The Case for Explicit Coupling Constraints

Mikal Ziane,† Mel Ó Cinnéide†

May 13, 2013

Abstract

A software element defined in one place is typically used in many places. When it is changed, all its occurrences may need to be changed too, which can severely hinder software evolution. This has led to the support of encapsulation in modern programming languages. Unfortunately, as is shown in this paper, this is not enough to express all the constraints that are needed to decouple programming elements that evolve at different paces.

In this paper we show that:

• A language can be defined to easily express very general coupling constraints.

• Violations to these constraints can be detected automatically.

We then demonstrate several places where the need for coupling constraints arose in open-source Java projects. These constraints were expressed in comments when explicit constraints would have enabled automatic treatment.

1 Introduction

During the software maintenance process, the repeated addition of functionality can lead to a loss of quality in the underlying design. This problem, known as software decay [11], occurs when changes are made to a program without due consideration to its overall structure and design rationale. Previous work has addressed this problem by using automated refactoring to restore design quality [20, 21]. In this paper we present an approach that aims to prevent the decay from occurring in the first place.

1 E-mail: mikal.ziane@lip6.fr
2 E-mail: mel.ocinneide@ucd.ie
When discussing internal design quality, software coupling is one of the most important issues to consider. Indeed, the importance of software coupling has been appreciated since the early 1970s when the pioneering work on modular decomposition and structured design was performed [22, 27]. When modules are loosely coupled, a change in one module is not likely to require that changes be made to other modules. When the reverse is the case, i.e., when a high degree of coupling exists between modules, the result is that maintenance work tends to cause more source code modifications, and indeed an increased error rate [15].

Further evidence of the importance of coupling lies in the fact that a large number of object-oriented design principles and design patterns deal with coupling. Many of the original Gamma et al design patterns [12] can be used to decouple program elements from each other [31]. An example of a widely-accepted design principle that is fundamentally to do with coupling is the Dependency-Inversion Principle [19]. Its goal is to prevent high-level modules from depending on low-level modules, so that low level modules can change without causing a ripple of changes up through the higher-level modules.

In spite of the importance of software decoupling, it is imperfectly supported in current programming languages. In Java, attributes of a class can be made private or protected, and within packages classes can be made non-public. However, many common decouplings cannot be expressed. Consider for example the UML class diagram for the Factory Method class in figure 1. An obvious decoupling required here is that the ConcreteProduct class be decoupled from the Creator class. This decoupling would prevent a maintenance programmer, either accidentally or through misunderstanding, from creating a coupling between these two classes.

The only way to express this decoupling in Java is to place the classes in different packages. However, the package construct is usually used to reflect the overall system architecture, so using it to capture a class-level decoupling like this is not a workable solution. Furthermore, the ConcreteCreator class would have to occupy the same package as the ConcreteProduct class, which may be very undesirable.

In the absence of language support for this type of decoupling, two other options can be used. The original programmer or system architect who intends two modules to be decoupled, and to remain decoupled, can express this either in documentation, or by relying on the insight of future maintenance programmers to understand the intention of their design. Neither solution is ideal. Comments are often ignored[1] and maintenance programmers cannot

---

---

1Indeed, Agile practices suggest that comments can be a sign of poor design, and that
be relied upon to appreciate and observe the coupling limitations implied in the original design. The fact that design decay does occur over time suggests that these techniques are not working in practice. A further weakness of this implicit expression of coupling constraints is that tool support for decoupling remains an impossibility.

In order to address these issues, we introduce the concept of a coupling constraint. A coupling constraint expresses the requirement that one program element (package, class or method) should not know about another program element (package, class, method or field). Coupling constraints are defined by the original programmers or system architect and may be automatically checked whenever the software is later updated by a maintenance programmer. If a coupling constraint is violated, the maintenance programmer will need to find the appropriate way to achieve their goal, or refactor to code to make the desired coupling possible without violating coupling constraints.

In the case of the Factory Method example above, all the implementations of the Product interface should be hidden from the Creator class. This includes the existing ConcreteProduct class, and all future implementations that may be added. Less obviously, the Product interface should also be decoupled from the ConcreteCreator classes. This permits the ConcreteCreator classes to create instances of the implementing classes of Product, but not to invoke their methods. The details of the coupling constraints required to achieve this will be provided in subsection 2.

where possible the design should be refactored to make the comment unnecessary [11]. In the case of the comments under discussion, the goal is to alert maintenance programmers to avoid particular couplings; the design may be completely adequate and no refactoring required.
The remainder of this paper is structured as follows. In section 5 we review related work in the area of software coupling. In section 2 we describe our notion of coupling constraint in detail, present the graphical technique we use to depict coupling constraints, and present a precise definition of the coupling constraints used in this paper. In section 4 we evaluate our work by seeking examples of coupling constraints in open source software and demonstrate how these can be detected using our prototype software tool, Lutin. Finally, in section 6 we present our overall conclusions and discuss future work in this area.

2 Static dependencies and access graphs

What is a static dependency to an entity? We assumed that, aside from the mere duplication of code which we are not addressing in this paper, a static dependency involves using an entity e by its name. If e is removed or even changed, each occurrence of its name may lead to compilation errors.\footnote{We do not currently take into account occurrences in literal strings nor in comments.}

We are thus only considering entities with a name which, following Java’s terminology \cite{Java}, we call declared entities (packages, classes, interfaces, class members ...). Names may be partially implicit in programs but we assume that a deterministic procedure can statically (i.e. before execution) produce a fully qualified name from a partial name and its context.

Static names may still be ambiguous with respect to inheritance polymorphism, which is resolved by dynamic binding, but this is intentional as our goal is to pinpoint static dependencies. We thus introduce the following definitions.

\textbf{Definition 2.1 (Owner and declaration scopes of an entity)}

Each declared entity is owned by a scope which, intuitively, is the smallest scope that strictly includes the declaration of the entity.\footnote{The Java Language specification defines the scope of an entity as "the region of the program within which the entity [...] can be referred to using a simple name, provided that it is visible" \cite{Java}§6.3.} The declaration of the entity is also typically a scope itself: the declaration scope of the entity.\footnote{The term "definition scope" would be better suited for languages like C and C++ where a declaration is not the same as a definition in which case declared entities should probably be renamed as defined entities.}

For instance, the declaration scope of a method is the whole method declaration including its body, if there is one, while the owner scope is the class or the interface bearing the declaration.
Definition 2.2 (Static dependency to a declared entity)
A static dependency to a declared entity $e$ in a program $P$ is any occurrence of the name of $e$ in $P$. An entity $c$ statically depends on $e$ when there is at least one static dependency to $e$ in the declaration scope of $c$.

Access graphs
In order to define coupling constraints as independently as possible from any particular programming language, programs are abstracted by a relational structure: an access-graph. Access graphs also make it easier to reason about static dependencies in programs, by focusing on the relevant concepts.

Nodes in access graphs denote declared entities while relations either bind entities which use other entities or are useful to qualify which entities are allowed to use other entities. Several dependencies to the same target entity that occur in the same source entity will appear as a single edge from the node of the source entity to that of the target entity.

The central relation of access graphs is the $\text{uses}$ relation.

Definition 2.3 (The $\text{uses}$ relation of a program)
Let $P$ be program. A declared entity $c$ of $P$ $\text{uses}$ another declared entity $e$ of $P$ when $c$ statically depends on $e$.

Definition 2.4 (Access Graph of a program)
An access graph $g = \langle \text{Nodes}, \text{Relations}, \text{uses}_g \rangle$ of a program $P$ is a graph whose nodes are declared entities of $P$ and with a special relation $\text{uses}_g$ which is the $\text{uses}$ relation of $P$ restricted to these entities.

Coupling constraints will be defined below as logical formulas that forbid some $\text{uses}$ edges in access graphs. In addition to the $\text{uses}$ relation, other relations (e.g. inheritance or aggregation) are typically included into access graphs to qualify what $\text{uses}$ edges are allowed or forbidden: the only requirement is that these relations can be automatically computed from a given program.

Access graphs are useful to define the semantics of coupling constraints, to reason on them, and to display what depends on what or which dependencies violate a given coupling constraint. Note however that access graphs may be displayed partially to improve readability.

Consider the Java program of figure 2. The $\text{ImageMgr}$ class manages Image documents (instances of the $\text{ImageDoc}$ class). In order to prepare

$^5$Trivial dependencies such as the mandatory occurrence of a name in its own declaration are omitted.
Figure 2: Image Manager example
the evolution of the program to support different kinds of documents, the `ImageMgr` class should not depend on the `ImageDoc` class but on a more stable abstraction.

The access graph of figure 3 was computed by the Puck tool, a spawn of Lutin written using JL[26] and JastaddJ[10]. It displays the uses relation as full lines and the contains relation (see section 3.1) as dashed lines. Squares are classes or packages, diamonds are methods or constructors and ovals are data members. The red edges are dependencies that violate a coupling constraint as will be explained below.

Six dependencies are pinpointed as problematic with respect to the (yet

---

6Using JL reduced the complexity of Puck over Lutin by at least an order of magnitude as the uses relation appeared to be supported by JL as the ... uses attribute! On the other hand JL’s uses attribute binds a declared entity to the occurrences of its name not to the entities (the scopes) which bear these occurrences.
informal) coupling constraint that the *ImageMgr* class should not be used (directly) from the class *ImageDoc*. The name of the class *ImageDoc* is used (as a type name) in the *ImageMgr* constructor, in the *ImageMgr.display* method as well as in the declaration of the *ImageMgr.images* attribute. In addition, *ImageMgr.images* method uses the *ImageDoc.getName* method and the *ImageDoc.addImage* method uses the constructor *ImageDoc* but also while doing that, also the name of the class *ImageDoc* itself.

### 3 Coupling constraints

Coupling constraints are now defined as logical expressions that forbid some *uses* edges in access graphs. The only couplings that will be considered are those that are compatible with the syntax and especially the access rules of a programming or a modeling language. Thus, coupling constraints further restrict the couplings among the syntactically correct ones.

It is beyond the scope of this paper to discuss which changes in a given access graph, if any, are able to fix the violations coupling constraints. However, the point of coupling constraints is to guide refactoring by rigourously determining which refactoring combinations are intended among the possible ones.

Coupling constraints can be interpreted at two different levels of abstraction: access graphs or programs. For instance, $\text{hiddenFrom}(b, a)$ first means that in the considered access graph an *uses* edge from $a$ to $b$ would be *incorrect*. An incorrect edge is a potential target to apply a refactoring transformation. In this paper, such edges are displayed in red. Second, given a program $P$, $\text{hiddenFrom}(b, a)$ means that in $P$ the occurrences of the name of $b$ in the scope of $a$ are incorrect.

#### The need for a logical language

An elementary formula like $\text{hiddenFrom}(b, a)$, is typically not enough to express a useful constraint for at least two reasons.

- First, a given program element typically needs to be hidden from a large number of other elements, possibly including elements which will be added to the program after the constraint was defined.

7Remember though that the forbidden *uses* edges are called *incorrect* only with respect to a given set of coupling constraints and this has nothing to do with behavior preservation or the syntax of the programming language which are always assumed to be respected.
• Second, $\textit{hiddenFrom}(b, a)$ forbids $a$ from using $b$ but says nothing about the nested elements in $a$ or $b$. Access to sub-elements often need to be restricted when access to their owners is.

We thus now introduce a first-order logical language to express coupling constraints and then higher-level predicates to ease the declaration of the most common constraints.

### 3.1 First order language

The first-order language that we propose to define coupling constraints includes:

- a set of constants,
- a set of variables,
- the usual logical symbols,
- a signature: a set of binary relational symbols including a special relation $\textit{uses}$ and a set of predicates including three special binary predicate $\textit{hiddenFrom}$, $\textit{hideFrom}$ and $\textit{canSee}$.

Given an access graph, this language can be interpreted this way: the variables and constants denote nodes or sets of nodes of the graph and binary relations denote sets of edges. The $\textit{uses}$ relation of the language denotes the uses relation of the graph. Predicates are interpreted the usual way.

The point of the language is to express which $\textit{uses}$ edges are correct and which are not. This is done by the introduction of constraints from which $\textit{hiddenFrom}$ facts can be deduced.

**Definition 3.1 (violation of a set of coupling constraints)**

*Given a set of coupling constraints $C$, a $\textit{uses}(a, b)$ edge of an access graph is a violation of $C$ if $C \Rightarrow \textit{hiddenFrom}(b, a)$.***

### 3.1.1 Dealing with exceptions

In order to allow for the introduction of local exceptions to global decoupling policies that should remain unchanged, it is advised to use the softer $\textit{hideFrom}$ predicate rather than $\textit{hiddenFrom}$ directly. Exceptions can then be introduced using the $\textit{canSee}$ predicate. Note that all the high-level predicates defined below are defined using $\textit{hideFrom}$ rather than $\textit{hiddenFrom}$.

The following axiom defines the relationship between the three predicates.
Definition 3.2 (hiddenFrom axiom)
\(\text{hideFrom}(b, a) \land \neg \text{canSee}(a, b) \Rightarrow \text{hiddenFrom}(b, a)\)

Using canSee should be done very sparingly, though, as it bypasses all the constraints that rely on hideFrom. A more cautious way to introduce exceptions consists in including them directly in coupling constraints as allowed by most of the high-level predicates defined in this document. To avoid making constraints depend on specific nodes, one can define them using variables that denote sets of nodes.

3.1.2 Other low-level relations and predicates

The simplest way to forbid node \(a\) to use node \(b\) is simply to declare the hideFrom\((b, a)\) constraint. First order constraints can also be written the usual way to hide an arbitrary set of node from other nodes. For this purpose, functional or relational symbols can be added to the logical language as long as they are unambiguously defined on access graphs and programs. For instance, the isClass unary predicate can be added to denote nodes that are classes.

Two relations are particularly useful: contains and isA. Contains can be given a rather generic definition and is quite convenient to hide a whole scope including the elements defined in it. In practice though, it is often contains*, the reflexo-transitive closure of contains, that is actually used in coupling constraints.

The precise definition of isA depends on the programming language but it is quite important for two reasons. Firstly, it is convenient to forbid the use of all the subtypes of a given type including those that have not been defined yet.

Secondly, and more importantly, the isA relation is central to solving coupling problems trough dynamic binding. If a method call \(a.m(...\) is forbidden, where \(a\) is of static type \(A\), then a common refactoring consists in declaring \(a\) to be of type \(T\), where \(T\) is a super type of \(A\) which either exists or needs to be inferred (with the appropriate methods) and introduced. So, while the isA relation is not absolutely necessary to define coupling constraints it is often essential to their satisfiability.

Definition 3.3 (contains)
A declared entity \(e\) contains a declared entity \(e'\) iff \(e\) is the owner scope of \(e'\).

Definition 3.4 (isA)
A declared entity \(s\) isA \(t\) iff both are types and \(s\) is defined as a subtype of \(t\).
This implies that wherever an expression of type $t$ is expected, an expression of type $s$ may occur.

### 3.2 Higher-level predicates and relations

The `hideFrom` predicate is quite low-level and it is often more convenient to rely on higher-level predicates and relations. The following definitions are given in first-order logic and have been implemented in prolog (see the Appendix).

**Definition 3.5 (Virtual scopes and virtual\_contains)**

A virtual scope is an arbitrary collection of declared entities that are put together so that they can easily be considered as a whole in coupling constraints. The virtual scope becomes a node that virtually contains its elements.

\[ \text{virtualScope}(s, \text{elements}) \equiv \exists \text{node node} = s \land \forall e \in \text{elements} \rightarrow \text{virtual\_contains}(s, e) \]

An example of virtual scope is given in section 4.1.1 where one of the layers of a layered architecture is not a scope but a collection of scopes.

In order to deal with virtual scopes and actual scopes uniformly in constraints it is convenient to introduce a generalize contains relation which also supports set (or any kind of collection) membership so that sets of entities and single entities can be dealt with uniformly too.

**Definition 3.6 (generalized contains)**

\[ \text{gContains}(a, b) \equiv (b \in a) \lor \text{contains}(a, b) \lor \text{virtual\_contains}(a, b) \]

A constraint `hideScope(s, facades, interlopers, friends)` hides a scope $s$, except for a set of facades, from a set of scopes (the interlopers) except from a set of friends which are not interlopers after all. Simpler versions of this predicate are also convenient:

- `hideScope(s)` that hides a scope $s$ from anything outside of it (i.e. from anything that $s$ does not gContains),
- `hideScopeBut(s, facades)` that hides $s$ except for a set of facades,
- `hideScopeFrom(s, interlopers)` that hides $s$ from a set of scopes (the interlopers),
- `hideScopeButFrom(s, friends)` that hides $s$ but from a set of scopes (the friends).
Definition 3.7 (hideScope)

\[ \text{hideScope}(\text{scope, facades, interlopers, friends}) \equiv \]
\[ \forall e \forall i (gContains^*(s, e) \land gContains^*(\text{interlopers}, i) \land)
\[ \neg gContains^*(\text{facades}, e) \land \neg gContains^*(\text{friends}, i) \land \neg gContains^*(s, i)) \]
\[ \rightarrow \text{hideFrom}(e, i) \]

3.3 Using coupling constraints and access graphs

Consider again the program of 2. Declaring an explicit coupling constraint works in two ways. First it makes explicit in an unambiguous way the decoupling intention of the architect of the application. Second it allows the automatic detection of the dependencies that do not comply with this constraint.

Depending on the intention of the developer the ImageDoc class could be hidden either from the ImageMgr class specifically or from every name space (but itself) in the program. Both constraints are equivalent for the program we are considering but if more classes are added it will be necessary to clarify which ones can access ImageDoc. Let us assume that the second option has been chosen and that the following constraint is added:

\[ \text{hideScope(ImageDoc}) \]

This not only means that the ImageDoc identifier cannot be used outside its own scope, but that the identifiers defined in the ImageDoc scope cannot be used outside ImageDoc either. For instance, the occurrence of getName line 16 is not allowed because as the static type of the d variable is ImageDoc, it statically denotes the ImageDoc.getName method. The bold identifiers in figure 2 are those whose occurrence is not allowed by the coupling constraint.

On the access graph of figure 3 each red edge denotes at least one violation of the coupling constraint.

4 Evaluation

Our approach to evaluation is to demonstrate firstly that there is a need for coupling constraints, then to show how these coupling constraints can be detected, and finally to evaluate our approach to detection on an open source example.

To determine if there is a need for coupling constraints, we consider what a programmer might do if they encounter the need for a coupling constraint in their code that cannot be expressed in the programming language itself. We hypothesise that a diligent programmer might express it as a comment to alert future maintenance programmers not to create the coupling in ques-
tion. Such comments, if discovered, could provide insight into what type of coupling constraints are required in practice.

For our case study, we examined in detail one medium-sized open-source Java application, namely DSpace version 1.5.1 [6]. DSpace is an open-source Content Management System written primarily in Java. It was originally developed jointly by MIT Libraries and Hewlett-Packard before being released into open source. It comprises just under 100 KLOC of Java code and contains 75 KLOC of comments, and so provides a rich domain in which to seek comments that relate to coupling constraints.

Our aim was to find comments that express the need for coupling constraints. We filtered the comments initially using coupling-related terms namely “access,” “coupling,” “coupled,” “depend,” “know,” and “visibility.” We then inspected each comment manually to determine if it was in fact related to coupling or not. The results of this analysis are presented in section 4.1. In subsection 4.2 we illustrate how we can detect violations of these coupling constraints and finally, in subsection 4.3 we discuss our results.

4.1 Coupling constraints found in DSpace

In the following subsections we present examples of the type of coupling constraints that were found in DSpace and, in each case, show how the constraint can be represented in our formal notation. All the evidence presented here is based on comments found in the source code, except for the first example in section 4.1.1 which is based on DSpace design documentation.

4.1.1 Decoupling from a Package

Decoupling between packages is of the upmost importance as it relates to the system architecture, and problems at this level cannot be easily resolved with local measures. As can be seen in Figure 4, DSpace uses the standard 3-tier layered architecture. A key aspect of this architecture is that each package (layer) should use only the package immediately below it. This implies that a package should be decoupled from all the other packages, except the package immediately below it. These coupling constraints can be expressed thus:

```
virtualScope('org.dspace.business',
['org.dspace.administer',
 'org.dspace.authenticate',
...]).
hideScopeFrom('org.dspace.app',
 ['org.dspace.business','org.dspace.storage']).
hideScopeButFrom('org.dspace.business',
```
Figure 4: Logical Architecture of the DSpace Application

['org.dspace.app']).
hideScopeButFrom('org.dspace.storage',
['org.dspace.business']).

The first declaration defines the Business Logic Layer as a virtual scope as it is in fact not a single package in DSpace but a collection of packages. The first constraint says that the Business Logic layer and the Storage layer may not use the Application layer. The second constraint hides the Business Logic layer to anything outside its boundaries but the Application Layer. The last constraint does similarly with the Storage layer which can only be accessed from outside its boundaries by the Business Logic layer.

Since the layered architecture is common, we also introduced a higher-level predicate, layers, so that the five constraints above could be replaced by just one:

layers(['org.dspace.app',
 'org.dspace.business',
 'org.dspace.storage']).
Another example of decoupling from a package was discovered in the METSEExport class, where the following comment appears:

We don’t pass up a MetsException, so callers don’t need to know the details of the METS toolkit.

The METSEExport class provides high-level wrapper methods to access the METS toolkit, and the comment expresses the constraint that classes that use the METS toolkit should not be exposed to any exceptions defined by the toolkit. More generally, it means that the classes in the METS package, except for METSEExport, should be hidden from the other classes of the Application layer. This can be expressed as follows:

hideScopeBut(‘org.dspace.app.mets’, ['METSEExport']).

4.1.2 Decoupling from a Class

In chapter 6 of the DSpace documentation, the following comment appears:

The BitstreamStorageManager provides low-level access to bitstreams stored in the system. In general, it should not be used directly; instead, use the Bitstream object.

This warns programmers not to use the BitstreamStorageManager class directly but to use instead the Bitstream class. Looking at this in terms of coupling constraints, what is required is that all DSpace classes other than the Bitstream class should be decoupled from the BitstreamStorageManager class. This can be achieved thus:

hideScopeButFrom(‘org.dspace.storage.bitstore.BitstreamStorageManager’,
['org.dspace.content.Bitstream']).

If a programmer accidentally uses BitstreamStorageManager from another class in the application, a coupling constraint violation will be raised.

Another example appears in the same class, BitstreamStorageManager, where the following comment appears:

The dependency on the checker package isn’t ideal...

On closer inspection, the dependency in question is actually on the class BitstreamInfoDAO. In terms of the coupling constraint required here, it is simply a matter that the BitstreamInfoDAO class should be hidden from the class BitstreamStorageManager which can be expressed thus:

hideFrom(‘BitstreamStorageManager’, ‘BitstreamInfoDAO’).
4.1.3 Decoupling one Method from another

In the Bitstream class, the following comment appears in the create method:

...This method ... does not check authorisation; other methods such as Bundle.createBitstream() will check authorisation.

This implies that the Bitstream::create method should not access the method that checks authorisation, namely authorizeAction in the AuthorizeManager class, because other methods are responsible for performing this check. This decoupling can be expressed thus:

hideScopeFrom('AuthorizeManager.authorizeAction', 'Bitstream.create').

This constraint prevents the maintenance programmer from erroneously invoking authorizeAction in the Bitstream::create method, believing authorisation to be part of creating a bitstream. If this dependency is created, the subsequent coupling constraint violation will direct the programmer to seek another solution.

Another example of decoupling from a method is found in the following comment that appears in the DAVEPersonEPerson class:

Give read-only access to the contents of an EPerson object...

The purpose of the DAVEPersonEPerson class is to serve as an Adaptor for the EPerson class, i.e., to prohibit access to the mutator methods in EPerson. A coupling constraint can be used to good effect here, in order to state that particular clients of the EPerson class are to be decoupled from its mutator methods. This avoids the necessity of creating a new interface, or relying on programmer discipline to preserve the decoupling.

Closer examination of the DSpace code reveals that the DAVEPersonEPerson class has two clients, namely Item and WorkflowItem. Also, the Eperson class contains ten mutator methods, which we refer to as EPerson_mutators. The required coupling constraint can then be expressed:

declareSet('EPerson_mutators', ['EPerson.setEmail', ...]).
hideSet('EPerson_mutators').

By defining these coupling constraints, we ensure that the client classes are not erroneously updated to access mutator methods in the Eperson class. Furthermore, the rather artificial DAVEPersonEPerson class can now be deleted from the program as its role has been assumed by these coupling constraints.
4.2 Detecting violations of DSpace coupling constraints

In the preceding section we presented evidence from the DSpace documentation that programmers see the need for coupling constraints and sometimes express them as comments. Due to the lack of language or tool support for coupling constraints, this is the only option open to them. It may be claimed that expressing coupling constraints as comments is an adequate solution. Maintenance programmers will read the comments, take heed of their advice and avoid the undesirable couplings.

To test if this is the case have used two different tools to check the constraints described in section 4.1. Both tools have been run on DSpace code to detect if the coupling constraint has been observed or not. The reason for using two tools was that the Lutin/puck prototype was not, until recently, mature enough to deal with software as large as DSpace.

So a first series of experiments were conducted using FindBugs [8], an open source static analysis tool for Java. More recently Lutin/puck was ported to JL and a GUI frontend was added to filter packages or classes so that it became possible to display only some of the nodes or edges of a huge access graph.

The same series of experiment was then run using puck. One advantage of puck over the FindBugs approach is that the constraints are written in prolog and match very closely those of this document while when using FindBugs a specific detector has to be implemented for each constraint.

A FindBugs detector examines a Java program looking for a specific set of patterns or rules by matching program bytecode against a list of specified "bug" patterns. A bug in this context is really a code smell, i.e., an undesirable design construct. The input to each detector is an XML file that provides the necessary parameters. Creating this XML file from the coupling constraints is straightforward, so we omit this detail.

In the following subsections we provide the results for coupling constraints in each of the main categories, namely decoupling from a package (section 4.2.1), decoupling from a class (section 4.2.2) and decoupling from a method (section 4.2.3).

4.2.1 Detecting package decoupling violations

In section 4.1.1 we noted several cases where DSpace packages should be decoupled from one another. Here we take one of those cases, build a detector for it and run the detector to determine if the coupling constraint is violated or not. We choose the requirement from figure 4 that the Storage layer should only be accessed from the Business Logic layer.
When this detector was executed on DSpace, five distinct violations were found in four separate packages (app.statistics, app.oai, app.util and app.webui.jsptag). It is remarkable to find the essential architecture of the application being violated at all. Each of these violations represents an instance of the Application layer bypassing the Business Logic layer and accessing the Storage layer directly. In each case, the offending access was to Storage layer functionality required by the Application layer, but that was not exposed by the Business Logic layer.

These violations would be of great concern to a software architect, as they are signs that the architecture is starting to decay. Indeed, the two violations from the app.webui.jsptag package also involved the duplication of an entire method in the Application layer, which is another clear indication of architectural decay.

Fixing these problems at this early stage is probably not a major challenge. The access to the desired functionality in the Storage layer should be exposed to the Application layer by the Business Logic layer, in keeping with the layering principle.

4.2.2 Detecting class decoupling violations

In section 4.1.2 we saw the need to decouple the BitstreamStorageManager class from all DSpace classes other than the Bitstream class.

On creating and running the detector for this decoupling constraint, five violations were found. They originated in five separate classes, namely BitstreamDAO, BrowseListTag, Bitstream, Cleanup and ItemListTag. In four cases the violation would appear to have been accidental, i.e., the programmer simply neglected to read the comment or failed to realise the import of the comment.

In the case of the violation in the BrowseListTag class it is evident that the programmer wished to circumvent explicitly the authorisation required by the Bitstream class, and so accessed the BitstreamStorageManager class directly. This suggests that the design decision expressed in the original comment is too constraining for the programmers to work with. The reporting of a violated coupling constraint in this context suggests that the access to the Bitstream and BitstreamStorageManager classes may need to be redesigned.

4.2.3 Detecting method decoupling violations

In section 4.1.3 the DAVEPersonEPerson was described. The sole purpose of this class is to provide read-only access to an instance of the EPerson class.
We built a detector for this coupling constraint and executed it. No violations were discovered. To ensure that the detector was correct, we injected several random violations all of which were detected correctly.

4.3 Discussion

Our analysis of developer comments in DSpace reveals a need for decoupling constraints. We found several cases where the developer wanted to constrain the future evolution of the program so as to avoid certain undesirable couplings, and expressed this as a comment. We only lay claim to the existence of this need; we have not tried to quantify it. We anticipate that our approach has a very high false negative rate. Most coupling constraints are probably not documented, and of the few that are, our blunt keyword search no doubt detected only a percentage of them.

We selected three coupling constraints to analyse further. A detector was developed that could detect violations of each of the chosen coupling constraints. We expected that in a well-regarded application like DSpace, no violations would be found. We were surprised to discover that two of the three coupling constraints were violated, and a total of ten violations were found. This is clear evidence that expressing coupling constraints in comments alone is not sufficient that further tool support is necessary to ensure that coupling constraints are maintained during program evolution.

5 Related Work

In spite of its maturity, coupling remains a topic that attracts the interest of researchers. In this section we review related work in this field and demonstrate that coupling constraints, their detection, and their consequences, have not been addressed in the literature.

One of the earliest works in automated detection of object-oriented design problems is that of Ciupke [4]. It aims to check a program for violations of object-oriented design principles, for example, to test if all fields are private in their class. These design principles are formulated as Prolog clauses and Ciupke shows how they can be detected in real applications. We also model the program being examined as a set of Prolog clauses and use Prolog queries to detect design violations. However, only one of the constraints Ciupke deals with is a coupling constraint, namely that a class should not know about its subclasses. He does not consider application-specific constraints which are the focus of our work.

Guéhéneuc and Albin-Amiot [13] also deal with the detection of design
problems. They argue that intra-class design problems have been well-studied and focus their attempts instead on detecting and correcting inter-class design defects. We share their viewpoint that “inter-class design defects are difficult to define independently of the application and its context.” However, they hypothesize further that design patterns embody quality architecture and that transforming structures that closely resemble design patterns to the normal pattern structure will improve architectural quality. The recognition of the problem of over-engineering caused by “pattern happy” developers \[16\] renders the first hypothesis suspect. Regarding the second hypothesis, patterns have many variations in their implementation structure, so a structure that is close to the prototypical pattern implementation may be perfectly valid in its context and not an appropriate for target for restructuring. By way of comparison, our approach is relatively agnostic in terms of design quality model, only assuming that in certain application-specific contexts, it is useful to decouple one program element from another.

The extent to which modules with poor structural measures (size, coupling, cohesion, inheritance) contribute to maintenance problems has been a topic of research for some time. Briand et al performed an empirical evaluation of object-oriented design measures to determine their ability to predict fault-proneness \[3\]. They found many coupling and inheritance measures to be correlated with the probability of fault detection in a class. In later work, Koru and Tian analysed data from two large open-source projects and found that although there is indeed a correlation between modules with poor structural measures and change-proneness, the most change-prone modules were not those with the worst structural measures \[17\]. Yu et al analyse intermodule coupling and show how the use of global variables in the Linux kernel has led to tighter coupling than was heretofore understood to be the case \[28\][29]. They suggest that this coupling raises concerns about the long-term maintainability of Linux. From our perspective, these various studies serve to confirm the importance of coupling.

Arisholm et al. investigated the use of dynamic analysis to improve the measurement of intermodule coupling \[2\]. Static object-oriented coupling measurements do not take polymorphism into account, and thus are prone to estimating incorrectly the true extent of interclass coupling. They demonstrate that dynamic measurements are better indicators of complexity than static measurements. In later work, Liu, Liu and Ana demonstrated that cheap, static analysis such as Rapid Type Analysis can compute dynamic coupling measures with almost perfect precision \[18\]. Our focus is on compile-time dependencies in order to reduce the ripple effect when one module is changed, so the use of static measures is more appropriate.

The concept of change coupling is introduced by Ratzinger, Fischer, and
Gall [23]. Modules are changed coupled if they tend to be updated at the same time, according to source code repository (e.g., CVS) data. Modules can be change coupled and have no detectable dependencies in the source code – indeed this is by far the most insidious type of change coupling as it is undetectable by source code analysis. More recent work by Eaddy et al. [9] demonstrates that non-modular crosscutting concerns tend to increase the number of defects in a program. This is is likely to be related to change coupling, in that modules that take part in a non-modular cross cutting concern can be expected to be changed coupled as well. Approaches based on source code analysis, such as ours, cannot detect this type of coupling. It can only be detected by an analysis of source code repository data.

Zaidman and Demeyer use coupling measures in combination with data mining techniques to detect key classes in [30]. They found that classes that are strongly coupled with others are likely to be key in terms of comprehending the software system. In this context, it should be noted that strong coupling is not necessarily bad. As explained by Martin [19], a module such as an abstract class can have a high number of dependencies on it, but this is not a problem as long as the module is stable, i.e., not subject to change. However, if an unstable module is similarly highly-coupled, it is likely to cause a strong ripple effect as each time the unstable module is changed, its dependant modules are likely also require updating. In our work we make no assumption that strong coupling is bad of itself, but rather enable the programmer/architect to define that certain application-specific couplings are to be avoided.

The recent work of Sarkar et al. [25] is relevant to ours in a number of ways. They point out that traditional metrics focus on the class as the module, but in large software systems it is the coupling across larger packages that is more important. The main contribution of their work is to propose and validate a set of metrics that characterizes large object-oriented software systems with regard to such dependencies. For example, they introduce a metric called the “Module Interaction Index” that measures the extent to which modules are coupled only using their correct, published interface. An imperfect value for this measure indicates that undesirable inter-module coupling is taking place. Another metric, the “Not Programming to Interfaces Index,” measures the extent to which client code uses subclasses directly, rather than through the interface provided at the root of the inheritance hierarchy. Preventing design decay in terms of these metrics is possible using coupling constraints.

There is a large body of work in the field of Impact Analysis [24] which appears on the surface to be similar to our work. Impact analysis aims to discover the parts of a program that may be affected when a modification is performed. The analysis used may be static or dynamic, but in either case...
the goal is to find other modules whose behaviour might be affected by the modification. Our focus is rather on static, compile-time dependencies, which have no impact on behaviour. For example, the static dependency of a class A on a class B can be removed by creating an interface to B and updating A to depend on this new interface. This refactoring will not however affect the possibility of a change to the class B having an impact on A, as the runtime object structures are identical in both cases.

There is some support for coupling constraints available in current software tools. In the Eclipse IDE [7], it is possible to allow only limited access to classes/packages that are included from other projects. If the client code creates a dependency on a type or class that is not permitted, the Java compiler will report a warning or error. This is in effect a limited form of coupling constraint in that it can only be applied between a project and packages/classes that are from another project. For example, to limit an Eclipse project from accessing JRE classes outside of java.io.*, the following access rules should be added to the JRE classpath in the project:

\begin{verbatim}
Accessible : java/io/*
Forbidden : **
\end{verbatim}

Another example is the import control feature provided with CheckStyle [5], an open source tool that checks Java code for a variety of coding problems. The import control feature checks that all import statements follow the layering and import rules defined in a project XML file. The motivation behind this tool is similar to ours: to prevent a programmer carelessly creating an undesirable dependency on a class in a package. Our work goes much further than this, by considering decoupling between all program elements, not only packages.

Finally, there is of course some support for coupling constraints in the programming languages through various mechanisms to restrict the visibility of program elements [1]. This support is unfortunately not sufficient as will be shown in section 4 through the comments that programmers felt were needed to warn about unwanted couplings.

Following [1], we have called our graphs “access graphs” because our fundamental relation, uses, binds program elements to the scopes which use (access) them. Our access graphs are simpler, though, than those of [1] because the uses relation abstracts various kinds of accesses. All that matters to us here, is that the name of a program entity appears or not in some scope, thereby exposing or not the scope to changes of the program entity.
6 Conclusions and Future Work

In this paper we have:

- defined the concept of static dependency as the occurrence of a name in a scope,
- defined the concept of access graph to reason about static dependencies,
- defined a logical framework to express coupling constraints that forbid some static dependencies,
- demonstrated the need for coupling constrains by finding occurrences of them in comments in Dspace,
- expressed these constraints using our language,
- found several violations of these constraints in DSpace using FindBugs and our own tool, Puck.

We thus draw the conclusion that coupling constraints should be made explicit so that they are both easy to understand by human developers and supported by tools that can detect their violations.

This should greatly help software designers analyze the impact of changes as advised for instance by [19]. They will try and keep them local by hiding the scopes which are expected to change from scopes which change at a different pace. The hidden scopes may still be used indirectly from facades or trough abstractions of their types using dynamic binding (which creates no static dependency). The point of using explicit coupling constraints is that they point out precisely where indirections and abstractions are needed to avoid over-engineering. Finally, explicit coupling constraints help prevent the decay of software architectures by pointing out where coupling constraints are not enforced any more.

Future work includes the semi-automatic control of refactoring transformations to enforce coupling constraints and application to design patterns.

References

[1] Gilles Ardourel and Marianne Huchard. Access graphs: Another view on static access control for a better understanding and use. *Journal of Object Technology*, 1(5):95–116, 2002.
[2] Erik Arisholm, Lionel C. Briand, and Audun Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Trans. Softw. Eng.*, 30(8):491–506, 2004. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2004.41.

[3] Lionel C. Briand, Jurgen Wust, John W. Daly, and D. Victor Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3): 245 – 273, 2000. ISSN 0164-1212. doi: DOI:10.1016/S0164-1212(99)00102-8. URL http://www.sciencedirect.com/science/article/B6V0N-4007R6S-8/2/91c21a8abc2f4b0e47a1786370883746

[4] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. *TOOLS*, 1999.

[5] Open Source Community. *Checkstyle version 4.4*, 2008. URL http://checkstyle.sourceforge.net

[6] Open Source Community. *DSpace version 1.5.1*, 2008. URL http://www.dspace.org

[7] Open Source Community. *Eclipse Ganymede*, 2008. URL http://www.eclipse.org

[8] Open Source Community. *FindBugs version 1.3.7*, 2008. URL http://findbugs.sourceforge.net/

[9] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, 34(4):497–515, 2008. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2008.36.

[10] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *OOPSLA*, pages 1–18, 2007.

[11] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-48567-2.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley Publishing, 1995. ISBN 0201633612.
[13] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)).* Addison-Wesley Professional, 2005. ISBN 0321246780.

[14] Y. Guéhéneuc and H. Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. *TOOLS*, 2001.

[15] Chris Kemerer. Software complexity and software maintenance: A survey of empirical research. *Annals of Software Engineering*, 1(1):1–22, December 1995. doi: http://dx.doi.org/10.1007/BF02249043. URL http://dx.doi.org/10.1007/BF02249043.

[16] Joshua Kerievsky. *Refactoring to Patterns (Addison-Wesley Signature Series).* Addison-Wesley Professional, August 2004. ISBN 0321213351.

[17] A. G. Koru and Jeff (Jianhui) Tian. Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products. *IEEE Transactions on Software Engineering*, 31(8):625–642, 2005. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2005.89.

[18] Yin Liu and Ana Milanova. Static analysis for dynamic coupling measures. In *CASCON ’06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, page 10, New York, NY, USA, 2006. ACM. doi: http://doi.acm.org/10.1145/1188966.1188980.

[19] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, October 2002. ISBN 0135974445.

[20] Mark O’Keeffe and Mel Ó Cinnéide. Search-based refactoring: an empirical study. *J. Softw. Maint. Evol.*, 20(5):345–364, 2008. ISSN 1532-060X. doi: http://dx.doi.org/10.1002/smr.v20:5.

[21] Mark O’Keeffe and Mel Ó Cinnéide. Search-based refactoring for software maintenance. *J. Syst. Softw.*, 81(4):502–516, 2008. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2007.06.003.

[22] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

[23] Jacek Ratzinger, Michael Fischer, and Harald Gall. Improving evolvability through refactoring. In *MSR ’05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New
[24] Per Rovegård, Lefteris Angelis, and Claes Wohlin. An empirical study on views of importance of change impact analysis issues. *IEEE Trans. Softw. Eng.*, 34(4):516–530, 2008. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2008.32.

[25] Santonu Sarkar, Avinash C. Kak, and Girish Maskeri Rama. Metrics for measuring the quality of modularization of large-scale object-oriented software. *IEEE Transactions on Software Engineering*, 34(5):700–720, 2008. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2008.43.

[26] Max Schäfer, Andreas Thies, Friedrich Steimann, and Frank Tip. A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs. *TSE*, 2012. To appear.

[27] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Syst. J.*, 13(2):115–139, 1972.

[28] L. Yu, S.R. Schach, K. Chen, and J. Offutt. Categorization of common coupling and its application to the maintainability of the linux kernel. *Software Engineering, IEEE Transactions on*, 30(10):694–706, Oct. 2004. ISSN 0098-5589. doi: 10.1109/TSE.2004.58.

[29] Liguo Yu and Srini Ramaswamy. Introduction to extended common coupling with an application study on linux. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 192–197, New York, NY, USA, 2006. ACM. ISBN 1-59593-315-8. doi: http://doi.acm.org/10.1145/1185448.1185492.

[30] Andy Zaidman and Serge Demeyer. Automatic identification of key classes in a software system using webmining techniques. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(6):387–417, 2008. doi: 10.1002/smr.370.

[31] Mikal Ziane, Gilles Ardourel, Marianne Huchard, and Salima Chantit. Formalizing the decoupling constraints of design patterns. In *Proceedings of the 1st OOIS Workshop on Encapsulation and Access Rights in Object-Oriented Design and Programming*, pages 44–54. Springer-Verlag, September 2003.
Appendix: Prolog definition of the high-level predicates

% shortcut notations for edges
uses(A, B) :- edge(uses, A, B).
contains(Owner, Node) :- edge(contains, Owner, Node).
virtualContains(VScope, Node) :- edge(virtualContains, VScope, Node).

% additional relations
vContains(A, B) :- contains(A, B) ; virtualContains(A, B).
sContains(A, B) :- is_list(A), member(B, A), node(B).
sContains(A, B) :- node(A), 'vContains'(A, B).

% gather all the violations
checkConstraints(Violations) :-
    findall(edge(uses, A, B), (uses(A, B), hiddenFrom(B, A)), Violations).

hiddenFrom(B, A) :- hideFrom(B, A), \+ canSee(A, B).
hideFrom(B, Node) :- hide(B), node(Node).

hide(StringNode) :- hideString(String), node(StringNode, stringLiteral, String).

% this is expensive when the graph is large
outside(Scope, Nodes) :-
    findall(Node, (node(Node), \+ 'vContains*(Scope,Node)'), Nodes).

hideFrom(Element, Interloper) :- hideScope(Scope, Facades, Interlopers, Friends),
    'vContains*(Scope,Element),% Element is in Scope
    \+ 'sContains*(Facades,Element),% Element is not in one of the Facades
    'sContains*(Interlopers,Interloper),% Interloper is in one of the Interlopers
    \+ 'sContains*(Friends, Interloper),% but not in one the Friends
    \+ 'vContains*(Scope,Interloper).% Interloper is not in Scope

hideScope(Scope, [], Interlopers, Friends) :- hideScopeButFrom(Scope, Friends),
    outside(Scope, Interlopers).

hideScope(Scope, [], Interlopers, []) :- hideScope(Scope),
    outside(Scope, Interlopers).

hideScope(Scope, Facades, Interlopers, []) :- hideScopeBut(Scope, Facades),
    outside(Scope, Interlopers).

hideScope(Scope, [], Interlopers, []) :- hideScopeFrom(Scope, Interlopers).
hideScopeFrom(FirstLayer, OtherLayers) :- layers([FirstLayer|OtherLayers]).
hideScopeButFrom(NextLayer, [Layer]) :- layers(Layers),
member(Layer, Layers),
nth0(I, Layers, Layer),
J is I + 1,
nth0(J, Layers, NextLayer).

hide(Node) :- hideSet(Set), declareSet(Set, Nodes),
              is_list(Nodes), member(Node, Nodes).

% transitive closures
tclosure(Pred,A,B) :- call(Pred,A,B).
tclosure(Pred,A,C) :- call(Pred,A,B), tclosure(Pred,B,C).
’contains+’(A,B) :- tclosure(contains,A,B).
’vContains+’(A,B) :- tclosure(vContains,A,B).
’sContains+’(A,B) :- tclosure(sContains,A,B).

% recursive transitive closures
rtclosure(_,A,A) :- node(A).
rtclosure(Pred,A,B) :- tclosure(Pred,A,B).
’contains*’(A,B) :- rtclosure(contains,A,B).
’vContains*’(A,B) :- rtclosure(vContains,A,B).
’sContains*’(A,B) :- rtclosure(sContains,A,B).