DepMiner: A Pipelineable Tool for Mining of Intra-Project Dependencies

Vladislav Repinskiy  
Department of Computer Science  
University College London  
London, England  
vlad.repinskiy.17@ucl.ac.uk

Vladimir Kovalenko  
Intelligent Collaboration Tools Lab  
JetBrains Research  
Amsterdam, The Netherlands  
vladimir.kovalenko@jetbrains.com

Abstract—Dependency analysis is recognised as an important field of software engineering due to a variety of reasons. There exists a large pool of tools providing assistance to software developers and architects. Analysis of inter- and intra-project dependencies can help provide various insights about the entire development process. There is currently a lack of tools that would support researchers by extracting intra-project dependencies data in a format most suited for further analysis.

In this paper we introduce DepMiner[1] — an open source, language-agnostic tool for mining detailed dependencies data from source code, based on extensive static analysis capabilities of an industry standard IDE. DepMiner can be easily integrated into arbitrary mining pipelines to conduct large-scale source code processing jobs involving intra-project dependencies. It is easily extensible to support other languages of source code, different granularities of analysis, and other use-specific needs.

I. INTRODUCTION

Dependency analysis is a major topic in software engineering. The importance of dependency analysis comes from the variety of its applications in software maintenance and quality control. Dependencies data can be extracted and analysed to help ensure high standards of code quality, performance, and maintainability. Information about dependencies is used in mature code analysis tools [1], [2] as well as in the development of new software engineering techniques in research settings [3]–[7].

Extracting dependencies information from source code is challenging, as it involves building comprehensive code analysis pipelines. Technical work required to build such pipelines builds up to a substantial part of work in research settings. With no adaptive, reusable toolkit openly available for extraction of dependencies data, technical work of building mining pipelines diverts the focus of researchers from problems in their primary domain. However, reliable infrastructure for polyglot source code analysis exists: it is one of the pillars of modern IDEs. Reusing language analysis capabilities of IDEs for data collection tasks looks like the missing link towards drastically reducing the technical complexity of data mining in many areas of software engineering research.

In this paper we present DepMiner – an open source, reusable, extensible mining tool that uses static analysis ca-

1Anonymous. (2021, January 13). DepMiner. Zenodo. http://doi.org/10.5281/zenodo.4441367

2DepMiner Example Use Case. Anonymous. (2021, January 15). DepMiner Visualization App. Zenodo. http://doi.org/10.5281/zenodo.4441366
role in modular applications design and architecture, and some studies suggest that clustering is correlated directly to software quality [5]. Intra-project artefact dependency graphs have been proved crucial in detecting clusters in software [6], as well as in developing novel automated software clustering algorithms and tools [7]. Understanding dependencies data helps researchers throughout the whole variety of applied and pure studies.

B. Existing Approaches To Dependency Analysis

Many existing solutions and tools focus on data visualisation, generating UML diagrams and dependency graphs of the projects. Tools like Structure101 [18] and MagicDraw [19] aid software architects in planning, managing and assessing the architecture, that do not employ actual code analysis.

Others provide object oriented metrics and other quantifiable insights. For instance, metrics such as a number of methods and attributed, SLOC and the depth of inheritance tree of Java classes are typically used to address the matter of quality and the conformity of the project to the OOP design principles. There are also more general metrics such as cyclomatic complexity of a program. A consistent relationship between the usage of object oriented metrics and quality factors has been demonstrated, as observed by Darcy and Kemerer [20].

Another large portion of these tools focus primarily on visualisation of modular structure of the application, such as Sourcetrail [21] and discontinued OptimalJ [22].

Numerous instruments checking vulnerabilities in inter-project dependencies provide secure usage of external libraries and packages, for languages most dependent on it. Examples of those include OSSIndex [23], Dependency-check [24] and RetireJS [25].

The majority of automated tools are language-dependent, meaning that the tool can only function properly on the codebase maintained in a single language, or a collection of similarly constructed languages. For instance, JDepend [2], JavaNCSS [26] and Dependency Finder [27] support JVM language codebases, while CppDepend [1] works with C/C++ code.

Most modern IDEs also include similar functionality in their toolkit. For instance, IntelliJ IDEA provides reports on clustering and interdependencies of modules within one project, also suggesting and facilitating possible refactoring to restore application modularity without disrupting the functionality.

C. Motivation

Despite the widespread adoption of the aforementioned and other complementary instruments in the industry, they fall out of scope for quantitative, low-granularity alternative dependency analysis problems in research since they do now provide clear access to raw dependencies data for potential analysis. The majority of works revolving around the topic of inter and intra-project code dependencies attempt to ease the workload of software architects, lead developers and other professionals dealing with software architecture in its full complexity.

Existing tools for extraction of dependency graphs from static codebases rely on complex language-specific code analysis technology, and are thus not easily extendable to support multiple programming languages.

DepMiner is a reusable, open-source tool for extraction of call graphs that can be easily incorporated into pipelines for complex experiments, or extended to support new programming languages or other dependency types.

III. IntelliJ PLATFORM

DepMiner is based on IntelliJ Platform. It consists of an IntelliJ IDEA plugin and infrastructure to run the processing in headless mode, without spinning up the IDE interface. In this section, we provide a brief overview of IntelliJ Platform from the standpoint of use for code analysis tools like DepMiner.

A. IntelliJ Platform SDK

IntelliJ Platform is an open source system of abstract components comprising an infrastructure that provides rich language tooling support for developers in the form of a documented SDK [8].

The IDE itself can analyse project dependencies on different levels of granularity (module, file, line dependencies). However, it does not allow users to export this data outside of the IDE. Thus, it is not best suited for data mining settings, especially if the dependency analysis should be performed for multiple projects in one run. The core idea behind DepMiner is to utilize the capabilities of the IntelliJ Platform for mining of data that is useful in research contexts and would otherwise require building complex dedicated code analysis tools.

B. Program Structure Interface

One of the pillars of IntelliJ Platform is the Program Structure Interface (PSI). PSI is a platform layer responsible for representing source code in the form of meaningful semantic models. It is specifically designed to represent the codebase in a way suitable to support IDE code analysis features. PSI logic presents any code fragment as a AST-like syntax tree, where each node represents a certain code element and is augmented by numerous fields providing more detail about the element itself. There are different types of nodes for different types of code elements. For instance, a PSI structure of Java file would normally contain nodes of type PsiClass, PsiMethod, PsiField and so on. Figure 1 demonstrates a typical PSI structure of a Java file. Different types of nodes often have different sets of attributes, which in turn provide details of code element’s location in file, containing file name, language and such.

PSI trees and nodes are constructed on demand, which means that they are only built if and when any component or plugin in the IDE requires the PSI tree of a certain file or code region. Language-specific PSI parsers are responsible for constructing PSI trees of source code files. IntelliJ IDEA, for instance, can construct PSI trees for JVM-based languages,
but it can also be extended with a plugin that would provide support for files in other popular languages.

The ability to construct PSI trees for source code in different programming languages is implemented as part of language support for IntelliJ-based IDEs as either bundled or optional platform plugins. Therefore, relying on PSI, rather than on tailored language-specific parsers, for code analysis helps drastically reduce the complexity of code analysis pipelines.

C. Indexing and References

Indexing is the framework that lies in depth of dependency resolution in IntelliJ Platform, and thus in DepMiner. It provides a mechanism for querying whether a specific code element is present in a file, in a way that is scalable to large codebases performance-wise. This same mechanism allows to quickly search for the occurrences of the same token in multiple files. When a project is opened, the IDE goes through the entire file tree and indexes source code, constructing a map of keys found in each file and their associated values. Indices are then serialized and stored in a binary format, to allow efficient access and querying.

If a PSI element represents a usage of a certain element declared elsewhere in the code, it is linked to the declaration of the element through a PSI reference. However, references are not parts of the PSI tree per se. PSI elements that represent a usage of another element must provide a PSI reference to the declaration of itself on-demand, using the `getReferences()` method. However, PSI references do not contain actual location of the declaration of the element. Instead, they only provide logic on how to locate the declaration of the element for a certain type of element. However, in general location resolution is possible under the aegis of the aforementioned mechanism of indices.

For example, let us consider a PSI element representing a method call in Java. On invocation of `PsiElement.getReferences()`, element logic will retrieve the method name, and query the corresponding index on whether identical tokens exist anywhere in the project. Using this data, the platform can figure out whether any other files contain the aforementioned method name and construct corresponding PSI trees. Finally, it will traverse the files bottom-up, checking with each element if it is indeed a declaration of the inspected element. The process of invoking the logic stored in a certain reference is called resolving, and it can be initiated by calling the `resolve()` method on a PSI reference.

IV. DepMiner Design

A. Requirements

Initially, DepMiner was a component of our internal data mining pipeline that we later converted into a reusable tool. Apart from extraction of dependencies data, the fundamental feature we required of DepMiner is the ability to be integrated into complex data analysis pipelines.

We followed the following requirements to ensure that DepMiner can be reused in other settings:

- DepMiner must extract dependencies at all levels of granularity from project source code - i.e. tokens, functions and classes encapsulating them, files and directories.
- Must run from command line
- Must output results in JSON format
- Must support projects written in popular languages - Java and Python
- Should be extensible to support other languages
- Dependencies information must contain information about location and type of code elements

B. DepMiner Data Model

At the core of DepMiner lies its data model for dependency representation.

Location markers: Location of each code element is represented by a location marker, which consists of an absolute path to the file containing this element, and the range of lines in the document which a given element is encapsulating. Some code elements, such as tokens referring to variables, will only take up a small region of code or even a fraction of a single line. However, some larger elements like Java methods and classes might take up a significant portion of a file, and even represent a hierarchy by containing each other. Information on locations of code element is crucial for most applications. For instance, knowing that a certain method lies within a certain class is important for building UML-like representations of inter-class dependencies in Java code.

Type signatures: To create a full representation of an element, its location data is bundled together with its type signature. Type signature is a description of an element derived from a field of a corresponding PSI element: it is used to classify code element from the resulted output data provided by DepMiner. These signatures are language specific, with concrete type labels defined by the language implementation used to build the PSI tree. However, in inspected cases they allow to
gathers important information about code elements forming a dependency. For instance, after a Java project is scanned, it is common to see a dependency between code elements denoted `PsiClass` and `PsiReferenceExpression`. In case with Java, it would convey the usage of a Java class somewhere else in the code, for instance an object of a given class is created, or a method from the class is called. TypeSignature field can be extended to support any additional language specific logic necessary in the given circumstances.

**Dependency records:** Location markers are combined with type signatures into a code element record. Similarly, two code element records are bundled together to represent a dependency record. Dependencies are a final level of abstraction in DepMiner data model and the output data will be a list of dependencies. Figure 2 demonstrates the logical composition of a dependency record.

![Diagram](image)

Fig. 2. Representation of a dependency record in DepMiner's code.

C. **Working with DepMiner**

In essence, execution of DepMiner is execution of IntelliJ IDEA, extended by DepMiner's plugin, in headless mode, i.e. without spinning up the user interface. Headless runtime allows for pipelineable bulk processing without the need to interact with the IDE. In addition, headless mode improves performance by saving resources that would otherwise be occupied with loading and rendering the user interface.

DepMiner relies on Gradle for managing own dependencies and running. The IntelliJ Gradle Plugin [28] is designed for building and testing IntelliJ plugins and other IntelliJ Platform based systems. There exists a special Gradle task `runIde`, provided by the IntelliJ Gradle Plugin, that first builds and tests the plugin under development, and then launches another instance of IntelliJ IDEA with the plugin loaded. This allows developers to manually inspect the functionality and the UI appearance of plugin elements. DepMiner utilises the same `runIde` task to start an instance of IDEA and perform the analysis in headless mode. DepMiner is launched from the command line, through the invocation of a Bash script, which starts the `runIde` task with a proper format of arguments.

When an instance of IDEA is started, DepMiner opens and sets up a project model for the given directory, in order to trigger project indexing. Projects that contain files recognised by IDEs (such as `.project` for eclipse and `.iml` for IDEA) can be opened automatically by IntelliJ Platform utility functions. Projects that do not contain a project model definition, or store it in another format, are set up programatically by DepMiner. `ProjectSetupUtil.kt` file in its distribution contains the implementation of programmatic project setup.

Once the indices have been built, DepMiner builds PSI trees for all files in the project. Traversing all of the resulting trees, each element is inspected to check if any PSI references objects are attached to the object. If a reference is detected, the element is added to the final analysis scope, i.e., a list of elements to be inspected further. The final list of PSI elements with corresponding PSI references is iterated and each reference is resolved. Thus, usages of code elements can be connected to their declarations. For each reference in the list, DepMiner constructs a `Dependency` object. Finally, the final list of references is produced and stored in a JSON file.

To demonstrate how output produced by DepMiner can be potentially utilised and to support understanding of its output, we built a web app visualising the resulting dependency graph. The visualisation is available at [http://doi.org/10.5281/zenodo.4441366](http://doi.org/10.5281/zenodo.4441366).

**D. Extensibility**

DepMiner code can be easily extended in multiple ways to accommodate specific analysis needs. First, DepMiner can analyse source code in a variety of languages. To add support for additional languages, one can add a dependency on the corresponding plugin to the `build.gradle.kts` file in its distribution.

Additionally, it is easy to change the scope of analysis by implementing `AnalysisScope` interface. Extracting dependencies is a computationally expensive task and to avoid pipeline bottlenecks one might want to exclude certain files, directories or even certain line ranges within files.

Finally, the processing logic of DepMiner can be modified to enable tasks other than building a list of dependencies. In fact, it can be used as a basis for tools that extract any information that is available in the scope of the IDE from arbitrary software projects.

V. **CONTRIBUTION, QUALITY, AND DISTRIBUTION**

A. **Contribution**

DepMiner is the primary contribution of this work. It fills the gap in the diverse field of dependency analysis tools, excluding several drawbacks of the previously mentioned, especially from a standpoint of application in research, and has a wide range of potential applications for software engineering problems.

Besides being useful as an independent data collection tool, it constitutes a a step towards reusing language analysis capabilities of IDEs for mining data for research contexts.

B. **Performance and Quality**

Performance of DepMiner is highly influenced by the performance of IntelliJ Platform components in use. Constructing and analysing large PSI trees can use significant amounts of memory in large files. Resolving and searching for references is also computationally expensive. Thus, if the scope of dependency analysis can be limited, the implementation
of AnalysisScope should strive to do so as otherwise DepMiner might become a bottleneck in the analysis pipeline.

Essential functions of DepMiner are covered by functional tests. If DepMiner is extended in any way (e.g. to analyse projects in other languages), the test suite should be extended accordingly.

C. Distribution

Anonymized source code of DepMiner is available on Zenodo: http://doi.org/10.5281/zenodo.4441367

REFERENCES

[1] “Cppdepend/c/c++ static analysis and code quality tool,” [Online; accessed 10-January-2021].
[2] “JDepend,” URL: https://github.com/clarkware/jdepend [Online; accessed 22-January-2018].
[3] T. Zimmermann and N. Nagappan, “Predicting defects using network analysis on dependency graphs,” Proceedings - International Conference on Software Engineering, pp. 531–540, 2008.
[4] V. H. Nguyen and L. M. S. Tran, “Predicting vulnerable software components with dependency graphs,” 6th International Workshop on Security Measurements and Metrics, MetriSec 2010, 2010.
[5] B. Csaba, L. Schrettner, Á. Beszédes, J. Jáss, P. Hegedus, and T. Gyiomóthy, “Relating clustering measures and software quality,” Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR, pp. 345–348, 2013.
[6] J. Dietrich, V. Yakovlev, C. McCartiny, G. Jenson, and M. Duchrow, “Cluster analysis of Java dependency graphs,” SOFTVIS 2008 - Proceedings of the 4th ACM Symposium on Software Visualization, pp. 91–94, 2008.
[7] S. Mohammadi and H. Izadhkhah, “A new algorithm for software clustering considering the knowledge of dependency between artifacts in the source code,” Information and Software Technology, vol. 105, no. May 2018, pp. 252–256, 2019. [Online]. Available: https://doi.org/10.1016/j.infsof.2018.09.001
[8] “What is the intellij platform?” URL: https://jetbrains.org/intellij/sdk/docs/intro/intellij_platform.html Sep 2020.
[9] D. Garlan and D. Perry, “Introduction to the special issue on software architecture,” IEEE Trans. Software Eng., vol. 21, no. 4, pp. 269–274, 1995.
[10] D. E. Perry, T. B. Laboratories, M. Hill, and A. L. Wolf, “Foundations for the Study of Software Architecture,” Software Engineering Notes (SEN), vol. 17, no. 4, 1992.
[11] C. Pich, L. Nachmanoson, and G. G. Robertson, “Visual analysis of importance and grouping in software dependency graphs,” SOFTVIS 2008 - Proceedings of the 4th ACM Symposium on Software Visualization, pp. 29–32, 2008.
[12] V. Zapalowski, D. J. Nunes, and I. Nunes, “Understanding architecture non-conformance: Why is there a gap between conceptual architectural rules and source code dependencies?” ACM International Conference Proceeding Series, pp. 22–31, 2018.
[13] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, “In search of a metric for managing architectural technical debt,” Proceedings of the 2012 Joint Working Conference on Software Architecture and 6th European Conference on Software Architecture, WICSA/ECSA 2012, pp. 91–100, 2012.
[14] W. Zhang, H. Mei, and H. Zhao, “Feature-driven requirement dependency analysis and high-level software design,” Requirements Engineering, vol. 11, no. 3, pp. 205–220, 2006.
[15] R. Leitch and E. Stroulia, “Assessing the maintainability benefits of design restructuring using dependency analysis,” Proceedings - International Software Metrics Symposium, vol. 2003-January, pp. 309–322, 2003.
[16] J. Brondum and L. Zhu, “Visualising architectural dependencies,” in 2012 Third International Workshop on Managing Technical Debt (MTD). IEEE, jun 2012, pp. 7–14. [Online]. Available: https://ieeexplore.ieee.org/document/6226003
[17] M. Pinzger, K. Gräfenhain, P. Käib, and H. C. Gall, “A tool for visual understanding of source code dependencies,” IEEE International Conference on Program Comprehension, pp. 254–259, 2008.
[18] “Structure101,” URL: https://structure101.com/