Dependency Types and Subtypes in the Context of Architecture Reconstruction and Compliance Checking

Leo Pruijt
HU University of Applied Sciences
Utrecht, The Netherlands
leo.pruijt@hu.nl

Jan Martijn E.M. van der Werf
University Utrecht
Utrecht, The Netherlands
j.m.e.m.vanderWerf@UU.nl

ABSTRACT

Software architecture reconstruction and compliance checking rely on supporting tools that analyze the modules in the code and their dependencies. Tools may provide a dependency type for each dependency to provide more detail on the actual usage relation. This study is aimed on the identification of dependency characteristics which can be determined accurately and which might be interesting for architects and researchers in the context of architecture reconstruction and compliance checking. A classification is proposed which distinguishes dependency types, related subtypes, and several other characteristics. To enable reflection on the usefulness of the classified dependency details, a prototype implementation has been developed for the analysis of Java based systems. A frequency analysis of the classified dependency characteristics in three open source systems is presented, as well as an analysis of a set of rule violating dependencies in one of these systems.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques.

General Terms

Design, Verification.

Keywords

Software Architecture, Dependency Analysis, Static Analysis, Architecture Compliance, Architecture Conformance

1. INTRODUCTION

Dependency analysis is "the process of determining a program's dependences" [8]. Dependency analysis, which focuses on dependencies among classes and modules in the implemented code, takes a central position in software architecture reconstruction and architecture compliance checking (ACC). Many tools and techniques are available to analyze a software system and to reconstruct, visualize, or restructure its architecture [3]. Furthermore, tools are available to check conformance between architectural design and the implemented program code in order to prevent architectural erosion [13].

Static analysis tools focus on the module view of software architecture, where dependencies represent *uses relations*: "Module A *uses* module B if A *depends* on the presence of a correctly functioning B to satisfy its own requirements" [2]. In line with Briand et al. [1], we use the term *client-module* for the using module and *server-module* for the used module.

Dependencies may be typified to provide more details on the actual usage of the other module (at code level mostly another class). In a previous study [10] we have introduced a classification of dependency types, based on research papers and professional literature. We used the identified types to investigate whether ACC-tools were able to detect dependencies of all these types in the code. In the same study, we noted that dependency types were not standardized among the tested tools. We reported that only four tools[1] out of the seven tools in our test provide dependency types. Further, the number of types and the types themselves were varying per tool. Some of these types were very specific, while others covered many forms of code constructs. The following examples illustrate our findings and show that types like reference, access, or uses may represent quite different kinds of usage.

- Constructor calls were reported by Lattix as "Construct with Arguments", by SAVE as "ACCESS", by Sonargraph as "Uses new", and by Structure101 as "calls".
- Declaration of a local variable was reported by Lattix as "Class Reference", by SAVE as "ACCESS ", by Sonargraph as "Uses", and by Structure101 as "references".

In this study, we build on this work, but now with another focus. This time, we pursue the following research question: Which dependency types and other characteristics provide useful and accurate information for software architects and researchers? The two requirements enclosed in the research question, useful and accurate, both call for elaboration.

With useful, we mean that the dependency characteristics should be suitable to answer relevant questions. For example, the characteristics may be used to determine the severity of a violation of an architectural rule, the strength of coupling between two modules, or the degree of encapsulation of a module; both by human interpretation (requires user-oriented terminology) as well as by computation. A consequence is that dependency details that may be used as indicators for the degree of coupling, or encapsulation, should be separated strictly from each other. For example, usage of a variable of another class compromises encapsulation [18], thus is useful information. Furthermore, many details are interesting, since they may be used to determine the strength of coupling [1]; e.g., details about the used class (e.g., is it an interface, or implementation class), and details about the kind of usage (e.g., is it a call of normal method or of an inherited method).

In this context, accuracy means that the dependency details characterize the usage relation correctly, and uniquely. For

ECSAW '15, September 07 - 11, 2015, Dubrovnik/Cavtat, Croatia.
Copyright is held by the author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3393-1/15/09…$15.00
DOI: http://dx.doi.org/10.1145/2797433.2797491

[1] Lattix LDM - version 8.2.7 - lattix.com;
SAVE - version 1.7.1 - iese.fraunhofer.de;
Sonargraph Architect version 7.1.8 - hello2morrow.com;
Structure101 - version 3.5 - structure101.com.

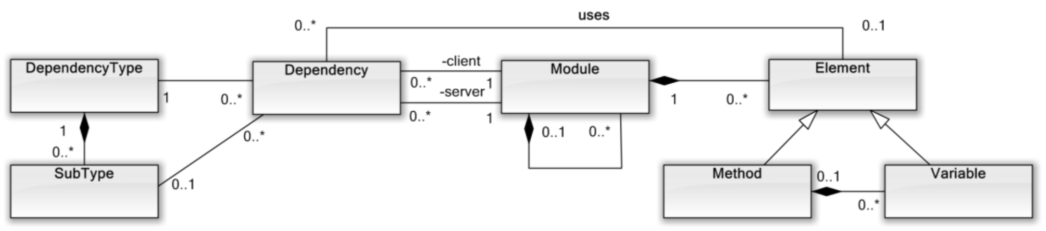

Figure 1. Conceptual model of a dependency and its context

example, if a dependency is typified as “access”, while this type characterizes usage of a variable of another class, than this kind of usage should really exist at code level. Further, its interpretation should be unique, i.e., type “access” should not be used for different kinds of usages, such as declaration of a local variable. In conclusion: accurate dependency types require well-chosen, distinctive identifiers.

To answer the research question stated above, we follow an approach of design science research [7]. We have not yet answered the question to its full extend. As a first step, we have focused on the identification of dependency characteristics which can be determined accurately and which might be interesting for architects and researchers. We have improved and extended the previous classification, and we have developed a prototype implementation in architecture compliance tool HUSACCT [11] that typifies dependencies in the code conform the classification in this paper. Furthermore, it generates dependency and violation reports with frequency statistics which provide interesting views on a system, and which may be useful for future research.

The contribution of this paper is threefold. First, we present a classification that distinguishes dependency types, subtypes, and some other characteristics, and we explain the principles behind the classification and the differences with the previous one. Second, we present the results of a frequency analysis of the classified dependency characteristics in three open source systems. Third, we present the analysis results of rule violating dependencies, subdivided in three groups for three different types of rules.

The next section introduces the classification of dependency types, subtypes and some other characteristics. Section 3 describes the analysis results of the application cases. Section 4 discusses related work, and Section 5 concludes the paper and presents ideas for future work.

2. Dependency Characteristics

A dependency essentially describes that a client-module uses another module, the server-module, as shown in Figure 1. At code level, all dependencies relate a client-class and a server-class, except in case of import dependencies, where the server-module may also be a package or namespace. In addition to the client-class and server-class, other characteristics may provide more details on the dependency; for example, details on the kind of the used class, details on how the server-class is used and which class-member is used, or details on how the dependency may be located or traced in the code. At code level, multiple dependencies may be present in one line of code.

2.1 Dependency Types

A dependency type characterizes how the server-class is used by the client-class at a certain location in the program code. In our classification, we distinguish seven types of dependency in object-oriented code: Access, Annotation, Call, Declaration, Import, Inheritance, and Reference. Compared to our previous classification [10], one type is added, namely Reference, and the names of the types Access, Call and Declaration are shorter.

The dependency types Annotation, Declaration, Import, and Inheritance are in concept the same and in practice quite simple to differentiate (code examples are provided in the previous paper). These four types have in common that they represent *preparing* activities, which do not take care of transformations.

The most important difference with our previous classification is the strict distinction between Access, Reference, and Call dependencies. These three types have in common that they represent *executing* activities, which take care of the transformations. Below, we elaborate on these three types.

A dependency of type *Access* represents the actual usage (e.g., read or write) of a variable of a server-class. In case of such an action, an Access dependency should link only to the server class that contains the variable, and does not consider the type of the accessed variable. Access of a variable of the client-class by the client-class itself is not interesting in case of architectural dependency analysis and is not reported as an Access dependency. However, if a class accesses a variable, there is a dependency to the type of that variable as well. This dependency on the variable type is useful to report, but as type Reference, not as type Access.

Dependency type *Call* represents the invocation of a method of the server-class. In case of such an action, a Call dependency should link only to the server class that contains the method, not to the return type of the invoked method. Calling a method of the client-class by the client-class itself, is not interesting in case of architectural dependency analysis and is not reported as a dependency. Again, the dependency on the return type of the method is useful and is reported as type Reference, not as type Call. In case of chained call and/or access statements, only a Reference dependency on the return type or type of the last element in the chain is useful, since dependencies on the preceding used types are reported as Call or Access dependencies.

A dependency of type *Reference* represents a link to the server-class or an object of type of the server-class in the context of an operational activity. At code level, references are often included in access statements or call statements, where they precede the actual variable or method to appoint the used class or object. In these situations, Reference dependencies are not useful to report, since they coincide with the Access and Call dependencies. Consequently, many tools do not report these dependencies (HUSACCT also does not). However, if a reference is not followed by an access or call statement, it is useful to be reported. For example, in case an object is passed as an argument.

2.2 Dependency Subtypes

A dependency subtype provides more detail on the usage of the server-class by the client-class at a certain location in the program code. A dependency type may have several subtypes, but a subtype belongs to one dependency type only, as depicted in Figure 1. To determine which kind of information should be expressed in dependency subtypes, we used the following criteria: 1) which information is available and reliable; 2) what might be interesting for practitioners and researchers?

Several solutions are possible, based on the available data. A conceptual model of the relevant concepts in the context of a dependency is depicted in Figure 1. The figure shows that details may be provided of the used server-class, or of the used elements (methods or variables) within this class. Reversely, details may also be shown on how and where within the client-class the usage exists. Ideally, separate fields could be presented for each viewpoint and characteristic. However, the user interfaces in a tool restrict the options, so we condense information that suits our criteria in one field only: subtype.

Our approach has resulted in twenty-nine subtypes, which are shown in the tables below. The dependency types Annotation and Import have no subtypes, currently. The subtypes of the types Access and Call provide information on the kind of used server-class, or in case of a "normal" class, provide information on the used element of the server-class. The subtypes of dependency type Declaration provide additional details on how the client-class uses the server-class. The subtypes of type Inheritance tell more about the type of the super class. Finally, also the subtypes of type Reference provide additional details on how the client-class uses the server-class. The first subtype of Access, Call, and Reference each represents the default value of the type, in case none of the other subtypes could be determined for sure.

2.3 Other Characteristics

Apart from dependency type and subtype, several other characteristics of a dependency may be interesting, for example line number, position within the line, or the containing method. In addition, we describe three more-complex characteristics that may provide interesting information, often in combination with dependency type, subtype, or one of the other characteristics.

2.3.1 Direct/Indirect

The characteristic "isIndirect = true" indicates that the server-class of the dependency cannot be determined without analysis of the code of another class. In case of a direct dependency, the server-class is traceable based on information in the source code of the client-class, but in case of indirect dependencies, this is not possible. For example, chained statements require the analysis of the code of the classes that own the used variables (to determine its type) and methods (to determine its return type) in the chain. Furthermore, in case of access of an inherited instance variable, at least the code of the parent super class needs to be analyzed to determine where the variable is implemented and what type is declared for the variable. A call of an inherited method requires a similar approach.

2.3.2 Inheritance Related

The characteristic "isInheritanceRelated = true" indicates that the dependency would not exist without inheritance. An inheritance related dependency is caused by: 1) class or interface Inheritance by means of an extends or implements statement; 2) Access of an inherited variable; or 3) Call of an inherited method. These cases may lead to indirect dependencies, like a dependency on a super-super class, a Reference to the type of an accessed inherited-attribute, or to the return type of a called inherited-method.

2.3.3 Inner Class Related

The characteristic "isInnerClassRelated = true" indicates that the client-class or server-class of a dependency is an inner class.

3. Application Cases

We have analyzed the source code of three open source applications. We used HUSACCT version 4.2 to analyze the code, generate dependency reports, and generate architecture violation reports. HUSACCT, a free-to-use, open source tool, downloadable via http://husacct.github.io/HUSACCT/, provides support for architecture compliance checking and architecture reconstruction. In HUSACCT version 4.0 we have improved the dependency analysis process of HUSACT to and beyond the point where all dependencies in the benchmark test and FreeMind test [10] are detected and reported, without false positives. Version 4.2 provides all the dependency types and subtypes conform the classification in this paper. Furthermore, it provides a dependency report and a violation report with frequency statistics of the types and the other characteristics, and in addition lists of all dependencies with their characteristics.

The analyzed open source subject systems are shown in Table 1 with their version, download address and size. The source code of these systems was downloaded at May 8, 2015. The size of the systems is presented in kilo lines of code (KLOC) of all lines (including blank lines and comments) in all files with extension "java", as measured by HUSACCT_4.2. To enable comparison of the results in the two following sub sections, we excluded the packages of subject system HUSACCT 2.0 that contained classes used for testing, or ANTLR-based generated lexer and parser classes (more than 100 KLOC).

The results are shown in Table 2-5. For each subject system, the numbers of reported dependencies are shown per type, subtype, or other characteristic. The last column in each table shows the average percentage of dependencies per characteristic (calculated as the sum of the three system specific percentages of dependencies with this characteristic, divided by three).

3.1 Frequency of Types and Subtypes

An overview of the frequency of dependency types and subtypes in three open source systems is provided in Table 2. The most relevant findings are described below.

1) All dependency types and subtypes in the classification are present in the subject systems, although in various frequencies. Notable is that nearly all types and subtypes were present in all three subsystems. In case of ANTLR, no calls of enumeration methods were reported, and in case of HUSACCT, no access of interface variables and no annotations were reported. In case of Spring, all dependency types and subtypes were reported.
2) Dependencies of the types Access, Annotation, Call, Declaration, Import, Inheritance, and Reference, represent

Table 1. Open Source Subject systems

System	Download address	Size (KLOC)
ANTLR 3.5	https://github.com/antlr/antlr3/releases	77
Spring 4.1.5	https://github.com/spring-projects/spring-framework/releases	893
HUSACCT 2.0	https://github.com/HUSACCT/HUSACCT/releases	60

respectively 7.6; 2.5; 37.6; 18.7; 9.0; 1.8, and 22.8 percent of all the dependencies.

3) Although usage of a variable of another class compromises encapsulation [18], it is quite common in the three systems. However, large differences exist between the three systems. In Spring, Access dependencies are limited to 3.5 percent only, while in HUSACCT they account for 6.1 percent, and in ANTLR for 13.2 percent.

4) In case of dependencies of type Access and Call, instance members and library members are used most often. Interface methods and variables are used at a limited scale only.

5) Declarations at the level of methods (subtypes Local variable, Parameter, and Return type) outnumber the other declaration subtypes by far.

6) References account for 22.8 percent of the dependencies. The majority (16.4 percent) is of subtype "Type of Variable", subdivided in direct (15.5 percent) and indirect (0.9 percent). The direct ones are caused predominantly by passing an object as an argument, in which case a dependency on the type of the passed object is registered.

An overview of the frequency of the other characteristics (direct, indirect, inheritance related, and inner class related) is provided in Table 3. The most relevant findings are the following:

1) The majority of dependencies are direct, but a considerable fraction of the dependencies in the three systems is indirect: 12.7 percent on average. The largest fraction of indirect dependencies is of type Call, followed by Reference, Access, and finally Inheritance. Chained statements and usage of

Table 2. Frequency of dependency types and subtypes in three open source systems

Dependency Type	Subtype	ANTLR	%	HUSACCT	%	Spring	%	Average %
All	All	27,046	100	39,349	100	421,001	100	100.0
Access	*All*	*3,583*	*13.2*	*2,393*	*6.1*	*14,652*	*3.5*	*7.6*
	Variable	201	0.7	43	0.1	462	0.1	0.3
	Instance Variable	2,457	9.1	1,051	2.7	3,613	0.9	4.2
	Instance Variable Constant	11	0	90	0.2	2,062	0.5	0.2
	Class Variable	45	0.2	12	0	352	0.1	0.1
	Class Variable Constant	508	1.9	212	0.5	2,025	0.5	1.0
	Enumeration Variable	4	0	111	0.3	1,600	0.4	0.2
	Interface Variable	120	0.4	0	0	1,640	0.4	0.3
	Library Variable	237	0.9	874	2.2	2,898	0.7	1.3
Annotation	-	*982*	*3.6*	*0*	*0*	*16,369*	*3.9*	*2.5*
Call	*All*	*9,790*	*36.2*	*15,373*	*39.1*	*157,406*	*37.4*	*37.6*
	Method	415	1.5	318	0.8	5,413	1.3	1.2
	Instance Method	3,824	14.1	4,188	10.6	60,890	14.5	13.1
	Class Method	548	2.0	1,148	2.9	11,160	2.7	2.5
	Constructor	2,315	8.6	1,042	2.6	24,642	5.9	5.7
	Enumeration Method	0	0	50	0.1	234	0.1	0.1
	Interface Method	1,025	3.8	1,055	2.7	15,340	3.6	3.4
	Library Method	1,663	6.1	7,572	19.2	39,727	9.4	11.6
Declaration	*All*	*5,130*	*19.0*	*7,451*	*18.9*	*76,599*	*18.2*	*18.7*
	Class Variable	42	0.2	48	0.1	767	0.2	0.2
	Exception	189	0.7	190	0.5	10,143	2.4	1.2
	Instance Variable	351	1.3	1,341	3.4	6,629	1.6	2.1
	Local Variable	3,098	11.5	2,977	7.6	32,929	7.8	9.0
	Parameter	1,088	4.0	1,863	4.7	17,613	4.2	4.3
	Return Type	362	1.3	1,032	2.6	8,518	2.0	2.0
Import	-	*1,059*	*3.9*	*4,883*	*12.4*	*45,252*	*10.7*	*9.0*
Inheritance	*All*	*277*	*1.0*	*633*	*1.6*	*11,824*	*2.8*	*1.8*
	Extends Class	80	0.3	62	0.2	1,313	0.3	0.3
	Extends Abstract Class	91	0.3	208	0.5	2,932	0.7	0.5
	Implements Interface	85	0.3	97	0.2	5,887	1.4	0.6
	From Library Class	21	0.1	266	0.7	1,692	0.4	0.4
Reference	*All*	*6,225*	*23.0*	*8,616*	*21.9*	*98,899*	*23.5*	*22.8*
	Type	46	0.2	55	0.1	9,177	2.2	0.8
	Type Cast	392	1.4	461	1.2	5,599	1.3	1.3
	Return Type	976	3.6	1,826	4.6	18,338	4.4	4.2
	Type of Variable	4,811	17.8	6,274	15.9	65,785	15.6	16.4

Table 3. Frequency of direct, indirect, inheritance related, and inner class related dependencies.

	ANTLR	%	HUSACCT	%	Spring	%	Average %
Dependencies, all	*27,046*	*100*	*39,349*	*100*	*421,001*	*100*	*100.0*
- Direct	23,123	85.5	34,763	88.3	370,652	88	87.3
- Indirect	3,923	14.5	4,586	11.7	50,349	12	12.7
Inheritance related dependencies, all	*2,778*	*10.3*	*2,274*	*5.8*	*46,637*	*11.1*	*9.1*
- Access of inherited variable	781	2.9	629	1.6	4,368	1	1.8
- Call of inherited method	1,469	5.4	701	1.8	28,401	6.7	4.6
- Inheritance relation	277	1	633	1.6	11,824	2.8	1.8
- Reference	251	0.9	311	0.8	2,044	0.5	0.7
Inner class related dependencies, all	*1,347*	*5*	*179*	*0.5*	*48,637*	*11.6*	*5.7*

inherited members predominantly cause the indirectness of Call and Access dependencies.

2) On average, 9.1 percent of the dependencies is inheritance related. The largest fraction of inheritance related dependencies is caused by calls (4.6 percent) of inherited methods. Notable is the difference between the three systems. HUSACCT has a low average percentage of inheritance related dependencies (5.8 percent), compared to ANTLR (10.3 percent) and Spring (11.1 percent).

3) On average, 5.7 percent of the dependencies are inner class related. Great differences exist in the average percentage per system between the three systems: HUSACCT has only 0.5 percent; ANTLR has 5 percent; and Spring has 11.6 percent.

3.2 Types and Subtypes in Violations

To illustrate potential usage of the dependency types and subtypes in the context of ACC, the results of a case are presented below. Table 4 provides an overview of the frequency of dependency types and subtypes in three sets of architecture-rule-violating dependencies, while Table 5 shows the frequency of the other characteristics (subtypes with null dependencies are not shown).

The violating dependencies in the table are detected in version 2.0 of HUSACCT itself. Two classes of 25-30 students in computer

Table 4. Number of violations per dependency type and subtype in HUSACCT_2.0

Dependency Type	Subtype	Back call	%	Facade	%	Not allowed	%	Average %
All	All	506	100	193	100	333	100	100.0
Access	*All*	*0*	*0*	*6*	*3.1*	*17*	*5.1*	*2.7*
	Enumeration Variable	0	0	6	3.1	0	0	1.0
	Library Variable	0	0	0	0	17	5.1	1.7
Annotation	-	*0*	*0*	*0*	*0*	*0*	*0*	*0.0*
Call	*All*	*268*	*53.0*	*73*	*37.8*	*92*	*27.6*	*39.5*
	Method	0	0	1	0.5	0	0	0.2
	Instance Method	207	40.9	9	4.7	0	0	15.2
	Class Method	30	5.9	15	7.8	0	0	4.6
	Constructor	31	6.1	10	5.2	1	0.3	3.9
	Interface Method	0	0	38	19.7	8	2.4	7.4
	Library Method	0	0	0	0	83	24.9	8.3
Declaration	*All*	*89*	*17.6*	*27*	*14.0*	*73*	*21.9*	*17.8*
	Class Variable	1	0.2	0	0	4	1.2	0.5
	Instance Variable	32	6.3	8	4.1	5	1.5	4.0
	Local Variable	22	4.3	9	4.7	28	8.4	5.8
	Parameter	28	5.5	8	4.1	18	5.4	5.0
	Return Type	6	1.2	2	1.0	18	5.4	2.5
Import	-	*65*	*12.8*	*65*	*33.7*	*66*	*19.8*	*22.1*
Inheritance	*All*	*0*	*0*	*1*	*0.5*	*11*	*3.3*	*1.3*
	Extends Abstract Class	0	0	1	0.5	0	0	0.2
	From Library Class	0	0	0	0	11	3.3	1.1
Reference	*All*	*84*	*16.6*	*21*	*10.9*	*74*	*22.2*	*16.6*
	Type	0	0	1	0.5	0	0	0.2
	Type Cast	7	1.4	8	4.1	2	0.6	2.0
	Return Type	5	1.0	8	4.1	14	4.2	3.1
	Type of Variable	72	14.2	4	2.1	58	17.4	11.2

Table 5. Number of violations for the attributes isIndirect, isInheritanceRelated, and isInnerClassRelated.

	Back call	%	Facade	%	Not allowed	%	Average %
Dependencies, all	*506*	*100*	*193*	*100*	*333*	*100*	*100.0*
Direct	426	84.2	155	80.3	301	90.4	85.0
Indirect	80	15.8	38	19.7	32	9.6	15.0
Inheritance related dependencies, all	*36*	*7.1*	*1*	*0.5*	*11*	*3.3*	*3.6*
Inheritance relation	0	0	1	0.5	11	3.3	1.3
Call of inherited method	36	7.1	0	0	0	0	2.4
Inner class related dependencies, all	*0*	*0*	*2*	*1.0*	*0*	*0*	*0.3*

science implemented version 1.0 and 2.0 in two consecutive years. Both years, six teams worked on different components of the application. Some teams have followed the intended architecture very well, while other teams introduced quite some violations. More recent versions of HUSACCT support eleven types of rules [9]. Violations of three types of rules are aggregated per type in the tables. The back call ban allows no usage of an element in a higher layer. The facade convention disallows incoming usage of a component other than via its interface. Finally, two not-allowed-to-use rules disallowed usage between a few components, while other rules of this types disallowed the usage of "library classes with user interface components" by subsystems not assigned with responsibility to construct user interfaces. The most relevant findings are the following:

1) Violations of back call rules are caused predominantly by the invocation of methods. Of the 506 violating dependencies, 286 instances (53 percent) are of type Call, while no Access or Inheritance dependencies are reported. Strikingly is that of all the calls, not one makes use of an interface, in order to reduce coupling. Based on the number of Declaration dependencies can be observed that in total 89 variables are of a prohibited type. Furthermore, 84 dependencies of type Reference are reported. Most of them represent objects of a prohibited type, passed as an argument.
2) The 193 dependencies that violate rules of type facade convention are of more varied types. Encapsulation of the components is compromised not only by Call, Declaration, and Reference dependencies, but even by Inheritance and Access; the latter caused by usage of enumeration variables. Interestingly, two dependencies are inner class related. These appeared to represent an import and an access of a variable of a nested enumeration within the encapsulated component. A positive observation is that more than half of the Call dependencies make use of an interface.
3) Of the 333 violations of is-not-allowed-to-use rules, 323 report on the usage of a class in a restricted user interface library. Consequently, it is not surprising that all 17 Access dependencies, all 11 Inheritance dependencies, and 83 of the 92 Call dependencies identify usage of a library.

4. Related Work

We did not design the classification in isolation. We based the classification of dependency types on professional literature, on information provided by the Java and C# communities, and on research papers that distinguish different dependency types, like [17], [5], [6], [16], [12], and [14].

We also gained knowledge by the study of existing tools in the field of architecture reconstruction and ACC. A few tools that we included in a previous study [10], distinguish interesting sets of dependency types. Especially the commercial tools Lattix and Sonargraph Architect provided extensive sets. The tested version of Sonargraph Architect distinguished the following types: Call, Cast, Catch, Field type, Implementation, Inheritance, Parameter type, Read, Returns, Throws, Type parameter, Uses, Uses annotation, Uses inline, Uses new, Write. Lattix provided the following types: Annotation (Runtime Visible, Runtime Invisible), Class reference, Data Member Reference, Constructs (Null Constructor, Construct with Arguments), Invokes (Virtual Invocation, Static Invocation, Interface Invocation, Native Invocation), Inheritance (Extends, Implements). Several similarities between the types in these sets and those in our classification are visible. First, details are provided on the type of the used class (only in case of an interface), or how and where within the client-class the usage exists. Second, most types in these sets are also recognizable in our classification. Third, Lattix structures its types also in two levels. Compared to these sets, our classification adds more structure, clarity, and detail. For instance, we make a clear distinction between dependency types that represent preparing activities and executing activities; we strictly separate the types Access, Call, and Reference; we provide a balanced spread of subtypes over the types; and we provide more details on the used server class or the used element within the server class.

With respect to the results of the frequency analysis of dependency types in the three open source systems, we found some similarities in other work. For example, Tempero et al. found that inheritance structures are used actively [15], also for what we typified as access of an inherited variable or call of an inherited method. In line with their research, our study has revealed a high percentage of inheritance related dependencies in the three analyzed systems; on average 9.1 percent.

Dyer et al. [4] showed that annotations were among the most used new features of the last three Java versions. In line with their work, our study showed that annotations with a reference to another type (internal or external) accounted for 2.5 percent of the dependencies in the three analyzed systems, on average.

5. Conclusions, Limitations and Future Work

In this study, we have focused on the identification of dependency details which can be determined accurately and which might be interesting for architects and researchers in the context of architecture reconstruction and compliance checking. First, we have discussed two requirements regarding dependency characteristics: useful and accurate.

Second, we have presented a classification of dependency types, subtypes, and other characteristics of dependencies. With respect to dependency types, we have provided distinctions and definitions that help to determine dependency types accurately and to interpret assigned types correctly. We have proposed to distinguish the dependency types Access, Call, and Reference strictly and have paid special attention to these three types, since they are easily confused, while they represent very different kinds of usage, with different strengths of coupling.

Third, to illustrate the usefulness of the classified dependency characteristics in the context of architecture reconstruction, we have presented the results of a frequency analysis per characteristic in three open source systems. Fourth, to illustrate the usefulness of the classified dependency characteristics in the context of architecture compliance checking, we have presented the results of a frequency analysis per characteristic in a set of more than thousand violating dependencies, subdivided in three subsets, for three different types of rules. We have shown that all dependency types and subtypes in our classification are used in the subject systems, although in various frequencies. Furthermore, we have shown that interesting findings can be derived from the frequency analysis results. For instance, we have shown that dependencies of type "Access" are used quite frequently (7.6 percent) within the three subject systems, although these usages compromise encapsulation. In case of violating dependencies the percentage is much lower (2.7 percent).

Several limitations apply to our work. First, we do not claim that the presented classification of dependency types and subtypes is complete and final. Especially the choice of the subtypes allows several variations; at this point future research is needed. Next, we think that it is important to realize that not all dependencies in the code are detected and reported by HUSACCT; same as with other tools [10]. Deficiencies may be present in HUSACCT itself or in the included open source lexer and parser functionality. In addition, we cannot guarantee that of all the reported dependencies in the code of the subject systems, the dependency types and subtypes are reported correctly, since many variations may be present per type at the level of program code. However, we have taken great care of the accuracy of dependency detection and typifying. We have extended our automated test sets to cover all types, subtypes and the other three characteristics, and we have performed extensive manual tests.

Our work is not finished with this paper. As future work, we intend to continue our research to answer the research question to its full extent. We want to find out which types, subtypes and other characteristics are most interesting to practitioners and researchers. In addition, we want to determine at what level of abstraction a characteristic is useful, or not. Some characteristics might be useful at the level of a solitary dependency, while others might be useful at a more aggregated level. Other questions that we have in mind are the following. Do the types, subtypes and other attributes form a useful base for metrics, e.g. to determine the level of coupling, cohesion, or encapsulation at different levels of aggregation, or to determine the severity of an architecture violation? Moreover, do the types, subtypes and other attributes form a useful base for architectural restructuring advice?

6. REFERENCES

[1] Briand, L.C., Daly, J.W. and Wust, J. 1999. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*. 25, 1 (1999), 91–121.

[2] Clements, P., Bachmann, F., Bass, L., Garlan, D., Merson, P., Ivers, J., Little, R. and Nord, R. 2010. *Documenting Software Architectures: Views and Beyond*. Pearson Education.

[3] Ducasse, S. and Pollet, D. 2009. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*. 35, 4 (2009), 573–591.

[4] Dyer, R., Rajan, H., Nguyen, H.A. and Nguyen, T.N. 2013. *A large-scale empirical study of Java language feature usage*.

[5] Feilkas, M., Ratiu, D. and Jurgens, E. 2009. The loss of architectural knowledge during system evolution: An industrial case study. *2009 IEEE 17th International Conference on Program Comprehension* (May 2009), 188–197.

[6] Ko, A.J., Myers, B.A., Member, S., Coblenz, M.J. and Aung, H.H. 2006. An Exploratory Study of How Developers Seek , Relate , and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering*. 32, 12 (2006), 971–987.

[7] Peffers, K., Tuunanen, T., Rothenberger, M.A. and Chatterjee, S. 2008. A design science research methodology for information systems research. *Journal of Management Information Systems*. 24, 3 (2008), 45–77.

[8] Podgurski, A. and Clarke, L.A. 1990. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*. 16, 9 (1990), 965–979.

[9] Pruijt, L. and Brinkkemper, S. 2014. A metamodel for the support of semantically rich modular architectures in the context of static architecture compliance checking. *WICSA 2014 Companion Volume* (Apr. 2014), 1–8.

[10] Pruijt, L., Köppe, C. and Brinkkemper, S. 2013. On the Accuracy of Architecture Compliance Checking: Accuracy of Dependency Analysis and Violation Reporting. *21st International Conference on Program Comprehension* (San Francisco, CA, USA, 2013), 172–181.

[11] Pruijt, L., Köppe, C., van der Werf, J.M. and Brinkkemper, S. 2014. HUSACCT: Architecture Compliance Checking with Rich Sets of Module and Rule Types. *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14* (Sep. 2014), 851–854.

[12] Saraiva, J., Soares, S. and Castor, F. 2010. Assessing the impact of AOSD on layered software architectures. *European Conference on Software Architecture* (2010), 344–351.

[13] De Silva, L. and Balasubramaniam, D. 2012. Controlling software architecture erosion: A survey. *Journal of Systems and Software*. 85, 1 (Jan. 2012), 132–151.

[14] Stafford, J.A. and Wolf, A.L. 2001. Architecture-level dependence analysis for software systems. *International Journal of Software Engineering and Knowledge Engineering*. 11, 4 (2001), 431–451.

[15] Tempero, E., Yang, H.Y. and Noble, J. 2013. What programmers do with inheritance in java. *27th European Conference on Object Oriented Programming* (2013), 577–601.

[16] Terra, R. and Valente, M. 2009. A dependency constraint language to manage object oriented software architectures. *Software: Practice and Experience*. 39, 12 (2009), 1073–1094.

[17] Tichelaar, S., Ducasse, S. and Demyer, S. 2000. Famix and xmi. *Proceedings Workshop on Exchange Formats*. (2000), 296–299.

[18] Wirfs-Brock, R. and Wilkerson, B. 1989. Object-oriented design: a responsibility-driven approach. *Object-oriented programming systems, languages and applications (OOPSLA '89)* (1989), 71–75.