A Simple Model for Reasoning about Limits on Coupling in Object-Oriented Software

Hayden Melton
Refinitiv
hayden.melton@refinitiv.com

Abstract—Given that all of a program’s code must be reachable from its main method, and that type-safety causes this reachability to substantially manifest as compilation dependencies among the program’s classes, a simple model for reasoning about limits on coupling among a program’s classes—a long-standing question in the software design literature—is proposed. The model takes the form of a directed graph with classes as nodes and four distinct forms of compilation dependencies as labeled edges. As manifested in compilation dependencies, the model suggests that there is a trade-off between direct and transitive coupling, and that certain forms of coupling in a class are prerequisites for the existence of other forms in that same class.

I. INTRODUCTION

Few concepts from the area of software design have received as broad and sustained treatment in the literature as that of coupling. Whether explicit or implicit in their respective expositions, the long-standing advice to reduce coupling between modules [1] seems to be the basis for many new features of programming languages [2] [3, ch.13], for specific design techniques such as design patterns [4], for entire design methodologies [5], and for many metrics [6, 7] and empirical studies of software structure [8, 9].

A question that has repeatedly been raised of coupling since when Stevens et al. first coined the term in the 1970s is: to what extent must coupling exist among a program’s modules?1 The intuition behind this question is that the modules of a program must all communicate with one another to some degree, because if a module does not communicate with any other module of the program it cannot be considered part of that program. So, although it is desirable to minimize coupling among a program’s modules, it does seem there are limits on the extent to which that can be achieved.

Disclaimer: Views expressed herein are those of the author and do not necessarily reflect those of Refinitiv.

1 Specifically, Fowler has said “it becomes clear that … you can’t treat coupling as something to always avoid. You can break a program into modules, but these modules will need to communicate in some way—otherwise, you’d just have multiple programs” [10]. Stevens et al. have said “Modules must at least pass data or they cannot functionally be a part of a single system. Thus connections that pass data are a necessary minimum” [1]. Berard has said “…we need to distinguish between two different categories of object coupling: necessary and unnecessary. Most, if not all, object-oriented applications can be viewed as systems of interacting objects. In such systems it is required (i.e., necessary) that objects be coupled—otherwise no interactions can take place” [11, p.93].

Over the years various bodies of work have been produced that can, more formally, begin to help us answer this question of coupling. One such thread of work is in reachability of code within a program. What we may infer from this thread is that if code is not reachable from the entry point of a program, it should not be considered a part of that program [12]. Another such thread of work is empirical studies of coupling in corpora of real-world software. What we may infer from this thread is that empirically we may not (yet) be capable of building real software that exhibits ‘low’ coupling [8, 9]. A final thread of work is on the nature of coupling itself. What we may learn from this thread is that coupling is a complex concept, and we should very carefully define and justify the specific manifestation of it in which we are interested [6, 7].

In this paper we are interested in reasoning about some lower limits on coupling specifically as it manifests as compile-time dependencies among the classes of object-oriented (OO) software. Unlike the empirical studies of coupling that have sought to do similar (see e.g., [8]), we take an entirely theoretical approach to reasoning about such limits. To do so, we introduce a model of an OO program that assumes for a class to be part of a given program some code in that class must be reachable from the program’s main method. We use the Java programming language2 to make the exposition of the model more concrete, though we believe with only minor modifications the model could be applied to other strongly-typed OO languages. We explain the intuition behind our model and ultimately use it to argue: (1) that within an OO program there seems to be a trade-off to be made between direct and transitive coupling, and (2) that certain forms of coupling seem to be more fundamental than others, in that the existence of one form in a class is predicated on the existence of another form in that same class. We identify the potential implications of these findings in a variety of contexts within software engineering.

The remainder of this paper is organized as follows. In Section II we describe our model in the context of Java. In Section III we identify some limits on coupling that seem to be indicated by the model. In Section IV we identify some practical implications of these limits. Finally, in Section V we draw conclusions and identify future directions of this work.

2 In the model’s exposition we focus only on features of the Java language that exist in it as of version 1.5 and exclude exceptions, but touch on the effects of them and other more modern language features generally in Section IV of the paper.
II. THE MODEL

We define a Java program—noting a distinction from software systems that comprise many individual programs, and similarly from libraries and frameworks that are effectively collections of related features but have no unifying main method—as one or more Java classes having a single main method from which the program is launched. Consistent with the literature on code reachability, a class is a part of the program if and only if some code in that class is reachable from the program’s main method. We will also assume for the sake of simplicity—though perhaps contrary to some recent empirical findings of real-world Java software—that our programs entirely avoid features of Java that are not type-safe, such as reflection, dynamic class loading, manual type-casting and type-testing. By making this assumption of type-safeness we can better ensure notions of reachability manifest as compile-time dependencies among classes. We focus exclusively on compile-time dependencies as a form of coupling—and not say runtime coupling among objects—for reasons set forth in [2].

Our model of a Java program is a labeled, directed graph, the edges of which are compile-time dependencies and the nodes of which are the classes comprising the program. The edges are labeled to distinguish between various forms of compile-time dependencies; the edges are directed because compile-time dependencies are not generally symmetric. For reasons that will become clear in due course, the labeling scheme for edges in the graph—and thus categorization of compile-time dependencies in the model—is as follows:

- **instantiates (ins)** An edge labeled ‘instantiates’ (‘ins’ for short) shall be drawn from the node of class A to that of class B if and only if the code `new B(...)` appears in the body of A.

- **non-private (np)** An edge labeled ‘non-private’ (‘np’ for short) shall be drawn from the node of class A to that of class B if and only if B appears as a type name in the parts of A that are visible to clients of A (i.e., B appears as: A’s supertype; or as an interface implemented by A; or as the return type, formal parameter or in throws clause of a method declared by A where that method has a public, protected or default access modifier; or as the declared type of a field with such an access modifier in A; and so on).

- **instance-ref (ir)** An edge labeled ‘instance-ref’ (‘ir’ for short) shall be drawn from the node of class A to that of class B if and only if B appears as a type in the parts of A that are visible to clients of A (i.e., B appears as: A’s supertype; or as an interface implemented by A; or as the return type, formal parameter or in throws clause of a method declared by A where that method has a public, protected or default access modifier; or as the declared type of a field with such an access modifier in A; and so on).

- **static-ref (sr)** An edge labeled ‘static-ref’ (‘sr’ for short) shall be drawn from the node of class A to that of class B if and only if in the body of A a field is accessed or method is called on B that is not static—i.e., that pertains to an instance of B.

An application of the model to the program listing in Fig.1 yields the labeled, directed graph shown at the bottom of the figure, noting that classes from the Standard Java API (such as String, Object—the implicit supertype of all listed classes, etc) have been omitted for clarity in the graph’s presentation. Note that, per our definition of what it means for a class to be part of a program, all classes contain code that is reachable in the directed graph model from the node whose class contains the main method (i.e., A). Note also that all our program does when it is run is to print out “Hello from B!” by invoking a sequence of method calls from A to B, from B to C, and from C to D. It is this sequence of method calls—and crucially calls to instance (cf. static) methods on the respective classes—that ultimately serve to make this program an instructive example of our model.

![Fig. 1. Application of the model to a simple Java program.](image)
answer is by obtaining a reference through the non-private interface of another class (i.e., the latter class instantiates the class of interest, and provides a reference to that of interest as the return type of a method). More concretely, in a hypothetical program comprising classes \( X, \ Y \) and \( Z \): class \( X \) requires a reference to class \( Z \) and obtains it by calling a method \( \text{getZ()} \) with return type \( Z \) on class \( Y \), with class \( Y \) instantiating and returning a reference to \( Z \) from that method. This technique of obtaining the reference through the interface of another class corresponds to the non-private (np) label in our model.

A more subtle way in which a reference to the instance of one class can be passed into another is illustrated in the code listing of Fig.2. This code is a refactored version of the program appearing in Fig.1, and demonstrates how a reference can be passed into a class ‘from outside’, as opposed to it being obtained from another class on which the requesting class transitivity depends. This specific technique has been described as both dependency injection and the dependency inversion principle [9, ch.10], and its refactoring involves extracting an interface \( IC \) from \( C \), then passing into \( B \) via a parameter to its constructor a reference to \( C \) having type \( IC \). This, in turn, eliminates \( B \)’s compile-time dependency on \( C \), and shifts the responsibility for instantiating \( C \) from \( B \) onto \( A \). We will have more to say on Fig.2 shortly, but the point for now is that \( IC \)—a parameter in the non-private constructor of \( B \), and thus corresponding to the label non-private (np) in our model—is used to pass in a reference to an instance of \( C \). \( B \)’s compile-time dependency on \( C \) is thus eliminated by this refactoring.

As for the remaining two edge labels in our model—instance-ref (ir) and static-ref (sr)—both are somewhat self-explanatory in light of the definition for what it means for a class to be a part of the program. Both represent execution of code—whether by field access or method call—in another class. The key distinction between them of course is that the former requires an instance of the class at the call site, the latter requires only the name of the class itself at that site. Indeed, it has been said that static methods and fields are akin to ‘globals’ in non-DD languages [9, ch.9], and this is why no corresponding discussion of how static references ultimately get to their call sites is necessary.

III. IDENTIFYING LIMITS ON COUPLING

The motivation for our model—and indeed the goal of this paper—is to try to identify limits on coupling in OO programs. What can our model tell us about such limits that improves upon the statements made to the same end in the literature (some of which are recited in footnote 1 of this paper)? First, and perhaps at the risk of applying circular logic given our definition of what it means for a class to be part of a program, is that if our program is subject to the type-safeness constraints of our model, each of its classes should be reachable via compile-time dependencies from the program’s main class. What this further implies, is that the program’s graph must be connected and that among its \( N \) nodes there must exist at least \( N - 1 \) edges.

More interesting than the somewhat superficial observations above though, is that the model seems to suggest that there may exist a tradeoff between direct and transitive coupling in a program’s overall structure. If one accepts that the predominant way in which classes in OO programs interact is by accessing one another’s instance members (i.e., fields and methods that are non-static), then the two program versions shown in Fig.1 and Fig.2 respectively can serve to illustrate this tradeoff. To this end, Table I shows the classes to which each is directly and transitively coupled in both versions of the program. Underlined classes in the table indicate an increase in coupling—specifically the class whose presence is associated with that increase—in one version of the program relative to the other.

![Fig. 2. Application of the model to a version of the simple Java program refactored to use dependency injection.]

<table>
<thead>
<tr>
<th>Class</th>
<th>Direct Fig.1</th>
<th>Direct Fig.2</th>
<th>Transitive Fig.1</th>
<th>Transitive Fig.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>{B}</td>
<td>{B,IC}</td>
<td>{B,IC,CD}</td>
<td>{B,IC,CD}</td>
</tr>
<tr>
<td>B</td>
<td>{C}</td>
<td>{IC}</td>
<td>{C}</td>
<td>{IC}</td>
</tr>
<tr>
<td>IC</td>
<td>{-}</td>
<td>{-}</td>
<td>{-}</td>
<td>{-}</td>
</tr>
<tr>
<td>C</td>
<td>{D}</td>
<td>{IC,D}</td>
<td>{D}</td>
<td>{IC,D}</td>
</tr>
<tr>
<td>D</td>
<td>{-}</td>
<td>{-}</td>
<td>{-}</td>
<td>{-}</td>
</tr>
</tbody>
</table>

TABLE I COMPARISON OF DIRECT AND TRANSITIVE COUPLING FROM EACH CLASS TO THE OTHERS IN THE PROGRAMS OF FIG.1 AND FIG.2.

What we can discern from Table I is that direct coupling may be ‘higher’ in the program of Fig.2, by virtue of \( A \) and \( C \) both directly depending on a larger number of classes than in the program of Fig.1. What we may also discern from the table is that \( B \) in the program of Fig.2 transitively depends on fewer classes than in the program of Fig.1. This reduction in transitive coupling would be more pronounced if both versions of the program also comprised further classes (say \( E \), \( F \) and \( G \)) and if these further classes were exclusively coupled—either directly or transitively—to \( C \) or \( D \).
To say the same thing, but in terms of cause-and-effect, extracting an interface $IC$ from $C$ (per Fig.2) eliminated $B$'s transitive dependency on all the classes that appear exclusively in $C$’s implementation—i.e., $C$ itself and $D$ (per Fig.1). The 'cost' of eliminating that transitive dependency on classes appearing exclusively in $C$’s implementation however was additional direct coupling, specifically: (1) the introduction of a new class (interface) $IC$ to which $A$, $B$ and $C$ became directly coupled; (2) the instantiation of $C$ inside of $A$—a coupling that did not previously exist in $A$, and that had to be introduced there to avoid $B$’s transitive dependency on $C$; and (3) the ‘widening’ of $B$’s non-private interface—and thus increase in its direct coupling—to include a parameter $IC$ so a reference to the instance of its implementation could be passed into $B$ from $A$.

A final observation we can seemingly make from the model on the limits of coupling follows from the discussion in the previous section on a `NullPointerException` occurring if a valid reference to an instance does not exist at the call site to a non-static field or method. What this seems to imply is that in our model, some forms of dependency are more fundamental than others in that the former may need to exist as a prerequisite to the latter. Indeed, from that discussion in the previous section, it would seem that the ‘instance-ref’ (ir) dependency of one class on another implies the existence of at least one other form of dependency defined by the model of that same class on the other.

IV. PRACTICAL IMPLICATIONS

We think that the model and the limits on coupling that seem to follow from it could have practical implications in quite a wide-variety of contexts within software engineering. In this section we identify and discuss some of these contexts.

Pedagogy of software design. Much of the instructional literature on software design that we have read espouses the benefits of ‘low coupling’ in a program, but fails to provide concrete techniques that students new to the topic are actually capable of applying to achieve said ‘low coupling’. Of the remaining literature that does provide such techniques, these techniques tend to be presented in terms of the ‘local’ structure of a program involving just a few of its classes, and without treatment of the technique’s effect on the overall structure of the program.

The Gang of Four note in their book Design Patterns, for instance, that while the goal is to “program to an interface, not an implementation”, “you have to instantiate concrete classes ... that is, specify a particular implementation somewhere in your system” [4][p.18]. They advocate using one of their ‘creational patterns’ to perform that instantiation, but those patterns like the others they describe involve dealing with just a handful of a program's classes. It is thus perhaps unsurprising a discussion of each pattern’s effect on coupling in the program’s overall structure is absent.

Of course there are further techniques in the literature beyond those that appear in Design Patterns [4] that purport to ‘reduce coupling’ also through the introduction of interfaces.

Our point here is that the model presented in this paper might serve to provide a unified way to reason about all these techniques, particularly in a manner: (1) that considers both the effects on coupling at both a ‘local’ and ‘global’ level, and (2) that is straightforward for students new to the discipline to apply. The benefits of models of “core constructs” in software design—such as coupling—have been long been espoused [15] but conspicuously seem to remain absent from the instructional literature.

Theories of coupling phenomena observed in real-software. Quite a large amount of research has been undertaken in the past decade-or-so in characterising attributes of the designs of real-world software systems [8, 9, 13, 16]. One thread in this body of work is the identification of power law distributions in direct coupling among a software system’s classes [8, 16] [9, ch.3]. Since power law distributions are quite prevalent in social systems, economic systems, physical systems and man-made systems quite a number of theories on how they come about have been proposed (e.g., ‘preferential attachment’). The theory of how they come about in direct coupling in software seems unsettled [16], but the limits on coupling identified by the model in this paper may help in the formulation of such theories, or in the elimination of existing ones. In particular, the limits on coupling identified here might impose new constraints that statistical models of software evolution (see e.g., [17, 18]) have not yet taken into account.

Along similar lines, and based upon empirical observation, it has been argued that a ‘high coupling’ might be unavoidable in real-software [8]. Performing a simulated refactoring on real-software whereby the limits on coupling identified here are respected, and the treatment performed is that of Fig.2 to all classes in a real-software system might provide further evidence to support or refute this, and might reveal distributions of coupling not previously observed. Further such simulations where coupling is broken by deliberately duplicating code [5, p.269] might also bear fruit.

Software visualization. One of the challenges in providing legible directed graph visualizations of the class structure of real-software systems is the sheer number of dependencies and therefore edges in those graphs [19] [20, p.6]. The model in this paper may aid the display of more legible visualizations by virtue of it collapsing dependencies into just four distinct categories, and by enabling the pruning of (redundant) instance-ref (ir) edges if the goal is to accurately show only transitive dependencies.

Software metrics. One problem in the field of software metrics pertains to deriving an entity population model [21], i.e., being able to interpret the meaning of a specific measurement’s value based on what the ‘normal’ value for that measurement across the population. To illustrate, by way of
analogy: it does not make sense to measure a person’s body temperature if there is no ‘normal value’ (98.6°F) against which to compare that measurement [21]. Where the model provided in this paper offers an alternative to the entity population model is that it seeks to identify theoretical lower limits on coupling; the extent to which a system exhibits coupling could then instead potentially be expressed with reference to those lower limits. The model might also find application in a ‘normalizing metric’ that allows a comparison of coupling in systems of different sizes (i.e., comprising differing numbers of classes) as per Lakos’ Normalized Cumulative Component Dependency (NCCD) metric [5, p.199].

Programming language features. While it remains true that all classes in a (type-safe) Java program must be reachable via compilation dependencies from its main method, Java’s implementation of exception handling provides a kind of ‘backdoor’ by which a reference to an instance of a class can get into another—something that was not previously discussed in Section II. In particular, the way exception handling works in Java is that the run-time type (cf. declared type) of the exception is compared to the type appearing in each catch clause in a manner equivalent to performing an instanceof test on it [22, p.352]. Given that the purpose of exceptions are to handle error conditions—and not ‘normal’ program control flow—this might not bear further discussion if not for the fact it is also highly reminiscent of how more modern language features such as multi-methods [23] and pattern matching [3, ch.15] work. One further application of the model may thus be in helping to better explain the specific mechanisms by which new language features such as these aid in reducing coupling, both locally and globally.

V. CONCLUSIONS AND FUTURE WORK

In this paper we presented a simple model of coupling as it manifests as compile-time dependencies among a program’s classes. Although we did not provide formal proofs for such, analysis of the model seems to indicate that there is a trade-off to be made between direct and transitive coupling in a program, and that some forms of compile-time dependency are predicated on the existence of other different forms in the class. Possible directions for future work are in proofs of the model and its findings, and in the application of the model in the specific ways described to: pedagogy of software design, theories of coupling phenomena observed in real-software, software visualization, software metrics, and programming language features.

Further future directions might include: refining the model to account for dependencies induced by manual casting operations [14], incorporating information about the extra nodes introduced in the program’s graph when dependency-breaking is undertaken through the introduction of interfaces [24], and use in reasoning about unanticipated instantiation [25].

REFERENCES


