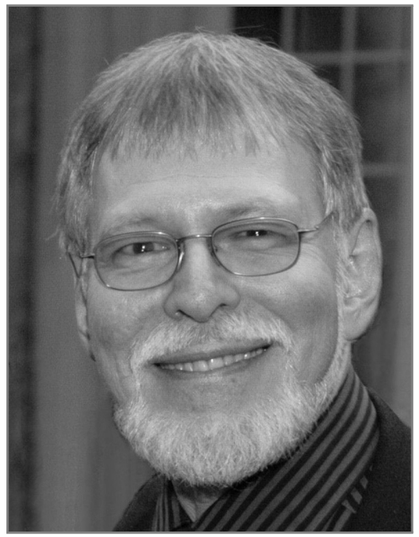
Coupling

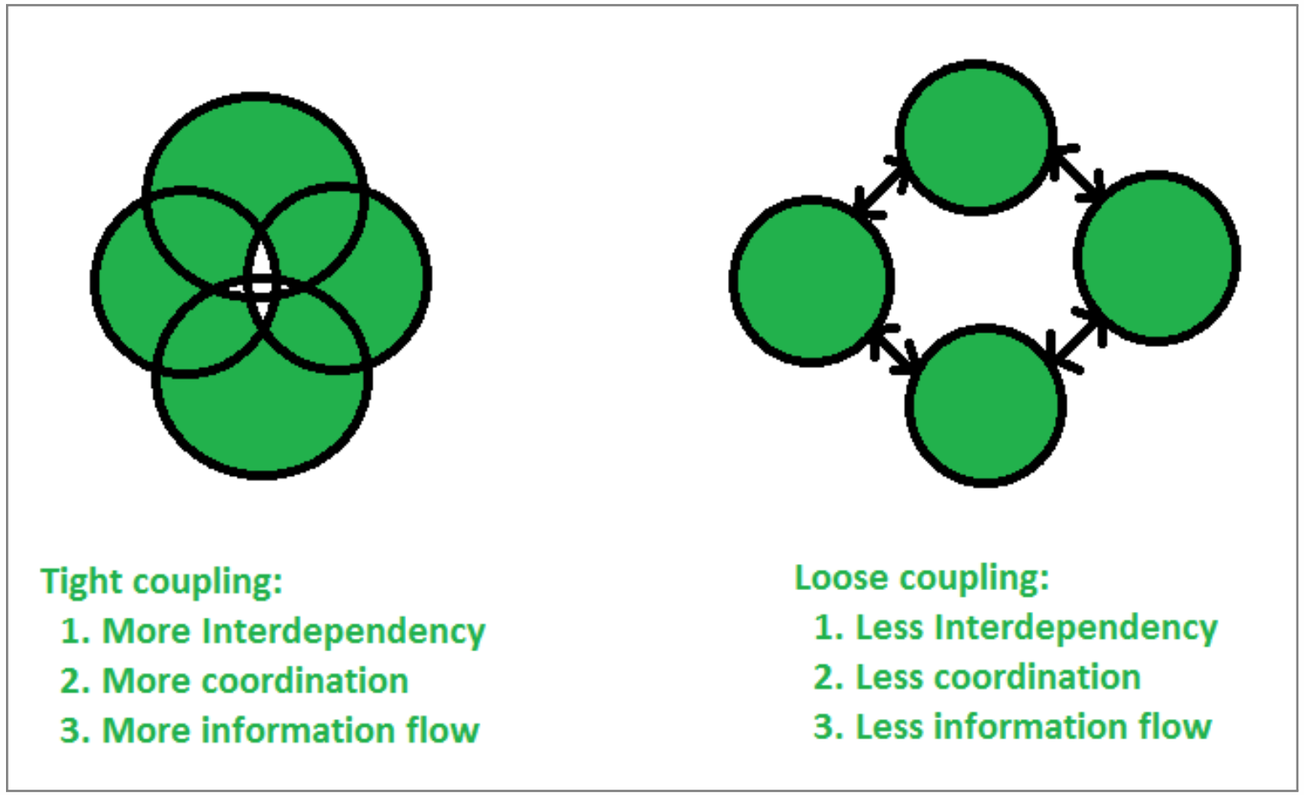
Yegor Bugayenko

Lecture #6 out of 24
80 minutes

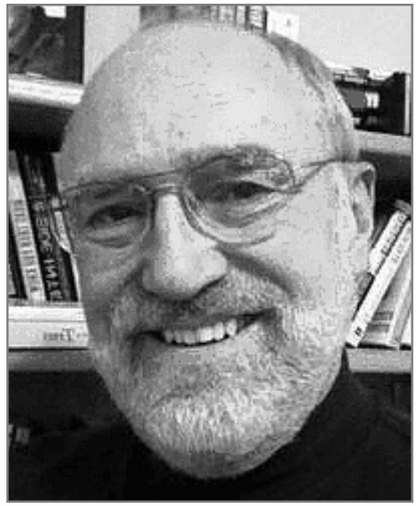


“The fewer and simpler the connections between modules, the easier it is to understand each module without reference to other modules.”

— Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine, *Structured Design*, IBM Systems Journal 13.2 (1974)

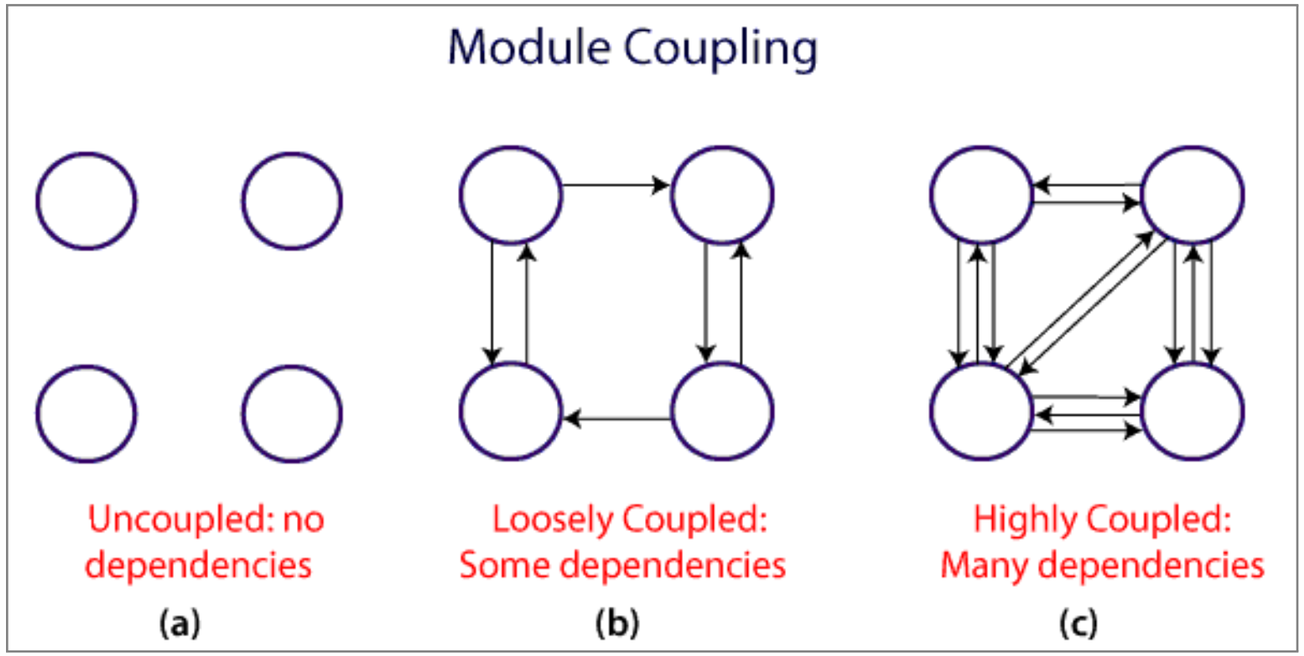


Source: https://www.geeksforgeeks.org/coupling-in-java/



“Coupling is the measure of the strength of association established by a connection from one module to another. Strong coupling complicates a system since a module is harder to understand, change, or correct by itself if it is highly interrelated with other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules.”

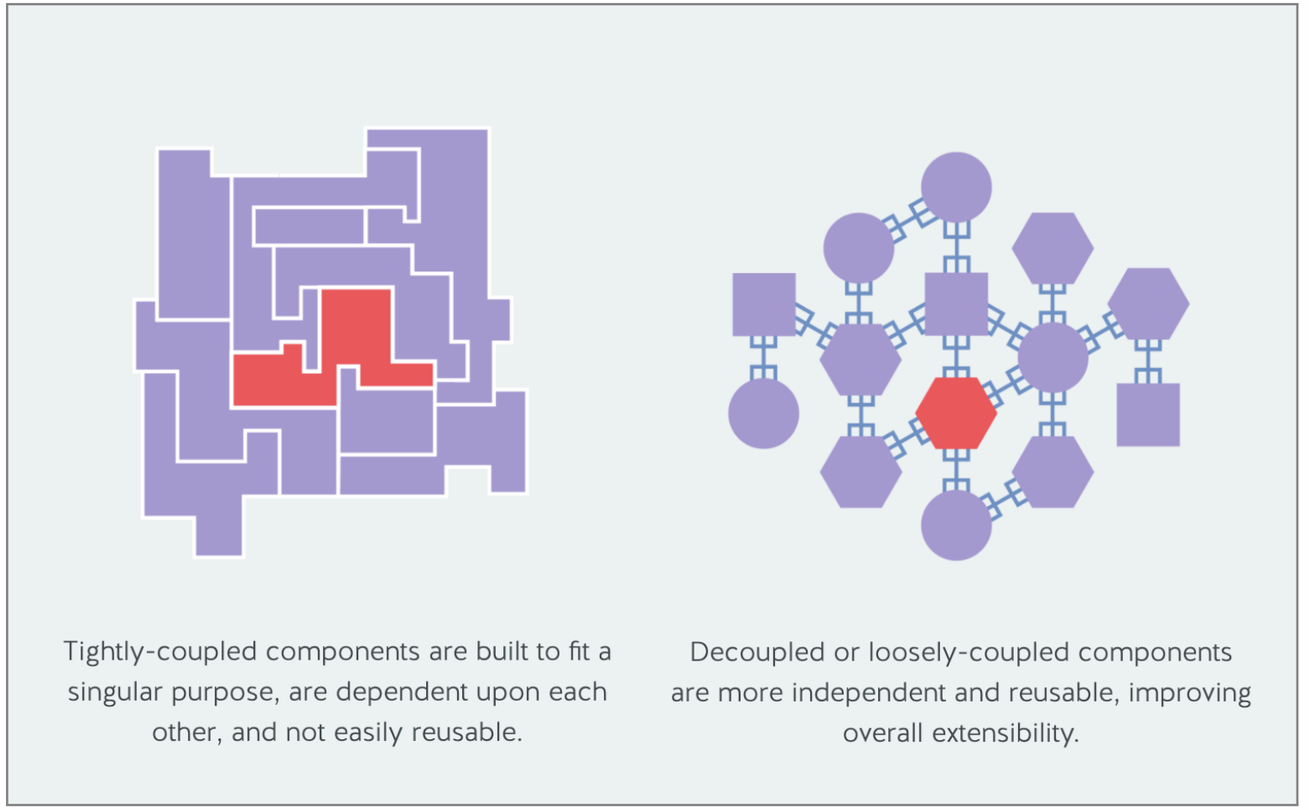
— Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine, *Structured Design*, IBM Systems Journal 13.2 (1974)



Source: https://www.javatpoint.com/software-engineering-coupling-and-cohesion

“The degree of coupling established by a particular connection is a function of several factors, and thus it is difficult to establish a simple index of coupling. Coupling depends (1) on how complicated the connection is, (2) on whether the connection refers to the module itself or something inside it, and (3) on what is being sent or received.”

— Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine, *Structured Design*, IBM Systems Journal 13.2 (1974)



Source: https://nordicapis.com/the-difference-between-tight-coupling-and-loose-coupling/

476 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 20, NO. 6, JUNE 1994

A Metrics Suite for Object Oriented Design

Shyam R. Chidamber and Chris F. Kemerer

Abstract— Given the central role that software development plays in the delivery and application of information technology, managers are increasingly focusing on process improvement in the software development area. This demand has spurred the provision of a number of new and/or improved approaches to software development, with perhaps the most prominent being object-orientation (OO). In addition, the focus on process improvement has increased the demand for software measures, or metrics with which to manage the process. The need for such metrics is particularly acute when an organization is adopting a new technology for which established practices have yet to be developed. This research addresses these needs through the development and implementation of a new suite of metrics for OO design. Metrics developed in previous research, while contributing to the field's understanding of software development processes, have generally been subject to serious criticisms, including the lack of a theoretical base. Following Wand and Weber, the theoretical base chosen for the metrics was the ontology of Bunge. Six design metrics are developed, and then analytically evaluated against Weyuker's proposed set of measurement principles. An automated data collection tool was then developed and the metrics were collected for an actual set of classes in two field sites in order to demonstrate their feasibility and suggest ways in which managers may use these metrics for process improvement.

***Index Terms*— CR categories and subject descriptors: D.2.8 [software engineering]: metrics; D.2.9 [software engineering]: management; F.2.3 [analysis of algorithms and problem complexity]: tradeoffs among complexity measures; K.6.3 [management of computing and information systems]: software management. General terms: Class, complexity, design, management, measurement, metrics, object orientation, performance.**

I. Introduction

It has been widely recognized that an important component of process improvement is the ability to measure the process. Given the central role that software development plays in the delivery and application of information technology, managers are increasingly focusing on process improvement in the software development area. This emphasis has had two effects. The first is that this demand has spurred the provision of a number of new and/or improved approaches to software development, with perhaps the most prominent being object-orientation (OO). Second, the focus on process improvement has increased the demand for software measures, or metrics with which to manage the process. The need for such metrics is particularly acute when an organization is adopting a new technology for which established practices have yet to be developed.

This research addresses these needs through the development and implementation of a new suite of metrics for OO design. Previous research on software metrics, while contributing to the field's understanding of software development processes, have generally been subject to one or more types of criticisms. These include: lacking a theoretical basis [41], lacking in desirable measurement properties [47], being insufficiently generalized or too implementation technology dependent [45], and being too labor-intensive to collect [22].

Following Wand and Weber, the theoretical base chosen for the OO design metrics was the ontology of Bunge [5], [6], [43]. Six design metrics were developed, and analytically evaluated against a previously proposed set of measurement principles. An automated data collection tool was then developed and implemented to collect an empirical sample of these metrics at two field sites in order to demonstrate their feasibility and to suggest ways in which managers may use these metrics for process improvement.

The key contributions of this paper are the development and empirical validation of a set of theoretically-grounded metrics of OO design. The rest of this paper is organized as follows. The next section presents a brief summary of the research problem, followed by a section describing the theory underlying the approach taken. Then Weyuker's list of software metric evaluation criteria is presented, along with a brief description of the empirical data collection sites. The Results section presents the metrics, their analytical evaluation, the empirical data and a managerial interpretation of the data for each metric. Some concluding remarks are presented in the final section.

II. Research Problem

There are two general types of criticisms that can be applied to current software metrics. The first category are those theoretical criticisms that are leveled at conventional software metrics as they are applied to traditional, non-OO software design and development. Kearney, et al. criticized software complexity metrics as being without solid theoretical bases and lacking appropriate properties [21]. Vessey and Weber also commented on the general lack of theoretical rigor in the structured programming literature [41]. Both Prather and Weyuker proposed that traditional software complexity metrics do not possess appropriate mathematical properties, and consequently fail to display what might be termed normal predictable behavior [34], [47]. This suggests that software

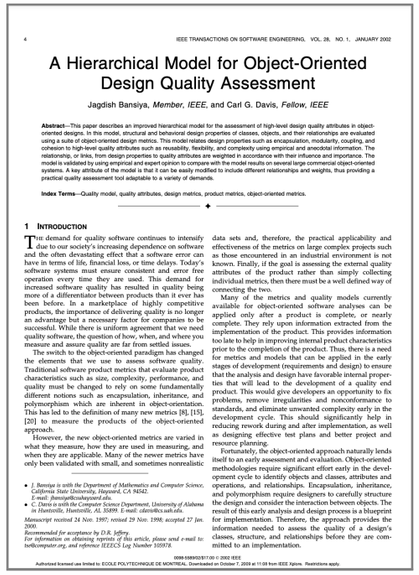
Manuscript received February 17, 1993; revised January, 1994. recommended by S. Zweben. This research was supported in part by the M.I.T. Center for Information Systems Research (CISR), and the cooperation of two industrial organizations who supplied the data.

The authors are with the Massachusetts Institute of Technology, E53-320, 30 Wadsworth St., Cambridge, MA 02139 USA. e-mail: shyam@sloan.mit.edu or ckemerer@mit.edu.

IEEE Log Number 9401707.

“Coupling Between Objects (CBO) — for a class is a count of the number of other classes to which it is coupled.”

— Shyam R. Chidamber and Chris F. Kemerer, *A Metrics Suite for Object Oriented Design*, IEEE Transactions on Software Engineering, 20.6, 1994



4 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 28, NO. 1, JANUARY 2002

A Hierarchical Model for Object-Oriented Design Quality Assessment

Jagdish Bansiya, *Member, IEEE*, and Carl G. Davis, *Fellow, IEEE*

Abstract—This paper describes an improved hierarchical model for the assessment of high-level design quality attributes in object-oriented designs. In this model, structural and behavioral design properties of classes, objects, and their relationships are evaluated using a suite of object-oriented design metrics. This model relates design properties such as encapsulation, modularity, coupling, and cohesion to high-level quality attributes such as reusability, flexibility, and complexity using empirical and anecdotal information. The relationship, or links, from design properties to quality attributes are weighted in accordance with their influence and importance. The model is validated by using empirical and expert opinion to compare with the model results on several large commercial object-oriented systems. A key attribute of the model is that it can be easily modified to include different relationships and weights, thus providing a practical quality assessment tool adaptable to a variety of demands.

Index Terms—Quality model, quality attributes, design metrics, product metrics, object-oriented metrics.

1 Introduction

The demand for quality software continues to intensify due to our society's increasing dependence on software and the often devastating effect that a software error can have in terms of life, financial loss, or time delays. Today's software systems must ensure consistent and error free operation every time they are used. This demand for increased software quality has resulted in quality being more of a differentiator between products than it ever has been before. In a marketplace of highly competitive products, the importance of delivering quality is no longer an advantage but a necessary factor for companies to be successful. While there is uniform agreement that we need quality software, the question of how, when, and where you measure and assure quality are far from settled issues.

The switch to the object-oriented paradigm has changed the elements that we use to assess software quality. Traditional software product metrics that evaluate product characteristics such as size, complexity, performance, and quality must be changed to rely on some fundamentally different notions such as encapsulation, inheritance, and polymorphism which are inherent in object-orientation. This has led to the definition of many new metrics [8], [15], [20] to measure the products of the object-oriented approach.

However, the new object-oriented metrics are varied in what they measure, how they are used in measuring, and when they are applicable. Many of the newer metrics have only been validated with small, and sometimes nonrealistic data sets and, therefore, the practical applicability and effectiveness of the metrics on large complex projects such as those encountered in an industrial environment is not known. Finally, if the goal is assessing the external quality attributes of the product rather than simply collecting individual metrics, then there must be a well defined way of connecting the two.

Many of the metrics and quality models currently available for object-oriented software analyses can be applied only after a product is complete, or nearly complete. They rely upon information extracted from the implementation of the product. This provides information too late to help in improving internal product characteristics prior to the completion of the product. Thus, there is a need for metrics and models that can be applied in the early stages of development (requirements and design) to ensure that the analysis and design have favorable internal properties that will lead to the development of a quality end product. This would give developers an opportunity to fix problems, remove irregularities and nonconformance to standards, and eliminate unwanted complexity early in the development cycle. This should significantly help in reducing rework during and after implementation, as well as designing effective test plans and better project and resource planning.

Fortunately, the object-oriented approach naturally lends itself to an early assessment and evaluation. Object-oriented methodologies require significant effort early in the development cycle to identify objects and classes, attributes and operations, and relationships. Encapsulation, inheritance, and polymorphism require designers to carefully structure the design and consider the interaction between objects. The result of this early analysis and design process is a blueprint for implementation. Therefore, the approach provides the information needed to assess the quality of a design's classes, structure, and relationships before they are committed to an implementation.

• *J. Bansiya is with the Department of Mathematics and Computer Science, California State University, Hayward, CA 94542. E-mail: jbansiya@csuhayward.edu.*
• *C. Davis is with the Computer Science Department, University of Alabama in Huntsville, Huntsville, AL 35899. E-mail: cdavis@cs.uah.edu.*

Manuscript received 24 Nov. 1997; revised 29 Nov. 1998; accepted 27 Jan. 2000.
Recommended for acceptance by D.R. Jeffery.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 105978.
0098-5589/02/$17.00 © 2002 IEEE

"Direct Class Coupling (DCC) — this metric is a count of the different number of classes that a class is directly related to. The metric includes classes that are directly related by attribute declarations and message passing (parameters) in methods."

— Jagdish Bansiya and Carl G. Davis, *A Hierarchical Model for Object-Oriented Design Quality Assessment*, IEEE Transactions on Software Engineering, 28.1, 2002

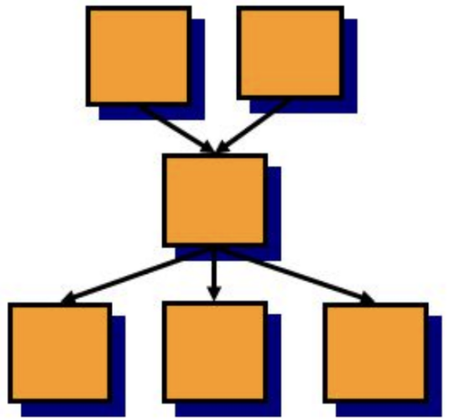
“The biggest problems come from uncontrolled coupling at the upper levels. I don’t worry about the number of modules coupled together, but I look at the pattern of dependency relationship between the modules.”

— Martin Fowler, *Reducing Coupling*, IEEE Software, 2001

“Low-to-medium fan-out means having a given class use a low-to-medium number of other classes. High fan-out (more than about seven) indicates that a class uses a large number of other classes and may therefore be overly complex. High fan-in refers to having a high number of classes that use a given class. High fan-in implies that a system has been designed to make good use of utility classes at the lower levels in the system.”

— Steven McConnell, *Code Complete*, 2004

Fan-in = number of ingoing dependencies
Fan-out = number of outgoing dependencies



Heuristic: a high fan-in/fan-out indicates a high complexity

(c) Natalia Kokash, Leiden Institute of Advanced Computer Science

An Evolutionary Study of Fan-in and Fan-out Metrics in OSS

A. Mubarak, S. Counsell and R.M. Hierons
Department of Information Systems and Computing, Brunel University
Uxbridge, UK. Email: steve.counsell@brunel.ac.uk

***Abstract*- Excessive coupling between object-oriented classes is widely acknowledged as a maintenance problem that can result in a higher propensity for faults in systems and a 'stored up' future problem. The aim of this paper is to explore the relationship between 'fan-in' and 'fan-out' coupling metrics over multiple versions of open-source software. More specifically, we explore the relationship between the two metrics to determine patterns of growth in each over the course of time. The JHawk tool was used to extract the two metrics from five open-source systems. Two questions were posed for each system. First, what are the characteristics of classes exhibiting the highest fan-in values? Second, do fan-in and fan-out increase in corresponding and consistent amounts over time? Results show a wide range of traits in the classes to explain both high and low levels of fan-in and fan-out. We also found evidence of certain 'key' classes (with both high fan-in and fan-out) and 'client' and 'server'-type classes with just high fan-out and fan-in, respectively. We provide an explanation of the composition and existence of such classes as well as for disproportionate increases in each of the two metrics over time.**

Keywords-coupling, Java, fan-in, fan-out, package.

I. INTRODUCTION

Excessive class coupling has often been related to the propensity for faults in software [5]. It is widely believed in the Object-Oriented (OO) community that excessive coupling between classes creates a level of complexity that can complicate subsequent maintenance and represents a 'stored up' maintenance problem. In practice, a class that is highly coupled to many other classes is an ideal candidate for re-engineering or removal from the system to mitigate both current and potential future problems. A problem that immediately arises however for the developer when considering re-engineering of classes with high coupling is: 'Do those classes have prohibitively large dependencies?' If so, then are those coupling dependencies 'incoming' or 'outgoing' dependencies? In theory, it is more difficult to modify a target class with high incoming and low outgoing coupling, since the former requires detailed and careful scrutiny of each of the many 'incoming' dependent classes and the possible side-effects of change.

In this paper, we investigate versions of five Open Source Systems (OSS) focusing on two well-known coupling metrics - 'fan-in' (i.e., incoming coupling) and 'fan-out' (i.e., outgoing coupling). We used an automated tool to extract each of the coupling metrics from those five systems. The research questions we explore are first, is it the case that classes with large incoming coupling naturally have low outgoing coupling and second, does this relationship worsen over time? In other words, does the potential maintenance problem become worse in terms of fan-in and fan-out values?

II. MOTIVATION AND RELATED WORK

The research in this paper is motivated by a number of factors. Firstly, previous research [15] has shown that there is a trade-off between coupling types – in particular, that between coupling through imported packages and the introduction of 'internal-to-the-package' coupling. In this paper, we explore the potential characteristics and trade-offs between fan-in and fan-out metrics over time. Second, we would always expect potentially problematic classes to be re-engineered by developers through techniques such as refactoring [9]; however, the practical realities of limited time and resources at their disposal means that only when classes exhibit particularly bad 'smells' (e.g. excessive coupling) [9] are they dealt with. In this paper, we explore, over time, whether smells given by too much coupling do become 'smellier' and, if so, in what proportions. Finally, the research is motivated by previous research [16] which showed that the fan-in and fan-out metrics tended to be relatively small for classes removed from a system. In other words, classes with either high fan-in and/or fan-out may be difficult to move or remove from a system. This question has inspired further examination of trends in the two metrics presented.

In terms of related work, the research presented relates to areas of software evolution, coupling metrics and the use of OSS [8]. In terms of software evolution, the laws of Lehman [2] provide the backdrop for many past evolutionary studies. Evolution has also been the subject of simulation studies [18] and this has allowed OSS evolution to be studied in a contrasting way to that empirically. The research presented in this paper delves into specific evolutionary coupling features

"We also found evidence of certain 'key' classes (with both high fan-in and fan-out) and 'client' and 'server'-type classes with just high fan-out and fan-in, respectively."

— A. Mubarak et al., *An Evolutionary Study of Fan-in and Fan-out Metrics in OSS*, Proceedings of the 4th International Conference on Research Challenges in Information Science (RCIS), 2010

Fan-out, as a metric, is supported by a few tools:

- Checkstyle for Java
- CCCC for C++, C, and Java
- module-coupling-metrics for Python

“Afferent coupling (denoted by **Ca**) is a metric that indicates the total number of other projects/boundaries that are dependent upon it. Efferent coupling (denoted by **Ce**) is another metric that is the verse of Afferent Coupling. It is the total number of projects that a given project depends on. Instability another metric that is a ratio: $\mathbf{I} = \mathbf{Ce}/(\mathbf{Ce} + \mathbf{Ca})$. This metric is a ratio between 0 and 1. With 0 meaning it’s totally stable and 1 meaning it’s unstable.”

— Derek Comartin, Write Stable Code using Coupling Metrics, 2021

Types of Coupling (some of them)

- Content Coupling is when one module modifies or relies on the internal workings of another module (e.g., accessing local data of another module).
- Global Coupling is when two modules share the same global data (e.g., a global variable).
- External Coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface.
- Control Coupling is one module controlling the flow of another, by passing it information on what to do (e.g., passing a what-to-do flag).
- Stamp Coupling is when modules share a composite data structure and use only a part of it, possibly a different part (e.g., passing a whole record to a function that only needs one field of it).
- Data Coupling is when modules share data through, for example, parameters. Each datum is an elementary piece, and these are the only data shared (e.g., passing an integer to a function that computes a square root).
- Message Coupling can be achieved by state decentralization (as in objects) and component communication is done via parameters or message passing (see Message passing).
- Subclass Coupling describes the relationship between a child and its parent. The child is connected to its parent, but the parent isn't connected to the child.
- Temporal Coupling is when two actions are bundled together into one module just because they happen to occur at the same time.

Source:
`https://wiki.edunitas.com/IT/en/114-10/Coupling-(computer-programming)_1430_eduNitas.html`

Fear of Decoupling

```
interface Money {
  double cents();
}

void send(Money m) {
  double c = m.cents();
  // Send them over via the API...
}

class OneDollar implements Money {
  @Override
  double cents() {
    return 100.0d;
  }
}
```

```
class EmployeeHourlyRate
  implements Money {
  @Override
  double cents() {
    // Fetch the exchange rate;
    // Update the database;
    // Calculate the hourly rate;
    // Return the value.
  }
}
```

“Polymorphism makes sofware more fragile ... to make it more robust!”

Temporal Coupling

Tight coupling (not good):

```
List<String> list =
  new LinkedList<>();
Foo.append(list, "Jeff");
Foo.append(list, "Walter");
return list;
```

Loose coupling (good):

```
return Foo.with(
  Foo.with(
    new LinkedList<>(),
    "Jeff"
  ),
  "Walter"
);
```

https://www.yegor256.com/2015/12/08/temporal-coupling-between-method-calls.html

Distance of Coupling

```
1  class Temperature {
2    private int t;
3    public String toString() {
4      return String.format("%d F", this.t);
5    }
6  }
7
8  Temperature x = new Temperature();
9  String txt = x.toString();
10 String[] parts = txt.split(" ");
11 int t = Integer.parseInt(parts[0]);
```

"The larger the number (or the mean of all numbers), the worse the design: in good design we are not supposed to take something out of a method and then do some complex processing. The distance metric will tell us exactly that: how many times, and by how much, we violated the principle of loose coupling."

https://www.yegor256.com/2020/10/27/distance-of-coupling.html

Read this:

Structured Design, Wayne P. Stevens, et al., IBM Systems Journal, 13.2, 1974

A Hierarchical Model for Object-Oriented Design Quality Assessment, Jagdish Bansiya et al., IEEE Transactions on Software Engineering, 28.1, 2022

An Overview of Various Object Oriented Metrics, Brij Mohan Goel et al., International Journal of Information Technology & Systems, 2.1, 2014

Analysing the Contribution of Coupling Metrics for the Development and Management of Process Architectures, Daniel Braunnagel et al., ECIS, 2015

New Metric: the Distance of Coupling (2020)

Fear of Decoupling (2018)

Reflection Means Hidden Coupling (2022)