Decoupling Constraints Detection to Facilitate Software Evolution

MSc Project Final Report

Robert J. Mujica

A thesis submitted in part fulfilment of the degree of MSc Advanced Software Engineering in Computer Science with the supervision of Dr. Mel Ó Cinnéide.

School of Computer Science and Informatics

University College Dublin

26 April 2011

Contents

1 Abstract

Coupling is one of the main properties to consider for developing high quality object oriented software. It describes how tightly a class, package or routine is related to other classes, package or routine. The objective is to create classes, packages or routines with small, direct, visible, and flexible relations to other classes, packages and routines, which is known as “loose coupling”. Most object-oriented languages provide some mechanisms to keep low coupling interaction between classes and routines like defining classes or routines as private or protected or internal, in which case it can be accessed by Classes within the same assembly or package. However, they are not flexible or rich enough to express loose coupling constraints among program elements, thus forcing developers to use inadequate methods like code comments or “exception throwing” among others. This work presents a practical solution to express and detect Decoupling Constraint violations. The solution defines a meta-language to express, detects and reports Decoupling Constraint violations on Java programs, thus leading programmers to find the appropriate solution. The report presents the outcome of evaluating a medium-sized open-source Java application called DSpace [14]. The result of this work demonstrates Decoupling Constraint is a useful concept that can help to avoid system decay. It also demonstrates the need of Decoupling Constraint to complement modern Object-Oriented languages and IDEs to express and detect decoupling rules between program elements in an automated fashion. Findings also demonstrate Decoupling Constraint violations detection helps to do design-oriented refactoring, particularly if they give enough information about the possible solution.

2 Introduction

Coupling was first defined in software engineering by Stevens et al when structured programming was the norm. It was defined as "the measure of the strength of association established by a connection from one module to another" [48]. Coupling indicates the dependency of a Class or routine to other Class or routine. Highly coupling classes are most likely to be affected by changes and defects from other Classes it also may decrease reusability of the classes and increase maintenance effort leading to the loss of the initial underlying design.

Coupling is such important software quality characteristic resulting in many different measurement approaches using for instance structural coupling metrics [8,29] , dynamic coupling measurements [6], evolutionary and logical coupling [20, 51], coupling measurements based on information entropy approach [3], and more recently systems developed using aspect-oriented approach [50].

The title of this dissertation plays on the title of another paper, "Decoupling Constraints to Facilitate Software Evolution" [35], which discuss the need and the existence of Decoupling Constraints to "express the requirement that one program element (package, Class or method) should not know about another element (package, Class, method or field)" [35]. The aim of this dissertation is to demonstrate we can express application specific Decoupling Constraints among software elements and detect any violation examining open-source Java projects.

2.1 Problem Statement

Decoupling Constraints although its importance and the significant progress made in the last decade in the field of object-oriented restructuring, especially with the introduction of *design pattern* [15], *refactoring* [31], along with tools for automatic detection of *"code smell"* [39, 40]., it has very little support in current object-oriented programming languages, the following highlight the limitations and the consequences of some of the current methods a system architect can use to express Decoupling Constraints:

2.1.1 Language specific features

Traditional programming languages provide some mechanism like access modifiers to protect a method or variable from being used by unauthorised objects, or by using the Obsolete attribute (specifically in C#) to marks a program entity as one that is no longer recommended for use. However, they are very limited to express Decoupling Constraint rules, for example, in Enterprise Applications it is common the concept of layers. Layers can be defined to a large extent by using the package construct of the programming language, but it is hard or almost impossible to define the dependencies rules between layers. So you might place all your UI code in App.UserInterface package and your domain logic in App.Domain package but there is not mechanism to indicate classes in App.Domain should not reference classes in App.UserInterface. To some extent this reflects the imperfect expressiveness of common object-oriented languages.

2.1.2 System documentation

One common approach to minimize "tightly coupling Classes" is to express system architect modules interaction in documentation and commenting the source code. However these techniques are not working in practice, according to a number of studies [44, 1, 26], maintenance activities constitute more than 70% of the total costs associated with the life cycle of a software system. A considerable part of maintenance efforts is concerned with fighting the phenomenon called

software decay [32] also known as design drift [15]. In spite of the significant progress in the field of object-oriented restructuring, it still remaining a labour-intensive and time consuming that involves broad and costly human expertise in both business-specific and technical. According to some studies 50% maintainer time is spent exclusively on understanding the code [44]. This work presents an approach to prevent software decay from occurring in first place.

2.2 Objectives

The main goals of this work are the following:

1. Propose and document a Decoupling Constraint detection mechanism to find them in *Java* code.

2. Demonstrate, document and experiment Decoupling Constraint detection using a well-known medium-sized open source Java projects. This discussion is also illustrated by a number of most common Decoupling Constraints already found in the work of Ó Cinnéide and Ziane [35].

2.3 Structure of the reminder of this document

The rest of this document is broken down into the following sections:

Section 3, Background

It provides the necessary background information and detailed motivation for the need of the work performed in this document.

Section 4, Approach

It explains our approach to implement Decoupling Constraint for Java applications. It begins discussing our solution, starting with the Decoupling Constraint covered by this work, it continues with the explanation of the mechanism to detect Decoupling Constraint looking at the Java bytecode, and then it explains the representation format to define Decoupling Constraints.

Finally it explains the methodology used to demonstrate and evaluate our Decoupling Constraint approach.

Sections 5, 6 and 7: examples of common Decoupling Constraints detection

The following three sections take a specific Decoupling Constraint and look at the implementation and evaluation scenarios using a well-known medium-size Java open source application. Section 5 explains our solution to do package level Decoupling Constraint, Section 6 details our approach to decouple Classes and finally Section 7 discuss our approach to decouple Class mutators from other Classes.

Section 8: General difficulties in implementing Decoupling Constraint detection

This section takes a look at some of the main challenges to implement Decoupling Constraints detection mechanism, especially about using 3rd party frameworks like *FindBugs*. Other issues, considered on this chapter include Decoupling Constraint authoring tools to help system architects

to express Decoupling Constraints rules, specifically IDE plug-in and custom DSL (Domain Specific Language).

Section 9: Conclusions and future work

In this section we give our overall conclusions and discuss future work on this area.

Appendix A shows decoupling from a package violations report screens and the corresponding outcome using command line option.

Appendix B shows decoupling Classes violations report screens and the corresponding outcome using command line option.

Appendix C shows decoupling Class mutator violations report screens.

3 Background

Although the importance of Coupling as a key characteristic to accomplish highly quality software systems, it still remains a subject for many researchers. This section evaluates previous work in this area and reveals the needs for Decoupling Constraint definition, their detection and the consequences of not being supported by modern object-oriented programming languages.

One previous work in the object-oriented reengineering field is that of Ciupke [39], which describes a method for the automatic detection of design problem in legacy code. Ciupke used predicated implemented as *Prolog* clauses to represent or formalize what he called "good design rules" about a system, where a violation of such a guideline or rule indicate a design problem. This work was limited to detect a subset of well-known object-oriented design rules violations regardless the nature of the application in study, it concentrates specifically around the use case of a *Class* should not know about its subclasses. Contrarily this work aims to detect application specific design rules violations.

Radu Marinescu [30] presented a detection strategy for formulating metrics-based rules that capture deviations from good design principles. First the source code is parsed and the design information (e.g. Classes, Methods, and Variables) of the system and about the existing relations (e.g., inheritance, call of methods) among the entities is extracted and stored in meta-model information. Second, the metrics used for the detection is computed. These metrics are implemented as SQL queries using the tables holding the design model. Then a *Strategy* defined by Radu as "a generic mechanism for analysing a source code model using metrics" could be run on the design model of the system. Each Strategy was aimed to capture a particular well-known flaw of object-oriented design found in the literature (e.g. god method, god class, feature envy, data class and others). In the end, the results are manually analysed based on the detection strategy. He does not consider application specific design constraints which are the main goal of this work.

Yacoub *et al* addressed the problem of measuring the quality of object-oriented designs using dynamic metrics [49]. They demonstrated dynamic metrics can be used to measure the actual runtime properties of an application in contrast to the expected properties measured by static metrics. In order to use them early in the development phase, the application has to be modelled as executable design, but they recognise the excess effort in developing executable designs is justifiable for complex real-time applications where deadlines should be investigated prior to the application developing phase. Contrarily our approach is focus on compile-time analysis to minimize the ripple effect while changing one program element, so the static analysis is more applicable.

Modern integrated development environment (IDE) tools support some level of Decoupling Constraints. Eclipse IDE [17] ships with a feature called "Type Access Rules" that allows developers to define rules to help restrict access to various part of a dependency. Dependencies can be anything from other projects in your workspace, JAR files, or even the JRE libraries. Developers can define access rules that warn about a "breach of contract", or even access rules that completely prevent you from accessing certain classes, throwing compiler errors as if it was a standard compilation problem. For instance, if you want your application to be truly portable to different JVMs, you should not reference any package that begin with "com" or "sun", so you could restrict it adding the following library access rules to your Java project:

Forbidden com/**
Forbidden sun/**

On the other hand, Microsoft Visual Studio 2010 [34], specifically its Ultimate edition (the most expensive version) comes with a new feature called Layer Diagrams. Layer Diagrams are a new type of diagrams that allows developers to describe the "logical" architecture of the application. Application code can be organized into different "layers", allowing a better understanding of what objects perform what tasks. Layer Diagrams can also be incorporated into the automated build process; allowing developers to validate no architectural constraints have been violated. However, Layer Diagram is just a dependency diagram because there's no inherent recognition for layer levels. Because there's no way of defining a level, there's no way to group layers within a specific level. It's also difficult to really enforce only a particular layer having references to particular external references. You can declare forbidden namespaces for a particular layer, but it's an inconvenient method of enforcing references to particular components only occur in a certain layer.

Both tools have the same purpose as ours, prevent a developer careless creating an undesirable dependency, but they are mainly focus at the coarser-grained modules level (packages and projects), our work goes much further, by taking into account Decoupling Constraints between fine-grained program elements.

4 FindBugs Custom Bug Detector for Decoupling Constraint Detection

This section introduces the proposed approach for Decoupling Constraint detection. The first part of the section describes the solution. The second part explains the methodology used to demonstrate and evaluate our Decoupling Constraint solution.

4.1 The Approach

The solution presented on this work was implemented with the aim of a tool called FindBugs [19] (version 1.3.9). FindBugs is an open source static analysis tool that examines a program class or JAR files looking for a specific set of patterns or rules by matching program bytecode against a list of bug pattern. In order words with static code analysis tools, a developer can analyse software without actually running the program. Unlike other static analysis tools, FindBugs does not focus on style or formatting. It specifically tries to find real bugs or potential performance problems. FindBugs uses the Byte Code Engineering Library or BCEL [10] to implement its detectors. All bytecode scanning detectors are based on the Visitor pattern [15], which FindBugs implements. It provides default implementations of these methods that developers override when implementing a custom detector.

Every Decoupling Constraint detector implemented for this work is represented by one custom FindBugs Bug detector. The following describes the steps for implementing each Decoupling Constraint detector:

1. **Identify Decoupling Constraints**, the three Decoupling Constraint scenarios presented on this report were based on those found by Ó Cinnéide and Ziane [35]. Although they uncovered a small number of Decoupling Constraints while evaluating two medium-sized open sources Java applications they were very significant as they represented a wide variety of types of decoupling.
2. **Identify Decoupling Constraint bug or rule pattern**, the general idea of a detector is to find certain patterns in the bytecode of a class, to do so it reads the .class files and put them through pattern matchers that are implemented using the Visitor pattern. While reading it will call appropriate Visitor methods, depending on what element (method, field declaration etc.) is at hand. Writing a detector means implementing one or more of those methods, building up an idea about what the class is intended to do. So in order to implement a Decoupling Constraint detector we need to generate and look at the Java bytecode, because this is what the detector will have to look at. To accomplish this work a Java project was created implementing every Decoupling Constraint scenario, so that the corresponding Java bytecode could be generated and used while implementing the corresponding Decoupling Constraint detector, the following is a Java bytecode sequence generated for one of the Classes used to implemented some of the Decoupling Constraints:

```
Compiled from "BitstreamStorageManager.java"
public     class     DecouplingClassesSampleCode.BitstreamStorageManager
extends java.lang.Object{
public DecouplingClassesSampleCode.BitstreamStorageManager();
  Code:
   0:  aload_0
   1:  invokespecial #1; //Method java/lang/Object."<init>":()V
   4:  aload_0
   5:  new    #2; //class DecouplingClassesSampleCode/BitstreamInfoDAO
   8:  dup
```

```
   9:  invokespecial #3;                                          //Method
DecouplingClassesSampleCode/BitstreamInfoDAO."<init>":()V
   12: putfield      #4;                                           //Field
infoDAO:LDecouplingClassesSampleCode/BitstreamInfoDAO;
   15: aload_0
   16: getfield      #4;                                           //Field
infoDAO:LDecouplingClassesSampleCode/BitstreamInfoDAO;
   19: ldc    #5; //String test
   21: putfield      #6;                                           //Field
DecouplingClassesSampleCode/BitstreamInfoDAO.Data:Ljava/lang/String;
   24: return

public void method1();
  Code:
   0:  aload_0
   1:  aconst_null
   2:  putfield      #4;                                           //Field
infoDAO:LDecouplingClassesSampleCode/BitstreamInfoDAO;
   5:  getstatic     #7;                                           //Field
DecouplingClassesSampleCode/BitstreamInfoDAO.className:Ljava/lang/Strin
g;
   8:  astore_1
   9:  return

public void method2();
  Code:
   0:  new    #2; //class DecouplingClassesSampleCode/BitstreamInfoDAO
   3:  dup
   4:  invokespecial #3;                                          //Method
DecouplingClassesSampleCode/BitstreamInfoDAO."<init>":()V
   7:  astore_1
   8:  aload_1
   9:  ldc    #5; //String test
   11: putfield      #6;                                           //Field
DecouplingClassesSampleCode/BitstreamInfoDAO.Data:Ljava/lang/String;
   14: return
}
```

This bytecode sequence corresponds to a class called BitstreamStorageManager, which has one Constructor and two public Methods (method1 and method2). On his Constructor it instantiates a BitstreamInfoDAO object, and then it assigns a String value to its Data public property. When developing a new FindBugs detector you need to look for two things: the special constants besides every Java line e.g. new, invokespecial, putfield, getfield, etc., and whether it is a Field or a Method call indicated by the commented string on the right side. Having those two inputs you know which Visitor method(s) you need to override and the constants you need to look for in order to find the set of patterns for each individual Decoupling Constraint detector.

The following is the corresponding Java source code:

```
package DecouplingClassesSampleCode;
public class BitstreamStorageManager {
	private BitstreamInfoDAO infoDAO;

	public BitstreamStorageManager(){
		infoDAO = new BitstreamInfoDAO();
		infoDAO.Data = "test";
	}
	public void method1(){
		infoDAO = null;
		String testField = BitstreamInfoDAO.className;
```

```
        }
        public void method2(){
                BitstreamInfoDAO local = new BitstreamInfoDAO();
                local.Data = "test";
        }
}
```

3. **Define Decoupling Constraints Representation**, once we know what we are looking for and how to find it using a custom FindBugs Bug Detector, we need to define a mechanism to express application specific Decoupling Constraints, so that every detector can use it as input to find possible violations. Out of the box FindBugs does not provide any mechanism to supply inputs to custom FindBugs detectors. This research makes use of eXtensible Markup Language (XML) to represent application specific Decoupling Constraint rules. The following highlight the key reasons for using it:

    - XML is a well-known standard for representing and exchanging data (especially on Internet). So developers are very familiar with it.
    - System architects can update the corresponding Decoupling Constraint XML file while defining system specific design rules.
    - Easy to persists, modern DBMS like Oracle, Microsoft SQL Server, IBM DB2 supports the persistence and search capability of XML data type [7]. For simplicity this research makes use of file system.
    - XML flexibility lets it serve as a meta-language for defining higher markup languages to presents Decoupling Constraints for example using DSL (Domain Specific Language) [2].

4. **Evaluation**, this work is evaluated using some Decoupling Constraints found by Ó Cinnéide and Ziane [35] in DSpace application. It also complements the evaluation using DSpace system architecture documentation [14] and some others "work-around" found on DSpace source used by developers to make up for the lack of Decoupling Constraints support. The following table summarise our case-study "DSpace" (version 1.5.2) application characteristics, which is also used to evaluate the Decoupling Constraint Detectors solution proposed on this work:

Total Lines of Code	Number of Classes	Number of Packages
105399	724	71

Table 1: DSpace high level metrics

4.2 Template for Describing decoupling Constraint detection

The following section describes the format used for describing each Decoupling Constraint Detector presented on this report:

1. **Problem overview**, this section provides details of the problem each Decoupling Constraint is trying to detect to facilitate software evolution.

2. **Solution**, This section provides details on how the Decoupling Constraint was implemented and the corresponding violations it tries to detect along with its representation.
3. **Case-study Evaluation**, This section provides details of the solution evaluation on some practical scenarios using the case-study application. It also evaluates the level of correctness of the solution by using two widely used metrics: Precision and recall. Every evaluation scenario is broken down into the following subsections:
    - Scenario Overview, detailing the scenario we are trying to detect.
    - Decoupling Constraint Definition, showing how we can represent such constraint using XML format.
    - Decoupling Constraint Violations Injection, listing some violations injected to test the precision of our solution.
    - Outcome Analysis, highlight the main findings of running our detector.

4. **Summary**, this section summarizes the results of using this Decoupling Constraint approach on the case-study.

5 Decoupling from a Package

This section describes the solution to implement Decoupling Constraint from a package. Section 5.1 describes the problem it tries to solve. Section 5.2 gives some details about the solution, the XML representation along with its scope. Section 5.3.1 describes an evaluation scenario from our case-study, which is based on DSpace system architecture documentation. The second evaluation scenario described on Section 5.3.2 is based on a comment located within a DSpace's Class. Section 5.4 presents a discussion of the findings and benefits of this Decoupling Constraint.

5.1 Problem Overview

Decomposing software systems into modules has been a central topic from the early days of computer science. The benefits of a good modular decomposition were already acknowledged in the early seventies: product flexibility, comprehensibility and reduced development time [13].

The Java language allows for decomposition by means of the package construct. Every Java class is part of a package, and packages are a level of abstraction beyond classes grouping sets of classes into functional units. Classes within a package may share information with each other that's protected from other packages. Packages can therefore be considered to be the modules of a Java program [29].

Consider the package diagram shown below, where all dependencies run in a single direction.

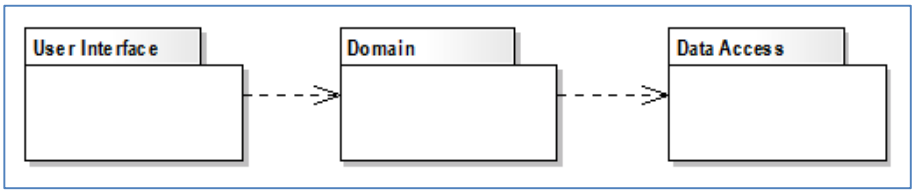

Figure 1: Package Diagram (Layered Architecture)

This diagram describes a familiar layered architecture [28], where User Interface module depends on Domain module and the Domain module depends on the Data Access module, it also shows us the Domain module insulates the User Interface from changes in the Data Access, thus if Database's interface changes, we don't need to worry about a change in the User Interface, as it will only change if the change in Data Access causes a big change in the Domain. Suppose the system needs to be able to connect to any DBMS, in which case this architecture would not be the more appropriate as a change in the Data Access module will involve changes in the Domain module, so the system architect decided to implement a data mapper pattern [28] in conjunction with a separated interface pattern [28], as shown in figure below:

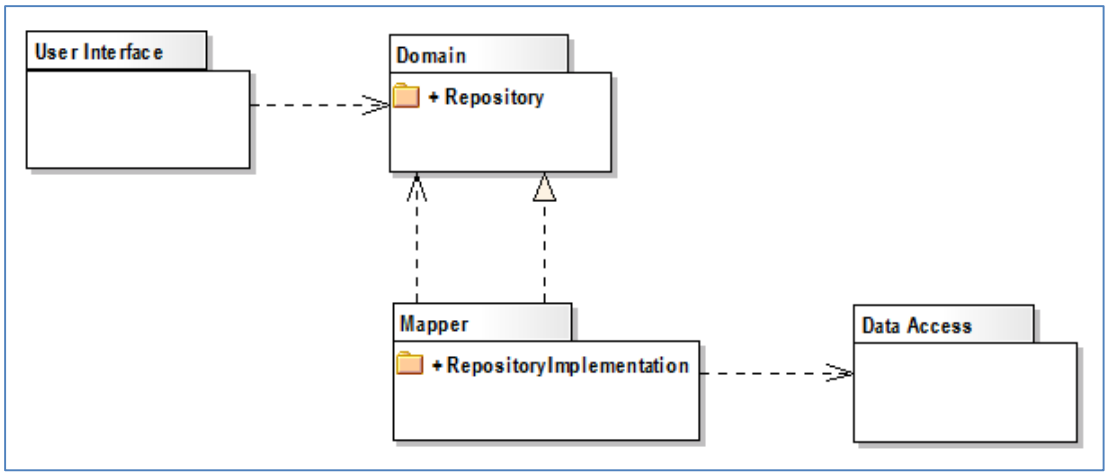

Figure 2: Package Diagram using Data Mapper and Separated Interface patterns

In this new layered architecture, the Domain defines and consequently depends on the interface (Repository) for the data mapper (RepositoryImplementation), which is then implemented in a separate package, thus breaking dependencies and reducing coupling from Domain to Data Access module. Hence changes to use different mapper implementation to support different RDBMS will not affect the Domain module.

Although the importance of decomposition and the dependency rules in this particular scenario, Java packages will not help to prevent programmers from accessing Data Access module directly from Domain, or even worse, from accessing Data Access module from User Interface module. Our Solution aims to prevent this type of coupling between packages; it will also help to enforce the implementation of enterprise architect design pattern like data mapper and separated interface patterns mentioned above, by validating at compile-time the source code against predefined Decoupling Constraints rules.

5.2 Solution

Implement a "Decoupling Constraint" detector using FindBugs to discover coupling between packages. The solution uses the Decoupling Constraint idea introduced by Ziane [35].

The solution uses an XML (see Representation below) file "to express the requirement that one or more packages should not know about other package other than some specific packages". Our solution aims to configure a specific number of packages access to a particular package. This would represent an enhancement from practical point of view to the initial idea of Ziane, which states *"A Decoupling Constraint expresses the requirement that one program element (package, class or method) should not know about another program element (package, class, method or field)"* [35], as it would minimize the work a programmer has to do to define this constraint.

The Figure below shows some Java Classes used to demonstrate how we can use this Decoupling Constraint Detector:

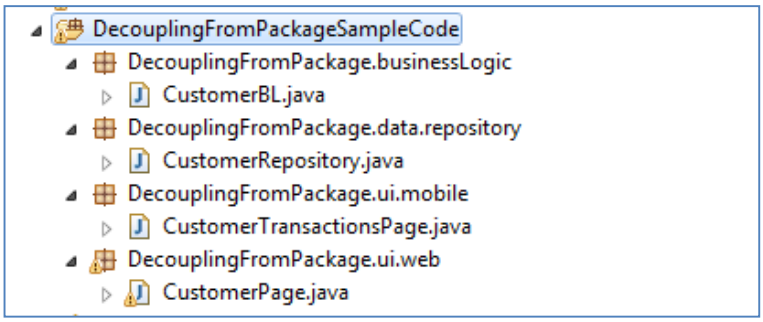

Figure 3: Java classes used to test Decoupling from a Package Constraint

- CustomerBL.java: Class located within the business logic layer (DecouplingFromPackage.businessLogic).
- CustomerRepository.java: Class located within data access layer (DecouplingPackage.data.repository)
- CustomerTransactionsPage.java, class located within UI layer (DecouplingPackage.ui.mobile)
- CustomerPage.java: class located within UI Layer (DecouplingFromPackage.ui.web).

In term of the Decoupling Constraint needed here, it is a matter that Application Classes should not access DecouplingPackage.data.repository library classes (data access layer) other than by using DecouplingFromPackage.businessLogic (business logic layer), so any UI library represented by DecouplingPackage.ui.mobile and DecouplingFromPackage.ui.web cannot access the data access layer directly,

The corresponding constraint is represented via XML in section below.

5.2.1 XML Representation

The following XML extract shows the schema to express this type of Decoupling Constraint:

DecouplingFromPackageConstraints.xml

```
<constraint
       targetPackage="DecouplingFromPackage.data.repository"
       allowedPackageList="DecouplingFromPackage.businessLogic;"
       message="only DecouplingFromPackage.businessLogic can access
       DecouplingFromPackage.data.repository Package"
/>
```

Figure 4: Decoupling from a Package Constraint definition

The following table describes XML elements and attributes:

XML elements	
targetPackage	This represents the constrained package.
allowedPackageList	List of packages allowed accessing the targetPackage.
Message	Custom message to be displayed in Bug Explorer to help developers to fix the violation.

Table 2: Decoupling from a Package Constraint XML description

5.2.2 Violation scope

The Decoupling Constraint detector should be able to report violations in any of the following scenarios:

1. Any import statement referencing the hidden package.
2. Declaration of a Private and/or Public field of a Class contained within the hidden package from any Class contained within the target package.
3. Any instantiation of a Class contained within the hidden package from any Class contained within the target package.
4. Local Method variable declaration and initialisation of a Class contained within the hidden package.
5. Method parameter of a Class contained within the hidden package.

5.3 Case-study Evaluation

This section describes the evaluation approach examining DSpace (version 1.5.2. This evaluation consists of two scenarios:

- Scenario 1: DSpace Layers Interaction Rules, it is based on DSpace system architecture [14]
- Scenario 2: Restrict Access to METS library Classes, this scenario is based on a comment found in a DSpace Class.

5.3.1 Scenario 1: DSpace Layers Interaction Rules

5.3.1.1 Scenario overview

The following diagram taken from DSpace documentation [14] shows the DSpace logical architecture:

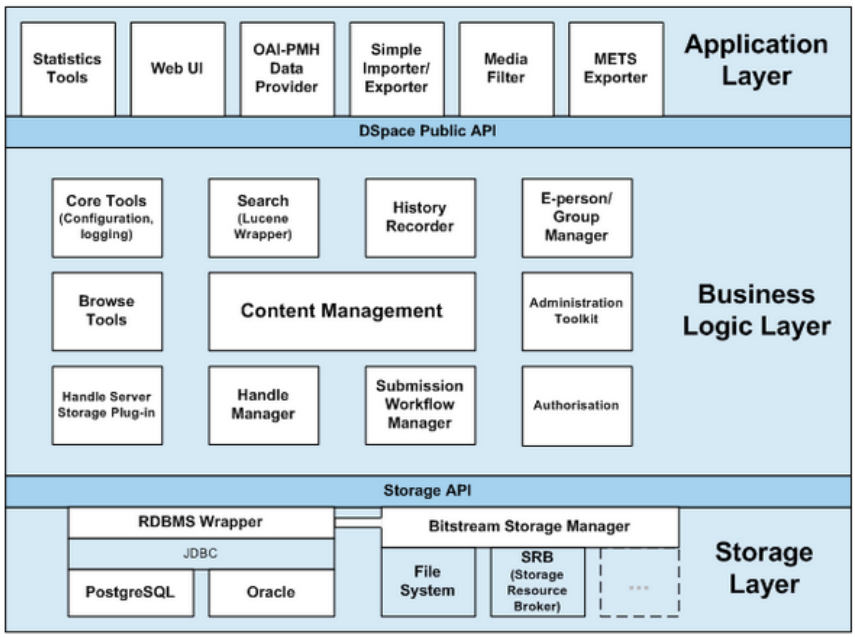

Figure 5: DSpace System Architecture

The storage layer is responsible for physical storage of metadata and content. The business logic layer deals with managing the content of the archive, users of the archive (e-people), authorization, and workflow. The application layer contains components that communicate with the world outside of the individual DSpace installation [13].

Each layer only invokes the layer below it; the application layer may not use the storage layer directly, for example. Each component in the storage and business logic layers has a defined public API. The union of the APIs of those components are referred to as the Storage API (in the case of the storage layer) and the DSpace Public API (in the case of the business logic layer). These APIs are in-process Java classes, objects and methods [14].

Packages within	Correspond to components in
org.dspace.app	Application layer
org.dspace	Business Logic layer (except storage and app)
org.dspace.storage	Storage layer

Table 3: DSpace layer distribution

Application layer

Subsystems in the application layer should be sub-packages of org.dspace.app. Subsystems in the application layer should not use the storage layer directly [13].

Business logic layer

It consists of a number of subsystems, stored in sub-packages of org.dspace, and some core classes in org.dspace.core. Sub-packages of org.dspace that are not in org.dspace.app or org.dspace.storage are business logic subsystems.

Storage layer

The storage layer currently consists of two subsystems: The relational database management system interface, and the bitstore. These are in packages org.dspace.storage.rdbms and org.dspace.storage.bitstore respectively.

5.3.1.2 Decoupling Constraint Definition

This implies a constraint that decouples the org.dspace.app packages, other than org.dspace and org.dspace.core, from org.dspace.storage and org.dspace.storage.bitstore. This can be expressed as:

DecouplingFromPackageConstraints.xml

```
<constraint
        targetPackage="org.dspace.storage.rdbms"
        allowedPackageList="org.dspace.eperson;org.dspace.authorize;org.dspace.
        browse;org.dspace.content;org.dspace.checker;org.dspace.storage.bitstor
        e;org.dspace.handle;org.dspace.content.dao;org.dspace.administer;org.ds
        pace.search;org.dspace.workflow;org.dspace.core"
        message="only org.dspace can access org.dspace.storage"
/>
<constraint
        targetPackage="org.dspace.storage.bitstore"
        allowedPackageList="org.dspace.eperson;org.dspace.authorize;org.dspace.
        browse;org.dspace.content;org.dspace.checker;org.dspace.storage.bitstor
```

```
        e;org.dspace.handle;org.dspace.content.dao;org.dspace.administer;org.ds
        pace.search;org.dspace.workflow;org.dspace.core"
        message="only org.dspace can access org.dspace.bitstore"
/>
```

Figure 6: DSpace Layers Interaction Rules representation

5.3.1.3 Decoupling Constraint Violations Injection

In order to test the precision of this Decoupling Constraint detector, the following violations were injected:

1. ***DSpaceServlet.java***, this class is located within org.dspace.app.webui.servlet package (application layer) and it is the base class for all DSpace servlets, its main responsibility is to initialise DSpace context for every http request to DSpace web portal. The following line was added to ***processRequest(...)*** function:

   ```
   DatabaseManager.find(context, "table", 0);//Decoupling from a
   Package Violation 1
   ```

   Clearly the Decoupling Constraint detector must report this line, as this class cannot access ***DatabaseManager*** class located on org.dspace.storage.rdbms package (storage access layer).

2. ***Authenticate.java***, this class is located within org.dspace.app.webui package (application layer) which is used for authenticating users. The following method was added:

   ```
   /**
        * Decoupling from a Package Violation 2
        * @param conn
        */
       private static void methodInjected(Connection conn){
          if(conn == null) return;
       }
   ```

3. ***Authenticate.java***, the following static field was also added:

   ```
   private static TableRow invalidField; //Decoupling from a Package
   Violation 3
   ```

5.3.1.4 FindBugs Outcome

Please refer to Appendix A, to view some sample screens and output violations report.

After running our Decoupling Constraint detector, it reported eight violations that can be broken down in two sections:

- Five existing DSpace violations.
- Three injected Violations.

DSpace violations

Surprisingly when running this Decoupling Constraint detector, it reported five existing violations that were already introduced by DSpace developers:

1. DSpaceContextListener: this class is located within org.dspace.app.util package and is in charge of initializing and cleaning resources used by DSpace when the web application is started or stopped. It calls the DatabaseManganer.shutdown() function which lives within org.dspace.storage.rdbms package which is not allowed to access to. Clearly this class located on Application Layer is not allowed to call this function, but it is also true it should dispose the database pool and as there is not any Business Logic class exposing this functionality, thus the developer in turn perhaps intentionally decided to "break this rule" instead of creating a new Class on the Business Logic exposing this functionality so DSpaceContextListener class can use it to free database resources.
2. LogAnalyser: This class is located within "org.dspace.app.statistics" package (thus part of Application Layer) and performs all the actual analysis of a given set of DSpace log * files. It call the DatabaseManager.querySingle(...) function for every log item to return in a TableRow data type the number of database items involved on every transaction. Instead the developer here had had implemented a Business Logic class to expose such "query" and returning a Java primitive type.
3. DIDLCrosswalk: this class located within org.dspace.app.oai package calls BistreamStorageManager.retrieve(...) org.dspace.storage.bitstore package, in this scenario I could not find any reference or usage of this class, perhaps this is a "deprecated Class".
4. BrowseListTag: this class located within "org.dspace.app.webui.jsptag" package calls BitstreamStorageManager.retrieve(..) method from within its private getScalingAttr(...) method. BitstreamStorageManager is a class located within org.dspace.storage.bitstore whose function is to store, retrieve and delete bitstreams.
5. ItemListTag: as BrowseListTag this Class is located within "org.dspace.app.webui.jsptag" package and calls "BitstreamStorageManager.retrieve(..)" method within its private getScalingAttr(...) method.

Note how the last two violations mentioned above (found on BrowserListTag and ItemListTag) reflect the decay of the original system architecture by not just only accessing data access layer subsystems where should not but also by duplicating the getScalingAttr(..) method in two different classes. The method is identical in both Classes, in which case one solution could have been the implementation of a Business Logic Class that could serve both Classes.

Violations Injected

The Decoupling Constraint detector also reported the three violations injected in three different contexts:

1. Accessing a static function:
```
DatabaseManager.find(context, "table", 0); //Decoupling from a Package Violation 1
```
2. Appearing on a function parameter list
```
private static void methodInjected(Connection conn){
    if(conn == null) return;
}
```
3. Declaring a static private field:
```
private static TableRow invalidField; //Decoupling from a Package Violation 3
```

5.3.2 Scenario 2: Restrict Access to METS library Classes

5.3.2.1 Scenario overview

This scenario was taken from Decoupling Constraints to Facilitate Software Evolution [35] paper, which in turn is based on a comment located within METSExport.java file:

"The org.dspace.app.mets package provides high-level wrapper classes like METSExport to access edu.harvard.hul.ois.mets library, this implies that the application classes should not be able to access the METS library classes, other than by using METSExport".

5.3.2.2 Decoupling Constraint Definition

The Decoupling Constraint mentioned above can be expressed as:

DecouplingFromPackageConstraints.xml

```
<constraint
       targetPackage="edu.harvard.hul.ois.mets"
       allowedPackageList="org.dspace.app.mets;"
       message="only org.dspace.app.mets.METSExport can access
       edu.harvard.hul.ois.mets"
/>
```

Figure 7: XML representation of hidden package Decoupling Constraint for this Scenario

5.3.2.3 Decoupling Constraint Violations Injection

In order to test the precision of this Decoupling Constraint detector, the following changes were made:

1. **WorkflowManager.java**, this class is located within org.dspace.workflow package., the following method were injected with a parameter of type edu.harvard.hul.ois.mets.Role :

```
/**
 * Decoupling from a Package Violation 4
 * @param role
 */
public static void fakeMethodInjected(Role role){
   role = null;
}
```

2. **JSPManager.java**, this class is located within org.dspace.app.webui package, which is used to display UI pages to users., the following field of type edu.harvard.hul.ois.mets.StructMap was injected:

```
private StructMap invalidField; //Decoupling from a Package
Violation 5
```

3. **JSPManager.java**, the following local variable was added and initialize to an existing method called showJSP(...)

```
XmlData invalidLocalVar = new XmlData(); //Decoupling from a Package
Violation 6
```

```
try {
            invalidLocalVar.wait();
      } catch (InterruptedException e) {
            e.printStackTrace();
      }
```

5.3.2.4 FindBugs Outcome

Please refer to Appendix A, to view some sample screens and output violations report.

There were ten violations that can be broken down in two sections:

- Seven existing DSpace violations.
- Three injected Violations.

DSpace Violations

As before the Decoupling Constraint detector reported a number of violations already introduced by DSpace developers, the following highlights two of the most relevant violations:

- AbstractMETSDisseminator class, which is located within "org.dspace.content" package and access a number of classes and enumerators located on "edu.harvard.hul.ois.mets" package. Our detector highlights and presents the following message: "Found Package Coupling AbstractMETSDisseminator.java:[line 306] only org.dspace.app.mets.METSExport can access edu.harvard.hul.ois.mets", thus programmers can infer a possible solution could be either moving it to org.dspace.app.mets package or to use METSExport package's classes instead.
- DSpaceMETSDisseminator, this Class is a subclass of AbstractMETSDisseminator class that was intended to be a useful sample of packager plugin to produce METS (Metadata Encoding & Transmission Standard), so by definition this Class is somehow tightly couple to org.dspace.app.mets classes, but it is located in the wrong package "org.dspace.content". Our detector reports every violation found on this class so that programmers can easily figure out this class should be part of org.dspace.app.mets package instead.

Violations Injected

The Decoupling Constraint detector reported the three violations injected in three different contexts:

1. Appearing on a function parameter list

```
private static void methodInjected(Connection conn){
      if(conn == null) return;
   }
```

2. As a Private field:

```
private StructMap invalidField; //Decoupling from a Package
Violation 5
```

3. As a local Function variable:

```
XmlData invalidLocalVar = new XmlData(); //Decoupling from a Package
Violation 6
```

5.3.3 Precision and Recall Analysis

In order to measure the level of correctness of this solution, two well-known metrics were used: **Precision and Recall** [40].

- Precision:

  Precision is the fraction of a search output that is relevant for a particular query [40]. Its calculation, hence, requires knowledge of the relevant and non-relevant hits in the evaluated domain:

$$precision = \frac{tp}{tp + fp}$$

- Recall:

  Recall on the other hand is the ability of a retrieval system to obtain all of most of the relevant items in the collection. Thus it requires knowledge not just of the relevant and retrieved but also those not retrieved, therefore in the present work Recall is not calculated as it is impossible to know the total number of relevant Decoupling Constraint violations in large applications such DSpace.

$$recall = \frac{tp}{tp + fn}$$

Where:

1. **tn / true negative:** case was negative and predicted negative
2. **tp / true positive:** case was positive and predicted positive
3. **fn / false negative:** case was positive but predicted negative
4. **fp / false positive:** case was negative but predicted positive

In the context of the present work, Precision is defined as:

$$precision = \frac{Relevant\ Positive\ Violation\ Cases}{Relevant\ Positive\ Violation\ Cases + Non - Relevant\ Violations\ Predicted\ Positive}$$

- Relevant Positive Violation Cases (tp) = 6, it is based on the Violations injected to DSpace described on Evaluation Scenarios 1 and 2 illustrated on Section 5.3.1 and 5.3.2 respectively.
- Non-Relevant Violations Predicted Positive = 0.

So

$$precision = \frac{6}{6+0} = 100\%\ positive\ predictions\ were\ correct$$

5.4 Summary

In this section, we evaluated the use of our Decoupling Constraint detector in DSpace Java application; interestingly we found a number of existing DSpace violations demonstrating the necessity of Decoupling Constraint in real applications.

Investigation of the DSpace violations revealed the most likely reason of such violations was due to lack of knowledge of the overall system design, which we categorized into the following common scenarios:

- **Implementations added into the wrong package**, for instance DSpaceMETSDisseminator Class implemented as part of org.dspace.content package, but clearly using METS related classes. Using a Decoupling Constraint developers are warned about this violation and can figure out very easily this class should be part of org.dspace.app.mets package instead.
- **Code duplication**, like the one found on ItemListTag and BrowseListTag classes, implementing the same function getScalingAttr(..) to access a prohibit BitstreamStorageManager Class methods. Having a Decoupling Constraint in place both developers are notified no just only about this undesirable dependency but also the possible solution, which in this particular case could be the implementation of a new Class within the mets package, implementing a method that can be reused by both Classes.
- **Method Returning a prohibit type**, for instance LogAnalyser class calls DatabaseManager.querySingle(...) function for every log item, which in turn returns a TableRow data type, which according to DSpace system architecture documentation it can be used only within data access specific packages. Using a Decoupling Constraint developers are warned that this dependency is undesirable and should be removed or fixed by probably implementing a new class or modifying an existing one in the business logic layer to expose the required functionality (querySingle() function) and returning a Java primitive type or a non-data access specific data type.

The findings also demonstrate our solution is able to detect violations between packages at very low level, e.g. we injected a violation by adding a private field of a type located within a prohibit packages; we also injected a violation by passing a function parameter of a forbidden type; this level of granularity become very important at the moment to report such violations to developers, so they know exactly where the offending lines are.

This evaluation also found that in most of the cases Decoupling Constraint at the package level should be complemented or reinforced by defining Decoupling Constraints at the Class level. This type of decoupling is discussed in the following section.

6 Decoupling Classes

This section describes a solution to implement decoupling between Classes. Section 6.1 describes the motivation of this Decoupling Constraint. Section 6.2 gives some details about the solution, the XML representation along with its scope. The evaluation scenario described on Section 6.3.1 is based on DSpace documentation. Finally Section 6.4 presents a discussion of the findings of this Decoupling Constraint.

6.1 Problem overview

When a class is heavily dependent on another class (e.g. it contains several method calls to the other class) it is said to be highly, or tightly, coupled to the other class. Consider the following common situation presented in Figure below. Here the **Client** class is coupled to the **Supplier** class in some way (e.g. a method in **Client** calls a method of **Supplier**). It is impossible to take the **Client** class and plug it into another system on its own. It will not compile and run without **Supplier**. If **Client** happens to be tightly coupled to **Supplier**, the chance is very high that a change in **Supplier** requires a change in **Client**.

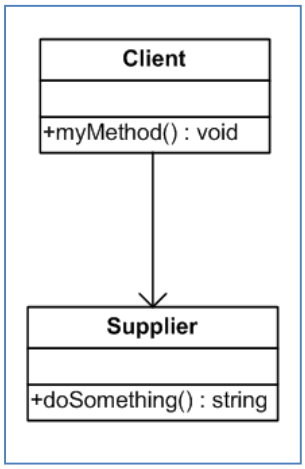

Figure 8: Coupling relationship from Client to Supplier Class

One common solution for this type of coupling is the use of Factory Method design pattern [15]. Factory Method allows us to create instances of concrete objects while depending only on abstract interfaces. So after doing some refactoring to the diagram above to implement Factory Method, our new Class Diagram looks like this:

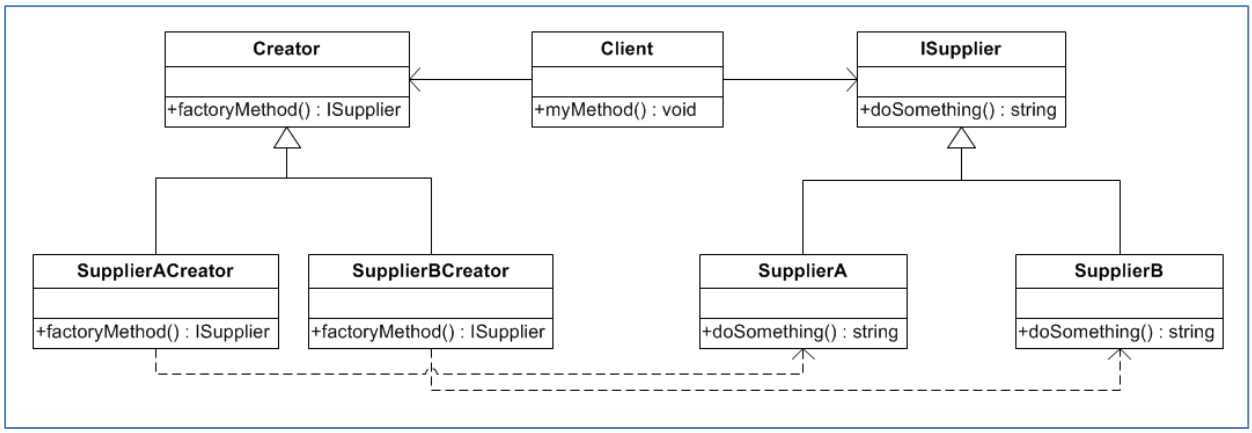

Figure 9: Factory Method to remove Coupling relationship from Client to Supplier Class

As you can see in diagram above the coupling problem between Client and Supplier is been fixed by changing Client dependencies to abstract interfaces, however a maintenance developer may accidentally add other undesirable coupling to the new Class Diagram, for instance coupling an implementation of ISupplier interface with Creator class, or even coupling ISupplier interface with concrete Creator classes. Our solution aim to prevent not just only the coupling between Client and Supplier happens in first place, but it also enforce decoupling between program elements used while doing refactoring or in this particular case while implementing Factory Method.

6.2 The Solution

Implement a Decoupling Constraint detector to discover class coupling. The solution uses an XML file to express the requirement that one Class (target class) should not know about other Class (forbidden class).

The Figure below shows some Java Classes used to demonstrate the usability of this Decoupling Constraint:

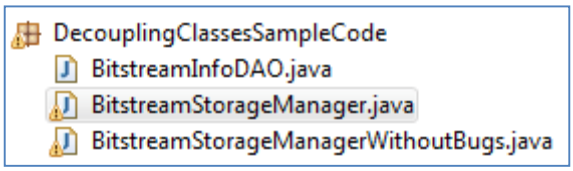

Figure 10: Java classes used to test Decoupling Classes Constraint

The following UML diagram shows the relationship among the Classes:

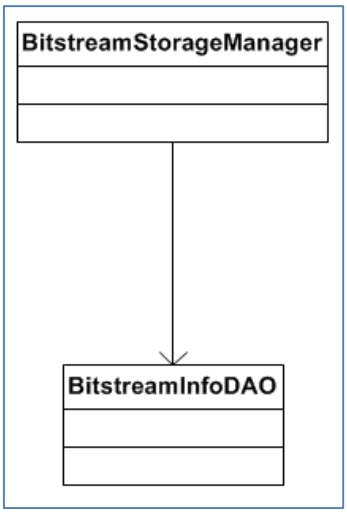

Figure 11: UML diagram of Java classes implemented for testing

As you can see BitstreamStorageManager class is coupled to BitstreamInfoDAO class, by either accessing Public field or a Public Method, or simply instantiating an Object of type BitstreamInfoDAO.

From the diagram above the Decoupling Constraint can be defined as follow:

BitstreamStorageManager *class should not access* ***BitstreamInfoDAO*** *Class.*

This constraint can be represented in XML format as shown in section below.

6.2.1 Xml Representation

DecouplingClassesConstraints.xml

```
<constraint
       targetClass="DecouplingClassesSampleCode.BitstreamStorageManager"
       forbiddenClass="DecouplingClassesSampleCode.BitstreamInfoDAO"
       message="DecouplingClassesSampleCode.BitstreamStorageManager Class
       cannot access DecouplingClassesSampleCode.BitstreamInfoDAO"
/>
```

Figure 12: Representation of Decoupling Constraint between Classes

XML Elements	
targetClass	The Class this Detector will apply this validation to.
forbiddenClass	The Class name the ***targetClass*** is not allowed to access to.
Message	Custom message to be displayed in Bug Explorer to help developers to fix the violation.

Table 4: Decoupling Constraint between Classes XML description

6.2.2 Violation scope

The Decoupling Constraint detector should be able to report violations if the following scenarios:

1. Declaration of a Private, Public or Protected field of the hidden/forbidden Class type from the target Class.
2. Instantiation of a hidden Class type object from within the target Class.
3. Access to a hidden Class Public field from the target Class.
4. Access to a Public static field or Function contained within the hidden Class from the target Class.
5. A Method parameter of the hidden Class type.

6.3 DSpace Evaluation

This section describes the evaluation of this Decoupling Constraint detector examining DSpace (version 1.5.2) source Java application. This evaluation consists of the following scenario:

6.3.1 Scenario: Decoupling BitstreamStorageManager Class from DSpace Classes

6.3.1.1 Scenario Overview

In the Chapter 6 of DSpace documentation the following description 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 in the* content management API *since that encapsulated authorization and other metadata to do with a bitstream that are not maintained by the BitstreamStorageManager.*"[11]

6.3.1.2 Decoupling Constraint Definition

In term of Decoupling Constraint required here, it is just a matter that BitstreamStorageManager Class should be decoupled from DSpace classes other than Bitstream class.

Now in order to define this constraint using our initial approach, the Decoupling Constraint definition in the xml file would look like this:

DecouplingClassesConstraints.xml

```
<constraints>
  <constraint
    targetClass="DecouplingClassesSampleCode.BitstreamStorageManager"
    forbiddenClass="Class1"
    message="Class1 Class cannot access BitstreamStorageManager"
  />
<constraint
    targetClass="DecouplingClassesSampleCode.BitstreamStorageManager"
    forbiddenClass="Class2"
    message="Class2 Class cannot access BitstreamStorageManager"
  />
.............................
</constraints>
```

Figure 13: Decoupling Class Constraint definitions using initial approach

However this solution would not be practical, as it would means we would need to add a constraint for every "forbidden class" within DSpace other than Bitstream, in other worlds, hundreds of Classes. One solution to this problem is to redefine the Decoupling Constraint representation consequently the algorithm, so that the definition of such constraint could be represented as shown in the following XML extract:

DecouplingClassesConstraints.xml

```
<constraints>
  <constraint
       targetClass="org.dspace.storage.bitstore.BitstreamStorageManager"
       allowedClassList="org.dspace.content.Bitstream;"
       message="org.dspace.content.Bitstream only can access
       DecouplingClassesSampleCode.BitstreamStorageManager"
  />
</constraints>
```

Figure 14: Decoupling Class Constraint definitions using new approach

By using this constraint representation we are able to define the same Decoupling Constraint using only few XML elements and attributes, thus simplifying system architect work.

6.3.1.3 Decoupling Constraint Violations Injection

In order to test the precision of this Decoupling Constraint detector, the following Decoupling Constraints violations were injected:

1. ***SitemapsOrgGenerator.java***, this class is located within org.dspace.app.sitemap package., the following private field of type org.dspace.storage.bitstore.BitstreamStorageManager was added:

   ```
   private BitstreamStorageManager managerTest; //Class Coupling Violation 1
   ```

2. ***SitemapsOrgGenerator.java***, the following call to a static function contained within BitstreamSotrageManager class was added to the SitemapsOrgGenerator constructor:

   ```
   BitstreamStorageManager.cleanup(true); //Class Coupling Violation 2
   ```

3. ***SitemapsOrgGenerator.java***, the following protected method was added with one parameter of type org.dspace.storage.bitstore.BitstreamStorageManager :

   ```
   /**
    * Class Coupling Violation 3
    * @param manager
    */
   protected void fakeBitstreamStorageManager(BitstreamStorageManager
manager){
      String v1 = manager.PropertyInjected; //Class Coupling Violation 4
      manager = null;
      if(v1 != null) v1 = null;
   }
   ```

In this new method we are injecting two violations, one as a parameter to the function and a second one when accessing manager.PropertyInjected public property.

6.3.1.4 FindBugs Outcome

Please refer to Appendix B, to view some sample screens and output violations report.

There were nine violations, which can be broken down in two groups:

- Five existing DSpace violations.
- Four injected Violations.

DSpace Violations

The Decoupling constraint detector reported five violations already introduced by DSpace developers:

1. BitstreamDAO Class provides data access for Bitstream through a function called getBitstream(int id), which is called by CheckerCommand Class. This function calls BitstreamStorageManager.retrieve**(...)**., and does not perform any authorization check. Instead CheckerCommand class should had been called Bitstream.retrieve() function.
2. BrowseListTag class in the same way as BitstewamDAO, it uses BitstreamStorageManager.retrieve**(...)** to retrieves the bits for the ID supplied. Curiosity analysing the calling function getScalingAttr**(...)** I noticed the following consecutive line was commented:

   ```
   //AuthorizeManager.authorizeAction(bContext, this, Constants.READ);
   ```

   This may explain why developer(s) did not use Bitstream.retrieve**()** function as it performs authorization check automatically. More over Bitstream.retrieve() do authorization check to the current context it is running under, but BrowseListTag needs to supplied the context, so developers could not use it unless a new overloaded version of retrieve function was implemented passing a context as parameter.

3. Cleanup class is used to Clean up asset store, it calls BitstreamStorageManager.cleanup**(...)** from within its main**()** function, perhaps this is not a violation as it is not being called from within DSpace application or probably a better solution could had been the implementation of a cleanup**()** function on Bitstream class.
4. DIDLCrosswalk class calls directly BitstreamStorageManager.retrieve**(...)**. the calling function instantiates a Context object manually so it can call this function, this could have be implemented by calling Bitstream.retrieve**()** instead, which under the cover instantiates a Context object. Again probably here a developer accidentally introduced this violation due to the lack of knowledge of the exiting API.
5. ItemListTag class as well as BrowseListTag class calls directly BitstreamStorageManager.retrieve**(...)** and commented the same line to skip authorization check:

   ```
   //AuthorizeManager.authorizeAction(bContext, this, Constants.READ);
   ```

It actually implements the same function signature getScalingAttr**(...)** as the one already described above, and both function looks identical. Eventually these could be avoided if developer had a Decoupling Constraint detector reporting these violations during development stage in which case they could have used Bitstream.retrieve**(...)** instead.

Violations Injected

The Decoupling Constraint detector reported the four violations injected in four different contexts:

1. As a Private field
```
private BitstreamStorageManager managerTest; //Class Coupling Violation 1
```
2. Appearing on a function parameter list and accessing it within the function:
```
protected void fakeBitstreamStorageManager(BitstreamStorageManager manager){

      String v1 = manager.PropertyInjected; //Class Coupling Violation 4

      ...

    }
```
3. Accessing a static function:
```
BitstreamStorageManager.cleanup(true); //Class Coupling Violation 2
```

6.4 Summary

In this section, we evaluated the use of our decoupling classes detector in DSpace Java application. Five existing DSpace violations were detected showing the effectiveness and the need of Decoupling Constraint to prevent and detect design rules violations.

Investigation of the five violations revealed that one of the violations found specifically within BrowseListTag class shows a developer intentionally introduced this violation as he commented a line to skip authorization check. By using Decoupling Constraint a developer can be presented with the following message “org.dspace.content.Bitstream only can access DecouplingClassesSampleCode.BitstreamStorageManager as this encapsulates authorization and other metadata … ”; based on this message the developer knows he not just only needs to use Bitstream Class but also he need to implement an overloaded version to an existing method (Bitstream.retrieve(..)) and perform authorization check.

This evaluation also shows some violations were introduced most likely due to lack of knowledge of the system (as described above on DIDLCrosswalk violation). Using an explicit Decoupling Constraint with enough information about the reason of the violation can help to direct developers to the correct solution, in this case it would be to call Bitstream.retrieve**()** function instead.

The evaluation also revealed some code duplication, like the one found on ItemListTag BrowseListTag classes, where not only the same function signature getScalingAttr(..) were implemented but also the same function body to access BitstreamStorageManager.retrieve**(...).** We also found implementation of some methods in the wrong Class, like the one found in Cleanup class, where using a Decoupling Constraint could have been pointed the developer to the right solution.

Finally the evaluation shows our solution can detect decoupling classes constraint violations in four typical context: as private field declaration, appearing on a function parameter list, accessing an object property and accessing a static function.

In the next section, we present a more granular level Decoupling Constraint: limiting access to mutator methods.

7 Limiting Access to Mutator Methods

This section describes a Decoupling Constraint implementation to limit access to mutator method. Section 7.1 describes the motivation of this Decoupling Constraint. Section 7.2 gives details about the solution, the XML representation along with its scope. The evaluation scenario described on Section 7.3.1 is based on some comments found in two DSpace Classes. In Section 7.3.2 we describe a second evaluation scenario based on the usage of exception throw artifact. Finally Section 7.4 presents a discussion of the findings of this Decoupling Constraint.

7.1 Problem overview

A very common approach to allow indirect access to private Class data is the use of public getters (accessors) to retrieve the data and return it to the caller and public setters (mutators) to allow clients to ask for data to be changed.

There are many supporters of getters and setters who generally argue that they increase the maintainability of software by creating a layer of abstraction. Johnson and Foote, for example, advise developers to "minimize access to variables" [42]; that is, to go through getters and setters rather than accessing variables directly. Moreover, setters allow a class to add validity checks to ensure that the data cannot get into an inconsistent state.

Even though the arguments in favour of getters and setters and their general use, many people argue that getters and setters break encapsulation and are bad Object Oriented (OO) design. The problems with getters and setters are summed up by Holub: *Though getter/setter methods are commonplace in Java, they are not particularly object oriented (OO). In fact, they can damage your code's maintainability. Moreover, the presence of numerous getter and setter methods is a red flag that the program isn't necessarily well designed from an OO perspective. ...Since accessors violate the encapsulation principle, you can reasonably argue that a system that heavily or inappropriately uses accessors simply isn't object oriented. ... My experience is that maintainability is inversely proportionate to the amount of data that moves between objects* [2].

One of the key arguments is that getters and setters break encapsulation. They can let any other part of the system to access the variables and change them. In addition to this problem, many argue that a need for getters and setters indicates that the data is not placed with related behaviour. Data should always be placed together with the methods that need to access it. The Law of Demeter [27] that says a module should not know about the innards of the objects it manipulates. It forbids the use of getters and setters to ensure that related data and behaviour are kept together. This means that an object should not expose its internal structure through accessors because by doing so is to expose, rather than to hide its internal structure. Thus a method should not invoke methods on objects that are returned by any of the allowed functions, in other words talk to friends, not to strangers.

Our solution is a hybrid approach from a practical point of view that consider valid the use of setters (talk to friends) to update private fields but in a limited way (not to strangers).

7.2 Solution

Implement a Decoupling Constraint detector using *FindBugs* to limit access to object mutators. The solution aims the following two contexts or representation formats:

- *Context 1*: No class should access *mutator* methods of a given Class X except A,B, C, in this scenario the designer or system architect wants to limit access from all system Classes except for a small subset of classes.
- *Context 2*: All classes can access the *mutator* methods of a given Class X except A,B, C, contrary to the scenario above, the system architect wants to limit access from a small set of Classes.

The solution uses two different XMLs schemas depending on the scenario it is used under. The Figure below shows some Java Classes used to demonstrate the need of this Decoupling Constraint Detector:

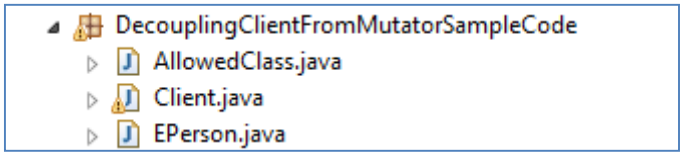

Figure 15, Java classes For limiting access to mutators

In this case the Decoupling Constraint rule can be expressed as "Client Class should be decoupled from EPerson's mutator methods, except for AllowedClass class".

Let's say a maintenance developer read the Decoupling Constraint statement mentioned above and add the following function to Client class :

```
public static void testMethod3() {
            AllowedClassTest test = new AllowedClassTest();
            test.getEPerson().m1();
      }
```

The implementation seems to be compliant with the Decoupling Constraint as it uses AllowedClassTest Class to get access to EPerson to finally execute m1 method contained on the latter one. However, it is actually violating the Decoupling Constraint, first of all this kind of code often called train wreck[1] is considered to be sloppy style and should be avoided [43], secondly Client is using an AllowedClassTest object to access a method contained on another object (EPerson) that is not supposed to access to, breaking The Demeter Law too. Our solution can detect and reports this violation either using direct access to a prohibit mutator or by using the train wreck style.

7.2.1 Xml Representation

The Decoupling Constraint mentioned above can be represented in either the following two formats based on the Contexts it is being used:

DecouplingClientFromMutatorConstraints.xml

```
<constraint
      className="DecouplingClientFromMutatorSampleCode.EPerson"
      mutatorsList="m1;m2"
      allowedClassList="DecouplingClientFromMutatorSampleCode.AllowedClass;"
                  message="Access to this Mutator is not allowed"
```

[1] *train wreck is considered a bad coding style that cause dynamic coupling between objects where any unrelated change somewhere else in the system can break the code where it is being used*

```
/>
```

Figure 16: Decoupling Constraint definition for Context 1

DecouplingMutatorFromClassesConstraints.xml

```
<constraint
      className="DecouplingClientFromMutatorSampleCode.EPerson"
      mutatorsList="m1;m2"
      message="This Class is not allowed to access this Mutator"
      forbiddenClassList="DecouplingClientFromMutatorSampleCode.Client;"
/>
```

Figure 17: Decoupling Constraint definition for Context 2

XML Elements	
className	The class this Detector will apply this validation to.
mutatorsList	Mutator names contained within the class name that this constraint will apply to.
forbiddenClassList	List of Classes that are forbidden to access the list of mutators.
Message	Custom message to be displayed in Bug Explorer to help developers to fix the violation.

Table 5: Context 1 Decoupling Constraint XML description

XML Elements	
className	The class this Detector will apply this validation to.
mutatorsList	Mutator names contained within the class name that this constraint will apply to.
allowedClassList	List of Classes that can access the list of mutators.
Message	Custom message to be displayed in Bug Explorer to help developers to fix the violation.

Table 6: Context 2 Decoupling Constraint XML description

7.2.2 Violation scope

The Decoupling Constraint detector should be able to report violations for the following scenarios:

1. Access to a mutator method defined within the target Class from a forbidden Class.
2. Access to a mutator method defined within the target Class from a Class other than the ones specified on the allowedClassList attribute.
3. Access to a mutator method defined within the target Class using the *train wreck* programming style.

7.3 DSpace Evaluation

This section describes the evaluation approach examining DSpace (version 1.5.2) source.

This evaluation consists of the following two scenarios:

- **Scenario 1**: taken from "Decoupling Constraints to Facilitate Software Evolution" [35] paper, which consists of an implementation of one class that acts as an "adapter" to access read-only properties / methods of another class.
- **Scenario 2**: based on a general "work-around" DSpace developers used to limit access to mutators by throwing exceptions.

7.3.1 Scenario 1: Provide Read-only access to EPerson Objects

7.3.1.1 Scenario Overview

This scenario was taken from "Decoupling Constraints to Facilitate Software Evolution" [35] paper, which in turn is based on the following comment located in DAVEPersonEPerson.java file of the DSpace application:

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

This comment is also related to the following comment located on DAVEPerson.java file:

> ** The "eperson" resource is a collection (in the DAV sense) of all E-people..*
>
> ** Its children are all the relevant <code>EPerson</code> objects. These*
>
> ** resources cannot be altered.*
>
> ** @see DAVEPersonEPerson*

Clearly the main goal of DAVEPersonEPerson class In DSpace is to provide read-only access to EPerson object from DAVEPerson class, see UML diagram below:

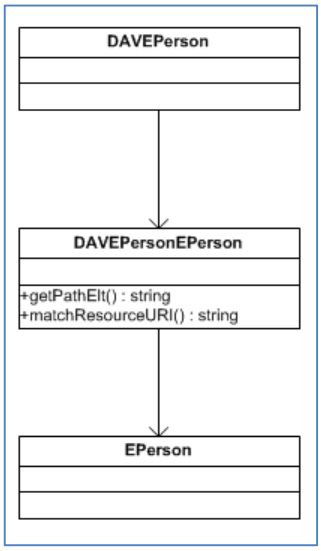

Figure 18: Class relationships / dependencies

7.3.1.2 Decoupling Constraint Definition

This scenario can be implemented by defining a decouple constraint as shown below:

DecouplingMutatorFromClassesConstraints.xml

```
<constraint
        className="org.dspace.eperson.EPerson.eperson"
        mutatorsList="setPassword;setLanguage;setEmail;setNetid;setFirstName;se
        tLastName;setCanLogIn;setRequireCertificate;setSelfRegistered;setMetada
        ta"
        message="This Class is not allowed to access this Mutator"
        forbiddenClassList="org.dspace.app.dav.DAVEPerson;"
/>
```

Figure 19: Decoupling Constraint definition using Context 2 approach

7.3.1.3 Decoupling Constraint Violations Injection

In order to test this detector the following violations were introduced in ***DAVEPerson.java*** class:

1. The following line was added to a protected function called children() :

```
EPerson self = this.context.getCurrentUser();
 self.setPassword("password" ); //Violation 1 introduced to test
'Limiting Access to Mutator Detector'
```

2. The following line was added to a protected function called propfindinternal*()* :

```
EPerson self = this.context.getCurrentUser();
self.setCanLogIn(true); //Violation 2 introduced to test 'Limiting
Access to Mutator Detector'
```

3. The following line was added to the only Constructor of DAVEPerson class:

```
EPerson self = this.context.getCurrentUser();
self.setRequireCertificate(true); //Violation 3 introduced to test
'Limiting Access to Mutator Detector'
```

7.3.1.4 FindBugs Outcome:

Please refer to Appendix C to view some sample screens and output violations report.

As expected, FindBugs found the three injected violations. It is also valid to say **DAVEPersonEPerson.java** class is no longer need as this Decoupling Constraint prevents the maintenance programmer from erroneously invoking EPerson's mutator methods in DAVEperson without it.

7.3.2 Scenario 2: Exception throws to restrict access to Classes Accessors and Mutators

7.3.2.1 Scenario Overview

While evaluating all "exception throws" statements in DSpace, a new "work-around" to make up for the lack of Decoupling Constraint support was found. Developers used exception throw artifact to report not supported or no needed operations to other developers.

The following table describes the different scenarios found in DSpace application where using "exception throws" work-around:

Throw Statement	Number Of Classes	Number Of matches	Description
throw new DAVStatusException(HttpServletResponse.*SC_NOT_IMPLEMENTED*.....	12	34	*DAVResource* class is a superclass for all DSpace "resources". This class defines a number of abstract methods that are not applicable for some read-only resource collection (subclasses) like *DAVEperson*, *DAVLookup* classes, where developers used this exception throw approach to restrict access to mutators operations.
throw new DAVStatusException(HttpServletResponse.SC_METHOD_NOT_ALLOWED	10	13	This "exception throwing" type is used for example by *DAVResource* subclasses that does not support some mutators or accessor operations against the "resource" it represents, e.g. *DAVItem*, *DAVBitstream*, *DAVPerson*, etc.

Table 7: Exception Throw findings

It is highly probable due to the number of classes using this coding artefact a developer either accidentally or through misunderstanding may call those "prohibit" methods and the only way they can be detected is in runtime. By using "limiting access to mutator" Decoupling Constraint, developers will get a warning when checking for Decoupling Constraint violation during compilation. Please refer to Appendix C to view some exception throw routines detected by this Decoupling Constraint.

Note as our solution aims to detect violations in compile time, it cannot detect whether the exception get throw in runtime, however it will help system architect to define, detect and report such work-around.

7.4 Summary

The evaluation performed on this section shows Decoupling Constraint from mutators is an effective technique to prevent access from one Class to another Class mutators, thus preventing the implementation of new Classes like the case of DAVEPersonEPerson class, which won't stop the developers from using the target class directly whatsoever.

The findings of this evaluation as well as the ones found on decoupling classes evaluation (Section 6), show the need to support two types of Decoupling Constraint representation format: one to limit access from all system Classes except for a small subset of classes and second one to allow access from all system Classes but a small set of Classes; thus minimizing the extra amount of work a system architect or programmer has to expend to define such decoupling rules.

The evaluation also showed Decoupling Constraint can be used to detect the use of obsolete or non-implemented mutator or accessors, so that they can be used during refactoring.

In the next section we explore the difficulties in implementing Decoupling Constraints and the possible techniques that can be used.

8 General Difficulties in Implementing Decoupling Constraint Detection

This chapter explores the difficulties in implementing, representing and detecting Decoupling Constraints in order to investigate the best approach from a practical point of view. It focuses on Java related technologies but it also covers some other non-Java related techniques that we consider relevant to this work.

- Section 8.1 reviews the different tools and techniques for parsing a Java project.
- Section 8.2 reviews the different techniques to express Decoupling Constraints.

8.1 Parsing Java Project

When implementing Decoupling Constraint detection, examining Java source code is one of the most important and complex components to be implemented, as it help to detect the different Decoupling Constraint violations patterns.

There are a number of Frameworks, techniques and 3rd parties tools that can help with not just only examining Java source but also to report in some standard format, the following section describe the pros and cons of using some of them:

8.1.1 FindBugs

FindBugs is a static analysis tool that examines Class files or JAR files looking for possible problems by matching the program bytecode against a list of bug patterns.

The following describes the main benefits of using this tool:

- Highly customisable, *FindBugs* utilizes the Byte Code Engineering Library or BCEL [10] to implements its detectors. All bytecode scanning detectors are based on the Visitor pattern, which FindBugs implements providing default implementations that you override when implementing a custom detector.
- Easy to integrate as part of build process, *FindBugs* can be executed from a command line and using Apache Ant [4] task.
- Easy to run as an Eclipse [17] plug-in, NetBeans [38] and using Maven [5], it also reports bugs on an Eclipse *FindBugs* Perspective view.
- It can generate XML or text output, including the exact offending line of code.

One of the main disadvantages of this tool is its lack of documentation.

Generally speaking any static analysis tool that provides some level of customisation is a good starting point to implement Decoupling Constraint detection as they provide some high level APIs that give you access to Class metadata and source code that simplifies the search algorithm.

8.1.2 Using Reflection

Reflection is a powerful feature contained in modern programming languages like Java and C#, which gives access to internal information or metadata for classes loaded into the VM. That is, Reflection lets you build flexible code that can be assembled at run time without requiring source code links between components.

In Java world it can be implemented by importing the "java.lang.reflect.*" package [23] or using open source Frameworks like JaxMe [24, 25], which help you to generate Java sources and to inspect Java source.

Reflection although the apparently flexibility for reading *Classes* metadata it would not be the best solution to implement Decoupling Constraint detection for the following reasons:

- The algorithm to find Decoupling Constraint violations needs to get access not just only to Class metadata but also to the actual source code, so that it can uncover some pattern to detect the violation, Reflection does not provide access to source code.
- Additional complexity and overhead as a "custom framework" is required to iterate through the Java project and to generate a decent report.

8.2 Expressing Decoupling Constraints

Undoubtedly there is a need for a language to express Decoupling Constraint that is not bound to a particular language. Base on this work we can infer a solution to represent Decoupling Constraints should have the following characteristic:

- Very rich or extensible, to support different program elements (package, class, method, attribute, etc.) and their attributes such as public, private, abstract, final and so on.
- Simple, to manipulate and define decouple constraints in a "short expression" or "small graph".
- Unambiguous syntax, to avoid Decoupling Constraint duplications.
- Scalable, so that it can support large applications.

Solutions to express Decoupling Constraint can be classified in two main areas:

8.2.1 Textual

8.2.1.1 Using Xml

This is based around notion of elements, each of which has a name, some attributes, and some child elements. Xml representation helps to address most of the characteristics mentioned above, except for being Scalable, as it will become unmanageable and highly error prone in large application.

8.2.1.2 Textual Domain Specific Language (DSL)

Jack Greenfield et *al* came to the conclusion a domain specific Language (DSL) is a language that enables the specification of software from a specific viewpoint; it defines abstractions that encode the vocabulary of the domain that is the focus of the viewpoint [22]. A general-purpose language (GPL) is not viewpoint based. Instead, as GPL supports only generic, undifferentiated specification. That is, DSLs help to reduce software development complexity by raising the abstraction of level towards an application domain. Unlike GPL such as OMG's Unified Modelling Language (UML) [47], DSL brings the model much closer to the domain experts and enables easier maintenance and evolution of such models

The most common DSLs in the real world are textual, as it provides a bridge between the textual nature of the code and the ability to be comprehended by non-technical stakeholders. Additionally, textual DSLs are much simpler to create than creating a visual representation of the same data.

Please refer to Michael Pfeiffer and Josef Pipchler paper [32] for further details about the different tools to support textual Domain-Specific Languages and the criteria to be considerate to select one or another.

8.2.2 Graphical

8.2.2.1 Graphical DSL

Another type of DSL is the graphical domain specific languages, is a DSL that is not textual, but rather uses shapes and lines in order to express intent. A good example would be UML. That is a DSL for describing a system. There has been a lot of effort invested in making possible to write your own graphical DSL.

Microsoft has Visualization and Modelling SDK – Domain Specific Language (VMSDK) [34], which is supported by the latest version of Visual Studio (2010) [34]. VMSDK provides the following main features:

- Editor to define graphical notation and to view the model.
- Serialization methods that save the model in readable XML.
- Facilities to generate program code and other artifacts using text templating.

On the Java side, Eclipse has an "Eclipse Modeling Project" which provides a lot of capabilities to develop Graphical and Textual DSL. Those capabilities can be described in the following sections:

- Abstract Syntax Development
  - Eclipse Modelling Framework (EMF): it is a modelling framework and code generation facility to develop tools and other applications [16].

- Concrete Syntax Development
  - Graphical Modeling Project (GMP): provides generative components and runtime infrastructure for developing graphical editors [21].
  - Textual Modeling Project (TMF): provides tools and frameworks for developing textual syntaxes and corresponding editors [46].

- Model Transformation
  - Model To Model Transformation (M2M): framework for model-to-model transformation languages [36].
  - Model to Text Transformation (M2T): provides framework and standards to generate textual artifacts from models [37].

Based on a comparative qualitative study [9] where a number of students were divided into 2 groups for developing DSL using a different tool, and also answered a survey about their experience with the tool, Eclipse Modeling Plug-Ins has been better accepted by the students than Microsoft DSL Tools. This study found that Eclipse Modeling Plug-Ins was better accepted

because Eclipse provides an open environment to integrate external plug-ins, however this is something Microsoft is catching very quickly with the new Visual Studio 2010 Extension Manager [18].

8.3 Summary

In this section we discuss the main challenges in implementing Decoupling Constraint detection. This discussion highlights the different technology that can be used to parse source code and to express Decoupling Constraint rules.

Based on our work we can state Decoupling Constraint must be defined unambiguously by system architect or programmers while designing or implementing system functionality, thus the mechanism to define them must be very simple but rich syntactically to be able to support any system specific constraint. The adoption of modern modelling tools and frameworks plays a key role to not just only raise the abstraction towards an application domain but also to easily integrate and even generate Decoupling Constraints automatically, thus simplifying, minimizing and promoting their adoption.

Our findings also show the simplicity and detection speed of Decoupling Constraint algorithms depends on the mechanism or Framework adopted to parse source code; the richer source context and extension points it has the easier and faster the algorithm will be. FindBugs provides a lot of flexibility based on the Visitor pattern and Visual Studio 2010 Static Code analysis tools uses a new technique called Introspection which unlike Reflection technique used by previous version provides much richer analysis infrastructure and uses the Visitor pattern too.

9 Conclusions and Future Work

In this research we presented a custom FindBugs-Bug-detector-based approach for Decoupling Constraint detection. The specific contribution of this work includes:

- **The definition of a detection technique for three representative Decoupling Constraints**: *Decoupling from a Package*, *Decoupling Classes* and *Limiting Access to Mutator Methods*.

- **Empirical evidence that Decoupling Constraints exist and can be expressed and detected in real-world applications,** this work also demonstrated Decoupling Constraint is a useful concept that can be integrated as part of modern IDE for easy use.

- **Decoupling Constraints as a mechanism to validate system architecture constraints** during compile-time, as it allows system specific decoupling rules definition at both coarse and fine-granular program elements.

- **Decoupling Constraints as a *design-oriented refactoring***, by reporting in compile-time design rules violations and by giving enough information about the possible solution.

- **Use of Static Code Analysis tools**, as a practical solution with a high level of precision to implement Decoupling Constraint Detection.

- **Decoupling Constraint as effective technique to prevent software decay,** as it reports any violation to the initial design rationally as changes are made, it also prevents the implementation of unnecessary Classes or Interface to restrict access from a Class to another class methods. This study found when a Decoupling Constraint rule was broken, the offending lines (method or Class) have some level of code duplication.

- **Decoupling Constraints can be expressed,** using XML format. It also found the importance of highly expressive metalanguage to facilitate the definition of Decoupling Constraint rules for any program element.

- **Decoupling Constraints Detection as part of Development Life Cycle**, Decoupling Constraint Detection can be integrated as part of software development life cycle as it can be run when compiling the application on Eclipse IDE or via command line as part of the continuous integration process.

This research work has examined one open-source Java application using this detection method and the results reported on this work show so far that this technique can be successfully applied. Despite the fact that more experimental study is needed further case-studies like evaluating other Java applications and also implementing other type of Decoupling Constraints will validate and refine the approach.

One interested point for future investigation is the auto-generation and execution of language independent Decoupling Constraint Detectors based on system models using new technology such as DSL or Metaprogramming. By creating some sort of metadata or design model layer, system architect will model or document Decoupling Constraints while designing, thus providing a more abstract parsing mechanism that is language independent and also generating detailed system design information enough to automate Decoupling Constraint generation.

10 References

1. A. Eastwood. Firm fires shots at legacy systems. *Computing Canada*, 19(2):17, 1993.
2. Allen Holub. Why getter and setter methods are evil. *JavaWorld*, 2003.
3. Allen, E. B., Khoshgoftaar, T. M., and Chen, Y., *Measuring coupling and cohesion of software modules: an information-theory approach, in Proceedings of 7th International Software Metrics Symposium (METRICS'01)*, April 4-6 2001, pp. 124-134.
4. Apache Ant, *http://ant.apache.org/*.
5. Apache Maven, *http://maven.apache.org/*.
6. Arisholm, E., Briand, L. C., and Foyen, A., Dynamic coupling measurement for object-oriented software, *IEEE Transactions on Software Engineering*, vol. 30, no. 8, August 2004, pp. 491-506.
7. Bertino, E.; Catania, B. Integrating XML and databases. *IEEE Internet Computing*. 2001.
8. Briand, L. C., Devanbu, P., and Melo, W. L., *An investigation into coupling measures for C++, in Proc. of International Conference on Software engineering (ICSE'97)*, Boston, MA, May 17-23 1997, pp. 412 - 421.
9. Building Tools for Model Driven Development. Comparing Microsoft DSL Tools and Eclipse Modeling Plug-Ins, *http://users.dsic.upv.es/workshops/dsdm06/files/dsdm06-11-Pelechano.pdf*.
10. Byte Code Engineering Library, *http://jakarta.apache.org/bcel/*.
11. Chapter 6. DSpace System Documentation: Storage Layer, *http://www.dspacedev2.org/1_5_2Documentation/ch06.html#docbook-storage.html* .
12. D.L. Parnas, Carnegie-Mellon University, On the criteria to be used in decomposing systems into modules, *Communications of the ACM*, Vol. 15, No. 12, 1972, pp. 1053 – 1058.
13. DSpace Public API Javadocs. Located on /dspace/dspace-api/src/main/java folder.
14. DSpace System Architecture, *http://www.dspacedev2.org/1_5_2Documentation/ch08.html*.
15. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series. Addison Wesley, 1996.
16. Eclipse Modeling Framework, *http://www.eclipse.org/modeling/emf/*.
17. Eclipse, *http://www.eclipse.org/*.
18. Extension Manager in Visual Studio 2010, *http://www.infoq.com/articles/extension-manager-2010*.

19. FindBugs, *http://findbugs.sourceforge.net/*.

20. Gall, H., Jazayeri, M., Krajewski, J., *CVS Release History Data for Detecting Logical Couplings, 6th International Workshop on Principles of Software Evolution (IWPSE'03)* Sept. 1 - 2, 2003, pp. 13 - 23.

21. Graphical Modeling Project, *http://www.eclipse.org/modeling/gmp/*.

22. Jack Greenfield and Keith Short, with Steve Cook and Stuart Kent, *Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools*.

23. Java Reflection, *http://java.sun.com/developer/technicalArticles/ALT/Reflection/*.

24. Java Source Reflection, *http://ws.apache.org/jaxme/js/index.html*.

25. Java Source Reflection, *http://ws.apache.org/jaxme/js/jparser.html*.

26. L. Erlikh. Leveraging legacy system dollars for e-business. *IEEE IT Pro*, May/June:17–23, 2000.

27. Law of Demeter. *http://c2.com/cgi/wiki?LawOfDemeterAndCoupling*, 2008.

28. M Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional (November 15, 2002).

29. M.Fowler, Reducing Coupling, *IEEE Software* July August, 2001, pp. 102-105.

30. Marinescu, R., *Detection strategies: metrics-based rules for detecting design flaws*.

31. Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

32. Michael Pfeiffer and Josef Pichler, *A Comparison of Tool Support for Textual Domain-Specific Languages*, *http://www.dsmforum.org/events/DSM08/Papers/1-Pichler.pdf*.

33. Microsoft Visual Studio 2010, *http://www.microsoft.com/visualstudio/en-us/*.

34. Microsoft Visualization and Modelling SDK – Domain Specific Language, *http://msdn.microsoft.com/en-us/library/bb126259.aspx*.

35. Mikael Ziane, Mel O' Cinneide, *Decoupling Constraint to Facilitate Software Evolution*.

36. Model to Model Transformation, *http://www.eclipse.org/m2m/*.

37. Model to Text Transformation, *http://www.eclipse.org/modeling/m2t/*.

38. NetBeans IDE, *http://netbeans.org/*.

39. O. Ciupke. *Automatic detection of design problems in object-oriented reengineering. In Proceedings of Technology of Object-Oriented Languages and Systems - TOOLS30*, pages 18–32, 1999.

40. Precision and Recall, *http://webology.ir/2005/v2n2/a12.html*.

41. R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, ”Politehnica” University of Timisoara, 2002.

42. Ralph Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22– 35, 1988.

43. Robert C Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. 2008

44. S. Huff. Information systems maintenance. *The Business Quarterly*, 55:30–32, 1990.

45. T. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, 10(5):494–497, 1984.

46. Textual Modeling Project, *http://www.eclipse.org/modeling/tmf/*.

47. Unified Modelling Language, *http://www.uml.org/*.

48. W. P. Stevens, G. J. Myers, and L. L. Constantine, Structured design, *IBM Systems Journal*, vol. 13, pp. 115–139, 1974.

49. Yacoub, S.M. ; Ammar, H.H. ; Robinson, T. Dynamic metrics for object oriented designs. *IEEE Software Metrics Symposium*, 1999. Proceedings. Sixth International.

50. Zhao, J., Measuring Coupling in Aspect-Oriented Systems, in Proc. of 10th *IEEE International Soft. Metrics Symposium (METRICS'04)*, Chicago, USA, 2004.

51. Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S., Mining Version Histories to Guide Software Changes, *IEEE Transactions on Software Engineering*, vol. 31, no. 6, June 2005, pp. 429-445.

Appendix A: Decoupling from a Package Violations Report

Figures in this appendix show decoupling from package violations report displayed on Eclipse Bug Explorer perspective and a Table with the “offending lines” when using FindBug command line:

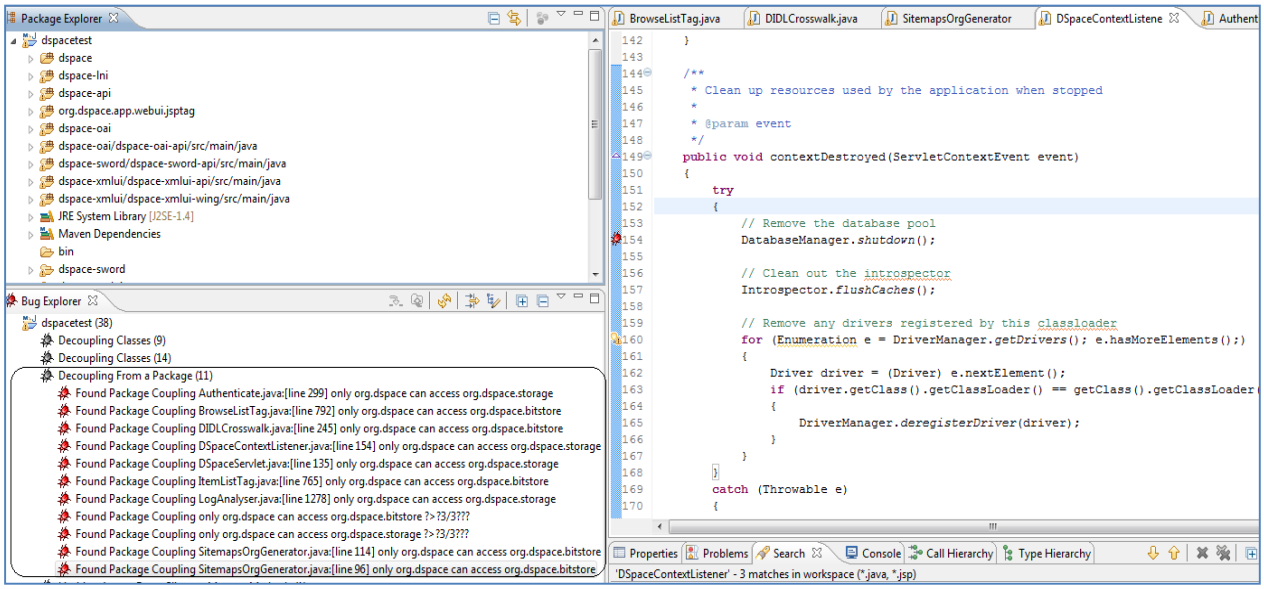

Figure A.1 – Decoupling from a package violations report

DSpace Violations

W:/Robert Stuff/Thesis/DSpaceWS/dspacetest/dspace-oai/dspace-oai-api/src/main/java/org/dspace/app/oai/DIDLCrosswalk.java:245 **Found Package Coupling DIDLCrosswalk.java:[line 245] only org.dspace can access org.dspace.bitstore**

W:/Robert Stuff/Thesis/DSpaceWS/dspacetest/org.dspace.app.webui.jsptag/org/dspace/app/webui/jsptag/BrowseListTag.java:792 **Found Package Coupling BrowseListTag.java:[line 792] only org.dspace can access org.dspace.bitstore**

W:/Robert Stuff/Thesis/DSpaceWS/dspacetest/dspace-api/org/dspace/app/util/DSpaceContextListener.java:154 **Found Package Coupling DSpaceContextListener.java:[line 154] only org.dspace can access org.dspace.storage**

W:/Robert Stuff/Thesis/DSpaceWS/dspacetest/dspace-api/org/dspace/app/statistics/LogAnalyser.java:1[19]8 **Found Package Coupling LogAnalyser.java:[line 1[19]8] only org.dspace can access org.dspace.storage**

W:/Robert Stuff/Thesis/DSpaceWS/dspacetest/org.dspace.app.webui.jsptag/org/dspace/app/webui/jsptag/ItemListTag.java:765 **Found Package Coupling ItemListTag.java:[line 765] only org.dspace can access org.dspace.bitstore**

Table A.1: Tables showing Decoupling from a package violations from command line

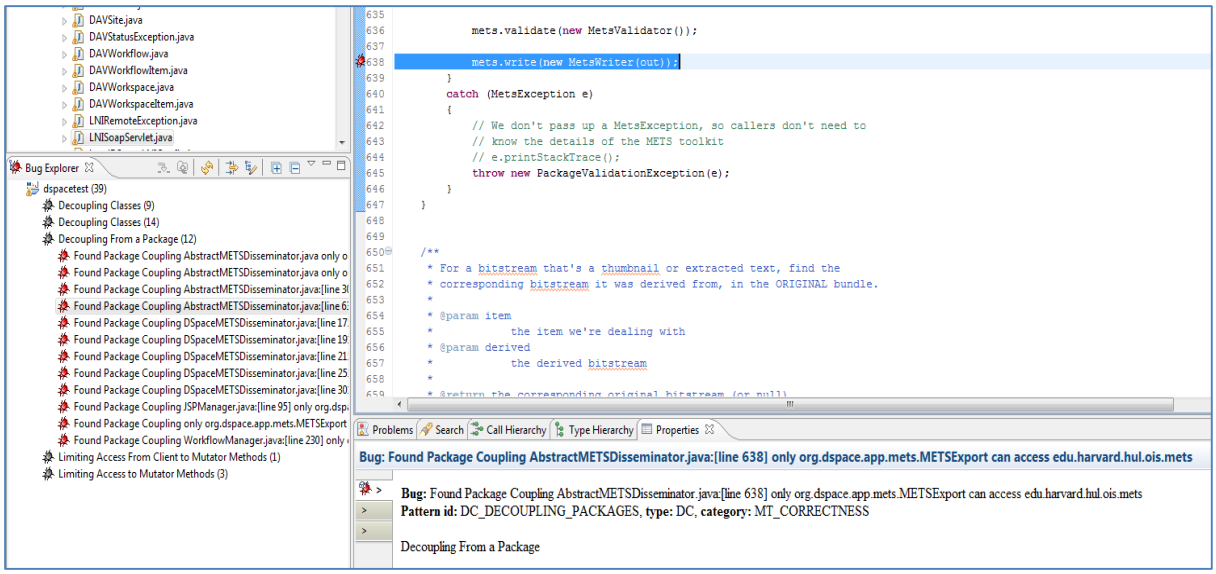

Figure A.2 – Decoupling from a package violations report. On the right hand side it shows one violation in AbstractMETSDisseminator class

DSpace Violations

```
W:/Robert Stuff/Thesis/DSpaceWS/dspacetest/dspace-api/org/dspace/content/packager/AbstractMETSDisseminator.java:638
Found Package Coupling AbstractMETSDisseminator.java:[line 638] only org.dspace.app.mets.METSExport can access edu.harvard.hul.ois.mets
W:/Robert Stuff/Thesis/DSpaceWS/dspacetest/dspace-api/org/dspace/content/packager/DSpaceMETSDisseminator.java:175
Found Package Coupling DSpaceMETSDisseminator.java:[line 175] only org.dspace.app.mets.METSExport can access edu.harvard.hul.ois.mets
W:/Robert Stuff/Thesis/DSpaceWS/dspacetest/dspace-api/org/dspace/content/packager/DSpaceMETSDisseminator.java:191
Found Package Coupling DSpaceMETSDisseminator.java:[line 191] only org.dspace.app.mets.METSExport can access edu.harvard.hul.ois.mets
W:/Robert Stuff/Thesis/DSpaceWS/dspacetest/dspace-api/org/dspace/content/packager/DSpaceMETSDisseminator.java:215
Found Package Coupling DSpaceMETSDisseminator.java:[line 215] only org.dspace.app.mets.METSExport can access edu.harvard.hul.ois.mets
W:/Robert Stuff/Thesis/DSpaceWS/dspacetest/dspace-api/org/dspace/content/packager/DSpaceMETSDisseminator.java:301
Found Package Coupling DSpaceMETSDisseminator.java:[line 301] only org.dspace.app.mets.METSExport can access edu.harvard.hul.ois.mets
W:/Robert Stuff/Thesis/DSpaceWS/dspacetest/dspace-api/org/dspace/content/packager/AbstractMETSDisseminator.java:306
Found Package Coupling AbstractMETSDisseminator.java:[line 306] only org.dspace.app.mets.METSExport can access edu.harvard.hul.ois.mets
W:/Robert Stuff/Thesis/DSpaceWS/dspacetest/dspace-api/org/dspace/content/packager/DSpaceMETSDisseminator.java:255
Found Package Coupling DSpaceMETSDisseminator.java:[line 255] only org.dspace.app.mets.METSExport can access edu.harvard.hul.ois.mets
```

Table A.2: Tables showing Decoupling from a package violations report using command line

Appendix B: Decoupling Classes violations report

Figures in this appendix show decoupling Classes violations report displayed on Eclipse Bug Explorer perspective and a Table with the "offending lines" when using FindBug command line:

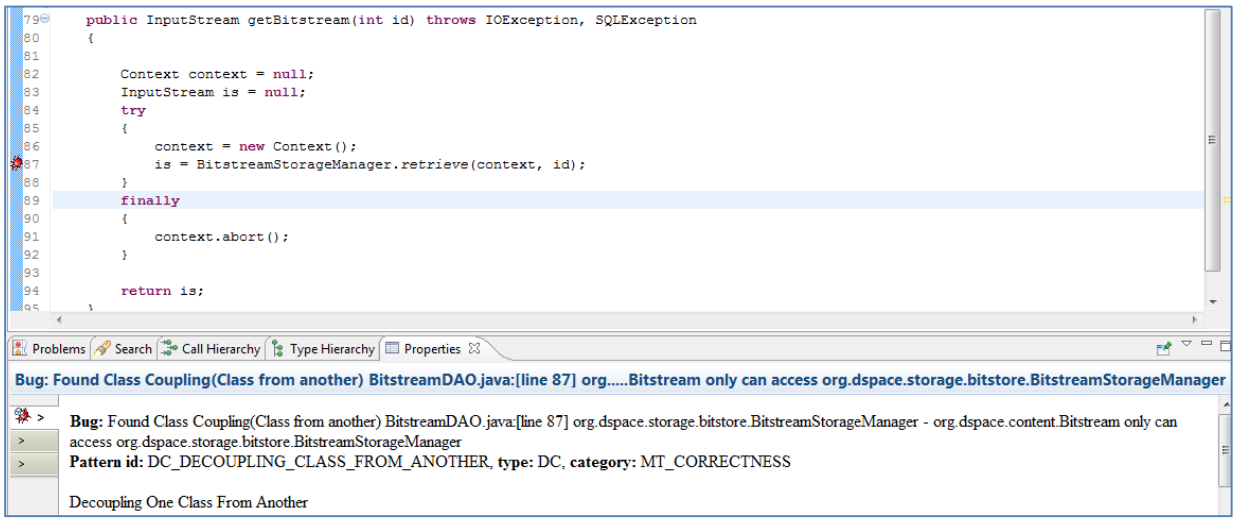

Figure B.1: Decoupling Classes violations report. It shows one violation in BrowseListTag class

DSpace Violations

```
W:/Robert Stuff/Thesis/DSpaceWS/dspacetest/dspace-api/org/dspace/checker/BitstreamDAO.java:87 Found Class Coupling(Class from another) BitstreamDAO.java:[line 87] org.dspace.storage.bitstore.BitstreamStorageManager - org.dspace.content.Bitstream only can access org.dspace.storage.bitstore.BitstreamStorageManager

W:/Robert Stuff/Thesis/DSpaceWS/dspacetest/dspace-api/org/dspace/storage/bitstore/Cleanup.java:109 Found Class Coupling(Class from another) Cleanup.java:[line 109] org.dspace.storage.bitstore.BitstreamStorageManager - org.dspace.content.Bitstream only can access org.dspace.storage.bitstore.BitstreamStorageManager

W:/Robert Stuff/Thesis/DSpaceWS/dspacetest/dspace-oai/dspace-oai-api/src/main/java/org/dspace/app/oai/DIDLCrosswalk.java:245 Found Class Coupling(Class from another) DIDLCrosswalk.java:[line 245] org.dspace.storage.bitstore.BitstreamStorageManager - org.dspace.content.Bitstream only can access org.dspace.storage.bitstore.BitstreamStorageManager

W:/Robert Stuff/Thesis/DSpaceWS/dspacetest/org.dspace.app.webui.jsptag/org/dspace/app/webui/jsptag/BrowseListTag.java:792 Found Class Coupling(Class from another) BrowseListTag.java:[line 792] org.dspace.storage.bitstore.BitstreamStorageManager - org.dspace.content.Bitstream only can access org.dspace.storage.bitstore.BitstreamStorageManager
```

```
W:/Robert
Stuff/Thesis/DSpaceWS/dspacetest/org.dspace.app.webui.jsptag/org/dspace/app/w
ebui/jsptag/ItemListTag.java:765 Found Class Coupling(Class from another)
ItemListTag.java:[line 765]
org.dspace.storage.bitstore.BitstreamStorageManager -
org.dspace.content.Bitstream only can access
org.dspace.storage.bitstore.BitstreamStorageManager
```

Table B.1: Tables showing Decoupling Classes violations from command line

Appendix C: Limiting Accesss to Mutators violation Report

Figures in this appendix show Bug Explorer perspective reporting the "offending lines" after running FindBugs to detect access to mutator violations:

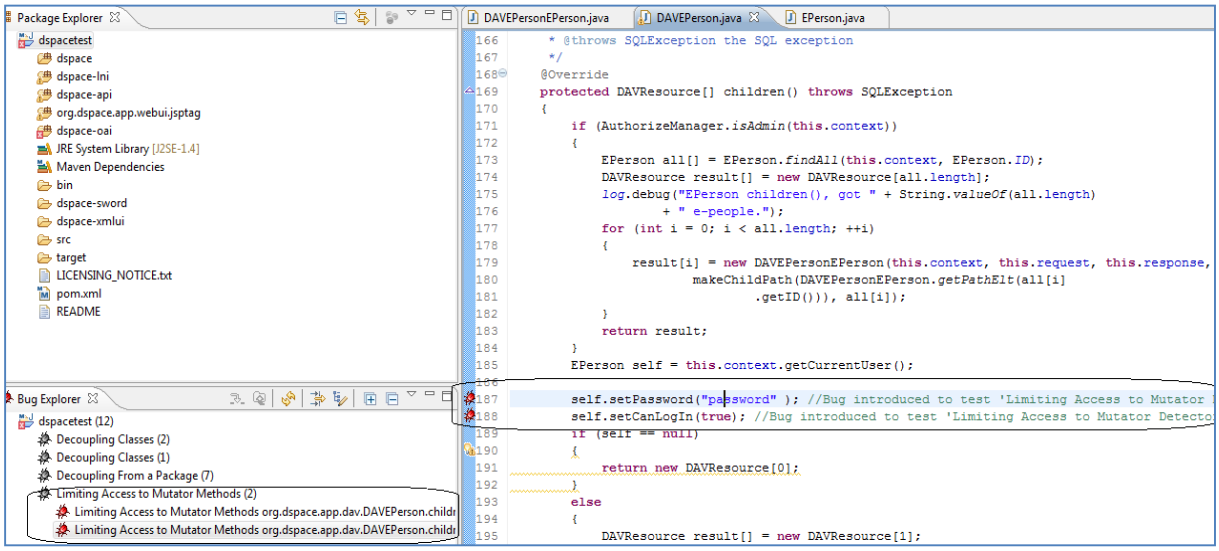

Figure C.1: Access to mutator violations report. It shows 2 violations in DAVEPerson class

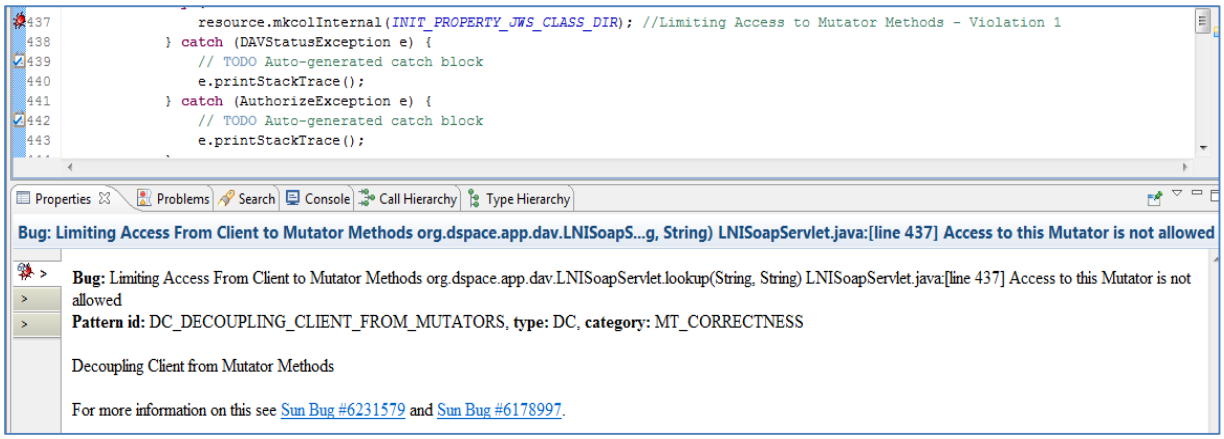

Figure C.2: Access to exception throw routines.

The figure above shows one violation detected accessing a method called "mkcolInternal()", which returns the following exception:

```
throw new DAVStatusException(HttpServletResponse.SC_METHOD_NOT_ALLOWED,
                             "MKCOL method not allowed for Lookup.");
```

Using a decoupling from mutator developers will get a warning message as shown in figure above.