On the Congruence of Modularity and Code Coupling

Fabian Beck Stephan Diehl
Computer Science
University of Trier, Germany
{beckf,diehl}@uni-trier.de

ABSTRACT

Software systems are modularized to make their inherent complexity manageable. While there exists a set of well-known principles that may guide software engineers to design the modules of a software system, we do not know which principles are followed in practice. In a study based on 16 open source projects, we look at different kinds of coupling concepts between source code entities, including structural dependencies, fan-out similarity, evolutionary coupling, code ownership, code clones, and semantic similarity. The congruence between these coupling concepts and the modularization of the system hints at the modularity principles used in practice. Furthermore, the results provide insights on how to support developers to modularize software systems.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Software science*

General Terms

Design, Experimentation, Measurement

1. INTRODUCTION

Applying the age-old warfare strategy of *divide and conquer* to a software system, its complexity can be managed by modularizing it into smaller parts. Every non-trivial software system is modularized on different levels of abstraction and employing different techniques. For instance, classes enclose related functionalities, packages summarize classes into subsystems, components provide reusable units of source code, or aspects represent cross-cutting features.

When modularization came up, Parnas [19] introduced the concept of information hiding, which became a fundamental paradigm for designing software systems. Shortly thereafter, Stevens et al. [23] proposed the principle of low coupling and high cohesion: While modules should only be loosely coupled, the elements of a module are supposed to be highly cohesive. Moreover, Conway [11] observed that modularity of a product is connected with the structure of the organization that designs the product. These principles and theories put forward different criteria to modularize a system. *But we do not know which design principles are really used in practice.*

In this paper we want to approach this question and validate known design principles by looking at the different types of relationships that connect the entities of a software system. These relationships, which we call *code couplings*, may explain the design decisions that constitute the modularization of a system. In particular, we will compare a set of different code coupling concepts to the package structures of Java systems. This comparison will be based on a metric that measures the congruence between coupling and modularity. We will show that different principles impact the studied software systems, but none of them dominates the modular structures.

In the rest of this paper, we first briefly review different principles on modularizing software systems as well as different code coupling concepts (Section 2). Then, we present our study analyzing the relationship between the coupling concepts and the modularization of the systems (Section 3): We contrast the information contained in the coupling concepts (Section 3.1). Based on a measure of congruence between the coupling concepts and modularity, we compare the coupling types to find out which concepts impact the modularization most (Section 3.2). By classifying the modules of the systems, we furthermore derive coupling patterns for different module types (Section 3.3). While the discussion shows some limitations of the study, it also raises possible implications of the results for designing and maintaining software systems (Section 4). Finally, we present related research (Section 5) and conclude the paper (Section 6).

2. MODULARITY AND COUPLING

"For human beings, the only way to manage a complex system or solve a problem is to break it up." [3]

Modularizing a system to manage its complexity is not unique to software engineering, but is part of every engineering or design process. Baldwin and Clark [3] summarize this idea in an interdisciplinary theory on modularity. They argue that a well-modularized system or design allows concurrently working on different parts in parallel, makes a larger complexity manageable, and hides uncertainties into modules.

2.1 Modularity Principles

In software engineering, different design principles were proposed on how to create a good modular structure of software systems. We will briefly outline these principles in the following.

2.1.1 Low Coupling and High Cohesion (P1)

Stevens et al. propose to focus on the data communication dependencies of a program [23]. The target characteristic of good modularization is that the elements of each single module are highly cohesive (i.e., connected by many dependencies), while elements belonging to different modules should be low coupled (i.e., connected by few dependencies)—short, *low coupling and high cohesion*. This principle might be considered as the default approach when analyzing or retrieving the modularization of a software system.

2.1.2 Information Hiding (P2)

Parnas introduced the principle of *information hiding* [18, 19]—hiding design decisions inside modules. Slim interfaces should represent the facade of the module. The design starts with identifying a set of important design decisions that may change during development. Each of these decisions should be encapsulated in an independent module. As a result of this process, changes in the software system should be limited within the boundaries of a module. Information hiding later on became one of the main principles of the object oriented paradigm [16].

2.1.3 Conway's Law (P3)

Conway formulated a general law on the congruence between organizations and the products they design [11]: "[...] organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations."

Conway's law does not directly state guidelines to modularize a software system. Nevertheless, it implies the principle that the design of a system, and hence its modularization as a part of the design, should match the structure of the organization. Otherwise, the intended design would conflict with the social preconditions and may not be enforceable in practice.

2.1.4 Others (P4)

Besides these three often cited principles, there are other explicit modularity principles or principles that at least impact modularity.

a) Martin [15] introduced three package design principles: the *reuse-release equivalence principle*, the *common-reuse principle*, and the *common-closure principle*. These principles propose that the package reflects the granule of reuse, release, and change.

b) Architectural styles like *client-server* or *pipes-and-filters* provide a tight framework for modularization; layered architectures impose rules on the structural dependencies of the modules.

c) The idea of separating concerns, which led to the development of aspect-oriented programming [25], could govern the modular structure of a software system in form of non-cross-cutting concerns.

d) Domain knowledge is one of the key strategies to identify classes in object-oriented analysis [16].

In summary, various principles exist on how to modularize software systems. They focus on different aspects of software engineering: While the principle of low coupling and high cohesion looks at the source code, information hiding and Conway's law take the development process into account. In turn, others reflect the domain the software is part of (identification of classes) or consider the ecosystem around the software (common-reuse principle). It is reasonable to assume that these principles are not independent but interact with each other. A consistent theory or framework to formally summarize or compare these principles, however, does not exist.

2.2 Code Coupling Concepts

If applied, modularity principles, as those discussed above, leave traces in the structure and evolution of a software system. Since modularizing a system means grouping its elements based on their connecting properties, the principles are closely related to the relationships of code entities. We will use the general term *coupling* to refer to such a relationship. While the frequently used term *dependency* implies a cause and effect of the connecting property, the term *coupling* is not necessarily directed and is neutral with respect to causality. Please note, that *code coupling* has here a different meaning than the term *coupling* as used in the principle of low coupling and high cohesion (*P1*): While, in the principle, it only denotes cross-module relationships, we will use the term in general for relationships.

In this study, we focus on object-oriented software systems, namely, systems mainly written in Java. Such systems are modularized into packages that themselves may include further packages. These hierarchically organized packages form a mid-level of abstraction and will be considered as the modularization of a system in this study.

Classes and interfaces are the basic elements of a package. As a matter of simplification, we will use the term *class* interchangeable for *classes and interfaces*. Together with a coupling concept, the classes form a graph data structure. The package modularization defines a hierarchy on the classes. Formally, this data structure can be described as a five-tupel $G = (\mathcal{C}, P, E, I, \mu)$—called a *directed weighted compound graph*—where $\mathcal{C}$ is the set of classes, P is the set of packages, $E \subset \mathcal{C} \times \mathcal{C}$ is the set of directed couplings, $I \subset (\mathcal{C} \cup P) \times (\mathcal{C} \cup P)$ contains the inclusion edges of the package hierarchy, and $\mu : E \to \mathbb{R}_0^+$ is the coupling weight function. Furthermore, $((\mathcal{C} \cup P), I)$ forms a tree with leaves $\mathcal{C}$ and inner vertices P.

Quite a number of different coupling concepts have been proposed in software engineering literature. In the following we will introduce the set of coupling concepts that we will analyze in our study. We will also discuss possible relationships between the coupling concepts and the modularity principles as conjectures.

2.2.1 Structural Dependencies (SD)

A method calls another method, a class extends another class, or a class aggregates objects of another class—all this creates a direct dependency between two classes. These static structural code dependencies are most frequently used when analyzing or leveraging code coupling.

In our study we distinguish the following three types of structural dependencies.

SD.Inh *Two classes are coupled (directed) if one class extends the other class. If coupled, the coupling weight is 1.*

SD.Agg *Two classes are coupled (directed) if one class aggregates another by having class variables using the other class. The coupling weight is the number of class variables relying on the other class.*

SD.Use *Two classes are coupled (directed) if one class uses the other class in a method as a local variable, a method parameter, or by calling a method of the other class. The coupling weight is the number of methods using the other class.*

We use the tool *DependencyFinder* to retrieve these structural dependencies from compiled Java bytecode.

Conjecture: Structural dependencies are a straightforward approach to measure the data communication for the principle of low coupling and high cohesion (*P1*). Introduced prior to object-oriented design, it is, however, not clear how to rate the different types of the object-oriented structural dependencies when applying the principle.

2.2.2 Fan-Out Similarity (FO)

Structural dependencies might not only *directly* couple two code entities but also *indirectly*: Two entities connected to the same set of other entities might indicate a similar purpose or functionality of these two entities. We refer to these indirect structural couplings as *fan-out similarity*.

A feature vector describes the fan-out for each class. Each feature represents the number of direct structural dependencies (SD) to a particular class. If two vectors of two classes are similar, the classes reference similar other classes. To quantify the similarity, we employ the cosine similarity measure, which computes the angle between the vectors but does not take their lengths into account [24].

We again distinguish three types of structural information. Orthogonally, external fan-out similarity (abbreviated E), which is based on references to external libraries, contrasts internal fan-out similarity (abbreviated I), which only considers the dependencies between the classes of the system.

FO.InhE, FO.IhnI *Two classes are coupled (undirected) if they extend the same class or implement a similar set of interfaces. The coupling weight is the cosine similarity of the* ***inheritance*** *feature vectors.*

FO.AggE, FO.AggI *Two classes are coupled (undirected) if they aggregate a similar set of classes. The coupling weight is the cosine similarity of the* ***aggregation*** *feature vectors.*

FO.UseE, FO.UseI *Two classes are coupled (undirected) if they use a similar set of classes. The coupling weight is the cosine similarity of the* ***usage*** *feature vectors.*

Conjecture: Rather than optimizing coupling and cohesion (P1), grouping together similar classes with respect to fan-out similarity aligns the structural dependencies of the grouped classes. High similarity regarding external fan-out shows the common use of the same external functionality, which might hint at being covered by the same concern (*P4c*)—cross-cutting concerns have been detected based on this idea [8].

If we assume that a class of the system reflects a design decision, then, depending on the same class means depending on that design decision. Following the principle of information hiding (*P2*), the two elements depending on the same internal class should be placed into the same module to hide the design decision. Hence, similar internal fan-out similarity could be an indicator for information hiding as Schwanke argues [22].

2.2.3 Evolutionary Coupling (EC)

If two classes are frequently changed together during development, this reveals an implicit dependency. Since this information stems from the evolution of the software, we call the coupling concept *evolutionary coupling*, which is also known as *co-change coupling* or *logical coupling*.

We measure evolutionary coupling in terms of two established measures, *support* and *confidence*. The support value of a coupling counts how often the two coupled software artifacts were changed together. Additionally, the confidence value of a coupling normalizes the support by the total number of changes of one of the artifacts. While *support* is a symmetric metric, *confidence* is not because of its asymmetric normalization (see [4] for details). Support and confidence produce two coupling variants that relate the same classes but differ concerning their weights.

EC.Sup *Two classes are coupled (undirected) if they were changed together during development. The coupling weight reflects the number of common changes (**support**).*

EC.Conf *Two classes are coupled (directed) if they were changed together during development. The coupling weight reflects the number of common changes relative to the number of changes of the first compared class (**confidence**).*

We use the approach by Zimmermann and Weißgerber [28] to extract the evolutionary class dependency graphs from the version archives. As common when mining software repositories, we omit large transactions (transactions including more than 25 classes) to reduce noise.

Conjecture: Information hiding (*P2*) argues that hiding design decisions increases the changeability of a system because changing a design decision causes only local changes in one module [19]. Hence, information hiding should rather lead to local changes, which result in local evolutionary coupling. Conversely, if the changes are local, this could also have other reasons, for instance, a strict module ownership policy, where each package has an owner who is responsible for it.

2.2.4 Code Clones (CC)

Code clones are fragments of code that are equivalent or similar. They introduce another concept of coupling: For example, changing a cloned code fragment often requires to change the other fragments of the clone group, too.

Comparing two classes, we take their overlap in terms of cloned code into account to get a similarity measure. Liviery et al. [14] introduced a coverage metric for this purpose. We employ this metric in a slightly adapted version:

$$\mu_{cc}(c_1, c_2) = \frac{\|cc(c_1, c_2)\|}{\|c_1\| + \|c_2\|}$$

where $\|c\|$ is the size of class c measured in number of tokens and $cc(c_1, c_2)$ denotes the set of code fragments covered by common clones between classes c_1 and c_2.

We use the code clone detection API *JCCD* [5] and apply two configurations: The first one detects Type I clones—exact matches of code fragments ignoring code layout and comments. The second one detects Type II clones—code fragments that are equivalent when generalizing from identifier names and also ignoring code layout and comments [20].

CC.I *Two classes are coupled (undirected) if they share* ***Type I clones****. The coupling weight is the clone coverage* μ_{cc}

CC.II *Two classes are coupled (undirected) if they share* ***Type II clones****. The coupling weight is the clone coverage* μ_{cc}

Conjecture: It may sound strange that the *bad* smell of duplicated code should indicate a *good* design principle. But code clones are often introduced for good reasons, for example, performance improvement, forking, generative programming, or reducing dependencies [20]. Depending on the purpose, code clones may disguise other couplings. For instance, copied code could replace an inheritance connection (*SD.Inh*) or a method call that would otherwise lead to a usage dependency (*SD.Use*) or a similar fan-out (*FO*).

2.2.5 Code Ownership (CO)

The main author of a code entity is the expert to be contacted when there is a problem or question concerning this code. If two code entities have the same author or authors, they could be covered by the same expertise.

To compare classes based on their owner, we list the developers who have ever changed a particular class and transform this list into a binary vector. In a more elaborate approach, we proportionally quantify the code ownership: For each class, the number of changes made by a particular author is divided by the total number of changes. Both metrics produce couplings relating the same classes but with different weights.

CO.Bin *Two classes are coupled (undirected) if they share common authors. The coupling weight is the cosine similarity of the* ***binary*** *ownership vectors.*

CO.Prop *Two classes are coupled (undirected) if they share common authors. The coupling weight is the cosine similarity of the* ***proportional*** *ownership vectors.*

Conjecture: Bowman and Holt [7] argue that, if two code entities have a common author, communication between the developers of two entities is likely. Hence, code ownership similarity is a way to map parts of the communication structure of the development team into the source code. Referring to Conway's Law (*P3*), the design of the system is said to follow this communication structure.

2.2.6 Semantic Similarity (SS)

Interpreting the source code of classes as plain text documents, the classes are coupled, like text documents, by their common vocabulary. In this case, the vocabulary consists of the identifiers used in the code and the words in the comments. Standard information retrieval methods for semantically relating documents can be applied.

Each class is described as a bag of words which is transformed into a vector where each dimension counts the number of occurrences of a particular term. To counterbalance the importance of frequently occurring terms across documents, we apply the *tf-idf* measure. Furthermore, we use *latent semantic indexing* (*LSI*) to consider related terms as a concept and to reduce noise.

SS.Tfidf *Two classes are coupled (undirected) if they share a similar vocabulary. The coupling weight is the cosine similarity of the* ***tf-idf*** *document vectors.*

SS.LSI *Two classes are coupled (undirected) if they share a similar vocabulary. The coupling weight is the cosine similarity of the* ***LSI*** *document vectors.*

In particular, we implemented an approach very similar to the one used by Kuhn et al. to cluster source code documents [13]: From the source code of a class including comments, we excluded the license text and, much more important in our scenario, the package name and the names of its ancestor packages. The preprocessing splits camel-case words into their components, removes meaningless words based on Java key words and a list of English stop words, and normalizes the words by reducing them to their stem based on the *Porter* algorithm. The resulting document vector is balanced by applying *tf-idf*, and finally transformed into an *LSI* model with 30 dimensions. We applied the Python APIs *stemming* and *gensim* to implement the process.

Conjecture: Among the presented coupling concepts, the domain knowledge incorporated in the package design (*P4d*) might be best reflected in the vocabulary used in the source code text. The terms describing a domain concept should also appear in the code as identifier names or in comments. The problem, however, could be that, besides these domain-related terms, many other circumstances influence the vocabulary, like reference to other code entities, information on the author, applied design patterns that bring in a certain terminology, etc.

3. STUDY

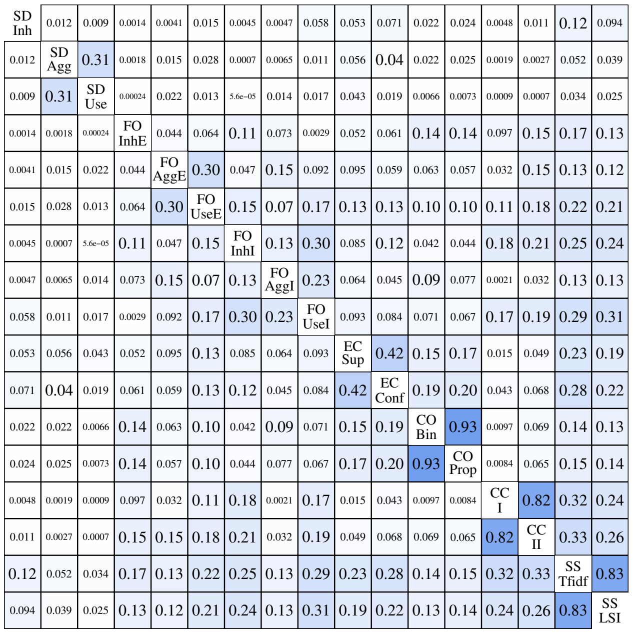
Based on the source code and its accompanying information, it is hard to directly conclude what principles were used to design the modular structure of a software system. Hence, we abstract the software project by reducing it to a set of different coupling concepts between the classes of the system. Analyzing the coupling concepts and their impact on modularity builds the foundation for indirectly drawing conclusions on principles applied. In particular, we want to empirically answer three questions:

- How do the different coupling concepts relate to each other? (Section 3.1)
- What is the impact of each coupling concept on the modularization? (Section 3.2)
- Are there specific differences in this impact for different package types? (Section 3.3)

The study examines the modularity of 16 open source software projects—Table 1 provides a complete list. Each of these projects is mainly written in Java and hosted at *sourceforge.net*. We selected a broad spectrum of application types, varying from small to mid-size projects. The number of packages that directly contain classes shows the size of the hierarchy, the number of classes provides an estimate of the size of the project. *CVS* or *SVN* repositories provide data about the development history of the projects. For the studied projects, the available development activity spans 10 months to up to 9 years or 126 to up to 4602 check-in transactions (only counting the transactions where relevant Java classes were checked in) involving up to 23 developers. To extract the coupling information, we mirrored the repositories and downloaded a compiled version of the software in form of a *jar* file. The structural dependencies (*SD*) as well as the fan-out references (*FO*) were derived from the *jar* file. By the date of release, we determined the corresponding version of the Java source code. This version is used to retrieve the code clone information (*CC*) and the semantic similarity (*SS*). Finally, the extraction of evolutionary coupling (*EC*) and code ownership coupling (*CC*) required to process the whole repository from the first check-in up to the version corresponding to the selected release. We used the released *jar* file as a reference to determine the set of classes considered for analysis (excluding contained third-party libraries). Other classes possibly contained in the repository like test classes, experimental functionality,

Table 1: Properties of the sample of software projects and their repositories.

Project	Description	Version	#Pack.	#Classes	Repos.	Time frame	#Trans.	#Dev.
Checkstyle	coding conventions	5.1	21	261	SVN	2001-06-22 – 2010-02-16	1335	6
Cobertura	test coverage	1.9.4.1	19	99	SVN	2005-02-12 – 2010-03-03	226	6
CruiseControl	continuous integration	2.8.4	27	295	SVN	2001-03-26 – 2010-09-16	1615	10
iText	*PDF* library	5.0.5	24	402	SVN	2007-12-20 – 2010-11-02	817	7
JabRef	*BibTeX* management	2.6	37	499	SVN	2003-10-16 – 2010-04-14	1348	23
JEdit	text editor	4.3.2	28	488	SVN	2001-09-03 – 2010-05-09	2927	23
JFreeChart	chart library	1.0.13	37	587	SVN	2007-06-19 – 2009-04-20	551	2
JFtp	*FTP* client	1.0	7	78	CVS	2002-02-06 – 2003-03-23	155	5
JHotDraw	GUI framework	7.6	65	679	SVN	2006-11-22 – 2011-01-09	302	2
JUnit	regression testing	4.5	23	119	CVS	2007-12-07 – 2008-08-08	126	2
LWJGL	gaming library	2.7.1	27	564	SVN	2002-08-09 – 2011-02-10	1557	11
PMD	code problems	4.2	47	565	SVN	2002-06-24 – 2008-03-26	2041	18
Stripes	web framework	1.5.5	19	238	SVN	2005-09-07 – 2011-01-04	812	7
SweetHome3D	interior design	3.1	8	167	CVS	2006-04-11 – 2011-02-13	1807	1
TV-Browser	program guide	2.7.6	62	485	SVN	2003-04-25 – 2010-12-19	4602	12
Wicket	web framework	1.2.2	86	622	SVN	2004-12-21 – 2006-08-27	3456	12



SD Inh	0.012	0.009	0.0014	0.0041	0.015	0.0045	0.0047	0.058	0.053	0.071	0.022	0.024	0.0048	0.011	0.12	0.094
0.012	SD Agg	0.31	0.0018	0.015	0.028	0.0007	0.0065	0.011	0.056	0.04	0.022	0.025	0.0019	0.0027	0.052	0.039
0.009	0.31	SD Use	0.00024	0.022	0.013	5.6e−05	0.014	0.017	0.043	0.019	0.0066	0.0073	0.0009	0.0007	0.034	0.025
0.0014	0.0018	0.00024	FO InhE	0.044	0.064	0.11	0.073	0.0029	0.052	0.061	0.14	0.14	0.097	0.15	0.17	0.13
0.0041	0.015	0.022	0.044	FO AggE	0.30	0.047	0.15	0.092	0.095	0.059	0.063	0.057	0.032	0.15	0.13	0.12
0.015	0.028	0.013	0.064	0.30	FO UseE	0.15	0.07	0.17	0.13	0.13	0.10	0.10	0.11	0.18	0.22	0.21
0.0045	0.0007	5.6e−05	0.11	0.047	0.15	FO InhI	0.13	0.30	0.085	0.12	0.042	0.044	0.18	0.21	0.25	0.24
0.0047	0.0065	0.014	0.073	0.15	0.07	0.13	FO AggI	0.23	0.064	0.045	0.09	0.077	0.0021	0.032	0.13	0.13
0.058	0.011	0.017	0.0029	0.092	0.17	0.30	0.23	FO UseI	0.093	0.084	0.071	0.067	0.17	0.19	0.29	0.31
0.053	0.056	0.043	0.052	0.095	0.13	0.085	0.064	0.093	EC Sup	0.42	0.15	0.17	0.015	0.049	0.23	0.19
0.071	0.04	0.019	0.061	0.059	0.13	0.12	0.045	0.084	0.42	EC Conf	0.19	0.20	0.043	0.068	0.28	0.22
0.022	0.022	0.0066	0.14	0.063	0.10	0.042	0.09	0.071	0.15	0.19	CO Bin	0.93	0.0097	0.069	0.14	0.13
0.024	0.025	0.0073	0.14	0.057	0.10	0.044	0.077	0.067	0.17	0.20	0.93	CO Prop	0.0084	0.065	0.15	0.14
0.0048	0.0019	0.0009	0.097	0.032	0.11	0.18	0.0021	0.17	0.015	0.043	0.0097	0.0084	CC I	0.82	0.32	0.24
0.011	0.0027	0.0007	0.15	0.15	0.18	0.21	0.032	0.19	0.049	0.068	0.069	0.065	0.82	CC II	0.33	0.26
0.12	0.052	0.034	0.17	0.13	0.22	0.25	0.13	0.29	0.23	0.28	0.14	0.15	0.32	0.33	SS Tfidf	0.83
0.094	0.039	0.025	0.13	0.12	0.21	0.24	0.13	0.31	0.19	0.22	0.13	0.14	0.24	0.26	0.83	SS LSI

Figure 1: Correlation between coupling graphs.

or unused legacy code are skipped. For simplification, we ignored anonymous and inner classes by considering them just as parts of the containing class.

3.1 Dimensions of Coupling

The first part of our analysis will compare the different coupling concepts themselves, yet ignoring the modular structure of the system. We may consider each concept as one dimension to describe a software system. If two concepts share some information, these dimensions would not be orthogonal, but correlated.

3.1.1 Method

For a project, each coupling concept provides a graph on the same set of classes. To compare the concepts, we describe each concept as a vector. A dimension of such a vector represents a pair of classes; its value reflects the weight of the edge between the two classes or is zero if there is no edge for this combination. Hence, a project consisting of n classes is described as an n^2-dimensional vector for each coupling concept. Concatenating theses vectors for all projects, we get a summarized vector $\vec{v}_{co}$ describing the coupling concept co. To equally weigh each project, we align the number of dimensions by randomly sampling 20,000 dimensions (with repetition) for each project. Finally, the Pearson correlation of two vectors $\vec{v}_{co_1}$ and $\vec{v}_{co_2}$ provides a measure of similarity for two concepts co_1 and co_2.

3.1.2 Results

We computed the pairwise correlation for all combinations of coupling concepts and visualize the resulting correlation matrix in Figure 1. High values indicate a strong (positive) correlation between two coupling concepts, which means, if two classes are coupled by the first concept, it is likely that they are also coupled by the second concept and vice versa.

The first, quite surprising observation is that the matrix is sparsely filled with high correlations—only few combinations of concepts near the diagonal are strongly correlated. These strong relationships can be identified as similarities between metrics measuring variants of the same concept, for instance, the two code ownership metrics *CO.Bin* and *CO.Prop*. But it is not always true that variants of the same concept provide similar data as the inheritance dependencies show: *SD.Inh* is only weakly correlated to the other variants of the structural dependencies. Such a statistical independence advocates to consider inheritance dependencies as an independent concept.

Off the diagonal, we mostly observe very weak correlations, which indicates that the coupling concepts indeed provide different coupling information. Though based on the same underlying information, structural dependencies are not considerably correlated with fan-out similarity. Analyzing the few somewhat stronger cross-concept correlations (0.10 – 0.35), we observe some interesting relationships:

- Ownership couplings (*CO*) correlate with evolutionary couplings (*EC*). An explanation is that both are based on the check-in information. Two files can only be coupled by evolution if they share at least one common author.

- Coupling by code clones (*CC*) shares some information with internal and external inheritance and usage fan-out (*FO.InhE*, *FO.UseE*, *FO.InhI*, *FO.UseI*) but clearly less with aggregation fan-out (*FO.AggE*, *FO.AggI*). The relationship to inheritance could be caused by code that implements or overrides the same methods in the same or similar way in two sibling classes of the inheritance hierarchy. Furthermore, the references used in code clones are necessarily equal and lead to a certain usage similarity, but do not directly imply similar aggregation.

- Semantic similarity (*SS*) is linked to most other coupling metrics, only structural aggregation and usage dependencies form an exception (*SD.Agg*, *SD.Use*). Since this similarity is based on the raw source code text, it partly aggregates other source code based coupling concepts (*SD*, *FO*, *CC*). Often the name of the author is also mentioned in the source code, which explains the correlation to code ownership (*CO*). For the correlation to evolutionary coupling (*EC*), we could not find an evident explanation.

In summary, the correlation matrix shows that the different coupling concepts are quite independent and thus each represents different coupling information. The correlations between metrics for variants of the same concept confirm the grouping of the metrics—except inheritance dependencies, they should not be equated with other structural dependencies.

3.2 Coupling-Modularity Congruence

The central part of this study is to relate the different coupling concepts to the modularity of the software systems. Thus, we have to find a measure of the congruence between a coupling concept and the modularization. This comes down to the problem of how to relate a graph structure (the coupling graph) and a hierarchy (the modularization).

3.2.1 Method

Usually, the extent to which the structure of a system matches its call graph is measured by coupling and cohesion as proposed by the modularity principle of low coupling and high cohesion (*P1*). Since we capture each concept as a graph $G = (\mathcal{C}, P, E, I, \mu)$ (Section 2.2), we are able to apply the same approach to measuring the congruence between coupling and modularity for all coupling concepts. Hence, in the following, *coupling* and *cohesion* are general properties independent of a coupling concept and do not extend the terms used in the modularity principle of low coupling and high cohesion.

An edge connecting two classes of the same package contributes to the cohesion of the package and moreover to the cohesion of the whole system. Just as well, an edge connecting two classes of different packages contributes to the coupling. This model, however, disregards the hierarchical structure of the packages. A more elaborate model also considers the distance of two packages in the hierarchy. Now, an edge e partly accounts to coupling and cohesion depending on the distance it spans.

$$\mathit{cohesion}(e) = \frac{\mathit{weight}(e)}{\mathit{distance}(e)}$$

$$\mathit{coupling}(e) = \mathit{weight} - \mathit{cohesion}(e)$$

The distance function $\mathit{distance}(e)$ counts the number of intermediate nodes on the shortest path through the package hierarchy that connects the source and target of an edge e. Thus, the distance is 1 if the source and target class are in the same package, 2 if, for instance, the source is in a sub-package of the package of the target, etc. The sum of cohesion and coupling always is equal the weight of the edge.

To express coupling and cohesion for a set of classes $C \subseteq \mathcal{C}$ (e.g., a package or the whole system), we sum up the respective cohesion and coupling values for all outgoing edges $\mathit{out}(C)$ and incoming edges $\mathit{in}(C)$ of the classes in the set.

$$\mathit{cohesion}(C) = \sum_{e \in \mathit{out}(C)} \mathit{cohesion}(e) + \sum_{e \in \mathit{in}(C)} \mathit{cohesion}(e)$$

$$\mathit{coupling}(C) = \sum_{e \in \mathit{out}(C)} \mathit{coupling}(e) + \sum_{e \in \mathit{in}(C)} \mathit{coupling}(e)$$

Finally, the cohesion in relation to the coupling expresses the congruence of the coupling graph and the modular structure of the system for a set of classes C, which we denote as *coupling-modularity congruence* (short, *congruence*).

$$\mathit{congruence}(C) = \frac{\mathit{cohesion}(C)}{\mathit{cohesion}(C) + \mathit{coupling}(C)}$$

High congruence values indicate high consistency between coupling and modularity. If the coupling graph is totally independent of the modularity the congruence, however, is not zero but larger because $\forall e \in E : \mathit{cohesion}(e) > 0$. Hence, we have to compute the congruence of such a graph for every project to get a project-specific baseline of the congruence metric. To this end, we take the complete graph which fulfills the independence of the coupling structure and the modularity—every other graph fulfilling the independence condition is just a modularity-independent random sample of the complete graph.

The congruence metric measures the quality of the coupling information with respect to modularity in the graph but not the richness or density of this information. This is intended because we do not want to mix quality and density. Nevertheless, we need a further metric to also measure the density of the congruence information.

The congruence density is high for a set of classes C if the congruence information is good for each class of the set. Hence, we take the average of this class-specific congruence to get a density measure, which we denote as *average class coupling-modularity congruence* (short, *average congruence*).

$$\mathit{congruence}_{avg}(C) = \frac{1}{|C|} \sum_{c \in C} \mathit{congruence}(\{c\})$$

When there is no information available for a class c, hence $\mathit{out}(\{c\}) = \mathit{in}(\{c\}) = \emptyset$, the function $\mathit{congruence}(\{c\})$ is undefined because $\mathit{cohesion}(\{c\})$ and $\mathit{coupling}(\{c\})$ are undefined. In that case, we consider instead the respective baseline value for $\{c\}$ to define the average.

3.2.2 Results

To analyze the relationship between coupling concepts and modularity, we compute the congruence and average congruence for all combinations of projects and coupling concepts,

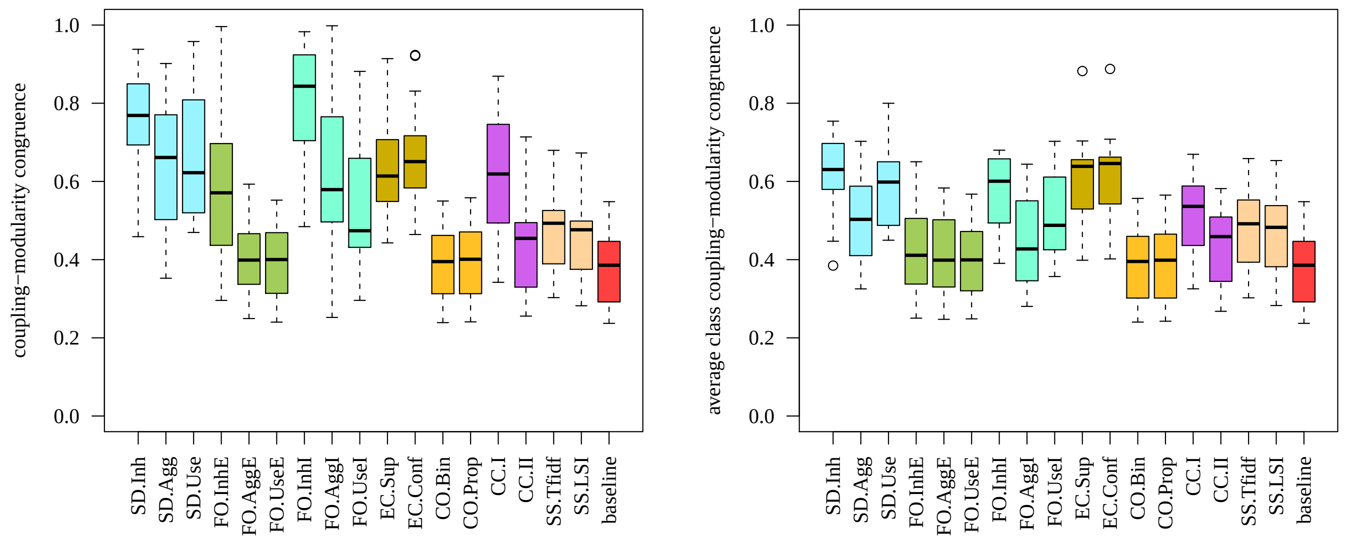


Figure 2: Coupling-modularity congruence and average class coupling-modularity congruence.

in each case considering all classes of the project. Summarized by concept, the result hence is a distribution of congruence values. We present these distributions as box plot diagrams in Figure 2—one diagram for congruence, the other for the average congruence. Such box plot diagrams sketch a distribution by plotting its main characteristic values: the median (horizontal line), the two mid-quartiles (filled box), and the outliers (dashed lines and circles).

The results of structural dependencies (*SD*) show high coupling-modularity congruence values clearly exceeding the baseline congruence. Hence, structural dependencies seem to preferably link classes of the same or neighboring packages. Among the three variants, the inheritance dependencies (*SD.Inh*) provide the best results. Taking also the information density into account, the average congruence values of structural dependencies are also among the overall best results. The usage dependencies gain against the aggregation dependencies.

For the fan-out similarity (*FO*), we observe large differences between external and internal fan-out. While the internal fan-out provides good congruence values, only the inheritance variant of the external fan-out (*FO.InhE*) is able to compete. External aggregation (*FO.AggE*) and usage fan-out (*FO.UseE*) do not seem to be a relevant criterion for modularity because they do not notably top the baseline values. The overall best congruence is reached by taking the internal inheritance fan-out (*FO.InhI*) into account. This fan-out similarity is also among the best average congruence values. Similar to the structural dependencies, again the usage information (*FO.UseI*) gains with respect to density and, together with the inheritance information (*FO.InhI*), shows competitive results for internal fan-out.

Evolutionary coupling (*EC*) is also congruent to the modularity of the systems—the congruence metric shows about equally high values as for structural aggregation and usage dependencies (*SD.Agg*, *SD.Use*). The difference between the two variants of the concept of evolutionary coupling is small but consistently better for the confidence metric (*EC.Conf*) than for the support metric (*EC.Sup*). Relating the common number of changes to the number of total changes as it is done in the confidence metric here shows a positive effect. With respect to density, the results for both evolutionary coupling metrics are very good—the median even is the highest compared to all other concepts.

Code ownership similarity (*CO*) does not impact modularity as clearly as most other analyzed coupling concepts. The congruence and average congruence only slightly exceed the baseline. This is, by the way, no problem of granularity—ignoring the lowest levels of packages, the congruence does not improve with respect to the baseline.

Instances of the same Type I clone can be often found in the same package as indicated by the high congruence values for this metric (*CC.I*). This is not true to the same extent for the more lax Type II clones (*CC.II*). Surprisingly, the density of Type I clones is high enough to also provide good average congruence values. Here, the inherently denser Type II clone information shows at least medium results.

Finally, the semantic similarity (*SS*) turns out to only be a second class factor for modularity. Though the congruence and average congruence clearly rise above the baseline, they cannot compete with the high values of other concepts. Notably, the more elaborate *LSI* variant of the semantic similarity (*SS.LSI*) consistently falls below the simpler *tf-idf* variant (*SS.Tfidf*). The negative impact of loosing some information by dimension reduction seems to outweigh the positive effect of noise reduction.

In general, we got the best coupling-modularity congruence for structural dependencies (*SD*), internal fan-out similarity (*FO.*I*), evolutionary coupling (EC), and Type I code clones (*CC.I*). Among the structural information, inheritance information outperforms for structural dependencies as well as for fan-out similarity especially with respect to quality. Since inheritance is nearly uncorrelated to the other structural information (Section 3.1), we observe here different highly congruent independent coupling concepts.

3.3 Module Type Congruence

The third and last part of the study investigates whether there are individual differences in the coupling-modularity congruence between different types of modules. A variation

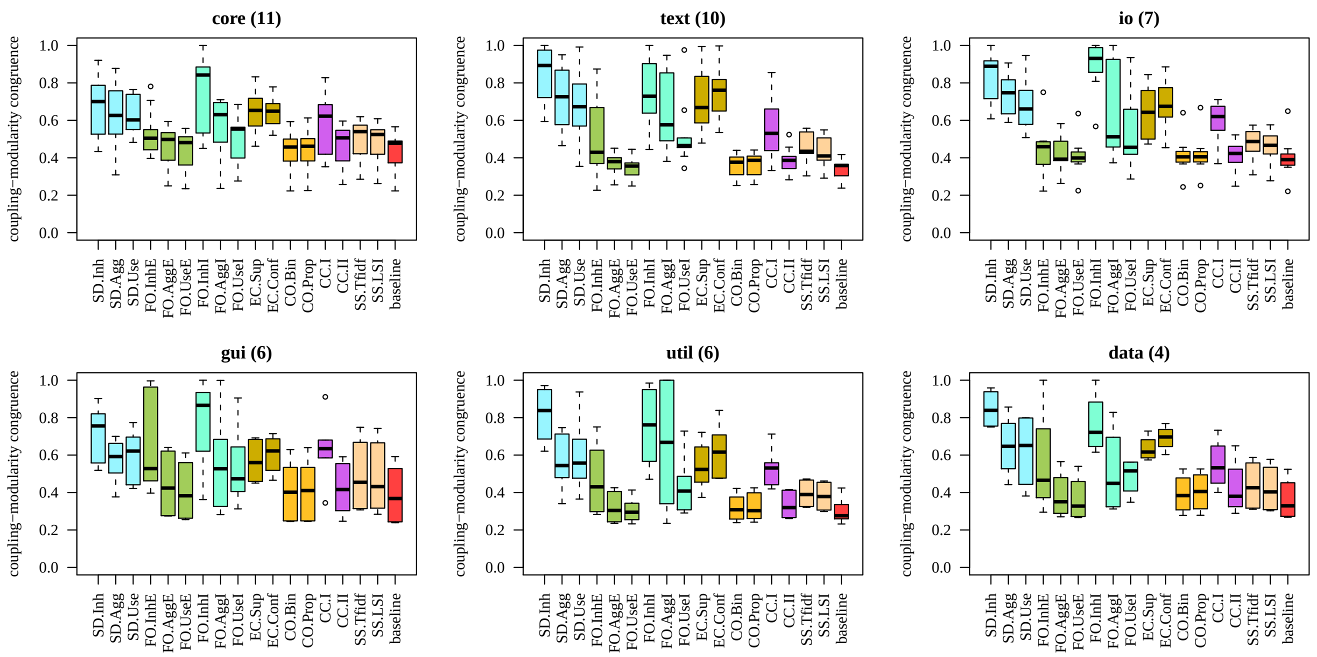


Figure 3: Coupling-modularity congruence for the most frequently occurring package types.

Table 2: Package types.

Type	Description
core	*implements the core functionality*
data	*encapsulates data and manages data structures*
event	*handles temporal events, actions, messages or exceptions*
graphics	*draws 2D or 3D graphics*
gui	*creates and controls the graphical user interface*
io	*reads or writes data from/to a device*
text	*parses, transforms, or creates textual data*
util	*provides supporting functionality*

could indicate that different design principles are used in different contexts.

3.3.1 Method

The first step towards such a module type congruence is to manually classify the packages of the software systems into distinctive types. We identified a set of package types that repeatedly appear in the projects (Table 2) and applied this classification to the main packages of the 16 software projects—sub-packages of the classified ones are assumed to belong to the same type. Some packages could not be unambiguously assigned to a type, but they can be safely omitted because the following analysis process does not rely on a complete classification.

For each project, we evaluate $congruence(C_t)$ where C_t is the set of classes included in a package assigned to type t to get a congruence value specific for a package type. We summarize each package type by considering all projects that at least contain 6 classes of this type.

3.3.2 Results

We present the results of this analysis by showing box plots of the six most frequently found package types (Figure 3). Each box plot depicts one type, the value in parentheses refers to the number of projects this type was detected in. The interpretation of the congruence values again depends on the baseline, and thus, the diagrams cannot be compared without considering the slightly different baseline values.

The box plot diagrams roughly show similar patterns compared to each other. This pattern reflects most of the observations we already described in the previous part of the study (Section 3.2). Hence, these general results virtually sustain for the different package types. Even utility packages, which are said to follow different laws than other packages [26], show this pattern together with considerably high congruence values. Nevertheless, there are small differences between the package types:

- Packages of type *io* and *gui* are exceptionally congruent with respect to the internal inheritance fan-out similarity (*FO.InhI*). Together with a high congruence value for *SD.Inh* this could hint at a special significance of inheritance in these packages.
- Overall the *core* packages show somewhat lower congruence values compared to the baseline. Here, factors that could not be grasped by the selection of coupling concepts might impact their design.
- For *text* packages, some metrics particularly exceed the baseline, including structural dependencies (*SD*), internal fan-out similarity (*FO.*I*), and evolutionary coupling (*EC*). This type of package seems to be particularly related to these coupling concepts.

4. DISCUSSION

So far, we have analyzed the results of the study with respect to the congruence of the coupling concepts and modularity. To finally get back to the modularity principles introduced in Section 2.1, we will next discuss the limitations of the results and interpret the results with respect to these principles. Beyond that, we will provide an outlook on how the results could be directly leveraged in applications regarding modularity.

4.1 Threats to Validity

The generalizability of the results is limited by the restricted set of software projects as well as by other design decisions of the study. The 16 software projects are open source systems written in Java and consist of up to 86 packages and up to 679 classes. Closed source projects might lead to different results especially with respect to evolutionary and code ownership couplings because they might follow a fundamentally different development process. Larger open-source projects may as well be organized differently. Furthermore, the results cannot be directly transferred to non-object-oriented systems because, there, concepts like inheritance do not exist.

We looked at modularity at a mid-level of abstraction and equate *module* with *package*. Hence, our results only apply to this level. For example, grouping methods into classes could follow totally different standards.

Though we covered many important coupling concepts, we did not exhaustively include all established concepts, for instance, dynamic dependencies or coupling by bug reports. Furthermore, the applied metrics are only heuristics to measure the respective concept. Although we tried to use established metrics, there could exist more appropriate metrics. Analogously, the introduced congruence metric might be unsuitable.

4.2 Modularity Principles

We must beware of over-interpreting the results with respect to modularity principles because we can only conjecture about the relationship of the principles to the coupling concepts. A high congruence for a particular concept might be caused by the application of a modularity principle, but could also be the consequence of another, unknown property. Furthermore, low congruence values do not necessarily indicate the absence of a particular modularity principle, but could also stem from weaknesses in measuring the principle by this concept. Nevertheless, the following implications can be considered as circumstantial evidence on the practical relevance of modularity principles and might be used as hypotheses for future research.

The high congruence values for structural dependencies (*SD*) suggest that the principle of low coupling and high cohesion (*P1*) is used to modularize software systems in practice as one of the dominating principles. The particularly high congruence values for inheritance (*SD.Inh*) together with its independence of the other structural dependencies (Section 3.1) highlights the role of inheritance. It appears to be the prevailing structural dependency type to implement the principle.

Also Information hiding (*P2*) seems to play a major role in the analyzed projects as the high congruence values for internal fan-out similarity (*FO*) and evolutionary coupling (*EC*) suggest. Again inheritance (*FO.InhI*) is outstanding as it provides the overall best coupling-modularity congruence result and might also be a distinguished aspect in the application of information hiding.

Code ownership (*CO*) does not play an important role in the package design of the investigated projects. A likely conclusion is that the impact of Conway's Law (*P3*) is small in contrast to other principles.

The external fan-out similarity (*FO.*E*), which we assume to be related to the separation of concerns (*P4c*), notably impacts modularity only with respect to inheritance (*FO.InhE*). The low result matches the observation that often concerns cross-cut the hierarchical structure of the system [25]. It would be interesting to also take aspects into account and compare the congruence of aspect-oriented and non-aspect-oriented systems with respect to external fan-out.

The role of domain knowledge (*P4d*) is hard to judge from the results of our study. The semantic similarity (*SS*) provides medium congruence. On the one hand, we may assume that domain knowledge is irrelevant and the medium congruence is just a result of the correlation to other high congruent coupling concepts (Section 3.1). On the other hand, we may assume that domain knowledge is a driving factor, but it is not appropriately reflected in the applied metric.

Finally, code clones (*CO*) clearly relate to the modular structure of the systems and are frequent enough to also provide medium data density (average congruence). Hence, they may significantly impact modularity principles, but we do not know their relation to the principles because code clones could hide different types of code coupling.

Concerning the module-type-specific properties, we were surprised by the stable congruence patterns that we observed for different types (Section 3.3). This is an indicator, that the the same modularity principles or guidelines are applied regardless of the package type.

4.3 Modularity Applications

Many tasks and problems in software engineering are related to modularity. First of all, finding a good modularization is a major part of designing a software system. The initial modularization might degenerate during the development process or could prove to be flawed so that a later remodularization would be necessary. In general, documenting the modularization is helpful for navigating and understanding the code.

Tools based on code couplings might support the developer in these modularity-related applications. Such tools, however, often considered coupling a one-dimensional concept, for instance, equating coupling with structural dependencies. In contrast, our results clearly show that coupling consists of several dimensions (Section 3.1), many of them impacting the modularity (Section 3.2). Exploiting combinations of coupling concepts could improve modularity tools like tools for software clustering, aspect mining, or component extraction. For instance, in the domain of software clustering, combining a few data sources has been already successfully applied [2, 4, 6]. Moreover, it is to assume that other applications of coupling data besides modularity are neither dominated by one coupling concept and could also profit from integrating different coupling concepts.

In detail, this study contains some findings that might be of particular interest for designing automatic modularization tools:

- It seems to be useful to discriminate structural dependencies (*SD*) further. At least inheritance (*SD.Inh*) should be considered as an independent concept because otherwise aggregation and usage dependencies might weaken the high impact of the inheritance dependencies.

- Including the highly congruent and dense Type I clone information ($CO.I$) is likely to have a positive effect because clones may disguise parts of the relevant coupling information.

- Using fan-out information (FO), it could be beneficial to discern external and internal fan-out because of its clearly different congruence to modularity that we observed in this study.

The conjectured relationships between the code coupling concepts and the modularity principles provide the opportunity to check whether the development team follows a certain principle. This could be a useful instrument for monitoring the development process. The congruence results of our study provide a reference of how modularity relates to coupling concepts in other projects.

We did not observe large differences for the coupling-modularity congruence between different package types (Section 3.3). This supports the decision of many existing automatic modularization approaches to handle all types of modules alike.

5. RELATED WORK

Abreu and Goulão [1] present an approach to find out whether coupling and cohesion are the driving forces for the modularity of a software system. They optimized the modularization with respect to this principle. The resulting modularization significantly exceeded the metric value of the original modularization. The authors conclude that "*the ideal of minimal coupling and maximal cohesion [...] does not match practitioners' reality at least in what concerns the modularity of object-oriented systems.*" Based on our new results, we can qualify this statement further: Coupling and cohesion still seem to be an important factors for modularity, but there appear to be other about equally important ones.

Sarkar et al. [21] discuss different principles used in module design, namely, *similarity of purpose*, *published API*, *compilability*, *extendibility*, *testability*, *acyclic dependencies*, and *module size*. They propose a set of metrics to quantify to what extent a modular structure follows these principles. The results of their case study validate the metrics with respect to measure modularity improvements. The applied metrics thoroughly investigate different kinds of couplings based on structural dependencies. By concluding that a single metric is not sufficient to estimate the quality of a modularization, they back our result that code coupling is a multi-dimensional construct. In contrast to our work, they focus on creating modularity quality metrics but do not aim at comparing different coupling concepts.

Evaluating different code coupling concepts to automatically retrieve the modular structure of a system by clustering, Beck and Diehl [4] compared structural dependencies and evolutionary couplings. They retrieved better results for structural dependencies because of a higher data density. Combining structural dependencies and evolutionary couplings improved the clustering result. Andritsos and Tzerpos [2] added meta information like developer, directory, number of lines of code, and the time stamp of the file to structural dependencies and were also able to improve the quality of the clustered modularization. Wierda et al. [27] showed how combining structural dependencies from multiple versions can be used to improve the clustering result. Bittencourt et al. [6] enhance the assignment of new classes to modules by enqueing semantic similarity and structural dependencies in the assignment process. These different improvements in automatically modularizing software systems by integrating different coupling concepts already exploit the multi-dimensionality of code couplings. Our study predicts that more rigorously integrating coupling concepts could lead to further improvements.

Comparisons between coupling concepts have been conducted not only to explain or construct modules. For instance, Cataldo et al. [9] found that high congruence between code couplings and the communication between developers leads to better productivity—for structural dependencies as well as for evolutionary couplings. In a further study [10], they focus on the impact of different coupling types on failures and conclude that evolutionary and work-related couplings have a higher impact on failures than structural dependencies.

The introduced congruence metric is similar to some metrics applied in software clustering. If we assume the package structure to be flat instead of hierarchical, our metric is closely related to the *clustering factor* metric used in the software clustering tool Bunch [17], to the *score of a cluster* defined for the *EVM* metric [12], or to the *intra modular coupling density* [1].

6. CONCLUSION

This work is a first step towards understanding, based on conjectured relationships to code couplings, how modularity principles are used in practice. We compared a broad spectrum of coupling concepts on a substantial set of software projects. The introduced measure of congruence connects code coupling and modularity while also taking the hierarchical structure of the modularization into account. The results of the study provide hints how modularity principles are applied in object-oriented open source software systems today. These findings could be used as hypotheses for future research. The study furthermore yields recommendations to enhance existing modularization approaches and tools.

Overall, we observed that none of the principles is exclusively dominating the modularity of the studied systems. Among the coupling concepts, a similar internal inheritance fan-out, in other words, the sibling relation in the internal inheritance hierarchy has the highest impact on modularity.

Acknowledgment

We like to thank Benjamin Biegel, Artur Lipinski, and Alexander Voskoboinik, who helped us retrieve the various kinds of coupling data.

7. REFERENCES

[1] F. B. Abreu and M. Goulão. Coupling and Cohesion as Modularization Drivers: Are We Being Over-Persuaded? In *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 47–57. IEEE Computer Society, 2001.

[2] P. Andritsos and V. Tzerpos. Information-Theoretic Software Clustering. *IEEE Transactions on Software Engineering*, 31(2):150–165, 2005.

[3] C. Y. Baldwin and K. B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. The MIT Press, 1st edition, Mar. 2000.

[4] F. Beck and S. Diehl. Evaluating the Impact of Software Evolution on Software Clustering. In *WCRE '10: Proceedings of the 17th Working Conference on Reverse Engineering*, pages 99–108. IEEE Computer Society, 2010.

[5] B. Biegel and S. Diehl. Highly Configurable and Extensible Code Clone Detection. In *WCRE'10: Proceedings of the 17th Working Conference on Reverse Engineering*, pages 237–241. IEEE Computer Society, 2010.

[6] R. A. Bittencourt, G. J. S. Santos, D. D. S. Guerrero, and G. C. Murphy. Improving Automated Mapping in Reflexion Models using Information Retrieval Techniques. In *WCRE'10: Proceedings of the 17th Working Conference on Reverse Engineering*, pages 163–172. IEEE Computer Society, 2010.

[7] I. T. Bowman and R. C. Holt. Software Architecture Recovery Using Conway's Law. In *CASCON '98: Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 6+. IBM Press, 1998.

[8] S. Breu and T. Zimmermann. Mining Aspects from Version History. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 221–230. IEEE Computer Society, 2006.

[9] M. Cataldo, J. D. Herbsleb, and K. M. Carley. Socio-Technical Congruence: a Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity. In *ESEM '08: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 2–11. ACM, 2008.

[10] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software Dependencies, Work Dependencies, and Their Impact on Failures. *IEEE Transactions on Software Engineering*, 35(6):864–878, Nov. 2009.

[11] M. Conway. How do Committees Invent? *Datamation Journal*, pages 28–31, 1968.

[12] M. Harman, S. Swift, and K. Mahdavi. An Empirical Study of the Robustness of two Module Clustering Fitness Functions. In *GECCO '05: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, pages 1029–1036. ACM Press, 2005.

[13] A. Kuhn, S. Ducasse, and T. Girba. Enriching Reverse Engineering with Semantic Clustering. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 133–142. IEEE Computer Society, 2005.

[14] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder: D-CCFinder. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 106–115. IEEE Computer Society, 2007.

[15] R. C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 1st edition, Oct. 2002.

[16] B. Meyer. *Object-Oriented Software Construction, 1st editon*. Prentice-Hall, 1988.

[17] B. S. Mitchell and S. Mancoridis. On the Evaluation of the Bunch Search-Based Software Modularization Algorithm. *Soft Computing*, 12(1):77–93, Aug. 2007.

[18] D. L. Parnas. Information Distribution Aspects of Design Methodology. In *IFIP Congress (1)*, pages 339–344, 1971.

[19] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[20] C. K. Roy and J. R. Cordy. A Survey on Software Clone Detection Research. Technical report, Queen's University at Kingston, Ontario, Canada, 2007.

[21] S. Sarkar, G. M. Rama, and A. C. Kak. API-Based and Information-Theoretic Metrics for Measuring the Quality of Software Modularization. *IEEE Transactions on Software Engineering*, 33(1):14–32, 2007.

[22] R. W. Schwanke. An Intelligent Tool for Re-Engineering Software Modularity. In *ICSE '91: Proceedings of the 13th International Conference on Software Engineering*, pages 83–92. IEEE Computer Society, 1991.

[23] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured Design. *IBM Systems Journal*, 13(2):115–139, 1974.

[24] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison Wesley, May 2005.

[25] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 107–119. ACM, 1999.

[26] Z. Wen and V. Tzerpos. Software Clustering based on Omnipresent Object Detection. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 269–278, Washington, DC, USA, 2005. IEEE Computer Society.

[27] A. Wierda, E. Dortmans, and L. L. Somers. Using Version Information in Architectural Clustering - A Case Study. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 214–228. IEEE Computer Society, 2006.

[28] T. Zimmermann and P. Weißgerber. Preprocessing CVS Data For Fine-Grained Analysis. In *MSR '04: Proceedings of the 1st International Workshop on Mining Software Repositories*, pages 2–6. IEEE Computer Society, 2004.