Measuring the impact of library dependency on maintenance

Núria Bruch Tàrrega
University of Amsterdam
nuria.bruchtarrega@student.uva.nl

Miroslav Živković
Software Improvement Group
m.zivkovic@sig.eu

Ana Oprescu
PCS, SNE, Informatics Institute
University of Amsterdam
a.m.oprescu@uva.nl

Abstract

Reusing code from open-source libraries is a useful practice for developers to avoid implementing the same functionalities multiple times. However, when a library is used in another software product, it creates a dependency that may spread the security vulnerabilities of the library to the product. Most package managers have dependency managers which only perform a binary evaluation of the dependencies. Thus, developers have no information about how much products depend on a library or how much effort would be needed to replace a dependency.

In this research, we propose a way to measure the degree of library dependency, as well as how much effort would be required to replace the usage of a library with another one. We leverage existing coupling metrics and revisit them in the context of library dependencies.

We present two metrics to measure the coupling generated by dependencies: method invocation and aggregation coupling, and briefly discuss the next steps.

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

1 Introduction

There are many open-source software (OSS) libraries available for the developers to reuse the features that these libraries implement [KGDP17]. This practice allows the reuse of previously developed code and, therefore, helps developers to avoid implementing the same functionalities multiple times.

When an open-source library is used in a software product (e.g. another library or an application), a dependency between the product and the library is created. This adds the task of managing these dependencies to the maintenance tasks of the product, and this task is not a trivial one.

Open-source libraries can have security vulnerabilities that may affect the products that depend on these libraries. For example, some security vulnerabilities can have a negative impact in terms of integrity, privacy, or availability.

Currently, developers have package managers at their disposal, to ease the task of managing the dependencies of their products. However, these package managers only evaluate whether a dependency exists or not and a more detailed risk evaluation is missing [HBG18]. There is no way to evaluate how much a product depends on a library.

Furthermore, the developers of a project may decide to replace one of the dependencies of the product with another one. This could happen when a library has vulnerabilities or is deprecated. However, replacing a dependency could be a costly process. For example, it may involve identifying which parts of the project are affected by the dependency, and which parts of the library are being used and need replacement.

This research has the goal of measuring the degree of library dependency and understanding how it affects the maintenance effort. In this work in progress a set of metrics is proposed to measure the library dependencies. These dependencies can take place between two libraries or between another type of software product and a library. We refer to dependencies between libraries interchangeably with dependency between a software product and a library.

Based on our problem statement, we define the following research question:

RQ1: *How can we measure the degree of source code dependency between a software product and a library?* The goal of this research question is to define the metrics to measure dependency. We focus on coupling metrics, which have been used for many years, in particular for Object-Oriented systems [BDW99]. The key difference is that these metrics have been used to measure coupling within a software product, and not between different software products. Therefore, the definition of coupling is changed, and so is the meaning of the metrics.

2 Background

Coupling is defined as the strength of the connection from one item to another, and is related to the maintainability of a software product [GC09]. To evaluate the connection or dependency between items that belong to the same product, a wide variety of metrics has been proposed to measure coupling. There are six main groups of coupling metrics: structural, dynamic, evolutionary and logical, information entropy approach, conceptual, and domain-specific [PM06]. The most largely studied by the literature is the structural coupling, and it is the type of coupling measured in this research.

Briand et al. [BDW99] defined a unified framework for coupling metrics, based on three previously existing frameworks [EKS94, HM95, BDM97]. They specify six criteria that define the type of coupling used for metric calculation.

Criterion 1: Type of connection, which is the connection mechanism that creates coupling.
Criterion 2: Locus of impact, or the point of view from which the coupling being measured. It can be import coupling, the point of view of the element that uses another element, the client element. Or export coupling, the point of view of the element that is being used by another one, the server element.
Criterion 3: The granularity of the measure, which includes two aspects: the aggregation level (e.g. method, class, system), and how the metric counts the connections between the two elements (e.g. count each connection individually or binary evaluation of the connection between two elements).
Criterion 4: The stability of the server, a stable server is not subject to modifications in the project at hand.
Criterion 5: Direct/Indirect coupling, if the metric accounts for indirect relationships, or only measures the direct ones.
Criterion 6: Inheritance, this criterion specifies how certain cases related to inheritance (e.g. inheritance and polymorphism) affect coupling.

3 Related Work

To the best of our knowledge, no studies measure the degree of dependency between libraries. However, some studies perform an evaluation of the dependencies according to certain characteristics.

Soto-Valero et al. [SVHMB20] conducted a study of bloated dependencies. Bloated dependencies, either direct or transitive, are those specified in the dependency set of a project, that are not used for either compilation or product deployment. The authors developed the tool *DepClean*, which analyses the dependencies of Java artifacts. This tool identifies bloated dependencies and generates an alternative dependency set without bloated dependencies. Further, *DepClean* creates a call-graph of the API members of the libraries and dependencies. However, the focus is on the bloated dependencies, not on measuring the degree of the dependencies.

Pashchenko et al. [PPP^{+}18] propose a method to analyze dependencies in which they distinguish between own and third-party libraries, as well as deployed and non-deployed dependencies. In addition, they remark the importance of halted dependencies, since the libraries that create these dependencies are no longer updated. However, Pashchenko et al. do not perform a call-level analysis of the dependencies, since their dependency resolution is based only on the *POM* file of the libraries. Hence, the transitive dependencies that are not really used in the studied library are still counted.

4 Definition of coupling

The first step towards creating a model to measure the dependencies is to define which meaning of coupling is involved in these dependencies. We use the framework explained in section 2, from Briand et al. [BDW99] to define coupling.

4.1 Criterion 1: Type of connection

With this criterion, it is defined which type of connection creates coupling between the two libraries. There are several clearly distinguished mechanisms that can create coupling [BDW99], and are listed below.

Given class a that belongs to library A, and class b that belongs to library B...

1. ... class a has an attribute of type b (Relationship of aggregation).
2. ... method of class a has a parameter of type b or has return type b.
3. ... method of class a has a local variable of type b.
4. ... method of class a calls a method which has a parameter of type b.
5. ... method of class a references attribute of class b.
6. ... method of class a invokes method of class b.
7. ... class a and class b have a relationship such as *uses* or *consists-of*.

Having a single metric that measures more than one of these types of connections is not recommended as this requires to figure out if every type of connection creates the same coupling, and whether it affects maintenance in the same way. It would not be possible to know how much of the coupling is created by which type of connection.

Therefore, different metrics should be used for different connections. To decide which types of connections to measure, we have decided to review the literature on coupling metrics, to understand which connections are the most measured and why.

Reference	1	2	3	4	5	6	7
[EKS94]	x	x	x	x		x	x
[HM95]	x	x	x		x	x	x
[BDM97]	x	x				x	
[WK00]	x	x					
[YTB05]	x	x	x	x		x	
[GS07]	x				x	x	
[GC09]	x	x	x	x	x	x	x
[HCN98]					x	x	
[DBDV04]	x				x	x	
[KKK+19]	x				x	x	

Table 1: Literature usage of the types of connection

Types 1 and 6 are the most used in the literature and therefore we define a metric to measure **type 6: method invocation**.

The second metric that we consider is **type 1: aggregation coupling**, for two main reasons. It is used as much as type 6 in the reviewed literature, and because in some cases measuring type 6 may not be enough to understand how much maintenance a library dependency may need. There is the possibility that a class has an attribute of another class, but never calls a method that belongs to that class.

The above-mentioned metrics are those that we initially consider in our work, and we will explain them in greater detail in Section 5. Nevertheless, it might be necessary to include additional metrics, to account for other connection types.

4.2 Criterion 2: Locus of impact

The goal of this measurement is to know how much a library depends on another. Therefore, the point of view of this evaluation is from the library that uses another one. Hence, the locus of impact of the coupling to be measured is **import**.

4.3 Criterion 3: Granularity of the measure

There are two aspects to define: (1) the aggregation level of the measure, and (2) how the metric counts the connections. First, we are going to discuss the aggregation level. Briand et al. define the following levels: Attribute, Method, Class, Set of classes, System.

Our goal is to measure the coupling between the set of classes of the client library and the set of classes of the server library. Therefore, the aggregation level used in this research is the **library** level. To maintain the precision of the measurement, the calculation of coupling for a more coarse-grained level, such as library, is done by aggregating the coupling of the more fine-grained aggregation levels, such as method and class.

There are two basic options for counting connections: A) counting every time an item is used, and B) count the number of items used. From a maintenance point of view, it matters whether a method is called once or multiple times, and we therefore use option A. To support fine-grained analysis, connections are counted from the smallest aggregation level, and aggregated up to the considered aggregation level. For example, when counting method invocations, we **add up the number of method invocations per each method of each class of a library**.

4.4 Criterion 4: Stability of the server

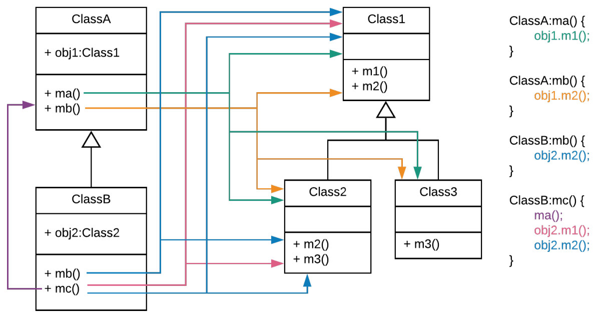
Briand et al. [BDW99] define stable classes as *"Classes that are not subject to change in the project at hand"*, hence we consider stable classes all the classes not implemented in the client library. Therefore, we measure coupling from non-stable classes to stable classes. However, the separation between stable and unstable classes is not enough. The goal is to measure coupling only with those **stable classes** that are part of third-party open-source libraries. Therefore, the standard classes of a programming language are not considered.

4.5 Criterion 5: Direct and indirect coupling

To make a decision about this criterion, we need to distinguish two alternative scenarios in which we want to measure coupling: direct dependencies and transitive dependencies. For the initial approach, we focus on the direct dependencies only. Therefore, the metrics measure **direct coupling**.

4.6 Criterion 6: Inheritance

Within this criterion, there are three aspects to decide about: how, if at all, does the metric distinguish between inheritance-based coupling and non-inheritance-based coupling? Does the metric account for polymorphism? And finally, what determines whether a method or attribute is part of a class or not?



```
ClassA:ma() {
    obj1.m1();
}

ClassA:mb() {
    obj1.m2();
}

ClassB:mb() {
    obj2.m2();
}

ClassB:mc() {
    ma();
    obj2.m1();
    obj2.m2();
}
```

Figure 1: Example of coupling special cases, based on example from Briand et al. [BDW99]

In order to answer the first question, we focus on the method `mc` of `ClassB` in Figure 1. This method invokes `ma` of `ClassA`, inherited by a class `ClassB`. This *inheritance-based* coupling is sometimes considered as a special case of coupling. When there is a change of an inherited method that a class uses, it requires the same maintenance effort as the method that is not inherited. Therefore, our metrics **include inheritance-based coupling without distinction**.

In case of polymorphism we look at the methods of `ClassA`. This class contains an attribute of type `Class1`, which could be of type `Class2` or `Class3`. We first analyze whether a call to a method of `Class1` would create coupling with `Class2` and `Class3`, and if it makes a difference when the method is overridden or not. The method `ma` invokes `m1`, which is not overridden by any of the descendants of `Class1`. When a change is made in `Class2` or `Class3` no change is required as the invoked method remains the same. In contrast, method `mb` calls `m2`, which is overridden in `Class2`. Here, the implementation of `m2` in `Class2` could be updated, and this may affect the way `ClassA` uses it, and therefore changes may be needed. Thus, it is necessary to **account for polymorphism**.

Further, a method belongs to the class that implements it (could be more than one since we account for polymorphism), or a method belongs to the class that it is referenced from. The last two lines of the method `mc` of `ClassB` call method `m1` and `m2` on an object of type `Class2`. Note that `m1` is only implemented in `Class1`, while `m2` is overridden in `Class2`. From a maintenance perspective, when the method `m1` is updated in `Class1`, this probably requires updating `ClassB` as well. However, changes in `Class2` do not generate a need to update the method call to `m1` in `ClassB`. Differently, when `m2` is updated in `Class1`, it will not make a difference for this call of `m2` since it is not executing the implementation of `Class1`. Therefore, **a method call creates coupling with the class that contains the implementation**.

5 Formal definition of the metrics

Based on the characteristics of coupling described in Section 4 we define two coupling metrics that differ with respect to the measured connection type.

5.1 Metric 1: Method invocation coupling (MIC)

The method invocation coupling between two libraries, with a direct dependency is calculated by the function $\texttt{MIC}(L_c, L_s)$, where L_c is the client library and L_s is the server library. According to criterion 3, the metric is calculated for each of the methods implemented in the classes included in the client library L_c. The set of methods implemented in library L_c is represented as $\texttt{M}(L_c)$. For each of the methods, the number of individual invocations to methods implemented in L_s are counted, and summed up.

$$\texttt{MIC}(L_c, L_s) = \sum_{m_c \in \texttt{M}(L_c)} \texttt{nII}(m_c, L_s)$$

The function $\texttt{nII}(m_c, L_s)$ calculates the number of individual invocations from the method m_c to methods of library L_s, considering all polymorphic implementations. In order to calculate $\texttt{nII}(m_c, L_s)$, for each of the stable methods invoked by m_c, the set of stable methods invoked by m_c is $\texttt{SIM}(m_c)$. A stable is a method which does not belong to L_c. For each one of the invoked methods (m_s), the number of invocations from m_c, denoted $\texttt{nI}(m_c, m_s)$, is multiplied by the number of polymorphic implementations of the method that are included in the server library (L_s). The number of polymorphic implementations is calculated by the function $\texttt{nP}(m_s, L_s)$.

To obtain the number of polymorphic implementations, we intersect the set of polymorphic implementations of a method m_s $(\texttt{PM}(m_s))$ with the set of methods implemented in the server library $(\texttt{M}(L_s))$. The cardinality of the intersection is the number of polymorphic implementations of the method m_s in L_s.

$$\mathtt{nII}(m_c, L_s) = \sum_{m_s \in \mathtt{SIM}(m_c)} \mathtt{nI}(m_c, m_s) * \mathtt{nP}(m_s, L_s)$$

$$\mathtt{nP}(m_s, L_s) = |\mathtt{PM}(m_s) \cap \mathtt{M}(L_s)|$$

5.2 Metric 2: Aggregation coupling (AC)

The second metric, which considers aggregation coupling between a client library (L_c) and a server library (L_s), is computed with the function $\mathtt{AC}(L_c, L_s)$. For this type of connection, the finest-grained level of aggregation is the class. Therefore, the function iterates through each class implemented in the client library. The set of classes implemented in library L_c is the set $\mathtt{C}(L_c)$.

For each one of these classes, the metric considers all the declared attributes of the class, the type of which is a stable class. The set of stable classes which are the type of an attribute declared in class c_c is denoted $\mathtt{SAT}(c_c)$. A stable class is a class which is not implemented in the client library (L_c).

In order to account for inheritance as described in the discussion of criterion 6, we multiply the number of times a stable class c_s is the type of a declared attribute in class c_c (denoted by ($\mathtt{nA}(c_c, c_s)$) by the number of descendants of the class c_s implemented in the server library L_s.

The number of descendants of a class c_s, implemented in the server library L_s, is calculated by the function $\mathtt{nDC}(c_s, L_s)$. To obtain the result, the set of descendants of the class (including the class) is intersected with the set of classes implemented in the server library. These sets are denoted $\mathtt{DC}(c_s)$ and $\mathtt{C}(L_s)$, respectively. The cardinality of the intersection is the result of the function $\mathtt{nDC}(c_s, L_s)$.

$$\mathtt{AC}(L_c, L_s) = \sum_{c_c \in \mathtt{C}(L_c)} \sum_{c_s \in \mathtt{SAT}(c_c)} \mathtt{NA}(c_c, c_s) * \mathtt{nDC}(c_s, L_s)$$

$$\mathtt{nDC}(c_s, L_s) = |\mathtt{DC}(c_s) \cap \mathtt{C}(L_s)|$$

6 Preliminary Results

The preliminary results have been obtained by means of a proof of concept implemented in Java using the *javassist*[1] library to analyze bytecode. We used Maven to download the client and server libraries that are part of the dependency tree of the client libraries.

Figure 2 illustrates current results. Some dependencies have zero value for both metrics. This could be due to client library that uses the server library but with a *type of connection* not measured by MIC or AC. This puts a requirement to implement other metrics to measure dependencies in PoC. Another explanation could be a possible relocation of a server library in the jar file of the client library during the build, which is not detected by the PoC. Finally, the results could mean the library is not using the dependency, i.e. it is a bloated dependency [SVHMB20].

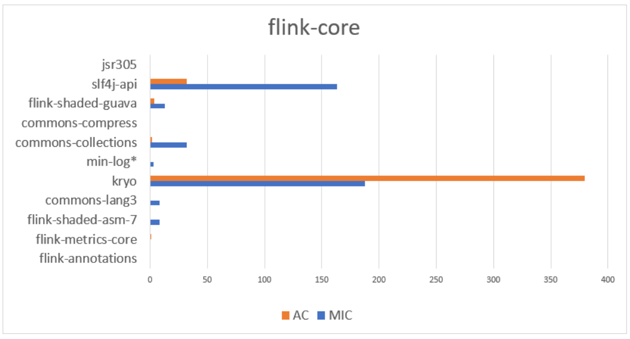


Figure 2: Results of MIC and AC for library `flink-core`

Most of the dependencies that have $\mathtt{AC} \neq 0$ also have $\mathtt{MIC} \neq 0$. Figure 2 shows a single case of a dependency (`flink-metrics-core`) that has $\mathtt{AC} = 1$ and $\mathtt{MIC} = 0$. In addition, there is only one case (`kryo`) in which the value of AC is greater than the value of MIC. However, the value of AC is mostly due to the detection of the descendants of the declared types (criterion 6). It might be interesting to compare which part of the measured coupling is due to accounting for inheritance. There are also cases where a dependency has $\mathtt{MIC} > 0$ and $\mathtt{AC} = 0$.

Figure 2 shows that the values of MIC range from 0 to 188, and AC from 0 to 380. However, for some other libraries that were investigated (but not shown here) this range is greater. For example, the MIC ranges from zero to 7601 for `puppycrawl-tools-checkstyle`.

7 Conclusion and Next Steps

We leveraged the framework defined by Briand et al. to formulate the definition of coupling to be measured. The framework was adapted to the use case of dependencies between libraries. The result is the definition of two metrics which measure different types of connection between libraries: method invocations and aggregation. An initial proof-of-concept (PoC) has been implemented and used with some example libraries.

We plan to extend the work presented by defining the metrics to measure transitive dependencies. Next, we will perform the theoretical validation of the metrics based on the properties defined by Briand et al. In addition, we will improve the PoC tool to calculate and compare the metrics defined.

[1] http://www.javassist.org/

Then, we will focus on a second research question. This second research question targets different cases of dependency replacement, how each one affects the code and how much effort is required for the replacement. Finally, we will compare the real effort invested in a replacement with the effort estimated by our model.

8 Acknowledgment

The work of the second author on this paper has been executed with the scope of the FASTEN project. The FASTEN project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825328.

References

[BDM97] Lionel Briand, Prem Devanbu, and Walcelio Melo. An investigation into coupling measures for c++. In *Proceedings of the 19th international conference on Software engineering*, pages 412–421, 1997.

[BDW99] Lionel C. Briand, John W. Daly, and Jurgen K Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on software Engineering*, 25(1):91–121, 1999.

[DBDV04] Bart Du Bois, Serge Demeyer, and Jan Verelst. Refactoring-improving coupling and cohesion of existing code. In *11th working conference on reverse engineering*, pages 144–151. IEEE, 2004.

[EKS94] Johann Eder, Gerti Kappel, and Michael Schrefl. Coupling and cohesion in object-oriented systems. Technical report, Citeseer, 1994.

[GC09] Varun Gupta and Jitender Kumar Chhabra. Package coupling measurement in object-oriented software. *Journal of computer science and technology*, 24(2):273–283, 2009.

[GS07] Gui Gui and Paul D Scott. Ranking reusability of software components using coupling metrics. *Journal of Systems and Software*, 80(9):1450–1459, 2007.

[HBG18] Joseph Hejderup, Moritz Beller, and Georgios Gousios. Prazi: From package-based to precise call-based dependency network analyses, 2018.

[HCN98] Rachel Harrison, Steve Counsell, and Reuben Nithi. Coupling metrics for object-oriented design. In *Proceedings Fifth International Software Metrics Symposium. Metrics (Cat. No. 98TB100262)*, pages 150–157. IEEE, 1998.

[HM95] Martin Hitz and Behzad Montazeri. *Measuring coupling and cohesion in object-oriented systems.* Citeseer, 1995.

[KGDP17] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 102–112. IEEE press, 2017.

[KKK$^+$19] Falko Koetter, Monika Kochanowski, Maximilien Kintz, Benedikt Kersjes, Ivan Bogicevic, and Stefan Wagner. Assessing software quality of agile student projects by data-mining software repositories. In *Proceedings of the 11th International Conference on Computer Supported Education-Volume 2: CSEDU, INSTICC*, pages 244–251. SciTePress, 2019.

[PM06] Denys Poshyvanyk and Andrian Marcus. The conceptual coupling metrics for object-oriented systems. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 469–478. IEEE, 2006.

[PPP$^+$18] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 42. ACM, 2018.

[SVHMB20] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. A comprehensive study of bloated dependencies in the maven ecosystem. *arXiv preprint arXiv:2001.07808*, 2020.

[WK00] F George Wilkie and Barbara A Kitchenham. Coupling measures and change ripples in c++ application software. *Journal of Systems and Software*, 52(2-3):157–164, 2000.

[YTB05] Hong Yul Yang, Ewan Tempero, and Rebecca Berrigan. Detecting indirect coupling. In *2005 Australian Software Engineering Conference*, pages 212–221. IEEE, 2005.