected statically.

GenDeMoG, which is a tool developed by Pfeiffer and Wąsowski [13], allows specifying intercomponent dependency patterns statically for artifacts in heterogeneous systems. The tool relies on parsing languages, querying source code objects each time, and retrieving the possible dependencies between different artifacts.

Mayer and Schroeder [8] created a generic approach to understanding, analyzing, and refactoring cross-language code by directly specifying and exploiting statically semantic links in multi-language programs. Their tool, XLL, was developed using QVT-Relations (QVT-R), where a set of rules for cross-language links per language is defined as a relation inside a transformation block, after parsing the source code, and introduces for the first time the idea of cross-language link correctness.

Dsketch is a tool created by Cossette and Walker [3] and used by developers to specify patterns and match blocks of code in any programming language using lexical matching. After identifying the artifacts, Dsketch starts looking statically for possible links between the languages of these artifacts, following a set of steps predefined by the author.

Soto-Valero et al. [18] suggested a new automatic approach to identify third-party Java dependencies statically in Maven projects, remove unneeded classes, repackage used classes into a different dependency, and regenerate the XML configuration file to refer to the new dependencies. The objective of their approach is to create a minimum project binary that only contains code required for the project and eliminates "bloated dependencies". As such, they introduce the notion of dependency correctness (or incorrectness in the case of bloated dependencies).

Kempt et al. [7] introduced an approach that could be applicable to enhance the refactoring of cross-language links between Java and Groovy. Their approach is completely static and relies on searching over source code objects, as Groovy and Java can easily interact with each other. In order to accelerate the searching engine, the authors

propose filtering the classes on which, for example, the method call is executed, creating a hierarchy scope, and starting a second search following the method inside the limiting hierarchical scope.

2.2 Categorization of the tools

We classify the presented work in different categories (summarized in Table 1[4]):

Dependency type: Many of the papers are considering dependencies existing in *cross-language* applications, like between Java source code and an XML configuration file, *multi-tier* like between a client application and a server one, *callbacks* from external libraries, or *generated files* like the R class in Android or the files generated by J2EE, or HTML and Javascript generated by PHP;

Analysis: We saw that both *static* and *dynamic* analyses could be used by the tools;

Matching strategy: There are two different strategies used: *Object matching* works on a model of the application and looks for specific objects in the model, *lexical matching* works directly on the source text (Java, XML, ...);

Engine: Finally, some approaches are based on user defined *rules* to identify dependencies, while others *automatically* discover them.

Table 1: Existing approaches(dependency types)

Name	Dependency Type	Analysis	Matching strategy	Engine
Yu and Rajlich [20]	*Data dependencies*	*Static*	*Object*	*Rule*
EdgeMiner [2]	Callbacks	Static	Object	Automatic
Dsketch [3]	Cross-language	Static	Object	Rule
Grichi et al. [4]	Cross-language	Static	Lexical	Rule
Hecht et al. [6]	all[a]	Dynamic	Object	Rule
Kempf et al. [7]	Cross-language	Static	Object	Rule
XLL [8]	Cross-language	Static	Object	Rule
BabelRef [12]	M-tiers[b], Gen.[c]	Dynamic	Object	Rule
GenDeMoG [13]	Cross-language	Static	Object	Rule
Polychniatis et al. [15]	Cross-language	Static	Lexical	Automatic
DeJEE [16]	Cross-language	Static/Dyn.	Lexical	Automatic
Soto-Valero et al. [18]	Cross-language	Static	Object	Automatic

[a] Multi-tiers, generated files, callbacks, cross-language
[b] Multi-tiers
[c] Generated files

We can see that only one paper considers multiple dependency types (Hecht et al. [6]: cross-language, multi-tiers, callbacks, and generated files). Also, two papers, Pfeiffer and Wąsowski [13], and Soto-Valero et al. [18], consider dependency correctness

[4] For completion, we include Yu et al. in the table although, as already said, they consider a different type of dependency

by identifying when a dependency should not exist. Additionally, we found that some approaches are generic [3, 8, 13, 15], while others work only for specific languages and frameworks [2, 4, 6, 7, 12, 16, 18, 20].

3 Structuring the domain

Having reviewed the literature on dependencies introduced by agents external to the source code, we will now discuss more in-depth the different categories of these *External Dependencies* that were introduced in the preceding section. We also discuss dependencies correctness and how an *External Dependency* detection tool can help, not only in understanding the software, but also in pointing out possible flaws.

3.1 Categorization

Different kinds of *External Dependencies* exist in software where artifacts are written in different languages or tiers need to interact:

Cross-language dependencies are those between artifacts written in different languages. For example, in the GWT framework where XML can be used to define the UI and Java is used to handle the behavior. The dependency between Java and CXML appears in the Java code with annotations such as `@UiField` and `@UiHandler`. If any change affects the XML artifact, the developer must apply in parallel the same change on the dependent Java artifacts, otherwise the dependency is broken and the application might fail at runtime. This seems to be the kind of *External Dependencies* most studied (see Table 1) Such dependencies may be difficult for the developers to detect as they imply a good understanding of the framework used [10]. For example, Android also expresses the GUI in an XML file and the behavior in Java code, but the dependencies are not materialized in the same way. Examples are not limited to GUI, other Cross-language dependencies can be found when an SQL query (in a String) references the tables and columns of an external database.

Multi-tiers dependencies appear in applications with a distributed architecture. For example, they can access data on one or more database servers, the business logic runs on an application server, the presentation logic is deployed on a web server, and the user interface runs on a web browser [11]. Thus, in a call established between a client and server application using Java RMI, the client program must be aware of the structure of interfaces that extend `java.rmi.Remote` in order to invoke its methods and make a successful call to the server tier. Again, such calls between tiers depend on the communication framework used, yet changes to one tier could require in parallel updates to the second.

Callback dependencies are often used by libraries to allow the user to get back control from library's elements. For example, in the JavaFX graphical library, a Button widget can give back control to the application upon end-user interaction through the `setOnAction(EventHandler)` callback. Kempf et al. [7] mention it for the Android framework. Again, each framework has a different set of callbacks that must be known to adequately identify the dependencies.

File generation dependencies appear when a framework or tool generates additional files, predefined in configuration files and parsed on deployment time, able to generate additional components linked with the existing ones of the project [6].

Such dependencies are hidden until the project is deployed or executed, and during the analysis phase, a developer or an analysis tool may not be aware of their existence. These dependencies may be accompanied by other kinds, for example, Android generates a `R` class that allows to link the Java code to XML artifacts (Cross-Language dependencies). In the context of J2EE, Hecht et al. use dynamic analysis to detect these File generation dependencies.

Documentation dependencies exist when the documentation refers to the source code. For example, the JavaDoc has special annotations to refer to classes, methods or their parameters. Some refactorings are able to detect and modify the comments when an artifact is renamed. These dependencies might be considered less critical because they don't affect the behavior of the application. Yet they are important for the readability and understandability of the source code. In this case, the "external agent" introducing the *External Dependencies* might be considered to be the human reader.

As noticed above, these categories are not mutually exclusive and actually frequently co-existent. We saw the example of Generated file dependencies and Cross-language dependencies, but a Multi-tier dependency will often come with a Callback dependency to allow one tier to answer to events in another tier.

3.2 Dependency correctness

A dependency exists between two artifacts (or entities) that we will call *reference* and *Resource*. Following Pfeiffer and Wąsowski definitions: "*a* [Resource entity][5] *is a fragment of code that introduces an identifiable object, a concept, etc.*", and "*a* [Reference entity] *is a location in code that relates to a* [Resource entity]". The definitions need to be more generic as we are not dealing solely with source code (XML files, documentation), but we will mostly restrict ourselves to *External Dependencies* which involve some source code, although we sometimes also consider dependencies between CSS specifications and HTML documents, which are typically not considered source code.

Most of the work cited in Section 2 focuses on identifying the *External Dependencies* to help fully understand the application and preserve the quality of the modified source code. We saw that Soto-Valero et al. [18] went one step further by considering the correctness of dependencies. Also, Pfeiffer and Wąsowski [14] state that the dependencies are always many-to-one between Reference entities and Resource entities.

The issue of correctness is important as it allows for checking (and possibly preserving) the quality of the application. The idea of cardinality (many-to-one) is an obvious candidate to express possible restrictions on the *External Dependencies*. However, although the many-to-one is the most frequent, we found the statement of Pfeiffer and Wąsowski to be incorrect in some cases; not all *External Dependency* are many-to-one.

We consider the cardinality of *External Dependencies* independently from the Reference side and the Resource side.

Resource side: The Resource entity might be mandatory or optional.

Mandatory: This is the most common case, the Resource entity referenced must exist or there will be an error at compilation or execution time.

[5] they call them "key"

Optional: Less common examples can be found in HTML tags (e.g., a specific `div`) considered as Resource entities. A CSS file can reference ("depend on") a non existent HTML tag, a JavaScript function could similarly look for the optional existence of a given node in the DOM of an HTML page.

Reference side: A Reference entity can be mandatory or optional, single or multiple:

Multiple: This is the most common case, a Resource entity can be referenced by multiple Reference entities.

Single: In specific cases such as ORM[6], there can be only one object referring to any element defined in a database.

Optional: Most of the time, Reference entities are optional. For example, one may not use a possible callback offered by a library.

Mandatory: There are cases where a Reference entity is mandatory. Its absence will often not produce a compilation or execution error, but it still denotes a flaw or a lack of understanding of the framework used. In the worst cases, this lack of understanding is circumvented by the developers with some needlessly complex and non-standard constructions in the source code.

The cardinality for any *External Dependency* is fixed by the framework considered.

Stating the cardinality of an *External Dependency* allows a tool to not only identify the dependencies (to help analyze the application), but also to verify them (to help discover flaws). We will see in our examples (Section 5) that we did find such flaws.

We propose two main categories of flaws/errors:

Excessive dependencies: They characterize a Resource entity that was created but never referred to [8]. In multi-tier applications, we had the case where many services were declared on the server side but never used on the client side. They were not possible extensions to be used by other client applications, but old code that ceased to be used ("dead services"). Other possible cases are when the developer made a mistake and the Reference entity points to a wrong (similar) Resource entity. Soto-Valero et al. [18] call them "bloated dependencies" when removing unused Java classes from imported jar files.

Missing dependencies: They characterize Reference entities that are referring to Resource entities that do not exist. Such dependencies are uncommon since most applications cannot run properly if resources are not available. Such dependencies have a higher chance of existing in the case of HTML and CSS, or comments. In this case, they would result in a wrong/unappealing rendering of a web application, and misleading comments. Note that wrong web page rendering can lead to an unusable web application in some cases.

4 External Dependencies Detection

This section explains how we implement a generic *External Dependency* detector. We first decompose the problem into two sub-problems, which allows us to develop a small library of reusable patterns. We then give the general architecture of our solution.

[6] Object Relational Mapping wikipedia.org/wiki/Object\T1\textendashrelational_mapping

4.1 Requirements

Going back to the categorization of work proposed in Section 2, we want a generic solution that allows us to identify dependencies with the following properties:

Dependency types: No limitation, we are interested in *External Dependencies* coming from multi-tier programming, cross-language programming, the generation of files during the installation process, and libraries making use of callbacks.

Matching strategy: The object matching strategy (on a model of the application) allows to identify more complex structures than lexical matching, for example searching for "a method in a class implementing a given interface and carrying a given annotation". This is why many publications have used it. However, building a model of an application may be a complex task, for example for less commonly used programming languages for which there is no generic parser and symbol resolver available. In these cases, lexical matching is a simpler solution, fast and flexible to implement [5].

We will allow both solutions, taking advantage of whatever tools are already available for a given language.

Engine: Some publications propose automatic solutions that are able to discover dependencies without the user specifying where to look for them. All of these are specific to one given framework except Polychniatis et al. [15] which may produce many false positives.

We will prefer a rule-based engine where we specify for each framework where to look for dependencies. This is the preferred solution in the domain (see Table 1).

Analysis: As for any software analysis approach, one can use static or dynamic analysis. The static approach is usually favored. It is easier to apply across various programming languages (and other languages too) and across application contexts. We will use this solution too.

File generation dependencies have been covered dynamically by Hecht et al. [6]. That could be a limitation of our approach for this dependency type.

4.2 Decomposing the problem

To simplify the implementation of a generic solution, we decompose the dependency identification problem into two parts: First, we look for *Containers*, software artifacts that may contain a Reference entity or a Resource entity. Containers are usually large artifacts, such as an entire file. Second, within a potentially interesting container, we look for *Entities* that will be either Reference entities or a Resource entities. For example, a (Reference) container in the GWT framework could be any Java class that contains the annotation `@uiField` or `@uiHandler`. These annotations are used in the GWT framework to establish the link between Java and XML.

This decomposition can help specifying the rules when using lexical matching.

We limit a container to be written in any single language, like a Java container, an XML container, etc. This simplifies the work of searching for entities in a container. Note that this may lead to a non-obvious situation where an SQL string inside a Java file or a comment inside any source code is considered a different container because the language is not the same. It may not be very important for lexical matching rules, but it is key for object matching rules that are based on models: They would require an SQL or a comment model to search in.

In the literature, many approaches search for some kind of "container" [4,7–9,13,17] by declaring specific search patterns or by discovering them with some heuristics. Thus,

Kempt et al. [7] propose first filtering containers that may contain such dependencies to accelerate the research.

There are two kinds of containers: Reference containers and Resource containers. In the GWT example above, the Java class containing the annotation is a Reference container and the Resource container will be an entire XML file describing the GUI.

Within these containers, we apply matching patterns to find the entities involved in *External Dependency*. Again, for the GWT example, inside the Reference container, we will be looking for specific attributes of methods carrying the GWT annotations.

4.3 Reusable patterns

Decomposing the problem into containers and entities allows us to have reusable patterns. For example, searching for an XML element according to one of its attribute names can be done whether this is an XML configuration file, or an XML layout file. It would therefore be applicable to *External Dependencies* detection in J2EE or GWT.

For a given language, the patterns can be used for container or entity detection. Each pattern can be expressed declaratively given the object type and its properties in form of key-value pairs. Consider the below pattern description for finding annotations in Java model: (i) the type of the object that we're looking inside is a `FamixJavaModel` (ii) in order to extract the annotations we use `allAnnotationTypes` property (iii) the pattern accepts annotation name parameter associated to value property of `FamixJavaAnnotationType` to search for annotations we are interested in (ie. 'UiField') (iv) the pattern returns the container name where the previously specified attribute is found, using `FamixJavaAttribute` parentType property, also expressed declaratively in the same pattern by using Temporary variables (ie. '@ClassObject')

This is especially important for object matching patterns, where we need to build a model of the source code. Working with this decomposition and reusable patterns allow to more easily add new frameworks.

4.4 Adonis

Our solution is implemented in a tool called Adonis (https://github.com/alesshosry/Adonis). First, Adonis takes as input two elements: the code to analyze and the framework identification (eg. GWT or RMI). The framework must be known by the tool: (i) programming language importer if we want to use object matching; (ii) detection rules for Reference containers, Resource containers, Reference entities, and Resource entities. This follows from the definition of *External Dependencies*, where one needs to know the usage rules of a framework to be able to identify the dependencies it introduces. Detection rules for a given framework can be based on reusable patterns for the considered programming language. That is to say, a pattern to detect a given Java class can be used by rules for the GWT or RMI framework (and any framework based on the Java programming language).

Second, Adonis offers two "engines": lexical and object based matching, depending on the rules describing the framework to analyze. The selection of an engine depends on the availability of meta-models and importers to generate source code models.

Third, Adonis applies the rules of the framework to identify Reference and Resource containers/entities. It maps Reference entities to their Resource entities, leading to the generation of *External Dependencies* that were implicit in the framework. By considering the restrictions defined in 3.2, it can also identify possible flaws.

5 Validation

We validate our approach on two projects taken from GitHub. The objective is to evaluate whether (i) the approach can identify *External Dependencies*, (ii) it works for different types of dependencies, (iii) we can reuse patterns across frameworks, and (iv) it can detect flaws (missing or excessive dependencies).

5.1 Validation setup

We are interested in how our approach can be adapted to different frameworks, if possible with different kinds of dependency types (cross-language, multi-tier, etc.). For practicality reasons, we chose two frameworks involving the same languages: Java and XML. Thanks to the Moose platform [1], we already have a Java importer and meta-model on which we can easily express object matching patterns. Building an XML model is easy since there are many libraries to parse XML.

Cross-language experiment For the cross-language example we will focus on GWT, a framework to develop Web applications using a combination of Java and XML. In GWT Java artifacts (class attributes or methods) are "linked" to widgets described in XML files through the annotations `UiField` and `UiHandler`. For example, Listing 1.1 shows how an attribute on line 3 is referring to a Window defined in Listing 1.2 as an XML element (lines 1 to 5). Similarly, the Java method on line 5 is linked to the Button widget on lines 2 to 4 of the XML file. GWT requires that the Java and XML files be located in the same folder and have the same name (without their respective extensions: `.java` and `.xml`).

Listing 1.1: GWT Java

```
1 public class ApplicationSettingsDialog implements Editor<ApplicationSettings> {
2   @UiField
3   protected Window window;
4   @UiHandler("saveButton")
5   public void onLoginClicked(SelectEvent event) {
6     window.hide();
7   }
8 }
```

Listing 1.2: GWT XML

```
1 <gxt:Window ui:field="window" pixelSize="300, 110" modal="true" headingText
      ="Global Settings" focusWidget="{saveButton}">
2   <gxt:button>
3     <button:TextButton ui:field="saveButton" text="Save" />
4   </gxt:button>
5 </gxt:Window>
```

The cardinality of these *External Dependencies* is many-to-one, as several Java associations can refer to the same XML attribute. The XML attributes (Resource entities) are mandatory, the Java ones (Reference entities) are optional, ie. Java part is not required to refer to all XML widgets.

We manually counted the number of Reference containers (ie. some java classes), Resource containers (ie. `.xml` files); Reference entities in the Java code; Resource entities defined in the XML files; and *External Dependencies* found (ie. the number of Reference entities with a matching Resource entity).

Multi-tier experiment For the multi-tier example, we will focus on the RMI[7] framework that is also based on Java. RMI allows to build distributed applications where a client part can call server methods running in a different JVM.

A Java interface is needed (`TheRemoteInterface` in the example below). It extends `java.rmi.Remote` and declares methods that can throw `java.rmi.RemoteException`.

On the server side, a class (`TheServerClass`) implements this interface and defines the service methods (line 1 in Listing 1.3). An instance of this class is registered in the RMI registry (lines 3 and 4 in Listing 1.3).

Listing 1.3: RMI server code

```
1 public class TheServerClass implements TheRemoteInterface { ... }
2 ...
3 Registry registry = LocateRegistry.createRegistry(<port number>);
4 registry.bind("rmi://localhost/TheServerClass", new TheServerClass());
```

On the client side, an instance of this interface is obtained from the RMI registry (Listing 1.4) and calls to its methods will be forwarded to the server application.

Listing 1.4: RMI client code

```
1 Registry registry = LocateRegistry.getRegistry(<number>);
2 TheServerInterface instance = (TheServerInterface) registry.lookup("rmi://
      localhost/TheServerClass");
```

Again, we manually counted the number of Reference containers (ie. some java interfaces), Resource containers (ie. java classes); Reference entities in the Java classes; Resource entities defined in the java interfaces; and *External Dependencies* found (ie. the number of Reference entities with a matching Resource entity).

5.2 Use cases

For the cross-language experiment, we chose the Traccar project[8] that uses the GWT framework. It was created in 2012, has 100 commits, and in total 34 Java classes, and 13 XML files. We counted the containers and entities manually and found in total 10 Reference containers out of the 34 Java classes and 10 Resource containers out of the 13 XML files. Some of the Resource entities (in XML) are not referred to in Java, and multiple Reference entities (in Java) may refer to the same Resource entity. The number of Resources, Referred Resources and External Dependencies are presented in Table 2. Each row in the table is for a pair of matching `.java` and `.xml` files (ie. Reference/Resource containers).

For the multi-tier experiment, we chose the UniScore solution[9]. It was created in 2020 and has a track record of more than 100 commits. Uniscore-Server is the server

[7] Remote Method Invocation
[8] https://github.com/traccar/traccar-web/tree/legacy
[9] https://github.com/redhawk96/UniScore

Table 2: GWT (cross-language) experiment

Containers	Resources	Referred Resources	External dependencies
ApplicationSettingsDialog	5	4	4
ApplicationView	12	4	4
ArchiveView	14	11	11
DeviceDialog	6	5	5
DeviceView	16	12	17
LoginDialog	6	5	6
StateView	5	4	4
UserDialog	7	6	6
UsersDialog	10	6	8
UserSettingsDialog	5	4	4

part sub-project of the UniScore solution. It is composed of two interfaces and 41 classes. Uniscore-Client is the client part sub-project of the UniScore solution. It has one interface and 50 classes. We manually counted the number of Resource containers that is, interfaces extending `java.rmi.Remote`. There is only one such interface in the project that declares 54 methods which are the Resource entities. In parallel the Reference containers are classes that use an instance of the Resource container. We found 14 such classes out of the 50 classes in the client part. For each Reference container, we also manually counted the number of entities, that is to say methods invoking Resource entities. The number of Resources, Referred Resources and External Dependencies are presented in Table 3. Each row in the table represents a Reference container.

5.3 Rules and Patterns

We developed patterns for Java and XML languages used by the two frameworks of our experiments. We defined two XML patterns and 12 Java patterns in all. Our strategy consists of filtering containers and finding entities inside each one.

For the cross-language experiment (GWT framework), we used one rule to find Resource containers (XML files): It looks for all XML files having a name matching a Java file. For the Resource entities, we use one rule: It looks for all XML nodes having a `UIField` or `Field` attribute. This rule uses twice the same XML pattern that looks for XML nodes having a given attribute.

We used one rule to find Reference containers (Java classes): It looks for all Java classes using a `@UiHandler` or `@UiField` annotation. This rule uses twice the same Java pattern that looks for annotation instances having a given name. For the Reference entities, we use one rule: It looks for the argument value of all `@UiHandler` or `@UiField` annotations in the container. Again, this rule uses twice the same Java pattern to look for annotation instances.

For the multi-tier experiment (RMI framework), we used one rule to find Resource containers (Java interfaces): It looks for all Java interfaces extending the `java.rmi.Remote` interface. This rule uses a Java pattern that looks for all implementations of a given Java interface. For the Resource entities, we use one rule: It looks for all Java public

Table 3: RMI (multi-tier) experiment

Containers	Resources	Referred Resources	External dependencies
SubmissionContentPanel	54	1	1
DashboardContentPanel	54	7	7
ExamContentPanel	54	8	9
LogoutNavigationPanel	54	1	1
LoginContentPanel	54	4	8
UniScoreClient	54	2	2
RemoveQuestionNotifier	54	2	2
DisplayQuestionsContentPanel	54	2	2
CreateQuestionContentPanel	54	2	2
DisplayQuestionContentPanel	54	2	2
StudentContentPanel	54	4	4
ModuleContentPanel	54	1	1
QuestionnaireContentPanel	54	4	4
SubmissionMailer	54	3	3

methods that declare throwing `java.rmi.RemoteException`. This rule uses two Java patterns, one looking for public methods and the other looking for methods declaring to throw a given exception.

We used one rule to find Reference containers (Java classes): It looks for all Java classes that use instances of at least one of the Resource containers. This rule uses one Java pattern to look for instances of a given interface and this pattern is called repeatedly for each of the Resource containers found. For the Reference entities, we use one rule: It looks for invocations of any of the Resource entities found. This rule uses a Java pattern to look for invocations of a given method.

5.4 Results

We ran Adonis for both experiments to detect the number of containers, entities and *External Dependencies* and compare them with what we found by manual count.

For the GWT experiment, Adonis was able to correctly detect the 10 containers (rows of Table 2). It also correctly detected all Resources and References.

The analysis of "Resources" and "Referred Resources" columns shows that multiple Resource entities are defined in XML but never referred to in Java. For example, for the `DeviceView` containers, only 12 out of 16 Resource entities were referred to. This is allowed in the GWT framework and is not considered a flaw.

Moreover, the comparison between "Referred Resources" and "External Dependencies" columns shows that the number of *External Dependencies* can be larger than the number of referred Resource entities, indicating that some Resource entities are referred by several Reference entities. For example, for the `UsersDialog` containers, there are 8 *External Dependencies* for only 6 Reference entities. Two references to `removeButton` and `addButton` can be found in the source code. Again, this is allowed in the GWT framework and is not considered a flaw.

For the RMI experiment, Adonis was able to correctly detect the 14 Reference containers (rows of Table 3). It also correctly detected all Resources and References entities.

The analysis of "Resources" and "Referred Resources" columns shows that multiple Resource entities are defined in the server but never referred to in the client. For example, in `DashboardContentPanel` container, we can see that only 1 out of 54 Resource entities is referred to.

In total, we found that only 26 Resource entities out of 54 were referred, which results the existence of 28 excessive dependencies. Moreover, the comparison between "Referred Resources" and "External dependencies" shows that the number of referred Resource entities can be less than the number of *External Dependencies*. For example, for the `DashboardContentPanel` class, we can see that none of the Resource entities was referred more than once, which is not the case for `ExamContentPanel` class, where we found `addLogActivity` was referred twice. This explains why we found 9 *External Dependencies* instead of 8.

6 Conclusion

Various frameworks are used to create various types of software. These frameworks rely on a set of rules specific to each one to establish dependencies. We conclude the existence of multiple types of *External Dependencies* such as multi-tiers, callbacks, and cross-language links in polyglot programming. We also state that even if each framework connects its languages or tiers in a unique way, a common approach emerges. This approach is based on finding the correct containers that lead to the identification of specific entities defined according to the framework's rules. Additionally, this approach defines restrictions that lead to error detection, making it important to know how to proceed with the re-engineering work. To design a detector that can achieve the usage goals successfully, we established a set of requirements and based on them, we developed our tool *Adonis* with the flexibility of defining or using existing patterns to limit the number of containers, detect entities, and link them following each framework rules that we experimented with two different projects.

References

1. Anquetil, N., Etien, A., Houekpetodji, M.H., Verhaeghe, B., Ducasse, S., Toullec, C., Djareddir, F., Sudich, J., Derras, M.: Modular moose: A new generation of software reengineering platform. In: International Conference on Software and Systems Reuse (ICSR'20). No. 12541 in LNCS (Dec 2020). https://doi.org/10.1007/978-3-030-64694-3_8
2. Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., Chen, Y.: Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In: NDSS (2015)
3. Cossette, B., Walker, R.J.: Dsketch: Lightweight, adaptable dependency analysis. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. pp. 297–306 (2010)
4. Grichi, M., Abidi, M., Jaafar, F., Eghan, E.E., Adams, B.: On the impact of interlanguage dependencies in multilanguage systems empirical case study on java native interface applications (jni). IEEE Transactions on Reliability **70**(1), 428–440 (2020)

5. Griswold, W.G., Atkinson, D.C., McCurdy, C.: Fast, flexible syntactic pattern matching and processing. In: WPC'96. 4th Workshop on Program Comprehension. pp. 144–153. IEEE (1996)
6. Hecht, G., Mili, H., El-Boussaidi, G., Boubaker, A., Abdellatif, M., Guéhéneuc, Y.G., Shatnawi, A., Privat, J., Moha, N.: Codifying hidden dependencies in legacy j2ee applications. In: 2018 25th Asia-Pacific Software Engineering Conference (APSEC). pp. 305–314. IEEE (2018)
7. Kempf, M., Kleeb, R., Klenk, M., Sommerlad, P.: Cross language refactoring for eclipse plug-ins. In: Proceedings of the 2nd Workshop on Refactoring Tools. pp. 1–4 (2008)
8. Mayer, P., Schroeder, A.: Cross-language code analysis and refactoring. In: 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation. pp. 94–103. IEEE (2012)
9. Mayer, P., Schroeder, A.: Automated multi-language artifact binding and rename refactoring between java and dsls used by java frameworks. In: ECOOP 2014–Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28–August 1, 2014. Proceedings 28. pp. 437–462. Springer (2014)
10. Mushtaq, Z., Rasool, G., Shehzad, B.: Multilingual source code analysis: A systematic literature review. IEEE Access **5**, 11307–11336 (2017)
11. Neubauer, M., Thiemann, P.: From sequential programs to multi-tier applications by program transformation. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 221–232 (2005)
12. Nguyen, H.V., Nguyen, H.A., Nguyen, T.T., Nguyen, T.N.: Babelref: detection and renaming tool for cross-language program entities in dynamic web applications. In: 2012 34th International Conference on Software Engineering (ICSE). pp. 1391–1394. IEEE (2012)
13. Pfeiffer, R.H., Wąsowski, A.: Taming the confusion of languages. In: European Conference on Modelling Foundations and Applications. pp. 312–328. Springer (2011)
14. Pfeiffer, R.H., Wąsowski, A.: Texmo: A multi-language development environment. In: European Conference on Modelling Foundations and Applications. pp. 178–193. Springer (2012)
15. Polychniatis, T., Hage, J., Jansen, S., Bouwers, E., Visser, J.: Detecting cross-language dependencies generically. In: 2013 17th European Conference on Software Maintenance and Reengineering. pp. 349–352. IEEE (2013)
16. Shatnawi, A., Mili, H., Abdellatif, M., Guéhéneuc, Y.G., Moha, N., Hecht, G., Boussaidi, G.E., Privat, J.: Static code analysis of multilanguage software systems. arXiv preprint arXiv:1906.00815 (2019)
17. Shen, B., Zhang, W., Yu, A., Wei, Z., Liang, G., Zhao, H., Jin, Z.: Cross-language code coupling detection: A preliminary study on android applications. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 378–388. IEEE (2021)
18. Soto-Valero, C., Harrand, N., Monperrus, M., Baudry, B.: A comprehensive study of bloated dependencies in the maven ecosystem. Empirical Software Engineering **26**(3), 45 (2021)
19. Vanciu, R., Rajlich, V.: Hidden dependencies in software systems. In: 2010 IEEE International Conference on Software Maintenance. pp. 1–10. IEEE (2010)
20. Yu, Z., Rajlich, V.: Hidden dependencies in program comprehension and change propagation. In: Proceedings 9th International Workshop on Program Comprehension. IWPC 2001. pp. 293–299. IEEE (2001)