Université de Lille, Faculté des sciences et technologies

Département de formation doctorale en informatique
UFR IEEA

École doctorale MADIS Lille

# External Dependencies in Programs: Specification, Detection and Incorrectness

## Dépendances Externes dans les Programmes : Spécification, Détection et Inexactitudes

## THÈSE

présentée et soutenue publiquement le 6 Décembre 2024

pour l'obtention du

**Doctorat de l'Université de Lille**

**(spécialité informatique et application)**

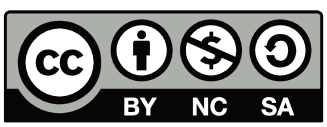
par

Aless Hosry

**Composition du jury**

| | | |
|---|---|---|
| *Président :* | Walter RUDAMETKIN | Professeur des universités<br>Université de Rennes, IRISA |
| *Rapporteurs :* | Salah SADOU | Professeur des universités<br>Université Bretagne Sud, IRISA |
| | Ghizlane EL BOUSSAIDI | Professeure<br>Université du Québec, ETS |
| *Examinateur :* | Walter RUDAMETKIN | Professeur des universités<br>Université de Rennes, IRISA |
| *Directeur de thèse :* | Nicolas ANQUETIL | Maître de Conférences<br>Université de Lille, CNRS, Inria, CRIStAL |

Centre de Recherche en Informatique, Signal et Automatique de Lille — UMR 9189
INRIA Lille - Nord Europe



# Acknowledgments

First and foremost, I would like to express my deepest gratitude to the jury members for their valuable time, insights, and feedback on my work. Your contributions have been invaluable to the completion of this thesis.

I am deeply grateful to my supervisor, **Nicolas**, for his continuous support, guidance, and patience throughout this journey. Your expertise and encouragement have been essential in helping me navigate the challenges of my research.

To my former supervisor **Vincent**, who continued to be a source of guidance and support. Your willingness to offer advice, despite the distance and circumstances, meant more to me than words can express.

I extend my sincere thanks to our team leader **Stéphane**, whose lessons on teamwork and collaboration have been instrumental in my personal and professional growth. A special thank you to Professor **Anne**, who entrusted me with the responsibility of teaching alongside her. Your support, guidance, and invaluable advice, both in teaching and in working as part of the team, have been crucial to my development as an educator and researcher.

To my team members, thank you for the stimulating discussions, the experiences shared, and the moments of teamwork and fun that made this journey all the more memorable.

I owe my deepest gratitude to my parents, **Edward** and **Marie**, who supported my decision to leave and pursue a PhD abroad. Your continuous support and belief in my potential gave me the strength to persevere. To my brother **Charbel**, who always stood by my parents side during my absence. I want to thank you for your support to me, for the funny and best moments we shared together. You will always be my beloved little brother ;)

To my extended family, starting with my beloved grandmother, **Teta Alice**, you are truly the best and I will never forget what you taught me! To my uncles, aunts, cousins, and "sisters", I love you all dearly. Your love and encouragement have been a constant source of strength.

To my former colleagues, thank you for your support and encouragement throughout this journey. Your belief in me helped me stay motivated.

To my friends in Lebanon and my second family in France, you have made so many things easier for me. Your kindness and friendship have been a blessing, and I am forever grateful for the comfort and joy you brought into my life.

To my beloved country, your resilience in these difficult times inspires me to keep pushing forward. I hold onto my love for you and hope for lasting peace.

# 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. Another example is software that connects to a database, where SQL queries are embedded within the source code of this software (such as in Java, .NET, etc.). These queries refer to database entities like tables and stored procedures. We call such dependencies external because they are introduced by some agent external to the source code. We call such dependencies external because they are introduced by some agent external to the source code. They are not easily detectable as they exist between parts (like different programming languages, different tiers ...).

In this thesis, we developed a generic tool named Adonis, which uses reusable patterns to identify dependencies. We implemented this tool in the Pharo programming language and validated it across various open source and industrial projects. During implementation, we realized the need for a search engine capable of identifying parts of external dependencies, regardless of their source code language, their depth within the code, or the complexity of their location. To address this need, we created MoTion, a declarative pattern matching language capable of defining patterns and matching objects or trees of objects in imported models in Pharo, as well as matching text strings using regular expressions. Additionally, we discovered that external dependencies are sometimes incorrectly established, potentially leading to program flaws. Identifying these dependencies is crucial for developers to make informed decisions on correcting or removing them to avoid potential issues or side effects. We developed an approach to detect such incorrect external dependencies, based on both literature and our research findings, and validated this approach on the same open source and industrial projects as for Adonis.

**Keywords:** External dependencies,incorrect dependencies, software quality, pattern matching.

# Résumé

Un logiciel réussi nécessite des modifications constantes. Pour garantir le bon fonctionnement continu des applications, les développeurs doivent bien les comprendre, notamment en ayant une carte précise des dépendances entre les parties qu'ils modifient. Cependant, certaines de ces dépendances ne sont pas facilement identifiables. Par exemple, dans une application Android, il existe des dépendances entre le code source Java et les parties XML, dont certaines sont matérialisées par une classe Java "R" générée. Un autre exemple est le logiciel qui se connecte à une base de données, où des requêtes SQL sont intégrées dans le code source de ce logiciel (comme en Java, .NET, etc.). Ces requêtes font référence à des entités de base de données telles que des tables et des procédures stockées. Nous appelons ces dépendances externes car elles sont introduites par un agent externe au code source. Elles ne sont pas facilement détectables car elles existent entre des parties (comme différents langages de programmation, différentes couches, etc.).

Dans cette thèse, nous avons développé un outil générique nommé Adonis, qui utilise des modèles réutilisables pour identifier les dépendances. Nous avons implémenté cet outil dans le langage de programmation Pharo et l'avons validé sur divers projets open source et industriels. Au cours de l'implémentation, nous avons réalisé la nécessité d'un moteur de recherche capable d'identifier des parties de dépendances externes, indépendamment de leur langage de code source, de leur profondeur dans le code ou de la complexité de leur emplacement. Pour répondre à ce besoin, nous avons créé MoTion, un langage de correspondance de motifs déclaratif capable de définir des modèles et de faire correspondre des objets ou des arbres d'objets dans des modèles importés dans Pharo, ainsi que de faire correspondre des chaînes de texte à l'aide d'expressions régulières. De plus, nous avons découvert que les dépendances externes sont parfois établies de manière incorrecte, ce qui peut entraîner des défauts dans le programme. Identifier ces dépendances est crucial pour que les développeurs puissent prendre des décisions éclairées sur leur correction ou leur suppression afin d'éviter des problèmes ou des effets secondaires potentiels. Nous avons développé une approche pour détecter ces dépendances externes incorrectes, basée à la fois sur la littérature et nos résultats de recherche, et avons validé cette approche sur les mêmes projets open source et industriels que pour Adonis.

**Mots-clés:** Dépendances externes, dépendances incorrectes, qualité logicielle, correspondance de motifs.

# Contents

# List of Figures

# List of Tables

CHAPTER 1

# Introduction

## Contents



## 1.1 Context

In modern software, the usage of multiple programming languages and multiple tiers to develop one program is dominant. This results in the existence of multiple components that depend on each other to ensure a proper functioning of this program. These components can be for example a Java program (first tier) intended to be used by a group of clients, that is connected to a database (second tier) and *depends* on it to handle the data provided by these clients. That is to say, a flow of dependencies exist between the Java program and the database. We refer to these dependencies in our thesis as *external dependencies*. The way those external dependencies are established is usually by agents like frameworks. Such programs require a solid understanding from the developers and maintainers that need to understand how different components depend on each others, more specifically the external dependencies that are established between the components. The identification of such external dependencies is becoming more and more challenging, specially there are few identification tools to show and lead the developers to their existence. And if they exist, they are limited to specific frameworks.

We propose in this thesis a new tool that reveals the existence of those external dependencies to the developers. Our tool has a simple goal: provide it with the necessary information like the program that you are analyzing and the agent used, and it will provide you with the external dependencies that you are searching for.

We were contacted by companies to analyze their programs. They provided us with some of their programs, as for the other programs that we could not have

access to, we searched for projects developed using the same agents on GitHub and started our work. During our first investigations, we discovered the existence of incorrect external dependencies because they violated the rules that defines how they should be established. So we decided in a later stage to adapt our tool in way to be able to filter the correct and incorrect external dependencies.

To conduct our investigation in this thesis, we used the Pharo programming language [Black 2009]. Pharo is an open-source, dynamically-typed, reflective, object-oriented programming language inspired by Smalltalk. It also serves as an integrated development environment (IDE) built entirely in itself. However, our findings are not confined to Pharo alone and can be applied to other programming languages like Java, or Python.

## 1.2 Problem Statement

In this thesis, we study how we can support the identification of the external dependencies between components. Specifically, we address three problems:

1. *Detection problem*. Understand how these external dependencies are established, and how the entities that relate those external dependencies can be found. Is there a way to create a generic tool, able to identify them regardless of the agent used? We address this problem in Chapter 3.

2. *Searching engine problem*. The entities that constitute these external dependencies are part of the programs. What are the searching engines that can be used to find tokens in programs? How can we express search patterns that are reusable? How they can be used in our case? Is there any limitations? We explore more this problem in Chapter 4.

3. *Correctness problem*. We need an approach to help us distinguish between correct and incorrect external dependencies. What are incorrect external dependencies? How to identify them? We address this problem in Chapter 5.

## 1.3 Contributions

The main contribution of this thesis are:

1. **Identification and detection of external dependencies**. We defined and categorized external dependencies. We created common approach that helped us structure the way to find those dependencies. We created a tool that we named Adonis, and validate it over public and private projects.

2. **Searching for tokens in programs**. We created a new declarative pattern matching tool capable of defining patterns suitable for the description of entities and we validate it on our projects and with other developers.

3. **Validation of external dependencies**. We created a novel approach, that clarifies what are the different incorrect external dependencies that can be found. We extended Adonis to filter the correct and incorrect dependencies then run the experiments over the same projects.

## 1.4 Thesis Outline

The thesis is organized as follows:

- In Chapter 1, we introduce the challenge of managing external dependencies, arise when different components like different programming languages or tiers are used in the development process.

- In Chapter 2, we review various approaches from the literature for detecting external dependencies and classify the approaches that address their correctness. Furthermore, we explore two primary methods for pattern matching: graph-based pattern matching and pattern matching in general-purpose languages (GPLs).

- In Chapter 3, we explore external dependencies in details and give multiple examples. We present Adonis, its usage and how patterns can be reused to allow a flexible extension of it to cover more external agents. We run experiments on public and private projects, present the results, validate them and show the needs of using pattern matching and revealing incorrectness.

- In Chapter 4, we present MoTion, as a new declarative pattern matching DSL (Domain-Specific Language) in Pharo. We discuss its implementation promoting its extensibility. We compare the traditional pattern matching language in Pharo with MoTion and show the results. We comment on some real use cases where MoTion was used by developers in software analysis tasks including Adonis. From these use cases, we derived some lessons learned on the most useful features of MoTion or what improvements can be done.

- In Chapter 5 we focus on identifying and detecting incorrect external dependencies by linking their correctness to the minimal and maximal multiplicities of involved entities. We introduced the types of incorrect dependencies on these multiplicities, and we validate the approach categorizing prior research and examples. We enhance Adonis to detect with common Pharo statements the incorrect dependencies and run experiments on public and private projects.

- Finally in Chapter 6, we conclude our thesis and present the future work.

## 1.5 List Of Publications

The list of papers published in the context of the thesis is listed below in chronological order:

**Pattern matching in Pharo.**
[Hosry 2023b] - Workshop Paper
*Aless Hosry, Vincent Aranega, Nicolas Anquetil, Stéphane Ducasse.*
International Workshop on Smalltalk Technologies: IWST 23, Aug 2023, Lyon, France.
https://hal.science/hal-04217930.

**External dependencies in software development.**
[Hosry 2023a] - Conference Paper
*Aless Hosry, Nicolas Anquetil.*
International Conference on the Quality of Information and Communications Technology, Sep 2023, Aveiro, Portugal.
https://hal.science/hal-04217300.

**MoTion: A new declarative object matching approach in Pharo.**
[Hosry 2024] - Journal Paper
*Aless Hosry, Vincent Aranega, Nicolas Anquetil.*
Journal of Computer Languages.
https://doi.org/10.1016/j.cola.2024.101290.

CHAPTER 2

# State of the Art

## Contents



In this chapter, we review various approaches from the literature for detecting external dependencies. Additionally, we classify research that addresses their correctness. Finally, we present two main approaches for pattern matching: graph pattern matching and pattern matching in GPLs. We considered both approaches to incorporate their features, aiming to develop a new pattern matching language in Pharo that helps identifying the parts forming an external dependency.

## 2.1 Existing Approaches

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.

### 2.1.1 Existing Detectors

Yu and Rajlich [Yu 2001] discussed "hidden dependencies". Their goal is to study such dependencies in order to enhance the software comprehension that is a prerequisite of any change. They define hidden dependency as a "relationship between

two seemingly independent components and it is caused by a data flow inside of a third software components". However, their work is limited to monolingual applications and the components within them.

Shatnawi et al. [Shatnawi 2019] developed an eclipse plugin named DeJEE that is able to analyze J2EE applications and draw a call dependency graph of the multilanguage dependencies in J2EE applications. Their approach relies on parsing the source and using KDM models to extract the dependencies. Their approach is applicable for J2EE and they intend to extend it to cover other frameworks. However, using KDM can be challenging because it standardizes the representation of model entities regardless of the programming language and runtime platform. This standardization can limit KDM's ability to accurately represent some components where dependencies exist, potentially resulting missing results.

BabelRef, developed by Nguyen et al. [Nguyen 2012], is a tool that automatically detects dependencies between generated client artifacts (HTML and JavaScript) and server-side artifacts (PHP). It uses object matching and applies dynamic analysis with a single tree-based structure known as the D-model [Nguyen 2011], specifically designed for server-side PHP code. However, a major limitation of this approach is the need to execute each PHP page to conduct a full analysis of the project. This requirement makes the tool less practical for those studying projects they cannot run, or for analyzing legacy programs, which often have complex structures that must be understood before each page can be executed. This approach is counterproductive for developers because these tools are generally meant to assist them in understanding the project, whereas here it necessitates a prior understanding of the project before they can use the tool.

EdgeMiner is a tool developed by Yinzhi et al. [Cao 2015] to automatically detect callbacks for Android framework applications created using 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. [Hecht 2018] developed another approach that detects the dependencies established in J2EE applications using codified rules. Their work includes detection of dependencies established between multiple languages or tiers, generated on run-time or established using callbacks. Still, the approach is limited to the J2EE framework, and it is not clear how it can be extended to other frameworks.

Polychniatis et al. [Polychniatis 2013] 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. This approach is relatively generic, and can be applicable for many frameworks as no parsing is required However, it lacks accuracy because many tokens can have identical values but be used in different contexts, similar to two buttons with the

same ID in a program that are distinct from one another.

Grichi et al. [Grichi 2020] 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. However, this approach is limited for developers who only have access to the latest version of the software and not its historical versions.

GenDeMoG, which is a tool developed by Pfeiffer and Wąsowski [Pfeiffer 2011], allows specifying intercomponent dependency patterns statically for artifacts in systems. The tool relies on parsing languages, querying source code objects each time, and retrieving the possible dependencies between different artifacts. However, GenDeMoG requires a customized "Component Descriptor Model" for each project to express patterns and match dependencies, even if the same framework is used across different projects.

Mayer and Schroeder [Mayer 2012] 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. However, relying solely on parsers and metamodels is not always a feasible solution, especially when dealing with concatenated strings in another source code (such as SQL queries embedded in Java), where parsing becomes particularly challenging.

Dsketch is a tool created by Cossette and Walker [Cossette 2010] 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. While relying solely on lexical matching can speed up execution and yield quick results, it may compromise accuracy.

Soto-Valero et al. [Soto-Valero 2021] 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 discussed dependency correctness (or incorrectness in the case of bloated dependencies).

Kempf et al. [Kempf 2008] 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.

Meurice et al. [Meurice 2016] created a static analysis approach that allows developers to identify and analyze database access locations from Java systems using Hibernate, JDBC and JPA. Their analysis relied on traversing ASTs and reaching the APIs used to call the databases and parsing the SQL queries. The queries being parsed are complex and concatenated in the source code. While their approach is beneficial when extracting comprehensive query information, it may be more effort than necessary for dependency analysis, where developers often only need to extract specific details like table names and columns.

### 2.1.2 Categorization of the Tools

We classify the presented work in different categories (summarized in Table 2.1[1]):

**Dependency type:** Many of the papers are considering dependencies existing in *cross-language* applications [Cossette 2010,Grichi 2020,Kempf 2008,Mayer 2012, Pfeiffer 2012, Shatnawi 2019, Polychniatis 2013, Soto-Valero 2021], like between Java source code and an XML configuration file, *multi-tier* [Nguyen 2012] like between a client application and a server one, *callbacks* [Cao 2015] from external libraries, or *generated files* [Nguyen 2012] 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* [Yu 2001,Cao 2015,Cossette 2010,Grichi 2020, Kempf 2008, Mayer 2012, Pfeiffer 2011, Shatnawi 2019, Polychniatis 2013, Soto-Valero 2021] and *dynamic* [Hecht 2018, Nguyen 2005, Shatnawi 2019] analyses could be used by the tools;

**Matching strategy:** There are two different strategies used: *Object matching* [Yu 2001, Cao 2015,Cossette 2010,Hecht 2018,Kempf 2008,Mayer 2012,Nguyen 2012, Pfeiffer 2011, Soto-Valero 2021] works on a model of the application and looks for specific objects in the model, *lexical matching* [Grichi 2020, Shatnawi 2019,Polychniatis 2013] works directly on the source text (Java, XML, ...);

**Engine:** Finally, some approaches are based on *rules* [Yu 2001, Cossette 2010, Grichi 2020,Hecht 2018,Kempf 2008,Mayer 2012,Nguyen 2012,Pfeiffer 2012] that should be defined by the users of such tools to identify dependencies, while others can *automatically* [Cao 2015,Shatnawi 2019,Polychniatis 2013, Soto-Valero 2021] discover them because rules are predefined by their implementors.

Table 2.1: Summary of external dependencies found in literature

| Name | Dependency Type | Analysis | Matching strategy | Engine | Supported agents |
|---|---|---|---|---|---|
| [Yu 2001] | Data Dependencies | Static | Object | Rule | Specific |
| [Cao 2015] | Callbacks | Static | Object | Automatic | Specific |
| [Cossette 2010] | Cross-language | Static | Object | Rule | Generic |
| [Grichi 2020] | Cross-language | Static | Lexical | Rule | Specific |
| [Hecht 2018] | all[a] | Dynamic | Object | Rule | Specific |
| [Kempf 2008] | Cross-language | Static | Object | Rule | Specific |
| [Mayer 2012] | Cross-language | Static | Object | Rule | Generic |
| [Nguyen 2012] | M-tiers[b], Gen.[c] | Dynamic | Object | Rule | Specific |
| [Pfeiffer 2011] | Cross-language | Static | Object | Rule | Generic |
| [Shatnawi 2019] | Cross-language | Static/Dyn. | Lexical | Automatic | Specific |
| [Polychniatis 2013] | Cross-language | Static | Lexical | Automatic | Generic |
| [Soto-Valero 2021] | Cross-language | Static | Object | Automatic | Specific |

[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. [Hecht 2018]: cross-language, multi-tiers, callbacks, and generated files). Also, two papers, Pfeiffer and Wąsowski [Pfeiffer 2011], and Soto-Valero et al. [Soto-Valero 2021], consider dependency correctness by identifying when a dependency should not exist. Additionally, we found that some approaches are generic (not specific to frameworks) [Polychniatis 2013, Pfeiffer 2011, Mayer 2012, Cossette 2010], while others work only for specific languages and frameworks [Yu 2001, Shatnawi 2019, Nguyen 2012, Cao 2015, Hecht 2018, Grichi 2020, Soto-Valero 2021, Kempf 2008].

### 2.1.3 Correctness of External Dependencies

In their work, other researchers noticed the existence of incorrect external dependencies. They all note the importance of detecting these incorrect external dependencies to improve the quality of the software projects.

Pfeiffer and Wąsowski [Pfeiffer 2012] created TexMo, as a prototype of multi-language development environment, to detect cross-languages links and help generating visualization, navigation and so on. They were the first ones to introduce the idea of correct dependencies between components where cross-languages links are established. They introduced the idea of multiplicities "TexMo relations are

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

always many-to-one relations between references and keys".

Mayer and Schroeder [Mayer 2012] created a generic approach to understanding, analyzing, and refactoring cross-language code by directly specifying and exploiting semantic links in multi-language programs using QVT-Relations (QVT-R). Their approach considered searching for incorrect cross-language links where artifacts can be "missing", and errors generated on execution time.

Soto-Valero et al. [Soto-Valero 2021] discussed "bloated dependencies", which are software libraries or packages included but not used at all by the project. They suggested a new automatic approach to identify third-party Java dependencies statically in Maven projects. Their methodology allows removing unneeded classes, repackaging used classes into a different dependency, and regenerating the XML configuration file to refer to the new dependencies.

Jafari et al. [Jafari 2021] studied the dependency smells generated after using the npm ecosystem to install libraries. They applied their study over JavaScript projects and identified 7 dependency smells: Pinned, URL, Restrictive Constraint, Permissive Constraint, No Package-Lock, Unused and Missing dependency similar to "bloated dependency" defined previously. According to them, those dependency smells can cause security problems, bugs, dependency breakages, and other maintenance issues.

Cao et al. [Cao 2022] studied the impact of dependency smells for Python projects. They identified 3 different types of dependency smells: "missing dependency", "bloated dependency" and "Version Constraint Dependency". Version Constraint Dependency is related to referring the same library but with different versions, multiple times in the same project. In this case, developers should put more efforts into maintaining numerous version constraints to ensure the consistency of dependency declarations in the project.

In summary, Table 2.2 presents all the types of incorrect external dependencies identified in the literature. They are organized into three columns, with each column grouping together types of incorrect dependencies that share the same type under different names.

Table 2.2: Summary of incorrect dependencies found in literature

| | | | |
|---|---|---|---|
| Mayer/Schroeder [Mayer 2012] | missing | | |
| Jafari et al. [Jafari 2021] | missing | unused | |
| Cao et al. [Cao 2022] | missing | bloated | vers. const. |
| Pfeiffer/Wąsowski [Pfeiffer 2011] | broken | | |
| Soto-Valero et al. [Soto-Valero 2021] | | bloated | |

## 2.2 Pattern Matching

In the context of pattern matching, there are two primary approaches: pattern matching with graph query languages, and pattern matching in General Purpose Languages (GPLs). Both of these approaches propose to the developer a set of possibilities when it comes to declaring a pattern, and how the result is returned. In this section, we review the work related to declarative matching over data in graphs and in GPLs.

### 2.2.1 Pattern Matching for Graphs

According to Krause et al. [Krause 2016]: In graph pattern matching, the task is to find all matches between a set of pattern variables and the nodes of a target graph that satisfy a set of conditions. A pattern consists of a pattern graph, a set of predicates, and an optional nested formula. The predicates and the formula are defined over a set of pattern variables that represent the pattern nodes, i.e., the nodes of the pattern graph. A match of a pattern with respect to a given target graph is a map of the pattern variables to nodes of the target graph (called the target nodes of the match), such that (i) each pattern node is matched to a target node of the same type, (ii) each pattern edge is matched to a target edge of the same type, (iii) all predicates are satisfied, and (iv) the logical formula is satisfied. Figure 2.1 illustrates a basic graph pattern derived from the authors work, showing how nodes and edges encapsulate the descriptions being searched for within a target graph. To note also that additional description using formulas and expressions can also be used to be able to specify a more flexible description.

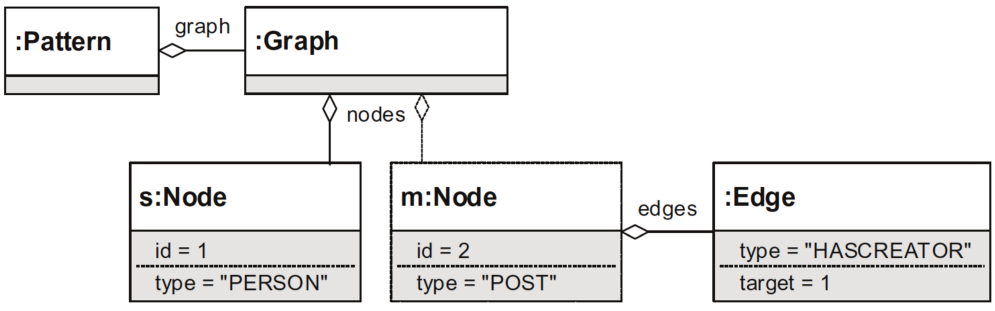



Figure 2.1: Example of Graph Pattern

A vast range of data from many domains can be represented by graphs [Libkin 2016] where Resource Description Framework (RDF) [Consortium 2014] graph and Property Graph (PG) [Angles 2017] are commonly used to represent this data [Deutsch 2022].

An RDF graph is equivalent to a set of triples, of node-labeled edge-node, to represent the data [Thakkar 2017,Deutsch 2022], and SPARQL [Consortium 2013] is the language that is capable of handling large-scale analytical operations over RDF graphs [Thakkar 2017]. It uses a syntax with patterns expressed in triple form (subject-predicate-object) [Krause 2016] using a combination of variables and specific resources or literals. The objects and subjects match nodes, and the predicates match edges.

```
1 SELECT DISTINCT ?name
2 WHERE {
3   ?instance athlete ?athlete .
4   ?instance medal Gold .
5   ?athlete label ?name .
6 }
```

Listing 2.1: SPARQL example

Listing 2.1 is a simple example of a SPARQL query to extract the names of every gold medal (excluding duplicates) from an RDF graph (see Figure 2.2). Line 1 is used to retrieve distinct (duplicates removed) names from the data returned by the query. The patterns are expressed in the `where` clause between lines 3 and 5. Names suffixed by "`?`" are variables that can match any node in the graph. Line 3 matches `?instance` and `?athlete` to 2 nodes related by an `athlete` edge. Line 4 requires that the `?instance` also be linked to a `Gold` node by the `medal` edge. Line 5 matches the `?name` variable to the node related to the `?athlete` node through a `label` edge. This `?name` is returned by the query (line 1).

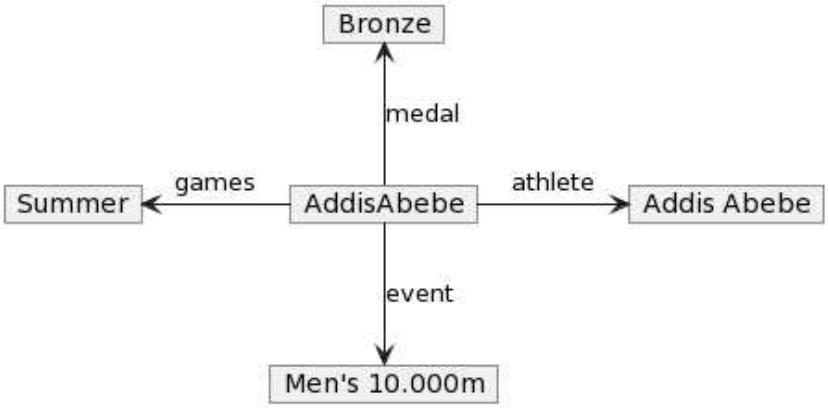



Figure 2.2: Example of a simple RDF graph

**Key points:**

- SPARQL allows expressing the structure of the data looked for without worrying about how deep in the whole graph of data it is or how complex it is.
- In lines 3 and 5, `?athlete` has been reused to match the same value multiple times in the same pattern.
- Additionally, patterns are expressed in a declarative way allowing developers to add as many patterns as they need.

A PG models the data as a mixed multi-graph, where both nodes and edges can be labeled and attributed [Deutsch 2022]. Various querying languages can be used for PGs like Gremlin APIs [Rodriguez 2015], and some declarative graph query languages like Cypher by Neo4j [Francis 2018], GSQL by TigerGraph [Deutsch 2020], PGQL by Oracle [Șimonca 2022], or Graph Pattern Matching Language (GPML) which is able to run CRUD operations [Deutsch 2022]. In addition, there are some experimental solutions in both industry and academia, like G-CORE [Angles 2018].

Gremlin is considered a graph traversal language, allowing the exploration of complex relationships through various nodes in the graph using traversals like Recursive Traversal [Rodriguez 2015].

```
1 g.V().has("name","marko").
2 repeat(out()).times(5).
3 values("name")
```

Listing 2.2: Gremlin recursive example

A Gremlin traversal is a sequential movement through the steps, which are represented by nodes and edges in a data graph. It initiates from all nodes in the graph and traverses the path until the endpoint predefined by the developer is reached successfully among the graph. Listing 2.2 is a recursive traversal expressed in Gremlin that consists of selecting 5 persons named *marko*. In line 1 `g.V()` is the starting point of the traversal where `g` refers to the target graph of the match and `V()` denotes all nodes selection of `g`. Then, `.has("name", "marko")` filters the nodes to only those for which the property `name` is equal to *marko*. In line 2, `repeat(out())` will repeat the previous match to get all possible results. Then `.times(5)` limits the repetition to 5 occurrences. And finally for line 3, `values("name")` property of value name is retrieved after moving recursively five steps forward.

**Key points:**

- The main advantage of this matching technique is that it helps the developers specify a path to be traversed, with unlimited numbers of nodes and edges.

It is adopted by many other languages for matching PG graphs like Cypher and PGQL.

- It is also possible to specify repeated searches to return all possible matches.
- Repeated searches risk falling into an endless cycle in the presence of cyclic relationships. Therefore, they can be limited to a maximum number of repetitions using `times()`.

### 2.2.2 Pattern Matching in GPLs

Pattern matching is one of the main features of functional programming languages [Ryu 2010] like Haskell[2]. With the evolution of object-oriented programming, pattern matching has increasingly found its way into this paradigm [Kohn 2020]. General Purpose Languages now have pattern matching capabilities like Java, which implemented pattern matching in the Amber[3] project; Python[4]; or Rust[5].

Such languages involve matching objects based on their types and/or field values, and many of them introduced the conditional match like "`switch` *subject* `case` *pattern*" in Java or Python.

```
1  record Point(int x, int y) { }
2  enum Color { RED, GREEN, BLUE; }
3  ...
4  String typ;
5  switch (obj) {
6    case null    -> typ="Null pointer";
7    case String s -> typ="A String";
8    case Color c  -> typ="A Color";
9    case Point p  -> typ="A Point";
10   case int[] ia -> typ="An Array of int";
11   default       -> typ="Something else";
12 }
```

Listing 2.3: Java Pattern Matching Example Java 21

Listing 2.3 exposes a pattern matching example in Java 21, where object matching is applied using the `switch case` statement. Lines 1 and 2, define the data structures that are used in the matching. Line 4 defines a variable that will hold a description of the type of `obj` (line 5). The switch statement performs the matching by looking for the first `case` that will match `obj` in lines 6 to 11.

**Key points:**

[2]Haskell https://www.haskell.org/tutorial/patterns.html
[3]Amber project https://openjdk.org/projects/amber/
[4]Python https://peps.python.org/pep-0000/
[5]Rust https://doc.rust-lang.org/book/ch18-00-patterns.html

- Object pattern matching helps define structural objects to be matched.
- Some languages allow to match an object not only on its structure, but also on the value an attribute should have.

We found several libraries that complement programming languages by introducing pattern matching capabilities. Tom [Pierre-Etienne 2003] [Balland 2007] and Rascal [Klint 2011] integrate with Java, Kiama [Sloane 2009] integrates with Scala, and pyZtrategic [Rodrigues 2024] integrates with Python. Tom and Rascal cover features that were not covered natively by Java like *Path traversal*, *Recursive traversal*, *Object matching* and *List pattern*. Kiama, Rascal and pyZtrategic enable transformation through the definition of strategies. These strategies employ pattern matching to identify the terms requiring transformation. Strategy execution in Kiama can proceed in different directions either top-down or bottom-up. This capability opens up the possibility of traversal in various directions, but it still requires additional investigation concerning its relevance in pattern matching.

### 2.2.3 Pattern Matching Language Features

We now consider what features have been proposed in different pattern matching tools/languages. This will be an inspiration for designing our own pattern matching language.

Klint et al. [Klint 2011] state that a rich pattern language should provide string matching based on regular expressions, matching of abstract patterns, and matching of concrete syntax patterns. To reach a stage where developers are able to express any pattern compatible with the shape of the object they are looking for, a list of "features" must be provided by the language and applied using multiple operators or methods.

Previously in the literature, some authors have listed features supported by Rascal or Python [Klint 2011, Kohn 2020].

We first consider features found in graph pattern matching. *Graph matching* is famous for specifying patterns similarly to database SQL queries.

***Declarative patterns*** help the developers define specific patterns that look like the results of matching, without caring about how these patterns will be matched. Using this paradigm leads to reduced development's time, increased maintainability, quick learning for pattern expression, and live preview changes without impacting the whole analysis process [Imbugwa 2021]. We oppose it to *Imperative paradigm* which consists of defining the computational steps to complete the matching process.

***Path traversal*** refers to visiting elements (i.e. nodes and edges) in a graph in some algorithmic fashion [Rodriguez 2012]. It helps the developers traverse nested structures while matching patterns.

***Recursive traversal*** is needed to apply recursive search over deep structures, especially when developers ignore the depth of elements being searched for.

***Repeated search*** is a feature related to the number of returned matches, where some languages can repeat the search to find all possible matches found, while others stop searching after finding the first match. A more flexible solution offers to specify the maximum number of matches that are expected, allowing to repeat the search without incurring the risk of infinite loops.

We now consider additional features of object matching. Some of these features would not make sense in graph patterns (such as *object matching*). We also add here some features that are inspired by pattern matching in functional languages such as non-linear patterns:

***Object matching*** is dedicated to match objects based on their types and properties, which can be methods with return values or instance variables.

***Literals*** (strings, numbers, integers) simply match themselves. They can be used to specify the value of an object's property.

***Non-Linear pattern*** (sometimes called *unification*) allows the developers to use the same variable multiple times that should always match the same value in the pattern.

**Wildcards** represent a placeholder, an anonymous property that can be matched and is not used afterwards.

***Nested pattern*** allows sub-patterns definition inside a pattern.

***List pattern*** supports the matching of a sequence of patterns taking into consideration their order. For example specifying that a pattern must be matched at the end of a list or in the middle.

***Logical matcher*** allows the possibility of combining multiple patterns in a boolean expression.

***Negation*** may allow to express a simpler pattern when searching for bindings that do not conform to a particular criteria.

### 2.2.4 Some Existing Object Pattern Matching Languages

We studied existing pattern matching languages to understand what features they offer. The goal of our matching language will be to offer all these features. We considered the top OO languages used in 2022 according to github[6]: C#, C++, Java,

[6]https://octoverse.github.com/2022/top-programming-languages, consulted on may 2nd, 2024

| Characteristics | C# | Java | Pharo (4) | Python | Ruby | Rust | Scala |
|---|---|---|---|---|---|---|---|
| *Paradigm* | D&I | D&I | I | D&I | D&I | D&I | D |
| *Path traversal* | | | | x | | x | x |
| *Recursive traversal* | | | x | | | | |
| *Repeated search* | | | x | | | | |
| *Object matching* | x | (1) | (5) | x | x | x | x |
| *Wildcard* | x | x | x | x | x | x | x |
| *Nested pattern* | x | x | x | x | x | x | x |
| *List pattern* | x | (2) | x | | x | | |
| *Literals* | x | x | x | x | x | x | x |
| *Logical matcher* | x | x | | x | x | x | x |
| *Negation* | x | (3) | | x | x | | |
| *Non-Linear pattern* | x | x | x | | | | |

Table 2.3: Pattern matching characteristics in Object Oriented programming languages.
(1) Matching Types, (2) Planned, (3) For types only, (4) RBParseTreeSearcher, (5) Only Pharo AST nodes.

Javascript, Python, Ruby, Typescript (three more languages are not OO: C, PHP, Shell). We added Rust and Scala that are well known for their pattern matching capabilities. And we added a library in Pharo (RBParseTreeSearcher) because this is the language we are working with.

Table 2.3 shows the features supported by different OO languages that have native pattern matching capabilities.

Without surprise, *Object matching* is well supported. *Nested pattern* and *Wildcard* are also two features that are common. On the other hand, it shows that features like *Path traversal*, *Recursive traversal*, *Repeated search*, *List pattern* and *Non-Linear pattern* are not universally supported by object matchers.

We are interested in creating an object matching language that could be used to match objects in models, taking into consideration that features like *Repeated search*, *Non-Linear pattern*, *List pattern*, *Recursive traversal* and *Path traversal* are also important for such matching languages, in order to provide developers with the possibility to create patterns in a flexible way, allowing deep matching for deeply recursive models.

## 2.3 Conclusion

In this chapter, we presented previous work related to external dependencies detection and pattern matching. For external dependencies, we reviewed earlier research across different categories, examined approaches that assess their correctness, and

discussed the types of incorrect dependencies identified by other authors. Regarding pattern matching, we covered prior work on graph matching and object matching in GPLs, highlighting the features supported by each approach.

CHAPTER 3

# External Dependencies Detection

## Contents

## 3.1 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 [Kaur 2015]. 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 component to rely on another component in order to fully operate as expected. Developers are more likely to miss "hidden dependencies" than explicit dependencies and thus introduce bugs into software [Vanciu 2010].

Yu and Rajlich [Yu 2001] talked about hidden dependencies as a relationship between two seemingly independent components. Such dependencies are hard to maintain and are considered by the authors as design faults, because they violate the rule: "if class A is unaware of the existence of class B, it is also unconcerned about any change to B". They give example of 2 components that affect the same data flow of a software. They both ignore the existence of each other, but both affect the same data flow through another third component, that makes calls to them. According to the authors, this results in hidden dependencies between the 2 components since they are managing the same data flow without knowing each other.

We are interested in dependencies that are hidden, and additionally introduced by external tools or agents like Android[1], GWT[2], J2EE[3], ODBC[4], 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 components that is created through an external agent. These external dependencies are established in between different programming languages or separate programs or various files. But "external" refers to the fact that the dependency is introduced by an external agent, not to the fact that the components 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 components 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 external agents allow different projects to collaborate, for example in client/server (multi-tier) applications.

In this chapter, we explore the external dependencies. Our objective is to equip

[1] Android android.com

[2] Google Web Toolkit gwtproject.org

[3] Java 2 Platform, Enterprise Edition oracle.com/java/technologies/appmodel.html

[4] Open Database Connectivity wikipedia.org/wiki/Open_Database_Connectivity

developers with a tool that identifies all external dependencies within their programs. Detecting these dependencies is not a simple task, as they often involve components spread across different places, potentially running on separate machines, and sometimes written in various languages at different positions or in different files in the same program. Understanding external dependencies is vital for developers working with unfamiliar or legacy programs. It provides insights into how the various components are interconnected and function together, facilitating a deeper understanding of the program's structure.

We found that there is no clear, unique definition in the literature of what an external dependency is. That's why we will go over the following points that outline our strategy in this chapter to detect external dependencies:

1. We list in this chapter all the categories of external dependencies we found in previous work [Yu 2001] and [Hecht 2018]. We add one more category that matches our general definition of external dependencies;

2. We list the commonalities between all the categories that we found and propose a single approach for external dependency detection;

3. We validate our approach with a tool and experiment it on 4 projects developed using 4 different external agents.

The rest of this chapter is organized as follows: Section 3.2 provides definitions of the domain and enumerates the various categories of external dependencies. In Section 3.3, we explore, for several external agents, how external dependencies are introduced. In Section 3.4, we introduce our approach and how we decompose our problem. This is followed by Section 3.5 where we present our tool, also we show how it was implemented and how it works. We experiment our tool in Section 3.6, where we present the validation process through 4 experiments, accompanied by a discussion of the results, followed by threats to validity in Section 3.7. Finally, we conclude the chapter with a summary and future directions in Section 3.8.

## 3.2 Structuring the Domain

In this section, we will further structure the domain by clarifying external dependencies and detailing their categories as derived from the literature.

### 3.2.1 Definitions

We define *external dependencies* as dependencies between two entities: one entity is defined, and the other refers the defined entity. We call the two entities involved in an external dependency, resource and reference. A *resource entity* is a software entity that exists independently within a program and can be referenced. Examples include database tables, classes, methods, attributes, or a GUI button defined in

XML. A *reference entity* is a location in source code or documentation that refers to a resource entity, such as a SQL query referencing a database table, a client program calling a method in a server program, or a program setting an action handler on a GUI button.

In the context of an external dependency, both entities are encapsulated in *containers*, they are part of the developed program. For instance in a client/server application, a client method may invoke an API method developed on the server side. The dependency exists between the statement on the client side invoking the API method, and the latter that is defined on the server side. The containers are both classes encapsulating the methods, whereas the resource entity is the API method defined on the server side, and the reference entity is the statement invoking this API.

Various *external agents* lead to the presence of these external dependencies. Examples include external agents that work with multiple programming languages, like Android[5], GWT[6], J2EE[7], Hibernate[8] etc., where each language answers a different need such as building user interfaces, handling logic and querying databases.

### 3.2.2 Categorization

In our literature review, we discovered that external dependencies cannot be confined to a single category, a finding particularly highlighted in the work of Hecht et al. [Hecht 2018]. We also observed that while some studies focused on detecting a single category of external dependencies, others addressed multiple categories, as summarized in Table 2.1 in Chapter 2. Below, we provide a list of the categories of external dependencies identified in the literature along with an additional category that we propose as another potential classification:

**Cross-language links** are those between components 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 XML appears in the Java code using annotations such as `@UiField` and `@UiHandler`, that are essential in GWT to be used to refer to the XML elements defined in the XML UI files (see Section 3.3.1).

If any change affects the XML component, for example renaming a referred XML element using one of these Java annotations, the developer must apply in parallel the same change on the dependent Java component, 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 2.1). Such dependencies may be difficult for the developers to detect as they imply a

[5]Android https://www.android.com
[6]Google Web Toolkit gwtproject.org
[7]Java 2 Platform, Enterprise Edition oracle.com/java/technologies/appmodel.html
[8]Hibernate https://hibernate.org

good understanding of the external agent used [Mushtaq 2017]. In another 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. This is not limited to GUI, other Cross-language dependencies can be found for example 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 store 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 [Neubauer 2005]. 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 their methods and make a successful call to the server tier. Again, such calls between tiers depend on the external agent used, yet changes to one tier could require corresponding 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. [Kempf 2008] mention it for the Android framework. However, we will exclude this category from our study because they are easier to detect as the relationship is directly stated within the code. . Our focus is on more external dependencies that we ignore which reference is targeting which resource and vice versa.

**File generation dependencies** appear when an external agent 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 [Hecht 2018]. 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 components (Cross-Language dependencies). In the context of J2EE, Hecht et al. use dynamic analysis to detect these File generation dependencies.

**Documentation dependencies** is a new category that we propose. It exists when the documentation refers to the source code. For example, the JavaDoc has special annotations to refer to classes, methods or their parameters. Some refactoring is able to detect and modify the comments when a component 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.

## 3.3 External Dependencies in Multiple External Agents

Numerous external agents support external dependencies. Throughout our research, we collaborated with several companies whose programs are built using a diverse range of these external agents. These companies required comprehensive analysis and engineering work on their programs, which motivated us to focus on these external agents for our studies. Additionally, our team is using Pharo in order to develop tools to enhance the analysis and engineering. Recognizing the potential of Pharo, we have also undertaken an analysis of Pharo's classes comments. In this section, we will introduce these external agents and outline the categories of external dependencies available for each one.

### 3.3.1 Google Web Toolkit (GWT)

Google Web Toolkit (GWT[9]), is a framework to develop web applications using a combination of Java and XML. In GWT Java components (class attributes or methods) are "linked" to widgets described in XML files through the annotations `@UiField` and `@UiHandler`. For example, Listing 3.1 shows how an attribute on line 3 is referring to a Window defined in Listing 3.2 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`).

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

[9]GWT https://www.gwtproject.org

```
9    }
10 }
```

Listing 3.1: GWT Java

```
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>
```

Listing 3.2: GWT XML

GWT external dependencies are classified as cross-languages links, involving dependencies between two different languages: Java and XML.

### 3.3.2 Remote Method Invocation (RMI)

Remote Method Invocation (RMI[10]) in Java 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 below example). 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 3.3). An instance of this class is registered in the RMI registry (lines 3 and 4 in Listing 3.3).

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

Listing 3.3: RMI server code

On the client side, an instance of this interface is obtained from the RMI registry (Listing 3.4) and calls to its methods will be forwarded to the server application.

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

Listing 3.4: RMI client code

[10]RMI https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/index.html

RMI external dependencies are categorized as multi-tiers, having calls between different tiers: the client and the server.

### 3.3.3 Hibernate

Hibernate[11] is an ORM framework to help developers map Java classes to database tables. It enables the transfer of data between the database and the program. Queries in the Java program are expressed in HQL (Hibernate Query Language) which is very close to SQL. The company which asked us for the analysis of their Hibernate project, uses an older version of Hibernate (3.1.3) which uses XML files to declare the mapping, so we consider it in this thesis. More recent versions of Hibernate use Java annotations instead.

To study the external dependencies of this framework, it is essential to analyze all the existing mappings. The HQL queries do not directly reference the database tables; instead, they use new names. Therefore, we must identify those mappings between the database, `.hbm.xml` files, Java classes and HQL queries. Since we have different components here, we recognize the existence of external dependencies which we classify into 4 groups as shown in Figure 3.1.

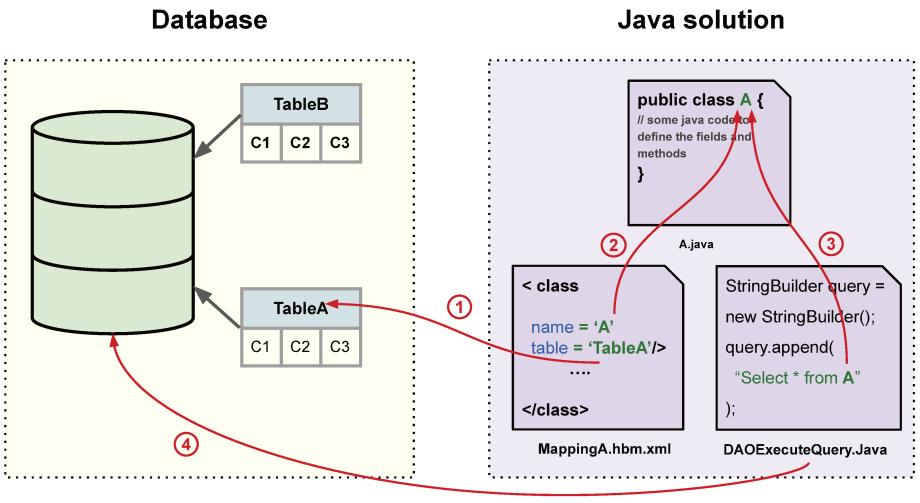



Figure 3.1: Different external dependencies groups

**Group 1:** The first group of external dependencies exist between the database, where entities like tables are defined, and the program where those entities are referred through the mapping files (`.hbm.xml`). These files declare the mapping between Java classes and database tables, and between Java class attributes and database columns. Consider for example the database

[11] Hibernate https://hibernate.org/orm/

table `tb_article` created in Listing 3.5 (line 1). In the mapping file (Listing 3.6) a mapping to this table is created (line 2). Here the database table is the *resource entity* and the *reference entity* is the part `table="tb_article"`. Note that there is another external dependency to the database column `id` in line 4 of Listing 3.6, but we will not consider it in our examples.

```
1  CREATE TABLE tb_article (
2    id SERIAL PRIMARY KEY,
3    ...
4
```

Listing 3.5: Database Table and Column Template

**Group 2:** The second group of external dependencies exists between some Java classes defined in the program, and the mapping files referring to these classes. In the same example, line 2 of Listing 3.6 has a reference to the Java persistence class `ArticleInfo` (defined in Listing 3.7). This class is the *resource entity*. Note that there is no direct external dependency between the Java class and the database table. Note also that more recent versions of Hibernate create such a direct external dependency because they put a Java annotation on the class referring to the database table. This clearly shows that external dependencies are fully dependent on the external agent used.

```
1  <hibernate-mapping>
2    <class name="ArticleInfo" table="tb_article">
3    <id name="id" type="java.lang.Integer">
4      <column name="id" />
5      <generator class="native" />
6    </id>
7    </class>
8    ...
9
```

Listing 3.6: Hibernate Mapping file

**Group 3:** The third group of external dependencies exist between the HQL queries and the mapped Java classes. For example, in Listing 3.8, line 3, `FROM ArticleInfo` creates an external dependency to the Java class defined in listing 3.7.

```
1  public class ArticleInfo {
2    private Integer id;
3    public Integer getId() {
4      return id;
5    }
6    public void setId(Integer id) {
7      this.id = id;
8    }
9    ...
10 }
```

Listing 3.7: Java persistence Class

**Group 4:** Finally, this group represents the external dependencies existing between the program and the database through all the previous groups. In other words, the external dependencies of this group exist between HQL queries and the database tables, without referring directly to those tables, but they are established through the Java persistence classes and the mapping files. They are needed because, with the older version of Hibernate (using mapping files), each of the three preceding ones alone cannot guarantee the correct execution of the program. For example, the HQL query in Listing 3.8 is referring to a persistence class **ArticleInfo** (line 2). This is a group 3 dependency that can be verified. But without the mappings (group 1 and 2 dependencies), there is no way to guarantee that the query can run.

```
1 StringBuilder hql = new StringBuilder("SELECT *");
2 hql.append("FROM ArticleInfo a");
3 hql.append("WHERE a.id = :articleID");
4
```

Listing 3.8: Embeded SQL Query Example

Note that group 4 dependencies are similar to what we would have if developers were using JDBC, directly referring to the database table in SQL (not HQL) queries without using Hibernate.

In summary, we identified various categories of external dependencies, ranging from multi-tiers (like group 1 and group 4) to cross-languages links (like group 2 and group 3). Each group was identified based on components, where an entity is defined in one component and referenced in another.

### 3.3.4 Pharo Comments

In this section, we will present the external dependencies that exist between Markdown-formatted comments and Pharo classes.

Comments are commonly used to improve program comprehension, helping with tasks such as usage and maintenance of the classes and their methods. In Pharo, each class has a comment section that can be filled with necessary information using Markdown. The external dependencies exist between the references to the classes from the comments. According to our classification in Section 3.2.2, these external dependencies fall under the "Documentation" category.

Figure 3.2 is an example of `SpListPresenter` class comment in Pharo, in Edit mode. To refer to the classes in Pharo from the comments, a developer can use the backticks (`) like `` `SpListPresenter` ``, `` `SpAbstractListPresenter` ``, `` `SpComponentListPresenter` `` and `` `SpTablePresenter` ``.

The display of this comment is shown in figure 3.3, where the class names are displayed in blue and clickable, which can help developers navigate to the corresponding class in Pharo. We identify in this example 4 external dependencies, where each one is a reference from the comment to a class defined in Pharo.

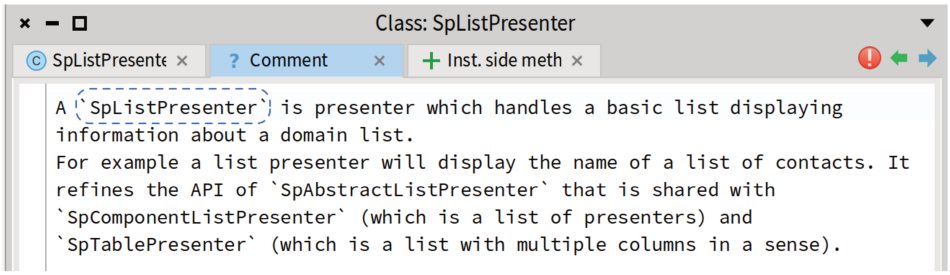



Figure 3.2: Example of Edit mode for 'SpListPresenter' class comment in Pharo

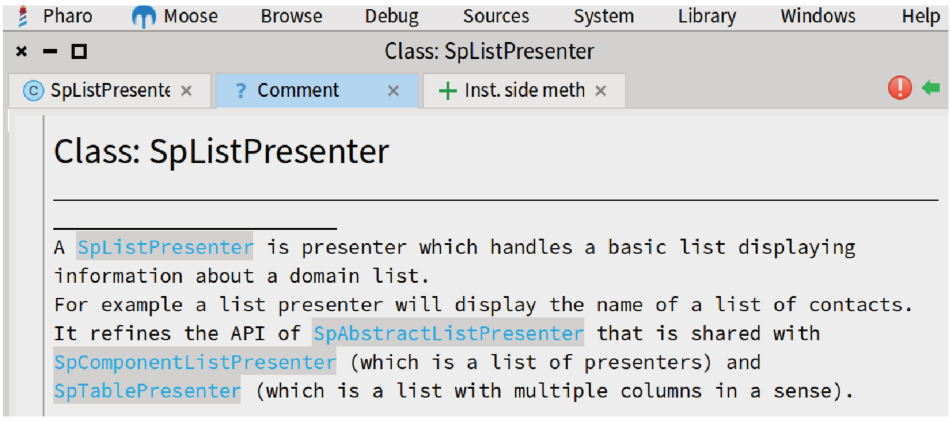



Figure 3.3: Example of View mode for 'SpListPresenter' class comment in Pharo

## 3.4 External Dependencies Detection

Various detectors in the literature have been implemented to detect several categories of external dependencies. The majority of them were focusing on specific category of external dependencies [Cao 2015, Cossette 2010, Kempf 2008, Mayer 2012, Pfeiffer 2011, Shatnawi 2019, Polychniatis 2013, Soto-Valero 2021]. The others covered additional categories [Hecht 2018, Nguyen 2012]. Additionally, not all the work implemented in the literature was intended to cover additional external agents [Nguyen 2012, Cao 2015, Soto-Valero 2021, Kempf 2008, Meurice 2016]. But some of them did [Shatnawi 2019,Polychniatis 2013,Grichi 2020, Pfeiffer 2011, Mayer 2012, Cossette 2010].

In our work, we want to implement an approach that is: (i) easily extendible to cover external agents, and (ii) generic for all categories we introduced previously in 3.2.2. In this section, we present the list of considerations which we want our approach to fulfill in order to respond to both needs. Then we decompose the problem in order to accelerate the process of finding external dependencies. Finally, we present the concept of reusable patterns, which allows a faster and easier extension of the tool to be created.

### 3.4.1 Key Considerations

In order to respond to our needs, we need to take into consideration several points, some of them were considered by previous researchers in their work:

**Dependency categories:** In our case, there is 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 documentations.

**Searching strategy:** We need a way to search in source files [Di Grazia 2023] in order to find the correspondent entities and relate them. For that we will proceed by using pattern matching, as it will help us describe patterns and apply the match over the source code, and find what we are seeking for.

**Matching strategy:** We have two different matching strategies: Lexical matching and Object matching.

In our case, we will use both strategies, taking advantage of whatever tools are already available for a given language, and how complex the position of an entity can be.

**Engine:** We saw in Section 2.1.1 of Chapter 2 that some publications [Shatnawi 2019, Polychniatis 2013, Soto-Valero 2021] 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 external agent except Polychniatis et al. [Polychniatis 2013] which may produce many false positives.

We prefer to implement a rule-based engine where we specify for each external agent where to look for dependencies.

**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. [Hecht 2018]. That is a limitation of our approach for this dependency type.

Following these key considerations, we decided to use Pharo[12] to implement our solution. One of the primary reasons for selecting Pharo is the Moose platform. Moose[13] is a free and open-source platform designed for software and data analysis. It provides a rich set of tools (like Moose importer and Moose query) and parsers that enable us to import and parse models, using metamodels (like FamixJava[14] to represent the structure and relationships of a Java program) facilitating software analysis and re-engineering tasks. This capability is crucial for our solution, specially when applying object matching strategy over models to find the resource and reference entities.

For object matching, we created and used MoTion[15], a new declarative pattern matching language developed in Pharo. MoTion is detailed in Chapter 4.

Additionally, like many languages, Pharo has a regular expression library and adds another layer of versatility to our approach. Regular expressions are a powerful tool for text processing, allowing us to perform lexical analysis across various programming languages, documentation, and tiers. This flexibility is essential for managing external dependencies and ensuring that our solution can adapt to different software environments.

### 3.4.2 Decomposing the Problem

To simplify the implementation of our solution, we decompose the dependency identification problem into two parts:

First, we look for *Containers*, software components that may contain a reference entity or a resource entity. Containers are usually large components, such

[12]Pharo https://pharo.org/

[13]Moose https://modularmoose.org

[14]FamixJava https://modularmoose.org/moose-wiki/Users/famix-java/famix-java

[15]MoTion https://github.com/alesshosry/MoTion

as a class or an entire file. In the literature, many approaches search for some kind of "container" [Grichi 2020, Kempf 2008, Mayer 2012, Mayer 2014, Pfeiffer 2011, Shen 2021] by declaring specific search patterns or by discovering them with some heuristics. We will adopt the approach of Kempf et al. [Kempf 2008] who propose first filtering containers that may contain such dependencies to accelerate the research. We limit a container to be written in any single language, like a Java container, an XML container, etc. 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. There are two kinds of containers: Reference containers and Resource containers. In the GWT example (listings 3.1 and 3.2), the Java class containing the annotation is a *Reference container* and the *Resource container* will be an entire XML file describing the GUI. In figure 3.4 explain how rules are applied to collect the containers and entities.

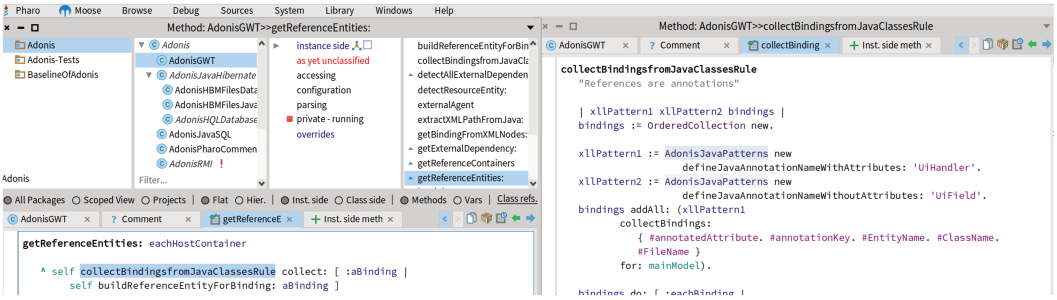



Figure 3.4: Adonis Rules and patterns

To find Resource containers (XML files) we used one rule: 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 `ui:field` 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. Two patterns are used inside the rule in order to find the annotations of `UiField` and `UiHandler` and extract the references to the XML file. For the reference entities, we use one rule: It looks for the argument value of all `@UiHandler` or `@UiField` annotations in the container.

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.

Now that we decomposed the problem, we can start locating containers and entities,by relying on pattern matching (lexical or object).

### 3.4.3 Reusable Patterns

Having a solution with no limitation for external agents, allows us to have reusable patterns, specially for external agents who share the same programming languages. For example, searching for an XML element according to one of its attribute names can be done whether this is an XML configuration file, an XML layout file or even an HTML file. It would therefore be applicable to external dependencies detection in J2EE, GWT, Android, Angular etc...

For a given language, 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, imported in Pharo using FamixJava MetaModel: (i) we need a way to precise the type of the object that we are trying to match, which is `FamixJavaModel`. After precising the type, (ii) we need to restrict our research; since we are looking for annotations in this object, we have to use the property `allAnnotationTypes` that return a collection of objects of type `FamixJavaAnnotationType`. In later steps we start restricting this property to be able find our match. After that, (iii) we need to precise that each object of this collection may have a property `name` equal to `UiField` or `UiHandler`, and another property `value` to extract in order to match it later with the description of the XML elements defined in the corresponding XML file. Such patterns can be encapsulated in methods with input parameters, which values can be set differently each time depending on the external agent in question.

Finally, the strategies presented in this section, will help the developers to enable a flexible extension of Adonis to support new external agents:

**Decomposing the problem** allows the generation of common methods like “findResourceContainers” and “findReferenceEntityFor(referenceEntity)” for each external agent, and inside, rules can be set to call specific patterns to locate those containers and entities.

**Reusable patterns strategy** allows them to reuse existing patterns, thus reduces time of development and testing of new patterns.

## 3.5 Adonis: External Dependencies Detector

We implemented our solution and name our tool Adonis[16]. In this section, we will explain how Adonis was implemented, what are the supported external agents and categories and then how it can be used.

### 3.5.1 Implementation

Adonis is implemented in Pharo as an open source and free package. The tool has a main class `Adonis`, from which derive subclasses representing each external agent. The main class has mandatory methods like `getReferenceContainers` and `getReferenceEntities` that should be implemented by each subclass, following our definition of external dependencies and our strategy of decomposing the problem to locate containers and entities, in addition to common methods to be used by subclasses like `buildResourceEntityFor(element)` and `defineExternalDependency`. Detection rules must be implemented in 4 methods for all external agents: `getReferenceContainers`, `getReferenceEntities`, `getResourceContainers` and `getResourceEntities`.

Adonis supports for the moment 4 different external agents (GWT, RMI, Hibernate and Pharo comments), thus 3 different categories of external dependencies as explained in Section 3.2.2: cross-languages links, multi-tiers and Documentation. Additionally, 3 external agents are in progress: GWT RPC[17] (Remote Procedure Call), Spring[18] and JDBC[19] (Java Database Connectivity) which is almost the same as Group 3 in Hibernate between HQL and Java classes, except that Java classes must be replaced by SQL Database tables.

For the matching strategies, we used both lexical and object matching using MoTion and regular expressions:

- **GWT:** we used Object matching. Parsers and metamodels for both languages (XML and Java) existed in Moose.

- **RMI:** again Object matching is used here. For RMI things are more complicated than GWT, as we needed to traverse the AST of the model in order to be able to find invocations of the client to the server, in addition to the registered server API in the registry.

- **Hibernate:** we used both, object matching and lexical. For XML mapping files, Java and the database, we used object matching. However, for HQL (which is the same as SQL), we decided to proceed with lexical matching,

[16] https://github.com/alesshosry/Adonis
[17] RPC https://www.gwtproject.org/doc/latest/tutorial/RPC.html
[18] Spring https://spring.io
[19] JDBC https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/

because the queries were formed by multiple concatenated Strings. Additionally, the only information that we wanted to extract from queries was the table names. We created adequate regular expressions to detect all the HQL queries that contain table names among Strings in the program, then started extracting all the mapped tables from the queries.

- **Pharo Comments** also here we used lexical matching to extract every CamelCase between single backticks( ` ).

### 3.5.2 Usage

Adonis is created to be executed automatically in order to facilitate its usage to the developers.

First, Adonis takes as input two elements: the program to analyze and the external agent identification (eg. GWT, RMI etc...), to know which subclass should be called. This is common to all external agents.

Figure 3.5 is an example of how Adonis can be used to detect external dependencies for GWT framework in Moose.

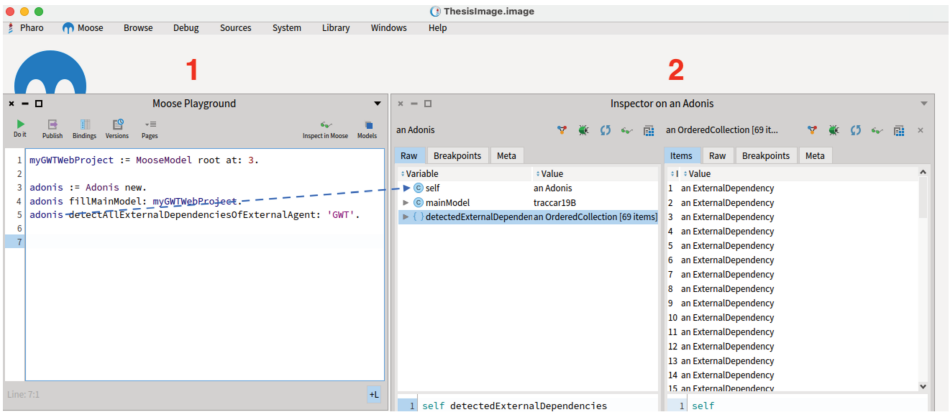



Figure 3.5: Adonis usage in Moose

As we explained before in Section 3.3.1, for GWT we chose to apply object matching. This is why we use Moose to import Java models. Being imported in Moose, it can now be used by encapsulating it in `myGWTWebProject` in line 1 of Moose playground (window 1). Then an instance of `Adonis` object is created in Pharo. With `#fillMainModel:` message, the imported model can be sent to the Adonis instance, and by calling `#detectAllExternalDependenciesOf ExternalAgent:` Adonis starts detecting dependencies between Java and XML.

Second, the results that are returned will be a collection of `ExternalDependency` objects. This is also common to all external agents. For example, for GWT the results are encapsulated in `detectedExternalDependencies` of the detector

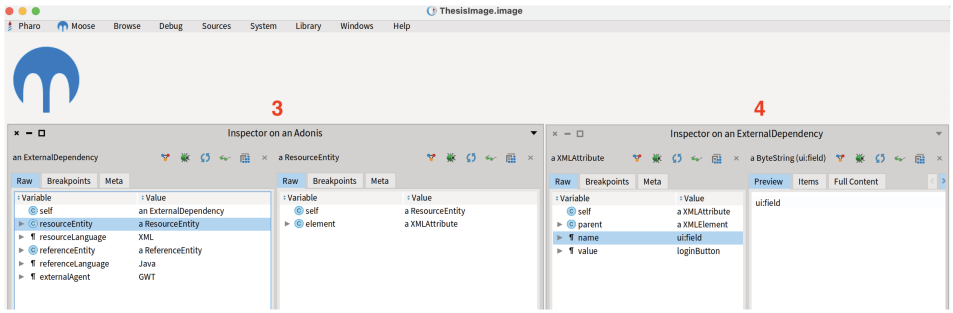



Figure 3.6: ExternalDependency Object

instance as a collection of `ExternalDependency` object as shown in figure 3.5 in window 2 after inspecting `adonis`.

In window 3, figure 3.6 we can see that an `ExternalDependency` has: a `resourceEntity` which is an object defined also in Adonis package, it contains the entity encapsulated in `element` of type `Object` (shown in window 4). For our example, the element is an `XMLAttribute` that has a `parent`, `name` and `value`, following the metamodel definition of XML parsed in Pharo. Additionally, it contains a `referenceEntity` which is also another object defined in Adonis package and has a `key` (of type `String` to fill the used key), the `elementName` (of type `String`) equivalent to the name of the resource entity being referred by the `key` and a `path` (of type `FilreReference`) containing the file where the reference exists. It contains also the used languages of the external agent (`referenceLanguage` and `resourceLanguage`) and the `externalAgent` as `Strings`.

## 3.6 Evaluating Adonis

In this section we explain how we conducted several experiments in order to validate if: (i) the approach can identify external dependencies, (ii) it works for different types of dependencies, and (iii) we can reuse patterns across external agents. To proceed, we first explain choosing the projects part of the experiment setup, how we conducted our experiment, then we present our findings and analyze them, and finally we show the results of Adonis and evaluate them.

### 3.6.1 Experiment setup

In this section we explain how we chose the projects and how we run our experiment.

We want to experiment Adonis on various projects from industry and open source. We also want various external agents to cover all the categories that we

presented previously in Section 3.2.2. We want real case (concrete) and legacy projects having various external dependencies.

In order to validate point (i) "the approach can identify external dependencies", we will rely on 5 metrics: we will count manually the number of resource containers, reference containers, resource entities, reference entities and external dependencies for each external agent and compare them with the results of Adonis. We chose to count the number of containers and entities because our approach is relying on targeting them first. Then it consists on relating the entities that construct the external dependencies conveniently, which is why we want to count them also. We will do this for all 4 projects, then run Adonis, extract the same data of the same metrics and check the accuracy of Adonis by comparing the results in terms of precision (the number of true positives divided by the number of true positives plus the number of false positives) and recall (the number of true positives divided by the number of true positives plus the number of false negatives). We will also track the execution time to assess how much we can reduce the time required for searching external dependencies.

To validate point (ii) "it works for different types of dependencies", we experiment with different projects developed using different external agents.

Finally, to validate point (iii) "we can reuse patterns across external agents", we will count the number of patterns in total and see how many patterns are reused.

### 3.6.2 Projects

In this section we present the projects used for our experiment and our findings (number of containers, entities and external dependencies) for each one them following a manual investigation.

Table 3.1 lists all different projects that we analyze. We have selected 4 different projects:

1. For GWT, we chose Traccar project[20]. It is an open project created in 2012, it has 3 different branches, but we analyzed the project using *legacy* branch because it was developed with GWT. Traccar-legacy has 100 commits and 2 contributors until 2015 (date of last commit on the branch). It has a total 34 Java classes, 2 KLOC and 13 XML files.

2. For RMI we chose the open source project UniScore [21], which is developed using two separate software: client and server. The project was created in 2020 and has a track record of 104 commits and 3 contributors. Uniscore-Server is the server part sub-project of 41 classes and 6 KLOC. Uniscore-Client is the client part sub-project of 50 classes and 7 KLOC.

[20]https://github.com/traccar/traccar-web/tree/legacy
[21]https://github.com/redhawk96/UniScore

Table 3.1: Selected projects to experiment Adonis

| External Agent | Project | Classes | KLOC | Open / Closed | Number of Commits | Number of Contributers |
|---|---|---|---|---|---|---|
| GWT | Traccar[22] | 34 | 2 | Open | 100 | 2 (Until 2015) |
| RMI | UniScore[23] | 91 | 13 | Open | 104 | 3 |
| Hibernate | – | 1K | 116 | Closed | – | – |
| Pharo Comments | Pharo[24] | 14K | 882 | Open | 29K | 209 |

Table 3.2: Results of manual count for containers, entities and external dependencies for all experiments

| External agent | Resource | | Reference | | External dependencies | Ex. deps. / Ref. ents. |
|---|---|---|---|---|---|---|
| | Containers | Entities | Containers | Entities | | |
| GWT | 10 | 60 | 10 | 69 | 69 | 100 % |
| RMI | 1 | 54 | 14 | 48 | 48 | 100 % |
| Hibernate G.1 | 1,140 | 86 | 51 | 51 | 42 | 82 % |
| Hibernate G.2 | 1,338 | 49 | 51 | 51 | 49 | 96 % |
| Hibernate G.3 | 1,338 | 49 | 334 | 424 | 414 | 97 % |
| Hibernate G.4 | 1,140 | 86 | 334 | 424 | 378 | 89 % |
| Pharo Comments | 1 | 14,159 | 557 | 5351 | 5282 | 98 % |

3. For Hibernate the company gave us access to a client registration program. It uses an Oracle database to store the necessary data. The program comprises 116 KLOC and includes a total of 1,338 classes and the database contains 86 tables.

4. For Pharo Comments experiment, we created a new clean image of Pharo 12, build number 1525, which is the latest stable Pharo version at the moment. In total this version of Pharo has more than 14K classes per image and 882 KLOC, more than 29K commits and 209 contributors.

Next, we show our findings for each project in table 3.2.

In an initial analysis for table 3.2, comparing the number of entities for each external agent reveals some patterns. For example, in GWT, the number of reference entities exceeds that of resource entities, suggesting that some resources are referred multiple times. However, in the case of RMI the opposite occurs, there are fewer reference entities than resource entities, indicating that not all resources are

referred. Additionally, when a certain number of reference entities target certain resources, we would expect to find the same number of external dependencies, which is the case of GWT and RMI where we found a ratio of 100% of reference entities are included in external dependencies. Yet, this is not true for Hibernate and Pharo Comments, where some reference entities point to non-existent resources, leading to lower number of external dependencies than expected, which explains why the ratios for these external agents vary between 82% and 98%.

Now we explain how we proceeded with our manual investigation and we discuss in details our findings.

For the **GWT** external agent, we identified 10 `.ui.xml` files (resource containers) and 10 Java classes (reference containers); We found 69 annotations (resource entities) in those Java classes and in parallel 60 XML elements (resource entities) defined in the XML files (XML elements referred by Java; others can be defined and have IDs without necessarily being referred); We found 69 external dependencies (the number of reference entities with a valid resource entity). During this investigation, we discovered that multiple resource entities are defined in XML but never referred to in Java. For example, for the `DeviceView` container, only 12 out of 16 resource entities were referred to. This is allowed in the GWT framework and is not considered a flaw (we will come back to this later in Chapter 5). Moreover, the comparison between "Resource Entities" 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 both Java annotations. For example, in the `UsersDialog` container, we found 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, we identified 1 java interface (resource container), and 14 java classes with calls to client servers (reference containers). In the java interface we identified 54 public methods signatures defined (resource entities) and 48 invocations for instances of this interface (reference entities). We identified a total of 48 external dependencies (the number of reference entities with a matching resource entity). We noticed that only 26 resource entities out of 54 were referred, which results in 28 resource entities that were created but never referred. Having this amount of unused resource entities is considered a bad practice in RMI since the API server methods were created but never used. Moreover, we also noticed that some server APIs were referred more than once. This explains why we found 26 referred resource entities, but also 48 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 in this class (reference container). This is considered a normal practice in RMI.

For **Pharo Comments**, we applied our study on one image (resource container),

that contains 14K Pharo classes (resource entities) intended to be referred by the comments. We extracted programmatically those classes. Then programmatically also we extracted all the comments that may contain references to such classes and found 557 comments (reference containers). We found a total of 5351 tokens as CamelCase between backticks (reference entities). We counted the external dependencies and found 5282 of reference entities with correspondent resource entities. With a manual investigation, we found that some of these entities are referring to missing classes that are deprecated and removed from the newer Pharo versions, whereas 6 others are referring just to descriptions of icons in Pharo IDE like "Help" and "Remove".

For **Hibernate**, we proceeded group by group but we cannot disclose the detailed results since it is a private project. Following are some data:

- For **Group 1**, between database and mapping files, we looked at the SQL schema and found that the database has 1,140 entities (resource containers) among them 86 SQL tables (resource entities). We identified using 51 XML mapping files (reference containers) and 10 attributes `table` within `class` XML element in these files (reference entities), meaning that not all tables are mapped in the program. We found 42 external dependencies (the number of reference entities with a matching resource entity). This means that 9 references in the mapping files were referring to tables that do not exist in the database. This is a bad practice in Hibernate and can cause issues at runtime.

- For **Group 2**, between the mapping files and the Java classes, we identified 1,338 Java classes (resource containers) and among them we filtered 49 classes as resource entities referred by the mappings. We had 51 reference containers and 51 resource containers identified in Group 1. This means that there are 2 classes referred in the mapping files but missing in the program. Also this is a bad practice and can cause issues on execution time. This resulted in 49 external dependencies.

- For **Group 3**, between HQL and the mapped Java classes, we have 1,338 resource containers and 49 resource entities counted previously. As for the references, we noticed that all the HQL queries were embedded in classes starting with "Dao". So to simplify our investigation, we searched among these classes. We found in these classes that HQL queries are concatenated. Some of them include references to the mapped classes without key words like "From" and "Update", which can be a limitation for Adonis, as we are doing a static analysis and relying on regular expressions to search for such keywords among the strings to identify table names. We extracted 334 queries fragments (reference containers) and extracted 424 table names (reference entities). This indicates that there are classes referred more than once in HQL queries. We ended up by counting 414 external dependencies

Table 3.3: Results of manual count for containers, entities and external dependencies for all experiments

| External agent | Resource Containers | Resource Entities | Reference Containers | Reference Entities | External dependencies |
|---|---|---|---|---|---|
| GWT | 10 | 60 | 10 | 69 | 69 |
| RMI | 1 | 54 | 14 | 48 | 48 |
| Hibernate G.1 | 1,140 | 86 | 51 | 51 | 42 |
| Hibernate G.2 | 1,338 | 49 | 51 | 51 | 49 |
| Hibernate G.3 | 1,338 | 49 | 331 | 421 | 411 |
| Hibernate G.4 | 1,140 | 86 | 331 | 421 | 375 |
| Pharo Comments | 1 | 14,159 | 557 | 5351 | 5282 |

which is the number of reference entities with a matching resource entity. We found 424-414 = 10 entities referring directly to the database instead of using the mappings, which is considered a bad practice in Hibernate. This was a lengthy task, but it was the only way to complete our validation.

- For **Group 4**, between the database and the HQL queries, we have the resources from Group 1 (database entities) and references from Group 3 (HQL queries in Java code). We linked the database tables, the mappings and the extracted table names and found 378 external dependencies. This indicates that 424-378 = 46 queries were used but referring incorrectly to the database. We investigated more, and found that this resulted from the incorrect mappings found for groups 1, 2 and 3.

At the end, the manual validation required the usage of multiple tools collect the data for different external agents, additionally between 20 and 25 hours of work to extract this data and to count the external dependencies. The results of this counting are published here [25].

### 3.6.3 Results of the Experiment

In this section, we will present the results provided by Adonis in Table 3.3. We compare the results with our manual findings in table 3.2. Then we show how many patterns were created and how they can be reused.

After running Adonis, we found for **GWT**, **RMI**, **Hibernate G.1**, **Hibernate G.2** and **Pharo Comments** experiment the exact same results as the manual count. However, for **Hibernate G.3** and **Hibernate G.4**, Adonis missed some reference

[25]Manual counting results https://doi.org/10.5281/zenodo.13820410

Table 3.4: Precision and recall ratios for Adonis

| External agent | Resource | | Reference | | External dependencies |
|---|---|---|---|---|---|
| | Containers | Entities | Containers | Entities | |
| | Pre./Rec. | Pre./Rec. | Pre./Rec. | Pre./Rec. | Pre./Rec. |
| GWT | 100% / 100% | 100% / 100% | 100% / 100% | 100% / 100% | 100% / 100% |
| RMI | 100% / 100% | 100% / 100% | 100% / 100% | 100% / 100% | 100% / 100% |
| Hibernate G.1 | 100% / 100% | 100% / 100% | 100% / 100% | 100% / 100% | 100% / 100% |
| Hibernate G.2 | 100% / 100% | 100% / 100% | 100% / 100% | 100% / 100% | 100% / 100% |
| Hibernate G.3 | 100% / 100% | 100% / 100% | 100% / 99% | 100% / 99% | 100% / 99% |
| Hibernate G.4 | 100% / 100% | 100% / 100% | 100% / 99% | 100% / 99% | 100% / 99% |
| Pharo Comments | 100% / 100% | 100% / 100% | 100% / 100% | 100% / 100% | 100% / 100% |

Table 3.5: Execution time for all experiments using Adonis

| External agent | Execution time (in seconds) |
|---|---|
| GWT | < 1 |
| RMI | 8 |
| Hibernate | 75 |
| Pharo | 2 |

entities and external dependencies, because three HQL query fragments were concatenated separately without any indicator suggesting they contained table names.

We also measured the execution time for each experiment, with the findings summarized in Table 3.5. At first glance, tasks that previously required hours, could now be completed in a maximum of 75 seconds for Hibernate, and under 9 seconds for the other experiments as shown in this table.

Following these results, we validated that Adonis is able to detect external dependencies successfully (point (i)), with promising precision and recall ratios for all projects varying between 99% and 100% as per table 3.4. Additionally, the implementation of Adonis covered various external agents. Each one of them covered one (and more for Hibernate) category of external agents. This validates point (ii), as despite the variety of these categories, we were able to create each time following the same methodology, a set of patterns, rules and methods to find containers, entities, and then external dependencies.

In terms of reusable patterns, we developed a total of 11 object patterns and 5 lexical patterns summarized in table 3.6. They are encapsulated in methods with input parameters, which can be used to construct the patterns, thus making them

Table 3.6: Patterns used in Adonis

| Pattern | Usage number | Comment |
|---|---|---|
| getCamelCases | 1 | |
| selectQuery | 1 | Reusable for JDBC |
| insertQuery | 1 | Reusable for JDBC |
| deleteQuery | 1 | Reusable for JDBC |
| joinQuery | 1 | Reusable for JDBC |
| defineDeclParamForType: | 1 | |
| getFileFromDirectoryBasedOnName | 1 | |
| defineJavaAnnotationNameWithAttributes: | 1 | |
| defineJavaAnnotationNameWithoutAttributes: | 1 | |
| defineJavaDeclStatementforString: | 1 | |
| defineJavaInheritanceFor: | 1 | Reusable for RPC |
| getFromFASTJavaMethodEntityAStringLike: | 1 | |
| defineJavaDeclStatement:forMethod: | 1 | |
| defineInvocationWithArgs:usingInvocator: | 1 | |
| defineInvocationWithoutArgs:usingInvocator: | 1 | |
| defineXMLPatternfollowingAttributeName: value:XMLType: | 2 | Reusable for Spring |

flexible patterns that can be reused for multiple matches. The lexical matching patterns are applied for HQL/SQL queries and Markdown comments. Currently, only 2 patterns were reused to search for XML attributes and XML files in GWT and Hibernate. This low reuse is due to the variety of programming languages used by the external agents, which means there are no patterns applied for the same language that can be reused multiple times, which limits the validation of point (iii).

For future external agents, the same XML pattern could be reused for Spring external agent that is under construction. Moreover, we have one (Java) pattern applied for RMI, which could also be used for RPC, as both require searching for inheritances from the `Remote` interface (for RMI) and the `RemoteService` interface (for RPC). This makes 3 out of 11 object patterns which could lead to an expected a total of 27% of reusable object patterns, where 1 of them will be reused 3 times.

Additionally, we have 5 lexical patterns: 4 for HQL and 1 for Markdown. Here we have 4 lexical patterns used for Hibernate, which should be reused for JDBC to extract table names from SQL queries, leading to an expected total of 80% of lexical reusable patterns.

## 3.7 Threats to Validity

In this section we present the threats to validity following the experiments and their validation.

**Internal Validity** We selected projects developed using various external agents, that use various languages (Java, Pharo, XML, Markdown and HQL) to prove generalizability. External dependencies are handled differently by these agents. Despite that were able to deal with this variety and detect external dependencies through Adonis.

As for the reusable patterns, we proved that XML pattern could be reused for different external agents that deal with XML. However, we do not know if others also can be reused. To avoid this possible bias, we are planning to expand the range of external agents.

**External Validity** We tried to ensure big and small projects, open and closed. We also tried to cover a wide range of external agents (RMI, GWT, Pharo and Hibernate) to cover all types of external dependencies discussed in this chapter (except callbacks).

**Construct Validity** We used metrics directly related to our research. We counted containers, entities, external dependencies and patterns, which are core to our approach. Execution time was also measured, though we recognize this metric requires further investigation, particularly for smaller projects like those using GWT and RMI. Larger projects would provide a more robust evaluation of this metric. These measures aim to accurately represent the effectiveness of our approach in detecting external dependencies.

**Conclusion Validity** Our approach demonstrated success in detecting external dependencies across the analyzed projects that use different languages. We know that if our approach could work for these languages, then it should work for others similar to them like C# and HTML. However, for GWT and RMI, we only tested small projects. We need to run experiments on larger ones to better understand how robust our approach is. We tried to expand more our research for Hibernate, but the projects [26] we found were not significant and did not help us to judge enough.

## 3.8 Conclusion

In this this chapter we saw that various external agents are used to create various types of software. These external agents rely on a set of rules specific to each one to establish external dependencies. We acknowledge the existence of multiple

[26] Hibernate open source projects experimenthttps://doi.org/10.5281/zenodo.13886550

types of external dependencies such as multi-tiers, cross-language links in polyglot programming and documentation. We also state that even if each external agent 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 external agent's rules. To design a detector that can achieve the usage goals successfully, we established a set of considerations 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 external agent rules that we experimented with two different projects. We found that there is a need to search over source files (wether source, documentations, comments ...) using object and lexical search. To apply that, we created a pattern matching that can apply object matching using its specific syntax and lexical matching using regular expressions. We explore more the pattern matching library in later chapter. We validated our tool by running 4 experiments using different external agents. Following our validation we found that the precision and recall ratios vary between 99% and 100%. Finally, following the completion of the experiments, we observed that several of the defined resource entities remained unused or were missed. This observation prompts a further discussion regarding the potential flaws and errors that may arise from external dependencies. We will explore more these issues and their implications in Chapter 5.

CHAPTER 4

# Declarative object matching in Pharo

## Contents

## 4.1 Introduction

We identified in the previous chapter the need for a good object matching tool in our detection of external dependencies. We now turn to this topic and present the pattern matching tool used in Adonis.

Pattern matching is the process of checking whether a given pattern matches a given value or not [Klint 2011]. Many pattern matching languages exist in the literature listed in Chapter 2, Section 2.2. They are used in different contexts with the main purpose: searching for tokens somewhere in files, graphs, objects... similar to a specific description expressed and called "pattern". We found in the literature an interesting list of features for pattern matching.

However, those features are split into two main groups: features frequent in graph pattern matching and features frequent in object pattern matching. Software models are huge graphs of objects. We need features from both kinds of pattern matchers.

In Pharo we have a pattern matching language to apply object matching which we refere in this thesis as "traditional pattern matching" and we have regular expressions to apply lexical matching. However we will see that it does not fit our needs to deal with the variety of external agents we are working with.

In this chapter we report on our efforts to implement a new object pattern matching tool in Pharo, that supports all features grouped in Chapter 2, Section 2.2. We explore an approach to pattern definition influenced by pattern matching for graphs and for objects to propose to developers a flexible way of searching and extracting information from objects. This is concretized in MoTion [1], a new declarative object matching language and tool implemented in Pharo. MoTion allows the definition of declarative patterns, combining features from both pattern matching for graphs and for objects.

The structure of the chapter is the following: In Section 4.2 we propose examples from the literature illustrating the need for pattern matching in modern software development. In Section 4.3 we present the traditional matching language in Pharo. Then, in Section 4.4 we describe the syntax of MoTion, a new declarative object pattern matching language implemented in Pharo. Some implementation details are discussed in Section 4.5 promoting the flexibility and extensibility of MoTion. We compare the traditional pattern matching language with Pharo and show the results in Section 4.6. We comment on some real use cases (Section 4.7) where MoTion was used by developers in software analysis tasks. From these use cases, we derived some lessons learned (Section 4.8) on the more useful features of MoTion or what improvement path lays before us. We close the chapter in Section 4.9 with our conclusions and discussion of future work.

[1] https://github.com/doubleBlind/MoTion

## 4.2 Motivation

When programming applications deal with large amount of data, developers often need to search for specific objects in the data. This happens when working in real-world domains such as biology, transport and social networks. An example of such graph analysis is illustrated by the work of Thakkar et al., describes how to extract the age of the eldest person knowing a person named "Marko" in a big graph of people [Thakkar 2017].

Software engineering daily activities also involve searching for source code elements. This is often done by representing the source code as a graph of objects (for example in an Abstract Syntax Tree) and searching for the "right object" inside this graph. For example, Mohamed and Kamel [Mohamed 2018] describe how to reverse engineer an application, looking for design pattern instances. For this, the authors suggested using static code analysis and following a set of heuristics, like identifying inheritance between classes and node selection inside methods based on their types. Searching is also needed when transforming (or refactoring, or restructuring) the code into a new form. This can be done by looking for a specific software element and transforming it into a new one, or by modifying it in a new form [Klint 2011].

Doing this kind of search in a GPL (General Programming Language) means going through sets of objects, selecting some of them, and then navigating to their children looking for specific properties (like attributes with a given value), ...

A better solution for this is to use a pattern matching language that will allow describing the sub-graph of objects one is looking for and letting it find all the matching occurrences in the whole graph of objects. The common pattern matching tools relieve the users from specifying how to traverse the whole graph, letting them concentrate on the description, in a specific notation, of the searched pattern.

```
1 public ColoredTree
2   makeGreen(ColoredTree t) {
3   return visit(t) {
4     case red(l, r) => green(l, r)
5   };
6 }
```

Listing 4.1: Rascal example

Listing 4.1 is an example extracted from the work of Klint et al. of a transformation rule expressed in Rascal, where method `makeGreen` defined in line 2 takes `ColoredTree t` as input parameter and replaces red nodes by green ones [Klint 2011]. The `visit` in line 3 is used to traverse all tree nodes and apply the rule expressed in line 4 using `=>` operator, where the Left Hand Side (LHS) pattern `red(l,r)` is transformed into the Right Hand Side (RHS) pattern `green(l,r)`. This example illustrates the capabilities of pattern matching to

represent the "shape" of data by describing a pattern: the LHS of line 4, and how information is extracted from this pattern: the `l` and `r` variables.

## 4.3 Traditional Pattern Matching in Pharo

Pharo already has a traditional pattern matching language used by the RefactoringBrowser 4.1 infrastructure [Anquetil 2022]. It employs a specific syntax for defining patterns to match them with Pharo source code AST using `RBParseTreeSearcher` class. The RefactoringBrowser also comes with a rewriter that allows to modify the source code matching a given pattern, but we will not explore this part here as we are only interested in the pattern matching capability.

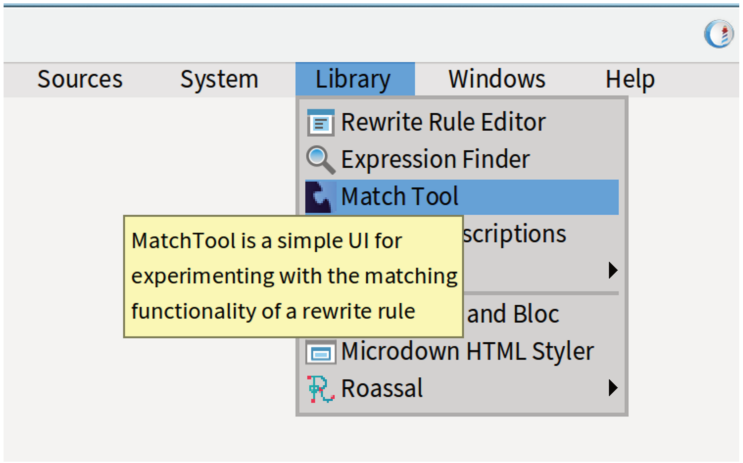



Figure 4.1: Pharo traditional patten matching tool

In this section we will present the syntax of the traditional pattern matching language in Pharo and then we will show different usages with Pharo methods using `RBParseTreeSearcher` class.

### 4.3.1 Syntax

The syntax of this language is based on specific symbols, enabling users to define patterns and match them against any source code within a Pharo image. Below is a list of these symbols, and we show how each one of them offers a distinct feature that highlights the traditional pattern matching capabilities in Pharo:

(‘) **Back-tick.** It is used to match any single node. For example: "`` `someName asString`" can match message `asString` sent to **any** receiver, disregarding its name. It can match Pharo source code like "`self asString`" and "`aParam asString`".

**(‘#) Literal pattern nodes.** To verify that the matched node is a literal, a backtick can be followed by the hash sign. “`‘#lit size`” is an example where `‘#lit` can match an integer `3`, a string `foo` or even a collection such as `#(a b c)`. It will not match “`self asString`”.

**(‘@) List pattern nodes.** Is used to match zero, one or more nodes in the AST. For example, “`|‘@args1 t1 ‘@args2|`” returns a successful match with `|t1 t2 t3 t4|` where `args1` will be empty and `args2` matches `t2 t3 t4`.

**(‘‘) Double Back-tick.** It is used to perform a recursive search. This implements the *deep match* characteristic. It entails searching for patterns not only on the surface level of the source code, but also examining their internal structure. For example, “`‘‘@vars + 1`” matches any selector followed by `+ 1`, additionally, it will internally check if this selector is already constructed by another selector followed by `+1`, such as `(myNum + 1) + 1 + 5`, where the first match of `‘@vars’` is `myNum + 1` and the second is `myNum` found by the deep match.

**(‘.) Statement pattern nodes.** To match statements, a developer can use a backtick followed by a period. Such patterns `‘.Statement1.` match a single statement in Pharo such as “`self assert: myVal size equals: 11.`” . It can also be combined with the list pattern (`‘@.`) to search for a list of successive statements, such as “`‘@.statements`” which will successfully match “`x := 1. y := 2. z := OrderedCollection new `”.

**(‘{ }) Block Pattern Nodes** Is a free-form test where, inside the curly braces, one puts a Pharo block, receiving a node as a parameter and returning a boolean whether this node matches or not. For example: “ `‘{:node|node isVariable and:[node isGlobal]} become:nil` ” is a pattern that matches a message `become:` with a `nil` argument, where the receiver is a global variable.

### 4.3.2 Examples

After describing the syntax and capabilities of the Pharo traditional pattern matching language, we will go through how to use this syntax with a handful of methods defined in `RBParseTreeSearcher` in this section. This purpose of this section is to how pattern matching can be used to identify a Pharo statement, a process that can be applied to any programming language, where external dependencies entities are embedded. While this chapter focuses on pattern “matching”, it is important to note that the same syntax can be used with a couple of methods implemented in `RBParseTreeWriter` class to perform some source code transformations.

```
1 searcher := RBParseTreeSearcher new.
2 searcher
3    matches: '@rcv put'
4    do: [ :aNode :answer | contextDictionary := searcher context ].
5 searcher executeTree: (RBParser parseExpression: 'self put').
```

Listing 4.2: Traditional pattern matching over source code sample in Pharo

Listing 4.2 provides an example of traditional pattern matching in Pharo of a match for a parsed tree of an expression, where the receiver "self", invokes the "put" message. This type of search for method invocations in source code is also useful in identifying external dependencies identification, such as searching inside client-server applications for server APIs invocations.

In line 1, `searcher` variable is declared as an `RBParseTreeSearcher` object. The message `#matches:do:` is sent to this variable in line 2 to start the match, where `matches:` accepts the pattern expressed using the syntax introduced in the previous section, and `do:` accepts a block of code that is executed only when the match succeeds. On success, all matches are stored in a dictionary (context), where each key represents a part of the pattern, like "@rcv" and the value contains the matched source code. In cases where the key matches multiple source code fragments, they are encapsulated in a list and stored in the value in pairs with the convenient key. In this example, when the block is executed, it enables storing all matches of `context` in `contextDictionary`. To start matching patterns, the developer needs to use method `#executeTree:` that takes as input a parsed source code object, such as line 5, using RBParser parseExpression.

In conclusion, this section introduced the traditional pattern matching language in Pharo. We provided a list of symbols that define its syntax and outlined the key features of the language. An example was also presented to demonstrate its usage for matching a method invocation. From this description, we identify a major limitation of the traditional pattern matching language: its focus on Pharo AST (Abstract Syntax Tree) matching. Every match must be compatible with the Pharo AST, and its syntax closely resembles Pharo, with only minor differences in the symbols used. Additionally, there is a significant limitation in feature availability, as several features listed in Chapter 2 Section 2.2 are not supported by this traditional pattern matching language.

## 4.4 MoTion

We just saw that traditional pattern matching in Pharo is not good enough. In this section we present MoTion, a new object pattern matching language in Pharo. MoTion is needed to apply generic matching following our needs for external dependencies detection.

A pattern matching language works on a finite set of objects that we will call a

*model*. Examples of models are: the Pharo AST of a method, the DOM of an XML document, the objects loaded from a JSON file,... MoTion can deal with Pharo objects independently of the model containing the data. It combines both features for graph pattern matching and object matching listed previously, and by doing so, it enables expressing patterns declaratively and applying matches to complex object structures.

In the next sections, we give a first overview of the pattern language with a concrete example. We also present the grammar of the pattern matching language and explain the semantics of each element of the language.

### 4.4.1 Simple Pattern Example

Before explaining in detail the syntax of MoTion, we give a simple example of a pattern used to detect all classes or interfaces that extend a given interface (named `'Remote'`). It works on Famix models [Anquetil 2020]. The pattern starts

```
1 FamixJavaModel % {
2  #'allTypes>entities' <=>
3  FamixJavaInterface % {
4   #'superInheritances>superclass>name'
5    <=> 'Remote' .
6   #'isStub' <~=> true.
7  } as: 'foundInterface'.
8 }
```

Listing 4.3: External dependencies searching pattern

by matching an instance of class `FamixJavaModel` (line 1) and looking in its `allTypes` property (line 2). This means that if it is not given a FamixJavaModel, it will not match anything. `allTypes` returns a special object that is a wrapper around a collection of all types defined in the model (classes, interfaces,...). To get this collection, we use its `entities` property. Note that if the object returned by `allTypes` has no `entities` property, then the pattern just fails to match anything.

The result of line 2 is a collection of objects on which we apply the operator "<=>" with the sub-pattern in lines 3 to 7. The operator "<=>" is polymorphic and for a collection of objects, it will try to match its sub-pattern to any object in the collection.

This sub-pattern matches an object of class `FamixJavaInterface` (line 3), and looks in its `superInheritances` property (line 4), then in the `superclass` property of the returned object, then in the `name` property of this other returned object. Here again, the operator ">" is polymorphic and handles collections of objects (in the pattern `superInheritances>superclass`) differently than single objects (in the pattern `superclass>name`).

Line 5, if the name matches the string 'Remote'[2], the engine checks for the next sub-pattern (line 6) which states that the matched object of line 3 (an instance of `FamixJavaInterface`) should also have a `isStub` property that should not match the boolean `true` (this is different from saying it should match `false` because it could also be `nil`).

Finally in line 7, if a matching `FamixJavaInterface` is found, it is bound to the key `foundInterface` in the final result. This result will be returned, and the object matched can be retrieved from it.

### 4.4.2 MoTion Grammar

In this section we present the grammar of MoTion in order to clarify its usage in upcoming sections. It must be noted however that, due to the nature of Pharo, this grammar is somehow artificial because it is not implemented in a specific parser. MoTion is implemented as extension methods on existing classes (see also Section 4.5). In this grammar, the non-terminals ⟨*PharoClass*⟩, ⟨*PharodLiteral*⟩, ⟨*PharoSymbol*⟩, and ⟨*PharoString*⟩, refer to normal elements of Pharo (respectively, a class name, a literal, a symbol and a string).

⟨*Pattern*⟩ ::= ⟨*LiteralPattern*⟩
| ⟨*PharoClass*⟩ ⟨*Percentage*⟩ '`{`' ⟨*Properties*⟩ '`}`'

⟨*LiteralPattern*⟩ ::= ⟨*PharoLiteral*⟩ '`asMatcher`'

⟨*Percentage*⟩ ::= '`%`'
| '`%%`'

⟨*Properties*⟩ ::= ( ⟨*Property*⟩ ⟨*SpaceShip*⟩ ⟨*Value*⟩ )*

⟨*Property*⟩ ::= ⟨*PropertyElement*⟩
| ⟨*Traversal*⟩

⟨*SpaceShip*⟩ ::= '`<=>`'
| '`<~=>`'

⟨*Value*⟩ ::= ⟨*PharoSymbol*⟩
| ⟨*Pattern*⟩
| ⟨*NonLinearPattern*⟩
| ⟨*ListPattern*⟩

⟨*PropertyElement*⟩ ::= ⟨*PharoSymbol*⟩

[2]The `Remote` interface of Java RMI.

⟨*Traversal*⟩ ::= ⟨*PathTraversal*⟩
| ⟨*RecursiveTraversal*⟩

⟨*PathTraversal*⟩ ::=
⟨*PharoSymbol*⟩ ( '>' ⟨*PathName*⟩ )*

⟨*PathName*⟩ ::= ⟨*PharoString*⟩ | '_'

⟨*RecursiveTraversal*⟩ ::=
⟨*PharoSymbol*⟩ '*' ( '>' ⟨*PathName*⟩ '*' )*

⟨*NonLinearPattern*⟩ ::= '#@' ⟨*PharoString*⟩

⟨*ListPattern*⟩ ::= '{' ( ⟨*ListItem*⟩ '.' )* '}'

⟨*ListItem*⟩ ::= ⟨*NonLinearPattern*⟩
| ⟨*PharoLiteral*⟩
| '_'
| '*_'

### 4.4.3 Pattern Operators

In this section, we outline the features offered by MoTion, which are essential for expressing any pattern required to detect external dependencies in our work. For each feature, we will demonstrate how it can be leveraged using the implemented syntax.

- ⟨*LiteralPattern*⟩s are Pharo literals used as patterns. For example `'A sample text here' asMatcher` and `1 asMatcher`. Literal patterns match exactly their literal value. This is useful for specifying the value that a property of an object must have.
- The ⟨*SpaceShip*⟩ operator tries to match a ⟨*Property*⟩ (of an object) on the left with a ⟨*Value*⟩ on the right. Note: the tilde version is a negation, it specifies that the ⟨*Property*⟩ should not match the ⟨*Value*⟩; It is the only way to specify a negation in MoTion.

  As noted before, it is a polymorphic operator depending on the content in the ⟨*Property*⟩. If this is an object, the operator tries to match this object to the ⟨*Value*⟩; If it is a collection, the operator tries to match any element of the collection to the ⟨*Value*⟩.
- To define an *object pattern*, one specifies its type using the ⟨*PharoClass*⟩ followed by the ⟨*Percentage*⟩ operator like in: `ClassA % {} .` '%' matches direct instances of the class, whereas '%%' matches instances of the class or any of its subclasses.

- These two operators can express sub-patterns and the properties of the matched object inside the curly braces. Object properties are instance variable accessors.

  The curly braces act as a conjunction of sub-patterns specifying the values that properties should match. It can be seen as a *Logical matcher*.

  The following pattern matches an object of class `ClassA`, with a ⟨*Property*⟩: `property1`, having the ⟨*Value*⟩: `aValue1`, and `property2` having the ⟨*Value*⟩: `aValue2`.

```
ClassA % {
  #'property1' <=> aValue1.
  #'property2' <=> aValue2.
}
```

  The sub-patterns could also be more complex (see below, *Nested pattern*).

  This mechanism contributes to the seamless addition of various properties, in a *declarative* way.

- The ⟨*Percentage*⟩, combined with the ⟨*SpaceShip*⟩ operator, also allows to express *Nested pattern* where a first object is matched, then a second object in one of the properties of the first is matched. One may express a sub-pattern on this second object. For example, the following pattern matches an instance of `ClassA` with `aValue1` in its `property1`, and an instance of `ClassB` in its `property2`. This second object must have `aValue3` in its `property3`.

```
ClassA % {
  #'property1' <=> aValue1.
  #'property2' <=> ClassB %% {
    #'property3' <=> aValue3.
  }
}
```

- *Non-Linear pattern* is obtained using the "`@`" operator followed by a name (for example: `@x`). This allows to store a matched object in the "variable" to reuse it somewhere else in the pattern.

- *Wildcard* ("_") can be used to indicate a property whose name is not known, when one only cares for its value:

```
ClassA % {
  #_ <=> aValue.
}
```

This pattern matches an instance of `ClassA` with an unnamed property matching the value `aValue`.

- The ">" operator implements *Path traversal* by allowing to "chain" multiple properties in a pattern. Such paths help reducing complex pattern's expression, by accessing a chain of objects and their properties:

  The following pattern first matches an instance of `ClassA`, then it takes the object in its `property1` and the value in `property2` of this second object. This value should match `aValue`.

```
ClassA % {
  #'property1>property2' <=> aValue.
}
```

  This notation allows expressing in a concise way a path in a graph of objects. The same result could be obtained with the pattern:

```
ClassA % {
  #'property1' <=> Object %% {
    #'property2' <=> aValue
  }
}
```

  Note that the ">" operator is also polymorphic. Similarly to "<=>", if one of the objects in the path is a collection, the operator will look for an element of this collection that allows to continue the search, that is to say that has a property matching the remaining part of the pattern.

- MoTion allows to perform *Recursive traversal* through a "$\star$" operator combined with the *Path traversal* operator ">". In a chain of objects, one may know the initial property and the final one, but not know how long the chain of objects is.

```
ClassA % {
  #'property1>repeatedProp*' <=> aValue.
}
```

  This pattern will match first an instance of `ClassA`, then the object in its property `property1` then it will match a chain of objects all having a property `repeatedProp` and one of them containing the value `aValue`. The match ends with this last object.

- The ⟨*RecursiveTraversal*⟩ operator may also be combined with a *wildcard* ("_").

```
ClassA % {
  #'property1>_*>propN' <=> aValue.
}
```

This pattern will match first an instance of `ClassA`, then the object in its property `property1` then a chain of objects with unknown properties ending with an object having a property `propN` with value `aValue`.

- It is possible to match *List pattern* using the ⟨*ListePattern*⟩ and declaring how the list should look like. Note that this is not the same operator as ⟨*Percentage*⟩ (see above). This operator allows to express that given elements in a list should match specific patterns.

```
{#'@x'. #'@x'}
```

This pattern matches a list containing exactly two elements that are the same (use of a named variable).

- The repetition operator ("*") may also be used in a list to indicate an unspecified number of elements.

```
{#'@x'. #'*_'. #'@x'}
```

This pattern, matches a list with first and last elements equal and of unspecified length (obviously at least 2).

Note that `'*_'` is used in list matching whereas `'_*'` is a repeated *wildcard* used in *Recursive traversal*.

- To express that one element is part of a collection, MoTion offers a shortcut. To check if the value `5` is part of a collection (contained in the property `someProperty` of an instance of `ClassA`) one can use the pattern:

```
ClassA % {
  #someProperty <=> {#'*_'. 5. #'*_'}
}
```

But, thanks to the already presented polymorphism, of the ⟨*SpaceShip*⟩ operator, the same can be expressed with a shortcut:

```
ClassA % {
  #someProperty<=> 5
}
```

This, however, could also match an instance of `ClassA` with a property `someProperty` containing exactly the value 5 (with no collection).

- Finally, there is another operator for *Logical matcher*: `orMatches:`. It allows to express a disjunction of two patterns (one or the other match). (Remember that ⟨*Percentage*⟩ implements a conjunction of patterns within the curly braces.)

```
ClassA % {
  #someProperty <=> (5 orMatches: 6)
}
```

This pattern matches an instance of `ClassA` with a property `someProperty` matching the value 5 or the value 6.

### 4.4.4 Using MoTion

First, one gets a "matcher" by calling the `asMatcher` method. We showed an example of this at the beginning of Section 4.4.3: "`1 asMatcher`" creates a matcher that only matches the value "1".

Second, a matcher has a `match:` method that allows it to try to match the argument.

The result of `match:` is a `MatchingResult`. It includes a boolean property `isMatch` indicating whether the match was successful or not. It also has the property `matchingContexts` which is a collection of `MatchingContext` objects. Each of these contexts includes again a boolean field `isMatch` and a dictionary of its `bindings`.

The following creates a matcher that matches anything and binds it to the "foo" symbol (first line). The pattern is run on the string `'text'`. The last line will answer `true` as the match was successful.

```
pattern := #'@foo' asMatcher.
result := pattern match: 'text' .
result isMatch.
```

To get the binding of `foo` in this small example, one would do (`bindings` returns a dictionary and `at:` is the standard method to access an element of a dictionary):

```
result matchingContexts first
  bindings at: 'foo'.
```

This will return the string `'text'`.

Bindings can also be created with the `as:` method. It is used to bind the result of a pattern that will be kept in the result's bindings. For example, in Listing 4.3, it is used to store the result of the sub-pattern in lines 3 to 7 (the matched FamixJavaInterface).

Finally, to simplify getting the result of the bindings one is mostly interested in, there is a method `collectBindings:` that accepts a collection of (interesting) keys as a parameter and returns their values matched by a pattern. In case there is no match, the return is an empty collection.

```
pattern := #'@foo' asMatcher.
results := pattern
collectBindings: {#foo }
for: 'text' .
```

This puts in `results` a collection of dictionaries (here there is only one) with the binding for the `#foo` symbol. The result is a collection because there could be several matchings (for example with a disjunction operator). The collection holds dictionaries because we could ask for several bindings in the first parameter of the method.

## 4.5 Implementation Notes

In this section, we present the implementation of MoTion as an open source library in Pharo, and show how it can be easily extended in order to enrich its syntax, thus providing more features for pattern matching.

MoTion uses the flexibility of Pharo syntax to implement the operators and enable the creation of additional operators or the specialization of existing operators. For example the "% {}" operator is implemented as a method on `Class`[3] so that this expression is valid in Pharo:

```
ClassA % {
  ...
}
```

We saw in the previous section that some operators are polymorphic ("<=>" and ">"). This is implemented through a polymorphic `#match:withContext:` method (not further described here).

[3] Actually, the method is `%` and the curly braces are the argument of the method.

In the following subsections, we propose some examples of implemented extensions, but more details about this implementation can be found on the open source hub of the project. In addition, a chapter will be published in an upcoming book: New tools in Pharo[4].

In summary the implementation relies on:

- A class `Matcher` responsible for matching a pattern to a model with the method `match:` (see also Section 4.4.4). It has 19 subclasses performing some operators (like `%` ) or literals as patterns,...
- Classes `MatcherResult` and `MatcherContext` that hold the result of a matching. An instance of `MatcherResult` is obtained as the returned value of the `match:` method (see above)
- Class `MotionPath` to implement the various path features: ⟨*PropertyElement*⟩ (ie. `#name`), *Wildcard* (ie. `#_`), *Path traversal* (ie. `>`),...

  It has six subclasses all implementing a method `resolveFrom:`.
- Six implementations of a method `asMatcher` added to pre-existing classes Array, Boolean, Class, Number, String, and Symbol. They convert a literal or Object to a pattern (ex: 'A sample text here' `asMatcher`).
- Methods `%` and `%%` implemented in Class to allow expressing *Object matching* (ie. ⟨*PharoClass*⟩ `% {}`).
- the ⟨*SpaceShip*⟩ methods (ie. `<=>` and `<~=>`) added to class `Object`

### 4.5.1 A Simple Extension

Listing 4.3 showed an example of a pattern on a Famix model. We actually implemented an extension of MoTion for Famix, since many external agents are using Java language which can be represented in Pharo using Famix metamodel.

In Famix, the properties of entities can represent:

- "Famix *properties*" that contain "Famix primitive types" (Numbers, String or Boolean);
- *associations* that point to another Famix entity (a FamixJavaMethod *invokes* multiple other FamixJavaMethods);
- *composition* relationships (a FamixJavaClass *contains* multiple FamixJavaMethods).

[4]https://github.com/SquareBracketAssociates/NewToolsInPharo, consulted on May 2nd, 2024; book in writing at this date.

Because the properties are meta-described, one can manipulate them programmatically. We therefore experimented with modifying the behavior of the path operator (“>”) to navigate only *composition* relationships. We also added another operator to preserve the previous behavior of the path operator.

### 4.5.2 Changing the Syntax

Another experiment could be to change the syntax of MoTion. This was a request from some users who wanted easier syntax for MoTion usage. For this, we proposed to replace the operators with keyword messages: The “% {}” operator could be replaced by the message “`instanceWithProperties:`”; “<=>” operator could be replaced by the message “`objectMatches:`”, and the <∼=> operator could be replaced by the message “`objectDoesNotMatch:`”.

We illustrate these changes on the pattern example of Listing 4.3. The result is shown in Listing 4.4 with the changes highlighted in bold. It’s important to emphasize that both patterns are valid and interpreted by MoTion in exactly the same way. We just added “synonym” methods.

```
1  FamixJavaModel instanceWithProperties: {
2    #'allTypes>entities' objectMatches:
3      ((FamixJavaInterface instanceWithProperties:{
4      #'superInheritances>superclass>name'
5      objectMatches: 'Remote' .
6      #'isStub' objectDoesNotMatch: true.
7    }) as: 'foundInterface')
8  }
```

Listing 4.4: MoTion operator replaced by Pharo message

The downside of this approach is that we need to put parentheses around the sub-patterns. Here, the “`as:`” message (line 7) could “collide” with the new keyword messages (`objectMatches:` on line 2 and `instanceWithProperties:` on line 3) and be mistaken for a composed keyword message (`objectMatches:as:` or `instanceWithProperties:as:`). This is an issue that did not arise with the use of symbols because of the precedence of binary messages in Pharo. Note also that, instead of the inner parentheses, we could actually have created the `instanceWithProperties:as:` method that would first call `instanceWithProperties:` and then `as:` on its result.

## 4.6 Comparison of MoTion and Traditional Matching in Pharo

By leveraging MoTion's syntax, users can specify complex patterns to match objects based on various criteria, as we chose MoTion syntax to be expressed declaratively. In terms of capabilities, MoTion has been impacted by other matchers like the ones stated in 2 in addition to RBParseTreeSearcher such as deep search and anonymous variables. Since both matchers have some common capabilities, we decided to perform a comparison of matching using the same Pharo 11 image for both of them. We took some basic source code templates, some of which are search rules implemented as examples in the rewrite tool.

### 4.6.1 Syntax and Expressiveness

- The first example, shown in Listing 4.5, checks if the source code has a selector `#ifTrue:ifFalse:`. With RBParseTreeSearcher, specifying patterns to detect the receiver and the list of arguments inside the blocks is mandatory, while in MoTion, this specification could be skipped as the developer is only interested in knowing if the selector is invoked in this code or not.

- Patterns of the second example in Listing 4.6, are inspired by the third rule of the rewriter tool, whose purpose is to check if `nil` exists in `ifNil` to remove it by applying a transformation rule. For this rule, the RBParseTreeSearcher pattern is more efficient as it is able to precisely position nil inside `ifNil:`. This cannot be done by MoTion, as it cannot precisely determine the nil position in the used blocks.

- The third example, shown in Listing 4.7, is inspired by rule 8 of the rewriter tool, which consists of matching a `select:` method that contains a receiver followed by not, to be replaced in a later stage by `reject:`. Again, the precision of `not` position is mandatory in this example, which is possible by RBParseTreeSearcher as it is able to express the possible existence of temporary variables and statement lists, followed by `not` positioned at the end inside the block. While that is not possible in MoTion, as the properties declared inside the pattern and associated with some values or subpatterns, do not take into consideration their order.

```
1 "-- RBParseTreeSearcher --"
2 '@receiv ifTrue: [ '@args1 ] ifFalse: [ '@args2 ].
3
4 "-- MoTion --"
5 RBMessageNode % {
6   #'selector>value' <=> #ifTrue:ifFalse:
7 }
```

Listing 4.5: Match #ifTrue:ifFalse:

```
"-- RBParseTreeSearcher --"
'@receiver ifNil: [ nil ] ifNotNil: '@arg

"-- MoTion --"
RBMessageNode % {
  #'selector>value' <=>#ifNil:ifNotNil:.
  #'arguments>body>statements>value' <=> nil
}.
```

Listing 4.6: Remove unessecary #ifNil

```

"-- RBParseTreeSearcher --"
'@receiver select: [ :'each |
        | '@temps |
    ''@.Statements.
    ''@object not ]

"-- MoTion --"
RBMessageNode % {
  #'selector>value' <=> #'select:'.
  #'children*>selector>value' <=> 'not'
}.
```

Listing 4.7: Replace #select: by #reject:

### 4.6.2 Matching Speed

Table 4.1: Speed comparison

| Pattern | RBParseTreeSearcher Speed for 100 000 executions per second | MoTion Speed for 100 000 executions per second |
|---|---|---|
| Listing 4.5 | 0.101 | 0.433 |
| Listing 4.6 | 0.98 | 1.705 |
| Listing 4.7 | 0.411 | 1.389 |

While both languages excel in pattern matching capabilities, RBParseTreeSearcher has been recognized for its superior speed compared to MoTion as it is dedicated to matching only Pharo ASTs. It is optimized for this only, while MoTion is entirely generic and can match any object at any depth, implying more computation and thus more time to execute.

### 4.6.3 Matching Characteristics

After comparing the speed of match for both languages, we list in Table 4.2 the characteristics explained in Chapter 2 and if they are applied for each one of them: Based on both comparisons, it is evident that RBParseTreeSearcher exhibits superior performance compared to MoTion, despite the latter being a more generic pattern matching language.

Table 4.2: Characteristics comparison

| Characteristics / Languages | MoTion | RBParseTreeSearcher |
|---|---|---|
| *Declarative patterns* | x | |
| *Path traversal* | x | |
| *Recursive traversal* | x | x |
| *Repeated search* | x | x |
| *Object matching* | x | Pharo AST |
| *Literals* | x | x |
| *Non-Linear pattern* | x | x |
| *Wildcard* | x | x |
| *Nested pattern* | x | |
| *List pattern* | x | x |
| *Logical matcher* | x | |
| *Negation* | x | |

## 4.7 Use Cases of MoTion

We used MoTion in some of our projects and presented it to other people to use in their projects. We report here some of these experiments. We will summarize the lessons learned from these experiments in Section 4.8.

### 4.7.1 External Dependencies

MoTion was used for in our main project "Adonis" as the searching engine for entities of external dependencies presented in Chapter 3. With MoTion we were able to deal with polyglot software, developed using several programming languages or tiers at the same time. This is the case for example of GWT applications that use Java and XML, or RMI systems where two applications (client and server) must cooperate.

In order to be able to detect dependencies in these projects, we used MoTion to create more generic patterns that could be reused for different frameworks. For example, searching for an XML attribute was used for GWT applications, but could be reused in other cases.

Our running example (Listing 4.3) comes from this experiment. It was already presented and explained in Section 4.4.1.

We noticed in this work, the use of multiple features of MoTion together for many patterns:

- structured patterns for complex objects such as `FamixJavaModel % {...}` (line 1) and `FamixJavaInterface % {...}` (line 3);
- traversal paths expressed in this listing in lines 2 and 4, that allow matching chains of objects;
- negative search (line 6).
- declarative pattern which allowed us to add as many properties as we want to specify more the description of the pattern

### 4.7.2 Refactoring Source Code

Our next use case is a developer who used MoTion for a refactoring task over a Java application with a model that contains more than 1.5 million entities.

The problem was to detect all the invocations of method `get` on an object `config` (the receiver) with argument a specific key (`config.get(aKey)`). This needed to be done on a representation of the AST of the method to be able to modify the AST after.

The developer created the pattern in Listing 4.8.

```
1 FASTJavaMethodEntity % {
2   #'children*' <=> FASTJavaMethodInvocation % {
3   #'receiver>name' <=> #'config'.
4   #name <=> #get.
5   #'arguments>primitiveValue' <=> aKey.
6   } as: #configInvocation
7 }
```

Listing 4.8: Reverse engineering pattern

The pattern was applied to FAST-Java, a member of the Famix family specializing in modeling Java ASTs. It starts by matching a `FASTJavaMethodEntity` (ie. a method node) and looks at its children for a `FASTJavaMethodInvocation` (line 2). Because this invocation could be at any depth in the AST, he used the "$\star$" operator (*Recursive traversal*). On the invocations matched, it looks for the receiver's name which should be "config" (line 3). The name of the invocation (method invoked) should be "get" (line 4). The argument of the invocation should be an object with the property "primitiveValue" matching `aKey` (line 5). Here, the key is a parameter that can change for different searches.

We noted in this work:

- The *Recursive traversal* which was necessary because the invocation is at different depths in the AST in different methods.
- The ease of use, the developer was able to work alone after a small presentation of MoTion syntax of only half an hour.
- The need for "named sub-patterns" that could be reused to compose complex patterns. Actually this feature exists but the developer did not think about it. The solution is simply to put a pattern in a variable that can be reused to build more complex patterns.

  As an example Listing 4.9 presents a rewritten version of the pattern represented in Listing 4.8 (even though there is no sub-pattern reuse there). Variables are highlighted to help read the pattern.

```
1  childrenPath := #'children*'.
2  receiverNamePath := #'receiver>name'.
3  argsVal := #'arguments>primitiveValue'.
4
5  subPattern := FASTJavaMethodInvocation % {
6    receiverNamePath <=> #'config'.
7    #name <=> #get.
8    argsVal <=> aKey.
9  } as: #configInvocation.
10
11 FASTJavaMethodEntity % {
12   childrenPath <=> subPattern.
13 }
```

Listing 4.9: Reverse engineering pattern decomposed

### 4.7.3 Backend for Other Pattern Matching

In our work [Hosry 2023b], we compared MoTion to `RBParseTreeSearcher`. It is a pattern matching language to search over the Pharo AST (and possibly rewrite the AST with `RBParseTreeWriter`).

We compared MoTion to `RBParseTreeSearcher` class, by applying a search over the same Pharo AST, with both matching languages. Listing 4.10, shows the two patterns, used to check if the AST contains a selector `#ifTrue:ifFalse:`.

The patterns in `RBParseTreeSearcher` syntax (line 1) look similar to the original Pharo source code, except that some operators are used to help describe the pattern. Here, we are using the backtick operator (`` `@ ``) to refer to a list of nodes in the AST. The pattern `` `@receiv `` can match multiple nodes that behave as a receiver for the `#ifTrue:ifFalse:` message, and the blocks can contain

multiple arguments inside, that are different from each other after naming them `args1` and `args2`.

In MoTion (line 3 to 5), we defined a pattern of type `RBMessageNode`, and used a traversal path to match the selector searched for.

```
1 '@receiv ifTrue: [ '@args1 ] ifFalse: [ '@args2 ].
2
3 RBMessageNode % {
4  #'selector>value' <=> #ifTrue:ifFalse:
5 }
```

Listing 4.10: Pharo AST matcher

We noted in this work:

- With RBParseTreeSearcher, specifying patterns to detect the receiver and the list of arguments inside the blocks is mandatory, while in MoTion, this specification could be skipped as the developer is only interested in knowing if the selector is invoked in this code or not.
- We found that RBParseTreeSearcher is faster. This should be expected as it is a matching language dedicated to only Pharo ASTs, while MoTion is a generic and can match any object at any depth, implying more computation and thus more time to execute.
- RBParseTreeSearcher lacks some capabilities, like the ability to express path traversal (see Table 2.3 for the features of this matching language).

## 4.8 Lessons learned

In this section, we summarized the take-outs of our experiments, both positive and negative.

### 4.8.1 Comparison with Pre-existing

We compared the performances of MoTion and RBParseTreeSearcher patterns for Pharo AST search. MoTion was slower. We suppose this comes from the fact that RBParseTreeSearcher is fine tuned for matching Pharo AST nodes. For example it lacks some capabilities that MoTion has and makes the matching more computationally demanding.

We are considering whether it would be possible to compile MoTion patterns to make them more efficient. This is the subject of future work.

We conducted an experiment comparing pattern matching (MoTion) with traditional programming approaches using the reverse engineering example described

in Listing 4.8. For the traditional programming, we implemented a new class with three methods totaling 30 lines of code.

These methods are contained within a single class. To use them, an instance of the class must be created.

The results obtained are equivalent to those achieved with the MoTion pattern encapsulation using `as: #configInvocation`.

In summary:

- The user code is longer, one class, three methods, 30 lines instead of a 7 lines pattern;
- *Recursive traversal* (`#'children*'` in MoTion) required to implement a recursive method;
- Searching in the list of arguments (`#'arguments> primitiveValue' <=> aKey`) required to loop over the values returned by `arguments` to check their `primitiveValue`;
- The *Non-Linear pattern* (`as: #configInvocation` in MoTion) simplifies collecting the results that can be later retrieved with `#collectBindings: for:`. Traditional programming requires taking care of how results are returned by each method to collect and return them at the end;
- The class and methods created are very specific to the problem considered and another pattern would require reinventing a new solution with a possibly very different strategy.

### 4.8.2 Most Used Features

In the experiments reported, we found the following features to be the most used:

**Object matching** is the main feature used in all patterns. This is due to the fact that we are in an object oriented language and the things to match are objects.

MoTion, however, unlike Tom or Rascal, can work with any object structure and doesn't require its own definition of the classes to express patterns on them.

We worked with many different models (FamixJava, FASTJava, XML-DOM, Pharo AST, mistletoe model) without having to specify anything specific in MoTion other than the patterns themselves.

**Repeated Search** helped developers to collect all the matches of a pattern. MoTion does not stop after the first match is found, it stops when the leaf is reached. For now, this option seems to be enough for most common uses. However, we set a future target of adding a feature to limit the number of

searches, which will be useful for some cases to prevent cyclic looping. A difficult issue will be to find a succinct syntax to express this feature.

**Traversal Path** is very useful to match nested objects by simply describing the path to reach them in a concise way. This makes the pattern more readable and understandable by the user.

It was used for example in Listing 4.3 (line 4) to navigate from an object, to its `superInheritances`, then the `superclass` of this object, and finally the `name` of the last object.

**Resursive traversal** is very helpful in searching for properties with unknown depth in a tree, or even for unspecified property with a concrete value. It was used for example in Listing 4.8 (line 2) to find an invocation that was at an unspecified depth in the AST.

There is a risk with this feature of entering an infinite loop if the graph is cyclic. In our example we used the `children` property which is a containment tree and assures us that the search will end.

For future implementations, we are planning to add a limit number for the recursive search like (`*numberOfSearches`) that will prevent it from running forever.

**Wildcards** not only helped developers express ignored properties, we also discovered another important usage of it. It was used by some developers to express multiple properties having common sub-properties, and the latter are the ones that interest the developers for search.

**Non-linear pattern** was very beneficial for developers dealing with cross-language applications, as it allowed them to express patterns in testing experiments to match the same value of a property in 2 different languages, like configuration keys defined in an XML file and referred to in a Java method.

Being able to express non-linear patterns was really useful in the context of matching YAML configuration files[5] (experiment not presented here), where those files can have very different structures and the same information needs to be matched in different parts of the data and at different depths.

On the opposite, list patterns were not used explicitly in our experiments. They appear in the "short cut form" in some patterns (eg. Listing 4.3, line 2) when a pattern matches a list and the "<=>" operators allow to look for one element of the list.

We used it only in one example (not presented here), in Iguala for the Napari project, to describe in one line the precedence of some values in a list.

[5] Specifically, GitHub Actions files.

### 4.8.3 Missing Feature

Debugging patterns is a known difficulty in pattern matching.

MoTion can return a false match in cases where patterns were expressed incorrectly (the pattern does not actually match what the user wants). Some help is required for the user to find these mistakes. We started to implement simple solutions, but more needs to be done.

Note, however, that the experiments were related to program analysis, and that may have biased the result based on the features most commonly used. To ensure and quantify well the degree of usefulness of each feature, a bigger study should be conducted, considering the use of MoTion in different contexts.

## 4.9 Conclusion

In this chapter, we introduce MoTion, a new generic object pattern matching language for Pharo Smalltalk. A pattern matching language specifically tailored to match Pharo ASTs is already included in Pharo using `RBParseTreeSearcher` class. We presented it and gave examples of matching Pharo ASTs, however it is not generic enough. On the other hand, MoTion can match ASTs and, more generally, any Pharo object, and it can be used on-the-fly, *i.e.* it is not required to redefine artifacts, like object signatures, to use it (opposed to Tom).

In order to create a new object pattern matching language that can offer developers some capabilities like searching among objects with deep depth, defining non-linear patterns, and applying list matching, we have extracted a couple of features known to be adopted by graph and object pattern matching and applied them to MoTion.

We gave an example of MoTion and then presented the syntax in detail. We explained the functionality of each operator and how a match can be applied using different Pharo messages implemented, like `#match:` and `#collectBindings: form:`.

We also presented the implementation of MoTion and how it can be extended if needed. We applied some small modifications to some operators to replace them with more human readable messages.

We compared MoTion with the traditional pattern matching language implemented n Pharo, and found that in terms of speed, the former is faster than the latter, however MoTion proved to be more generic and matching the characteristics stated in Chapter 2.

In order to prove its feasibility, we presented MoTion to a couple of developers familiar with Pharo, who work in the reverse engineering domain and software analysis for external dependencies extraction. Developers shared their positive experience and requested a couple of features, such as debugging, that we will introduce in our future work. We also compared MoTion syntax with RBParse-

TreeSearcher to check its feasibility for matching Pharo ASTs. We are planning for the future to change its backend to rely completely on MoTion, as we were able to find some cases where the match is not yielding the correct results at the end.

Ultimately, we concluded by enumerating the lessons we had learned that had not been taken into account when our matching language was first implemented. For example, developers employed wildcards to indicate anonymous properties as well as the fact that multiple properties occasionally share the same sub-property, the value of which is the one we are interested in matching.

Lastly, we have discussed external dependencies in Chapter 3 and how they can be detected using Adonis. We identified two main challenges: first, the need to locate the entities involved in external dependencies, which was addressed in this chapter by creating MoTion, the new declarative pattern matching tool in Pharo. Second, we need to assess the validity of external dependencies, including whether any entities might be missing, unused, or subject to other issues. In the next chapter, we will further explore the correctness of external dependencies to address this challenge.

CHAPTER 5

# Evaluating External Dependencies

## Contents



## 5.1 Introduction

Developers must have in-depth knowledge of programs in order to ensure they will continue to perform properly. This knowledge comes after understating how the different software components are connected to each other and how they interact, including external dependencies. In previous chapters we presented the external

dependencies, their multiple categories and our approach based on tracking the corresponding "containers" and "entities" in order to identify those external dependencies.

Our tool, Adonis, was evaluated using the projects introduced in Chapter 3. During this evaluation, we discovered cases where extra resource entities were defined but never used, as well as reference entities that pointed to missing resource entities. We classify such inconsistencies as incorrect external dependencies, and this chapter is devoted to their identification and detection.

To address the challenge of identifying and detecting incorrect external dependencies, we propose an approach with the following key contributions:

- A comprehensive identification of all types of incorrect external dependencies.
- A detailed evaluation of our approach's accuracy for detection of incorrect external dependencies using projects developed with various external agents that was conducted on the same set of projects previously used for detecting external dependencies.

The chapter is structured as follows: Section 5.2 sets the vocabulary used in the chapter and explains our approach to identify incorrect external dependencies. Section 5.3 lists for some external agents, the different types of incorrect external dependencies that can be found. In Section 5.4 we explain how Adonis and can be used to filter incorrect external dependencies. We evaluate Adonis in Section 5.5 where we list the details of the experiment and present the results of the detection with the limitations we faced and we list the threats of validity in Section 5.6. Finally, we end up with a conclusion, in Section 6.1.

## 5.2 External Dependencies Correctness

In this section, we explain the concept of incorrect external dependencies and their impact on a program. Additionally, we provide a detailed list of the types of incorrect dependencies we identified, drawing from both the literature and our experiments.

### 5.2.1 What are Incorrect Dependencies?

We saw in Chapter 2 that some studies on detecting external dependencies also took into consideration when these dependencies were incorrect. They looked for "missing", "broken", "bloated", "excessive", "unused" and "version constraint" dependencies. These incorrect dependencies can:

- cause errors at compile or execution time, when a resource entity is supposed to exist but is missing (eg. executing a SQL query embedded in Java program and referring to an absent database table) [Jafari 2021] ;

- make a program larger than necessary or a compilation longer than necessary, because some entities are included in the distribution but never used (eg. unneeded includes) [Soto-Valero 2021];
- cause inconsistent or incorrect behavior, when reference entities exist (eg. if there are several handlers for the same event on a button) and the external agent orders them randomly or chooses one incorrectly [Cao 2022];
- make the maintenance process more complex than required due to the presence of "dead dependencies" (dependencies to no longer existing resource entity) [Soto-Valero 2021].

We found that some past studies identify incorrect dependencies in a very specific context that would not be generally applicable. For example, in the work of Jafari [Jafari 2021], the "Pinned Dependency" smell —when one keeps referring to an old library version even if it was upgraded— is more related to libraries management practices than correctness of external dependency: the old library still exists, but it is considered bad project management practice to use it, what is incorrect is not the external dependency itself.

Our primary focus is on incorrect dependencies that go undetected by tools like compilers. For instance, consider embedded SQL queries that reference a missing table in a database. Since the SQL query is embedded as a string within the Java code, this type of incorrect external dependency isn't identified during compilation. Furthermore, as long as the method containing the query is not executed, the program runs without issue. The problem arises only when the method is executed, leading to a program failure.

### 5.2.2 Multiplicities

We saw in Section 2.1.3 in Chapter 2 several examples of incorrect external dependencies in the literature, for example when Pfeiffer and Wąsowski [Pfeiffer 2012] introduced the idea of multiplicity with errors for many-to-one external dependencies. According to them, an incorrect external dependency occurs when a required resource entity is missing, that is to say a mandatory minimal multiplicity of 1 is not met.

The idea of multiplicity is an obvious candidate to express possible restrictions on the external dependencies. However, it is not sufficient to look only for minimal multiplicity of one end (resource entity). We therefore decided to look at the minimal and maximal multiplicity at both ends of the external dependency. In each case, we give examples of resource and reference entities. Figure 5.1 illustrates this discussion:

**Optional entities** have a *minimum multiplicity* of $0$. This is the case of the reference entity in the figure.


Figure 5.1: Illustrating multiplicities in external dependencies

An example of optional *resource entity* is the case of HTML/CSS, a HTML tag (reference entity) may have an attribute `class="..."` referring to a nonexistent CSS class (resource entity). This is not considered an error because many HTML classes may exist that do not have corresponding CSS class.

An example of optional *reference entity* is that of a table in a database that is not used in any SQL query of a given project (although, if the table is not used in any query of any project, one might consider whether it is really needed).

**Mandatory entities** have a *minimum multiplicity* of 1. This is the case of the resource entity in the figure.

An example of a mandatory *resource entity* is for a database table used in an SQL query, the table must exist in the database or the query will not run.

An example of a mandatory *reference entity* could be in group 3 of our Hibernate example explained in Section 3.3.3 of Chapter 3: All Java persistent classes mapped to a database table should be used in the project. Not doing so would not produce any error, but it would probably be a bad smell (a dead class in the project).

**Unique entities** have a *maximum multiplicity* of 1. This is the case of the reference entity in the figure.

An example of a unique *resource entity* is server API: there can be only one service defined with the same signature on the server side.

An example of a unique *reference entity* is group 2 of our Hibernate example explained in Section 3.3.3 of Chapter 3: A Java persistence class can appear in only one mapping (it cannot be mapped to two different tables).

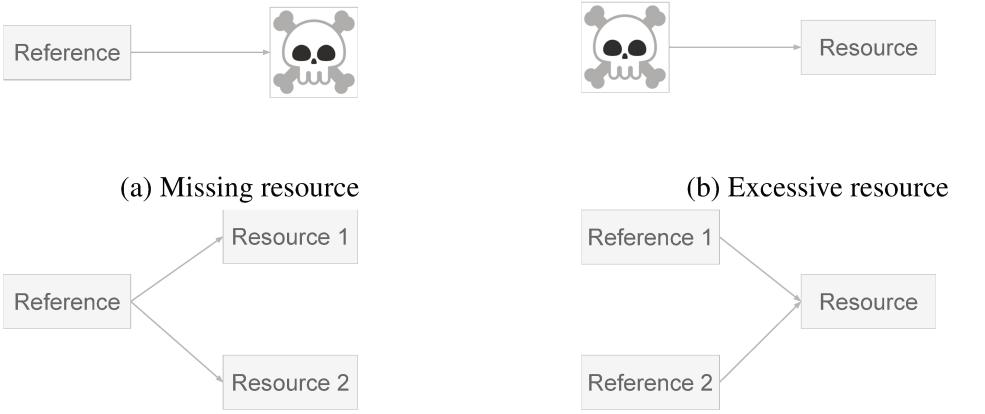
**Non-Exclusive entities** have a *maximum multiplicity* '$*$'. This is the case of the resource entity in the figure.

An example of a Non-Exclusive *resource entity* is in the Android framework, a drawable (ex: an image) used in a mobile app, must be saved in multiple folders (drawable-hdpi, drawable-ldpi) under the same name. The framework chooses the right *resource entity* at run time depending on the screen resolution of the terminal.

An example of a Non-Exclusive *reference entity* is when multiple SQL queries refer to the same database table.

### 5.2.3 Types of Incorrect Dependencies

Based on the previous description, we propose 4 generic types of incorrect dependencies corresponding to different cases when multiplicities are not respected. The types are illustrated in Figure 5.2.




(a) Missing resource

(b) Excessive resource

(c) Non-Exclusive resource

(d) Non-Exclusive reference

Figure 5.2: The four types of incorrect external dependencies

- When a minimum multiplicity is 1 (mandatory entity):
  - For an absent mandatory resource entity we name it a *Missing resource* (Figure 5.2a).
  - For an absent mandatory reference entity, we name it an *Excessive resource* (Figure 5.2b).
- When a maximum multiplicity is 1 (unique entity):
  - For violation of unique reference entity we name it a *Non-Exclusive Resource* (Figure 5.2c).
  - For violation of unique resource entity we name it a *Non-Exclusive Reference* (Figure 5.2d).

There are no incorrect dependency types for minimum multiplicity of 0 (optional) and maximum dependency of '$*$' (Non-Exclusive).

The identified incorrect external dependencies lead to various issues, though not all of the same severity. Some issues may cause compilation or runtime errors,

Table 5.1: Literrature external dependency errors and our classification

| | Missing Ressource | Excessive Resource | Non-exclu. Resource | Non-exclu. Reference |
|---|---|---|---|---|
| Mayer/Schroeder [Mayer 2012] | missing | - | - | - |
| Jafari et al. [Jafari 2021] | missing | unused | - | - |
| Cao et al. [Cao 2022] | missing | bloated | - | const. |
| Pfeiffer/Wąsowski [Pfeiffer 2011] | broken | - | - | - |
| Soto-Valero et al. [Soto-Valero 2021] | - | bloated | - | - |

while others are considered code smells. Our goal is to enable developers to detect these incorrect external dependencies to prevent such issues.

We now map our types to the incorrect external dependencies types to the types found in the literature (check Chapter 2 Section 2.2):

**Mayer and Schroeder:** In [Mayer 2012], Mayer and Schroeder introduce the idea of "Missing artifact". This corresponds to our *Missing resource* type and we actually took the term from them;

**Jafari et al.:** In [Jafari 2021] two dependency smells are discussed that can be mapped to our incorrect dependencies types: "Missing dependencies" map to our *Missing resource* type; "Unused dependency" maps to our *Excessive resource* type;

**Cao et al.:** In [Cao 2022], the authors have three types of dependency smells: "Missing dependency" maps to our *Missing resource* type; "bloated dependency" maps to our *Excessive resource* type, and; "Version Constraint Dependency" maps to our *Non-Exclusive reference* type;

**Pfeiffer and Wąsowski:** In [Pfeiffer 2012], our *Missing resource* type is called "Broken dependency";

**Soto-Valero et al.:** In [Pfeiffer 2012], the concept of "Bloated dependency" is proposed which maps to our *Excessive resource* type;

We found no mention in the literature of *Non-Exclusive resource*.

## 5.3 Incorrect Dependencies for different external agents

In this section, we present the various types of incorrect dependencies that can be found for different external agents mentioned in this thesis: GWT, RMI, Pharo comments and Hibernate.

### 5.3.1 Incorrect Dependencies in GWT

We present in this section for GWT framework, the types of incorrect dependencies that can be found between Java annotations (reference entities) in Java classes and XML elements (resource entity) in the XML layout files, according to our classification introduced previously. These types are summarized in Table 5.2 ("-" means the error type does not apply).

Table 5.2: Incorrect dependencies between Java annotations (reference entities) and XML elements (resource entities) in GWT

| Missing Resource | Excessive Resource | Non-exclu. Resource | Non-exclu. Reference |
|---|---|---|---|
| error | - | error | error |

We explain now for those external dependencies which type of incorrect dependencies might affect them and why:

- The minimum multiplicity for the *resource entity* (XML element) is 1, the element referred must exist or it would be a *Missing resource*;
- The maximum multiplicity for the *resource entity* (XML element) is 1, each XML element must have a unique ID or it would be a *Non-Exclusive resource*;
- The minimum multiplicity for the *reference entity* (Java annotation) is 0, not all XML elements need to be referred by Java annotations, as not all of those elements must be necessarily handled by Java, so no possible *Excessive resource*;
- The maximum multiplicity for the *reference entity* (Java annotation) is 1: the same element must be referred only once by each Java annotation (once by using `@UiHandler` and/or once by using `@UiField`), or it would be a *Non-Exclusive reference*;

From this, we deduce the following possible incorrect dependency: *Missing resource*, *Non-Exclusive resource* and *Non-Exclusive reference*.

### 5.3.2 Incorrect Dependencies in RMI

We will now present the types of incorrect dependencies for RMI, between the API services (resource entities) defined on the server side and the client invocations (reference entities) for these services, according to our classification. These types are summarized in Table 5.3 ("smell" means the type is not correct according to

Table 5.3: Incorrect dependencies between client ans server in RMI

| Missing Resource | Excessive Resource | Non-exclu. Resource | Non-exclu. Reference |
|---|---|---|---|
| error | smell | - | - |

the definition of the external agent but does not produce an error, and "-" means the incorrectness type does not apply).

We explain now for those external dependencies which type of incorrect dependencies might affect them and why:

- The minimum multiplicity for the *resource entity* (API service) is 1, the API service referred must exist or it would be a *Missing resource*;
- The maximum multiplicity for the *resource entity* (API service) is 1, the API service must be unique or it would be a *Non-Exclusive resource*. However this is handled in RMI as the services are defined by the interfaces where Java does not allow defining the same method signature more than one time in the same interface, therefore this cannot happen;
- The minimum multiplicity for the *reference entity* (client invocation) is 1, as each defined service must be referred by the client, otherwise, the program becomes larger unnecessarily and this would be an *Excessive resource* smell;
- The maximum multiplicity for the *reference entity* (client invocation) is $*$, the same server API can be referred as much as needed by the client program, therefore no possible *Non-Exclusive reference*.

From this, we deduce the following possible incorrect dependency: *Missing resource* and *Excessive resource*.

### 5.3.3 Incorrect Dependencies in Pharo Comments

We will now present the examples of incorrect dependencies between the referred classes (reference entities) in comments and Pharo actual classes (resource entities) according to our classification. These examples are summarized in Table 5.4 ("-" means the error type does not apply).

We explain now for those external dependencies which type of incorrect dependencies might affect them and why:

- The minimum multiplicity for the *resource entity* (Pharo class) is 1, the class referred must exist or it would be a *Missing resource*;

Table 5.4: Incorrect dependencies between Pharo classes and comments

| Missing Resource | Excessive Resource | Non-exclu. Resource | Non-exclu. Reference |
|---|---|---|---|
| error | - | - | - |

- The maximum multiplicity for the *resource entity* (Pharo class) is 1 or it would be *Non-Exclusive resource*. However, Pharo does not allow creating 2 classes with the same name, therefore this cannot happen.

- The minimum multiplicity for the *reference entity* (comment of a class) is 0, not all classes need to be referred by the comments; sometimes when a class is created it can be commented and referred using text like "this class is doing ...", therefore no possible *Excessive resource*.

- The maximum multiplicity for the *reference entity* (comment of a class) '$*$': the same class can be referred multiple times whether in the same comment or others, specially when it has a large usage like `Dictionary` class, therefore no possible *Non-Exclusive reference*.

From this, we deduce the following possible incorrect dependency: *Missing resource*.

### 5.3.4 Incorrect Dependencies in Hibernate

We will now present the types of incorrect dependencies in Hibernate according to the our classification. These types are summarized in Table 5.5 ("smell" means the type is not correct according to the definition of the external agent but does not produce an error, and "-" means the incorrectness type does not apply).

Table 5.5: Incorrect dependencies of Hibernate

| | Missing Ressource | Excessive Resource | Non-exclu. Resource | Non-exclu. Reference |
|---|---|---|---|---|
| Group 1 | error | - | - | smell |
| Group 2 | error | - | - | smell |
| Group 3 | error | smell | - | - |
| Group 4 | error | - | - | - |

We look at examples in different groups, explaining for each which type might affect them and why:

**Group 1:** These are the dependencies between the XML elements (reference entities) in the `.hbm.xml` mapping file to the database SQL tables (resource entities).

- The minimum multiplicity for the *resource entity* (database table) is 1, the mapped table must exist or it would be a *Missing resource*;
- The maximum multiplicity for the *resource entity* (database table) is 1 or it would be *Non-Exclusive resource*. However Oracle does not allow creating the same table more than once, except for different users. Therefore this cannot happen.
- The minimum multiplicity for the *reference entity* (XML element) is 0, not all tables need to be mapped in a specific project, therefore no possible *Excessive resource*.
- The maximum multiplicity for the *reference entity* (XML element) is 1: It is preferable to apply a single mapping to a database table in Hibernate, but it also allows several mappings to the same table. However this could be considered a bad practice in some situations, the developers might want to have it reported as a bad smell and it would be *Non-Exclusive reference*.

From this, we deduce the following possible incorrect dependencies: *Missing resource*, and *Non-Exclusive reference* (possible smell).

**Group 2:** These are the dependencies between the XML elements (reference entities) in the `.hbm.xml` mapping file to the Java persistent classes (resource entities).

- The minimum multiplicity for the *resource entity* (Java class) is 1, the class mapped must exist or it would be a *Missing resource*.
- The maximum multiplicity for the *resource entity* (Java class) is 1 or it would be *Non-Exclusive resource*. However, Java does not allow creating 2 classes with the same name, therefore this cannot happen.
- The minimum multiplicity for the *reference entity* (XML element) is 0, not all Java classes are mapped, only the persistent ones (since we did not consider the newer version of Hibernate with annotation in our example, there is no easy way to know which classes are persistent classes), therefore no *Excessive resource*.
- The maximum multiplicity for the *reference entity* (XML element) '$*$': There can be several mappings (thus several tables) for one class, but again this should probably be considered a bad smell, therefore a *Non-Exclusive reference*.

From this, we deduce the following possible incorrect dependencies: *Missing resource*, and *Non-Exclusive reference* (possible smell).

**Group 3:** These are the dependencies between the injected class names (reference entities) in HQL queries and the Java persistent classes (resource entities).

- The minimum multiplicity for the *resource entity* (Java class) is 1, the class used (queried) must exist or it would be a *Missing resource*.
- The maximum multiplicity for the *resource entity* (Java class) is 1, a Java persistence class name in a query must be non-ambiguous (refer to exactly one persistent class) or it would be *Non-Exclusive resource*. However, Java does not allow creating 2 classes with the same name, therefore this cannot happen.
- The minimum multiplicity for the *reference entity* (class name in HQL query) is 1: it is not mandatory in Hibernate that all persistent classes be used in queries, but not doing so can be considered a bad smell as the unused persistent classes are dead code*Excessive resource*;
- The maximum multiplicity for the *reference entity* (class name in HQL query) is '$*$', several queries can use the same persistent class, therefore no *Non-Exclusive reference*.

From this, we deduce the following possible incorrect dependencies: *Missing resource* and *Excessive resource* (possible smell).

**Group 4:** These are the dependencies between the injected class names (reference entities) in HQL queries and the SQL database tables (resource entities) that can be deduced from the "transitive closure" of the three other groups.

- The minimum multiplicity for the *resource entity* (database table) is 1, the table indirectly used (through the persistent class) in a query must exist or it would be a *Missing resource*.
- The maximum multiplicity for the *resource entity* (database table) is 1, a table name indirectly referred to in a query must be non-ambiguous (it must reference exactly one table) or there would be *Non-Exclusive resource*. However, we have seen previously that this is not the case, therefore it would not happen.
- The minimum multiplicity for the *reference entity* (HQL query) is 0, not all tables need to be (indirectly) used in queries, so no *Excessive resource*.
- The maximum multiplicity for the *reference entity* (HQL query) is '$*$', several queries can (indirectly) refer to the same table, so no *Non-Exclusive reference*.

From this, we deduce the following possible incorrect dependency: *Missing resource*.

In conclusion, it is evident that all four types of incorrect dependencies are deduced either from existing literature or from among variant external agents. Note that three types of incorrect dependencies are represented in past literature, and 1 type (Non-Exclusive Resource) was never considered.

## 5.4 Incorrect Dependencies Detection with Adonis

In this section, we first remind the readers of Adonis implementation already presented in Section 3.5 of Chapter 3, then we explain how this implementation was adapted and how we use it to detect not only correct but also incorrect external dependencies.

### 5.4.1 External Dependencies Detection with Adonis

Adonis is developed in Pharo and performs a static analysis to detect external dependencies. It has heuristics to link each reference entity to the corresponding resource entity.

- Adonis follows a two steps process where it searches first for containers holding resource/reference entities. This is done by using a pattern matching language in Pharo named MoTion (see Chapter 4), that facilitates traversing the models and the Abstract Syntax Trees (ASTs) when necessary.
- Once these containers are identified, it continues to search for entities through pattern matching.
- Adonis offers a "library" of heuristics (ie. searching patterns) for each external agent (GWT, RMI,...). Each specific external agent is handled by a class implementing the methods `#getReferenceContainers` and `#getReferenceEntities`.
- The result is a collection of `ExternalDependency` objects, each containing a `referenceEntity` and a `resourceEntity` objects.
- Adonis also offers a library of "atomic" patterns that can be reused between external agents. For example, a pattern designed to search over XML files based on a specific attribute can be applied to configuration files such as `pom.xml` for Maven projects, as well as UI files in Android and GWT projects.

### 5.4.2 Adaptation of Adonis to Reveal Incorrect Dependencies

We modified Adonis to handle both correct and incorrect external dependencies in two phases:

1. First Step: Adonis searches for all reference and resource entities and organizes them into a collection of `ExternalDependency` objects.

2. Second Step: Adonis applies filters per each external agent, resulting in two main collections: `detectedIncorrectExternalDependencies` which contains incorrect external dependencies, and `detectedExternalDependencies` which contains correct external dependencies as initially implemented (see above 3.5.1).

To generate the second collection, Adonis evaluates the first-step results per each object: when resource entity is nil, this indicates that the type of incorrect dependency is *Missing resource*. When reference entity is nil, this indicates that the type of incorrect dependency is *Excessive resource*. When duplicated references are found, this indicates that the incorrect dependency is of type *Non-Exclusive reference*. When duplicated resources are found, this indicates that the incorrect dependency found is of type of *Non-Exclusive resource*. The rest is considered correct external dependencies. These filters are applied per external agent, for example for RMI Adonis applies filters to search for *Missing resource* and *Excessive resource* only.

We created a class to represent each type of incorrectness and we explain below this implementation:

**Missing Resource:** We report reference entities without a corresponding resource entity when this is not allowed (for example a comment with a reference to a class that does not exist). To represent it, we created a new class `MissingResourceDependency` which contains only a `ReferenceEntity` object.

**Excessive resource:** We report resource entities without any corresponding reference entity when this does not make any sense (for example a server API that was created but not invoked by any client). To represent it, we created a new class `ExcessiveResourceDependency` which contains only a `ResourceEntity` object.

**Non-Exclusive resource:** We report multiple resource entities linked to one reference entity when it is not allowed (for example in GWT we cannot have duplicate XML element IDs in the same XML layout file). To represent it, we created a new class `NonExclusiveResourceDependency` which contains a `ReferenceEntity` object and a collection of `ResourceEntity` objects.

**Non-Exclusive reference:** We report multiple reference entities linked to one resource entity when it is not allowed (for example having duplicate Java annotations in the same Java class). To represent it, we created a new class `NonExclusiveReferenceDependency` which contains a `Resource Entity` object and a collection of `ReferenceEntity` objects.

## 5.5 Evaluating Incorrect External Dependencies Detection

In this section we explain how we conducted several experiments in order to validate if our modification helps revealing incorrect dependencies. So we first explain how the validation process was conducted; then we run Adonis and we try to reveal the results to be compared to what we extracted manually, then we end up by discussing the limitations of our approach.

### 5.5.1 Experiment setup

In this section we explain how we chose the projects and how we run the experiments.

Concerning the list of projects, we are using the same ones listed in Chapter 3, Section 3.6. In our initial analysis, we manually identified the existence of incorrect external dependencies, and now we seek to validate Adonis against those findings. To validate that "our modification helps revealing incorrect dependencies", we will rely on four metrics: we will count manually the number of incorrect dependencies for all four types identified previously in Section 5.2.3 and we will run Adonis on the projects and compare its results with our findings by measuring the recall and precision ratios. For the projects which we did not find certain types of incorrect external dependencies, but we know that these types could be found for the external agents used to develop them, we will inject them manually and see if Adonis is able to detect the expected injections.

### 5.5.2 Manual findings for projects

In this section we present our findings (number of *Missing resources*, *Excessive resources*, *Non-Exclusive resources* and *Non-Exclusive references*) for each project following a manual counting.

Concerning the counting of incorrect external dependencies for **GWT**, we expect to find incorrect dependencies of type *Missing resource*, *Non-Exclusive resource* and/or *Non-Exclusive reference*. However, following the experiment we found 0 incorrect dependencies in Traccar project. To be able to proceed with the validation process, we decided to manually inject such incorrect dependencies as per below explanations:

Table 5.6: Incorrect dependencies detection for public projects

| External agent | Incorrectness type | Original counting | After injection |
|---|---|---|---|
| GWT | Missing resource | 0 | 4 |
| | Excessive resource | - | - |
| | Non-Exclusive resource | 0 | 4 |
| | Non-Exclusive reference | 0 | 1 |
| RMI | Missing resource | 0 | 2 |
| | Excessive resource | 29 | 29 |
| | Non-Exclusive resource | - | - |
| | Non-Exclusive reference | - | - |
| Pharo Comments | Missing resource | 69 | 69 |
| | Excessive resource | - | - |
| | Non-Exclusive resource | - | - |
| | Non-Exclusive reference | - | - |

- *Missing resource:* We selected randomly two XML files out of 10 (UserSettingsDialog.ui.xml and UserDialog.ui.xml) and we removed four XML elements already detected as resource entities for external dependencies which IDs are "saveButton" and "cancelButton" in each file. Each one of them is referred once using `@UiHandler` key. This results in existence of four incorrect dependencies in our project of type *Missing resource*.

- *Excessive resource:* This type is not possible with GWT (see Section 5.3.1).

- *Non-Exclusive resource:* We selected randomly two XML files (ArchiveView.ui.xml and DeviceDialog.ui.xml) out of 10 and duplicated four XML elements which IDs are "saveButton", "cancelButton", "loadButton" and "clearButton". Each one of them is already referred once, so after the injection, we have now four incorrect external dependencies of type *Non-Exclusive resource*.

- *Non-Exclusive reference:* We selected randomly one Java class (UsersDialog) out of 10 and duplicated the references by using `@UiHandler` key for "addButton". We could not use `@UiField` because this key is not referring the XML element by embedding their names as String, instead it is relying on defining instances which cannot be duplicated in the same Java class. Now we have one new incorrect external dependency of type *Non-Exclusive reference*.

For **RMI**, the incorrect dependencies expected to be found are *Missing resource* and/or *Excessive resource*. The server APIs are intended for external use by the

client and are typically not used internally by the server. We counted 29 incorrect external dependencies of type *Excessive resource* in UniScore project, leading to a percentage of 37% of external dependencies that are incorrect in the project (29 incorrect / (29 incorrect + 48 correct) * 100). Those findings are the result following of our manual count. We did not find *Missing resource* dependencies. In order to proceed with the validation process, we injected such incorrectness manually. So we removed two method signatures from the interface on the server side (we could do it because both client and server are separated) and kept the calls to them from the client side. Now we expect to find for UniScore project two additional incorrect dependencies of type *Missing resource*.

For **Pharo Comments**, the incorrect dependencies expected to be found are of type *Missing resource*. Usually this type of incorrect dependencies is common in Pharo comments because of upgrades, where some classes were removed, but the corresponding comments were not updated. Although this issue does not impact Pharo's functionality, it can mislead developers by suggesting the presence of non-existent classes. We found 69 references in the comments without a resource, thus incorrect external dependencies of type *Missing resource*. After our investigation, we found that these incorrect external dependencies result because of removed classes between Pharo versions, or because also referring to button names using single backticks and CamelCase syntax, same as for referring classes.

For **Hibernate**, finding incorrect external dependencies is specific per each group. Table 5.7 presents the results of our manual finding. The cases that are not possible for Hibernate (as shown in Table 5.5) are marked as "-". The first column labeled as *Group*, contains the name of each type of group in hibernate. The second column labeled as *Incorrect dependency type* indicates the various types of incorrect external dependencies, as identified in Section 5.2. The third column contains the number found for each type. After our first report, the company updated its database and added additional tables. The numbers in parentheses are the new values obtained after getting a second version of the SQL schema model.

For **Hibernate Group 1**, we found a total of nine incorrect dependencies:

- *Missing resource:* We found nine *missing resource* incorrect dependencies. Those findings are for references from the mapping files to the database SQL tables. After getting the second version of the database, we found that seven SQL tables were created, reducing by that the number of *missing resources* incorrect dependencies to two.

- *Excessive resource:* This type is not possible Hibernate Group 1 (see Section 5.3.4).

- *Non-Exclusive resource:* This type is not possible Hibernate Group 1 (see Section 5.3.4).

- *Non-Exclusive reference:* We found 0 incorrect external dependencies of this type, meaning that every table in the database is mapped once. We selected

Table 5.7: External dependencies case studies of manual counting

| Dependency type | Incorrect dependency type | Total |
|---|---|---|
| Group 1 | Missing Resource | 9 (2) |
| | Excessive Resource | - |
| | Non-Exclusive Resource | - |
| | Non-Exclusive Reference | 0 |
| Group 2 | Missing Resource | 2 |
| | Excessive Resource | - |
| | Non-Exclusive Resource | - |
| | Non-Exclusive Reference | 0 |
| Group 3 | Missing Resource | 12 |
| | Excessive Resource | 2 |
| | Non-Exclusive Resource | - |
| | Non-Exclusive Reference | - |
| Group 4 | Missing Resource | 58 (12) |
| | Excessive Resource | - |
| | Non-Exclusive Resource | - |
| | Non-Exclusive Reference | - |

randomly one mapping file and duplicated it. By that we duplicate the reference to the same SQL table. Now we have one new incorrect external dependency of type *Non-Exclusive reference* for this group.

For **Hibernate Group 2**, we found a total of two incorrect dependencies:

- *Missing resource:* We found two *missing resource* incorrect dependencies. Those findings are for references from the mapping files to two missing Java classes.
- *Excessive resource:* This type is not possible Hibernate Group 2 (see Section 5.3.4).
- *Non-Exclusive resource:* This type is not possible Hibernate Group 2 (see Section 5.3.4).
- *Non-Exclusive reference:* We found 0 incorrect external dependencies of this type, meaning that each mapped Java class is referred once only, and each mapped table is referring to a single Java class. However, to validate our approach, we selected randomly one mapping file and duplicated it. By that we duplicate the reference to the same Java class. Now we have one new incorrect external dependency of type *Non-Exclusive reference* for this group.

For **Hibernate Group 3**, we found a total of 14 incorrect dependencies:

- *Missing resource:* We found 12 *Missing resource* incorrect dependencies. Those findings are for 12 queries, that were actually SQL queries, not passing through Hibernate but referring to the actual names of the tables in the database.

- *Excessive resource:* We found two incorrect external dependencies of type *Excessive resource*. This means that two Java classes were mapped but never used by the HQL queries. By checking more deeply, we realized that these two mapped classes, are actually the ones that are not mapped correctly to the database. Following further discussions with the developers from the private company, we found that these are old tables that were removed, but the developers forgot to remove their mappings from the Java program. They will not cause runtime issues, but they increase the program size unnecessarily and mislead developers.

- *Non-Exclusive resource:* This type not possible for Hibernate Group 3 (see Section 5.3.4).

- *Non-Exclusive reference:* This type not possible for Hibernate Group 3 (see Section 5.3.4).

Finally for **Hibernate Group 4**, where external dependencies are established through the previous groups, we found a total of 58 incorrect dependencies:

- *Missing resource:* We found 58 incorrect dependencies of type *missing resource* incorrect dependencies. With further investigation, we found 46 queries referring correctly to Java persistency classes, but these latter ones are not mapped correctly to database tables having nine references from the mapping files with *Missing resources* (Group 1 initial results). And we found 12 queries (from group 3), referring directly to the table names disregarding the mappings. This is an expected behavior for old systems where Hibernate was introduced late in the development process. After the update of the database, and adding missing tables, this number of *Missing resource* incorrect dependencies was reduced to 12 (row 1 of Group 4), because the classes referred by the queries are now mapped to existing tables in the database, keeping only queries with direct references to tables which is a flaw when using Hibernate.

- *Excessive resource:* This type is not possible for Hibernate Group 4 (see Section 5.3.4).

- *Non-Exclusive resource:* This type is not possible for Hibernate Group 4 (see Section 5.3.4).

- *Non-Exclusive reference:* This type is not possible for Hibernate Group 4 (see Section 5.3.4).

The numbers in parentheses reflect the results after updating the SQL schema model to include additional tables. SQL tables represent the resource entities for groups 1 and 4. The inclusion of the *Missing resources* in the updated schema reduced the number of incorrect dependencies to two for group 1 and 12 for group 4. Note that this update could not and did not correct the number of *Missing Resources* and *Excessive Resources* in groups 2 and 3. For this they would have to change their `.hbm.xml` files and/or Java code. Our findings for the initial status of the projects before injection is found here [1].

### 5.5.3 Results of the Experiment

In this section, we will present the results of incorrect external dependencies counting using Adonis. We compare those results to the manual findings in previous section and deduce the precision and recall ratios for incorrect external dependencies counting.

For GWT, RMI and Pharo comments, we had exactly the same results as shown in table 5.6. Since those results revealed with Adonis and Pharo conform to the manual counting, with no false positive results neither true negative, this leads to a precision and recall ratios of 100%.

However for Hibernate, the results were not the same for groups 3 and 4 as shown in Table 5.8 due to a limitation in our approach. This limitation is because the analysis would take all strings and also look for names of mapped class names in them, which limits the findings for *Missing Resources*.

We see in table 5.9 that most of the results are perfect (100% precision and recall). The few lesser recalls (groups 3 and 4) for *Missing resource* are due to the static analysis of the queries that are built from string concatenation. We knew that this was a limitation of our analysis, but it is difficult to do better with a statical analysis. Putting a spy on the database to intercept all incoming queries (as done by [Meurice 2016]) would ease this identification, but as all dynamic analyses, it would then depend on the scenarios ran being exhaustive (ie. invoking all possible queries). This is usually something that is also difficult to guarantee.

In summary, we adapted Adonis to detect incorrect external dependencies associated with various external agents. After running the tool and performing a manual verification, the results matched our expectations with a major limitation for *Missing resources* identification for group 3, but a recall and precision ratios varying between 79% and 100% for the others.

## 5.6 Threats to validity

In this section we present the threats to validity following the experiments and their validation.

[1] Manual counting results https://doi.org/10.5281/zenodo.13820410

Table 5.8: External dependencies case studies using Adonis

| Dependency type | Incorrect dependency type | Total |
|---|---|---|
| Group 1 | Missing Resource | 9 (2) |
| | Excessive Resource | - |
| | Non-Exclusive Resource | - |
| | Non-Exclusive Reference | 0 |
| Group 2 | Missing Resource | 2 |
| | Excessive Resource | - |
| | Non-Exclusive Resource | - |
| | Non-Exclusive Reference | 0 |
| Group 3 | Missing Resource | 0 |
| | Excessive Resource | 2 |
| | Non-Exclusive Resource | - |
| | Non-Exclusive Reference | - |
| Group 4 | Missing Resource | 46 (0) |
| | Excessive Resource | - |
| | Non-Exclusive Resource | - |
| | Non-Exclusive Reference | - |

Table 5.9: Incorrect dependencies of external agents

| | Missing Resource | Excessive Resource | Non-exclu. Resource | Non-exclu. Reference |
|---|---|---|---|---|
| | Pre./Rec. | Pre./Rec. | Pre./Rec. | Pre./Rec. |
| G.1 | 100% / 100% | - | - | 100% / 100% |
| G.2 | 100% / 100% | - | - | 100% / 100% |
| G.3 | 0% / 0% | 100% / 100% | - | - |
| G.4 | 100% / 79% | - | - | - |

**Internal Validity** In this chapter, we presented a list of incorrect external dependencies and their impact (errors or potential smells) that can result from them. While these are real ones that need to be discovered by the developers, further exploration with additional external agents is needed to see if more errors may emerge. We selected public and industrial projects that we ignore how they were developed, thus we ignore the existence of incorrect external dependencies. Our investigations (manual and using Adonis) proved the existence of some incorrect dependencies, but not all the types existed in the projects we chose. This is why we decided to go for random manual injection of the uncovered types of incorrect external dependencies and detect them using Adonis.

**External Validity** We chose a variety of projects, real concrete and with various sizes, in order to check if incorrect external dependencies exist. This variety, proved that despite the size of the project, incorrect external dependencies exist with ratios varying between a minimum of 0% for GWT and a maximum of 37% for RMI.

**Construct Validity** We used metrics directly related to our research regarding incorrect external dependencies. We counted *Missing resource*, *Excessive resource*, *Non-Exclusive resource* and *Non-Exclusive reference* and checked the ratios for precision and recall for each project. To count the results manually, we relied for small projects on counting the files and checking inside the references. For large projects, we proceeded differently. We relied for Pharo on extracting programmatically the comments and place them into Google sheets. We used one script to extract the references to classes and another one to create rows in excel and place values in cells. Then we used simple formulas to do the counting. And for Hibernate, we followed a trick provided by developers to check the queries only inside Java classes that start with "Dao". So we extracted them and put them again in Google Sheet and counted using formulas.

**Conclusion Validity** Regarding conclusion validity, our approach demonstrated success in detecting incorrect external dependencies across the analyzed projects, with a major limitation for Hibernate Group 3. However, to strengthen our statistical conclusions, we need to expand our analysis to a larger range of projects, and cover additional external agents to see if the 4 identified types of incorrect external dependencies exist.

## 5.7 Conclusion

In this chapter, we focused on identifying and detecting incorrect external dependencies. To achieve this, we proposed linking the correctness of external dependen-

cies to the minimal and maximal multiplicities of the entities involved, introducing four 4 types of incorrect external dependencies based on these multiplicities. We validated our approach by categorizing prior research and examples from various external agents.

Additionally, we enhanced Adonis, an open-source and extensible tool, to include the detection of incorrect external dependencies for the supported external agents. Upon running the tool, we successfully identified existing incorrect external dependencies, though not all of them existed in the provided projects. To thoroughly validate our approach, we applied targeted modifications to inject more incorrect external dependencies, re-ran Adonis, and compared the results to assess the tool's effectiveness in detecting the injections.

Finally, to validate the accuracy of our results, we manually counted all identified incorrect dependencies. We achieved perfect precision for many types of dependencies, although we did encounter some limitations.

CHAPTER 6

# Conclusion And Future Work

## Contents



In this chapter, we summarize the findings of our research and we present the future work following these findings.

## 6.1 Summary

In this thesis, we focus on detecting external dependencies, established between resource and reference entities through external agents like frameworks. We developed Adonis, a tool that identifies such dependencies, and tested it on various projects using different external agents.

Throughout our research, we encountered two main challenges: locating the entities forming the external dependencies and evaluating their validity, including missing, unused, or other issues.

To address the first challenge, we created MoTion, a declarative pattern matching tool in Pharo. MoTion combines features of graph and object matching, allowing the creation of flexible patterns that enable lexical and object matching. We used MoTion to define patterns that Adonis relies on to search for the relevant entities. For each external agent, we chosed lexical or object matching based on our needs.

Finally, we enhanced Adonis to detect both correct and incorrect external dependencies based on multiplicity. Revealing incorrect dependencies is crucial to help developers avoid potential negative impacts, such as runtime or compile-time errors, or subtle issues like code smells that could mislead developers in understanding the true nature of external dependencies. We validated Adonis across various external agents and projects, demonstrating its effectiveness in identifying both correct and incorrect dependencies.

In the following lines, we summarize the research problems discussed in this thesis and our contributions.

**Chapter 2**

- We presented the existing approaches for detecting external dependencies;
- We categorized the tools following different categories: dependency type, analysis technique, matching strategy and searching engine;
- We presented the tools that considered correctness of external dependencies in the literature and listed the types found;
- We also explored pattern matching tools for both graph matching and object matching and summarized their features.

**Chapter 3**

- We explored external dependencies in details and we listed four categories of them;
- We showed how external dependencies are established in external agents;
- We explained our approach to search for containers and entities, and how we can make an automated detection tool easily extendible by reusing predefined patterns;
- We showed our tool, Adonis, how ir was implemented following our approach and how it can be used, then we validate it on private and public projects and discussed the results.

**Chapter 4**

- We presented traditional pattern matching in Pharo and gave examples of it;
- We presented MoTion, a new declarative pattern matching language that can be used to search for tokens in objects;
- We explained the implementation and showed how it can be extendible to cover additional features or recover existing ones;
- We made a comparison between MoTion and the traditional matching and show the limitations of the latter;
- We listed the use cases of MoTion including how it is used in Adonis, and from these use cases we learned new lessons and got feedback from the users.

**Chapter 5**

- We explored incorrect external dependencies and explained how we can detect them based on multiplicity. From this we derived 4 different types of incorrect external dependencies;
- We showed for each external agent in study, what types of incorrect external dependencies may exist;
- We explained how we adapted Adonis to return all the results needed including correct and incorrect external dependencies;
- We validated the adapted Adonis on the same public and private projects Chapter 3 and discussed the results.

## 6.2 Future work

In this section, we present several perspectives that push us forward to continue our research concerning the external dependencies extraction and the pattern matching.

### 6.2.1 Pattern Matching

**Debugging** Debugging patterns is a well known challenge in the field of pattern matching. In MoTion, false matches can occur when the user incorrectly expresses a pattern, meaning the pattern does not accurately reflect their intended match criteria. This can lead to unexpected results, making it difficult for users to identify where the issue lies. This is the main request raised by the users of MoTion. To address this, we begun implementing basic debugging tools to assist users in pinpointing errors. That relies on commenting the parts of the patterns (mainly properties) an rerun the match again with less description (because less properties are used), thus higher chances of match. However, these solutions are just a starting point, and further research is necessary to provide more comprehensive support for identifying and resolving issues while pattern matching with MoTion.

**Limited recursive search** We are facing also another issue with MoTion, where looping takes long time in some cases when doing a recursive search. This is frequent when matching ASTs for example or for large models when using *Recursive traversal*. To stop that, we suggest to add a new feature that allows precising the number of recursive traversals, to set the limit of looping.

### 6.2.2 External Dependencies

**Artificial Intelligence as a searching engine?** We have seen that previously in the literature, researchers relied on lexical matching or object matching. Whereas we rely on both, by choosing the matching based on our needs. For example for the comments we chose lexical matching, since it was not worth it to parse comments in Pharo where regular expressions were enough. We did the same for HQL queries in Hibernate. However, in such cases, it would be worth to try using the Artificial Intelligence to have more accurate results and make sure we do not miss any information, like table name available in a String alone without any marker in comments without CamelCase format. We did minor experiment on AI platforms like ChatGPT, where we extracted the whole method from a Java class, containing concatenated HQL strings, and asked the platform to reveal the table names from them. We also extracted a complete comment for a Pharo class and asked the platform to tell us where the class names were in these comments. We got promising results, and we believe this is a new research perspective that should be explored in the future.

**Additional external agents?** Future work will focus also on extending our tool to support a wider range of external agents, such as Spring and JDBC. This expansion will allow us to validate the concept of reusable patterns, particularly as these agents cover languages like XML (previously addressed through GWT and Hibernate) and SQL (previously addressed through Hibernate). Additionally, it will provide an opportunity to further test the robustness of Adonis and assess the validity of its results for correct and incorrect external dependencies.

# Bibliography


[Angles 2017] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter and Domagoj Vrgoč. *Foundations of modern query languages for graph databases*. ACM Computing Surveys (CSUR), vol. 50, no. 5, pages 1–40, 2017.

[Angles 2018] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda*et al.* *G-CORE: A core for future graph query languages*. In Proceedings of the 2018 International Conference on Management of Data, pages 1421–1432, 2018.

[Anquetil 2020] Nicolas Anquetil, Anne Etien, Mahugnon Honoré Houekpetodji, Benoît Verhaeghe, Stéphane Ducasse, Clotilde Toullec, Fatija Djareddir, Jérôme Sudich and Mustapha Derras. *Modular Moose: A new generation of software reengineering platform*. In International Conference on Software and Systems Reuse (ICSR'20), numéro 12541 de LNCS, pages 119–134, December 2020.

[Anquetil 2022] Nicolas Anquetil, Miguel Campero, Stéphane Ducasse, Juan-Pablo Sandoval and Pablo Tesone. *Transformation-based Refactorings: a First Analysis*. In International Workshop of Smalltalk Technologies, 2022.

[Balland 2007] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau and Antoine Reilles. *Tom: piggybacking rewriting on java*. In Proceedings of the 18th international conference on Term rewriting and applications, RTA'07, pages 36–47, Berlin, Heidelberg, 2007. Springer-Verlag.

[Black 2009] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou and Marcus Denker. Pharo by example. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[Cao 2015] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna and Yan Chen. *EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework.* In NDSS, 2015.

[Cao 2022] Yulu Cao, Lin Chen, Wanwangying Ma, Yanhui Li, Yuming Zhou and Linzhang Wang. *Towards better dependency management: A first look at dependency smells in python projects*. IEEE Transactions on Software Engineering, 2022.

[Consortium 2013] World Wide Web Consortium*et al.* *SPARQL 1.1 overview*. 2013.

[Consortium 2014] World Wide Web Consortium*et al.* *RDF 1.1 concepts and abstract syntax*. 2014.

[Cossette 2010] Brad Cossette and Robert J Walker. *DSketch: Lightweight, adaptable dependency analysis*. In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, pages 297–306, 2010.

[Deutsch 2020] Alin Deutsch, Yu Xu, Mingxi Wu and Victor E Lee. *Aggregation support for modern graph analytics in TigerGraph*. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pages 377–392, 2020.

[Deutsch 2022] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels*et al.* *Graph pattern matching in GQL and SQL/PGQ*. In Proceedings of the 2022 International Conference on Management of Data, pages 2246–2258, 2022.

[Di Grazia 2023] Luca Di Grazia and Michael Pradel. *Code search: A survey of techniques for finding code*. ACM Computing Surveys, vol. 55, no. 11, pages 1–31, 2023.

[Francis 2018] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer and Andrés Taylor. *Cypher: An evolving query language for property graphs*. In Proceedings of the 2018 international conference on management of data, pages 1433–1445, 2018.

[Grichi 2020] Manel Grichi, Mouna Abidi, Fehmi Jaafar, Ellis E Eghan and Bram Adams. *On the impact of interlanguage dependencies in multilanguage systems empirical case study on java native interface applications (JNI)*. IEEE Transactions on Reliability, vol. 70, no. 1, pages 428–440, 2020.

[Hecht 2018] Geoffrey Hecht, Hafedh Mili, Ghizlane El-Boussaidi, Anis Boubaker, Manel Abdellatif, Yann-Gaël Guéhéneuc, Anas Shatnawi, Jean Privat and Naouel Moha. *Codifying hidden dependencies in legacy J2EE applications*. In 2018 25th Asia-Pacific Software Engineering Conference (APSEC), pages 305–314. IEEE, 2018.

[Hosry 2023a] Aless Hosry and Nicolas Anquetil. *External Dependencies in Software Development*. In International Conference on the Quality of Information and Communications Technology, pages 215–232. Springer, 2023.

[Hosry 2023b] Aless Hosry, Vincent Aranega and Nicolas Anquetil. *Pattern matching in Pharo*. In International Workshop on Smalltalk Technology (IWST'23), Grenoble, France, August 2023.

[Hosry 2024] Aless Hosry, Vincent Aranega and Nicolas Anquetil. *MoTion: A new declarative object matching approach in Pharo*. Journal of Computer Languages, page 101290, 2024.

[Imbugwa 2021] Gerald Birgen Imbugwa, Luiz Jonatã Pires de Araújo, Mansur Khazeev, Ewane Enombe, Harrif Saliu and Manuel Mazzara. *A case study comparing static analysis tools for evaluating SwiftUI projects*. In Journal of Physics: Conference Series, volume 2134, page 012022. IOP Publishing, 2021.

[Jafari 2021] Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab and Nikolaos Tsantalis. *Dependency smells in javascript projects*. IEEE Transactions on Software Engineering, vol. 48, no. 10, pages 3790–3807, 2021.

[Kaur 2015] Uttamjit Kaur and Gagandeep Singh. *A review on software maintenance issues and how to reduce maintenance efforts*. International Journal of Computer Applications, vol. 118, no. 1, pages 6–11, 2015.

[Kempf 2008] Martin Kempf, Reto Kleeb, Michael Klenk and Peter Sommerlad. *Cross language refactoring for eclipse plug-ins*. In Proceedings of the 2nd Workshop on Refactoring Tools, pages 1–4, 2008.

[Klint 2011] Paul Klint, Tijs van der Storm and Jurgen Vinju. *EASY Meta-programming with Rascal*. In João Fernandes, Ralf Lämmel, Joost Visser and João Saraiva, editeurs, Generative and Transformational Techniques in Software Engineering III, volume 6491 of *Lecture Notes in Computer Science*, pages 222–289. Springer Berlin / Heidelberg, 2011.

[Kohn 2020] Tobias Kohn, Guido van Rossum, Gary Brandt Bucher II, Talin and Ivan Levkivskyi. *Dynamic pattern matching with Python*. In Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages, pages 85–98, 2020.

[Krause 2016] Christian Krause, Daniel Johannsen, Radwan Deeb, Kai-Uwe Sattler, David Knacker and Anton Niadzelka. *An SQL-based query language and engine for graph pattern matching*. In Graph Transformation: 9th International Conference, ICGT 2016, in Memory of Hartmut Ehrig, Held as Part of STAF 2016, Vienna, Austria, July 5-6, 2016, Proceedings 9, pages 153–169. Springer, 2016.

[Libkin 2016] Leonid Libkin, Wim Martens and Domagoj Vrgoč. *Querying graphs with data*. Journal of the ACM (JACM), vol. 63, no. 2, pages 1–53, 2016.

[Mayer 2012] Philip Mayer, Andreas Schroeder and Welf Löwe. *Cross-Language Code Analysis and Refactoring*. In In Proceedings of the International Workshop on Source Code Analysis and Manipulation, 2012.

[Mayer 2014] Philip Mayer and Andreas Schroeder. *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, pages 437–462. Springer, 2014.

[Meurice 2016] Loup Meurice, Csaba Nagy and Anthony Cleve. *Static analysis of dynamic database usage in java systems*. In Advanced Information Systems Engineering: 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13-17, 2016. Proceedings 28, pages 491–506. Springer, 2016.

[Mohamed 2018] Khaled Abdelsalam Mohamed and Amr Kamel. *Reverse engineering state and strategy design patterns using static code analysis*. International Journal of Advanced Computer Science and Applications, vol. 9, no. 1, 2018.

[Mushtaq 2017] Zaigham Mushtaq, Ghulam Rasool and Balawal Shehzad. *Multilingual source code analysis: A systematic literature review*. IEEE Access, vol. 5, pages 11307–11336, 2017.

[Neubauer 2005] Matthias Neubauer and Peter Thiemann. *From sequential programs to multi-tier applications by program transformation*. In Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 221–232, 2005.

[Nguyen 2005] Tien Nguyen, Ethan Munson and John Boyland. *An Infrastructure for Development of Object-Oriented, Multi-level Configuration Management Services*. In Internationl Conference on Software Engineering (ICSE 2005), pages 215–224. ACM Press, 2005.

[Nguyen 2011] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen and Tien N Nguyen. *Auto-locating and fix-propagating for HTML validation errors to PHP server-side code*. In 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pages 13–22. IEEE, 2011.

[Nguyen 2012] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen and Tien N Nguyen. *BabelRef: detection and renaming tool for cross-language program entities in dynamic web applications*. In 2012 34th International Conference on Software Engineering (ICSE), pages 1391–1394. IEEE, 2012.

[Pfeiffer 2011] Rolf-Helge Pfeiffer and Andrzej Wąsowski. *Taming the confusion of languages*. In European Conference on Modelling Foundations and Applications, pages 312–328. Springer, 2011.

[Pfeiffer 2012] Rolf-Helge Pfeiffer and Andrzej Wąsowski. *Texmo: A multi-language development environment*. In European Conference on Modelling Foundations and Applications, pages 178–193. Springer, 2012.

[Pierre-Etienne 2003] Moreau Pierre-Etienne, Christophe Ringeissen and Marian Vittek. *A pattern matching compiler for multiple target languages*. In International Conference on Compiler Construction, pages 61–76. Springer, 2003.

[Polychniatis 2013] Theodoros Polychniatis, Jurriaan Hage, Slinger Jansen, Eric Bouwers and Joost Visser. *Detecting cross-language dependencies generically*. In 2013 17th European Conference on Software Maintenance and Reengineering, pages 349–352. IEEE, 2013.

[Rodrigues 2024] Emanuel Rodrigues, José Nuno Macedo, Marcos Viera and João Saraiva. *pyZtrategic: A Zipper-Based Embedding of Strategies and Attribute Grammars in Python.* In ENASE, pages 615–624, 2024.

[Rodriguez 2012] Marko A Rodriguez and Peter Neubauer. *The graph traversal pattern*. In Graph data management: Techniques and applications, pages 29–46. IGI global, 2012.

[Rodriguez 2015] Marko A Rodriguez. *The gremlin graph traversal machine and language (invited talk)*. In Proceedings of the 15th Symposium on Database Programming Languages, pages 1–10, 2015.

[Ryu 2010] Sukyoung Ryu, Changhee Park and Guy L Steele Jr. *Adding pattern matching to existing object-oriented languages*. In ACM SIGPLAN Foundations of Object-Oriented Languages Workshop, volume 5. Citeseer, 2010.

[Shatnawi 2019] Anas Shatnawi, Hafedh Mili, Manel Abdellatif, Yann-Gaël Guéhéneuc, Naouel Moha, Geoffrey Hecht, Ghizlane El Boussaidi and Jean Privat. *Static code analysis of multilanguage software systems*. arXiv preprint arXiv:1906.00815, 2019.

[Shen 2021] Bo Shen, Wei Zhang, Ailun Yu, Zhao Wei, Guangtai Liang, Haiyan Zhao and Zhi Jin. *Cross-language Code Coupling Detection: A Preliminary Study on Android Applications*. In 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 378–388. IEEE, 2021.

[Sloane 2009] Anthony M Sloane. *Lightweight language processing in Kiama*. In International Summer School on Generative and Transformational Techniques in Software Engineering, pages 408–425. Springer, 2009.

[Soto-Valero 2021] César Soto-Valero, Nicolas Harrand, Martin Monperrus and Benoit Baudry. *A comprehensive study of bloated dependencies in the maven ecosystem*. Empirical Software Engineering, vol. 26, no. 3, page 45, 2021.

[Thakkar 2017] Harsh Thakkar, Dharmen Punjani, Sören Auer and Maria-Esther Vidal. *Towards an integrated graph algebra for graph pattern matching with gremlin*. In Database and Expert Systems Applications: 28th International Conference, DEXA 2017, Lyon, France, August 28-31, 2017, Proceedings, Part I 28, pages 81–91. Springer, 2017.

[Vanciu 2010] Radu Vanciu and Václav Rajlich. *Hidden dependencies in software systems*. In 2010 IEEE International Conference on Software Maintenance, pages 1–10. IEEE, 2010.

[Yu 2001] Zhifeng Yu and Václav Rajlich. *Hidden dependencies in program comprehension and change propagation*. In Proceedings 9th International Workshop on Program Comprehension. IWPC 2001, pages 293–299. IEEE, 2001.

[Șimonca 2022] Iuliana Șimonca, Alexandra Corbea and Anda Belciu. *Analytical Capabilities of Graphs in Oracle Multimodel Database*. In Education, Research and Business Technologies: Proceedings of 20th International Conference on Informatics in Economy (IE 2021), pages 97–109. Springer, 2022.