Increasing Code Testability through Automated Refactoring

Final Year Project Final Report

Dermot Boyle

A thesis submitted in part fulfillment 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
29 April 2011
Abstract

Refactoring code to increase readability, maintainability, extensibility or to impose a standard style is an integral part of the software development process. Automated tools for refactoring address different issues but generally aim to deliver more readable, maintainable and extensible code. This study investigates the relationship between the ease or otherwise of test case construction relative to an improvement in cohesion values after refactoring using an automated standalone tool. The hypothesis is that automated refactoring which improves cohesion will lead to more testable code, from the point of view of the test case writer. The research includes a survey of volunteer software engineers who compared different versions of methods and classes and gave their opinions on differences and difficulties in the creation of unit tests against them. The results are inconclusive as regards the testability differences, but support the future conduct of a research project using a larger set of sample data and perhaps a more empirical measurement technique.
# Table of Contents

1. Introduction ........................................................................................................... 4
2. Testability & the measurement of testability ............................................................ 5
3. Refactoring ............................................................................................................... 7
4. Metrics and their application .................................................................................. 9
5. Code-Imp and how it refactors ............................................................................... 12
   5.1 Field-level Refactoring ..................................................................................... 13
   5.2 Method-level Refactoring .................................................................................. 13
   5.3 Class-level Refactoring .................................................................................... 13
6. The case study ......................................................................................................... 14
   6.1 The Plan ............................................................................................................. 14
   6.2 The sample application ...................................................................................... 15
   6.3 The Refactoring ................................................................................................ 16
   6.4 The Survey exercise format .............................................................................. 19
   6.5 The results ........................................................................................................ 21
7. Conclusions & Potential future work ..................................................................... 23
8. Acknowledgments ................................................................................................... 25
9. References ............................................................................................................... 25
10. Appendices ............................................................................................................ 28
    10.1 The raw refactoring output ............................................................................. 28
    10.2 The survey taken by the volunteer group ....................................................... 31
    10.3 The class code definitions from version “A” of the application - before refactoring.... 39
    10.4 The class code definitions from version “B” of the application - after refactoring has been applied ................................................................. 54
1 Introduction

This report documents research carried out on the feasibility of finding a reliable automated refactoring process for the improvement of the testability of program code. In general code refactoring attempts to improve some or multiple attributes of it. These attributes may be abstract concepts like “maintainability” and evaluated by the human eye or by feedback from maintenance coders. In this work the refactoring was based on the analysis of mathematical properties of the class hierarchy as measured using recognised metric formulae.

This work leverages from the existing large body of research into the accurate calculation of those metrics which attempt to measure the useful attributes of object-oriented code-bases. In the case study documented in this report software tool is used for automated code analysis and calculation of a specific code quality metric, as well for the refactoring of the code where the refactoring will improve the calculated metric value. Much of the available external research focuses on code analysis and calculation of metrics to evaluate various flavours of the cohesion or coupling attributes of code. Some of this work is in the area of applicability of different metrics to different code-bases depending on the domain. Perepletchikov, Ryan and Frampton in their 2007 paper [23] derived a new set of cohesion metrics for use when calculating the cohesion of service oriented architecture (SOA) systems. Here we look at general purpose metrics, created without a specific target code or application domain. In general software developers strive for high cohesion and loose coupling, as the accepted logic is that software with these attributes will be less buggy, more maintainable and ultimately more extensible. It is plausible then also that this same clean, well-structured code will allow the faster creation of simpler more robust unit tests, and that in fact the unit test code itself may be less buggy and perhaps also more cohesive.

This paper is concerned with cohesion metrics, and whether refactoring which improves cohesion also improves testability. The available formulae can be used to measure specific cohesion values along with any increases or decreases in these as a result of refactoring. There is a challenge in the measurement of testability as it is more a concept than a property. This is the main hypothesis for this paper: “Can we measurably improve the testability of program code through automated refactoring? Other researchers have used some of the properties of the code of a test case suite as measurements of complexity and equated that with testability (Badri, Badri and Toure [16]). The approach used for this study was to solicit the opinions of professional engineers. This report adds to the body of research supporting the equation of certain cohesion measurements with testability. It draws upon a case study where a code base was refactored to improve its cohesion properties, and the testability of that code was then assessed. It may seem obvious that small simple code elements will be easier to test than large complex ones. For a small method with few lines of code it will usually be quite a simple job to spot the inputs and decide on good and bad values to pass as tests. And conversely the creation of test code will be much more difficult if the method were to perform multiple unrelated tasks, or if it had redundant parameters which may cause initial confusion. Any refactoring which removes these unnecessary complexities and provides a cleaner code-base to the test case writer would seem a logical and even required step. It also seems like a step which might affect quite obvious differences between the two code-base versions. The automatic refactoring used in our case study did produce some dramatic changes to the code structure and class hierarchy, but it proved to be incorrect to assume that this would equate to a dramatically obvious increase in testability as assessed by our survey group. It transpired that changes which may facilitate the creation of better, simpler tests do not always immediately appear as such. Testability appears to be a more subtle property than others and code differences which improve it may appear as programmer styles on first reading. While there are certainly
basic rights and wrongs in the design of program code and logic, the design of execution flow allows the programmer a level of creative licence. However the structure of any suite of test cases will always be dependent on the program code against which it is written. So while the assumption that an increase in testability would be evident is a plausible one, there must be an allowance for this factor of “taste”; that one coder may just not favour some elements of another’s design and that this could have an effect on their perception of the ease of construction of test code for that design.

Based on the former assumption, in this study a group of volunteer software engineers were asked to give their opinions on different versions of classes and methods taken from an original code-base and a refactored version of it where the cohesion values had been improved through refactoring. There is a discussion of their responses in section 5 where the perceived differences or lack of differences in the ease of test case creation are also compared with opinions given on the code structural changes.

The rest of this paper is organized as follows: Section 2 discusses this elusive quantity; “testability” and it’s measurement. Section 3 discusses some of the available cohesion metrics and the metric used in the case study refactoring, with some explanation of its evolution and choice. In section 4 the Code-Imp automated analysis and refactoring tool is described as is its use. The case study itself is presented in Section 5 where there are some details of some of the more important refactoring changes obtained with Code-Imp and the small test application. Then Section 5 contains the results with some discussion or qualification of these and finally Section 6 wraps up this report with the authors findings and some thoughts or proposals for potential further work in this area.

2 Testability & the measurement of testability

This section looks briefly at the concept of testability, the factors which may affect it and the its measurement.

Bruce Lo and Haifeng Shi [3] in 1998 stated that “Testability measures the probability that potential faults in software will reveal themselves under software testing”. They argue that faults are detectable in code, whereas failures are the result of dynamic execution. Not all faults correspond to failures; faulty library code may never be called in an application. And vice versa; an external hardware or other issue can cause a failure which is not attributable to the code.

Robert V. Binder, in the 1994 work “Design for testability in object-oriented systems” [26] first published the “testability fish-bone” diagram, which he built upon what the ISO definition of testability called: “the attributes of software that bear on the effort needed to validate the software product” [10]. This fish-bone diagram is reproduced in Figure 1, with the major inputs being the documentation, the design implementation, the test suite itself, the test tools used and the process capability.

To define testability Binder had to recognise that there are more factors at play than just code design. This study is just about the effect that code structure can have on testability. The assumption is made that the other attributes on the fish-bones in Binder’s diagram [Figure 1] can be deemed to be either constant or negligible for different reasons. The testing criterion, i.e. the degree of validation or veracity required is irrelevant for the simple classes and methods tested in
this study as although there were no explicit guidelines given around testing criterion, the classes and methods were small and the requirement was understood - that all input states would be tested. The case study utilised the standard and well accepted junit automation tools and the application under test was our own small easily understood program; so the documentation and most of the test tools attributes are assumed to be out of scope. As regards the process capability area, because the survey group was a small group of volunteer professional software engineers then three of the sub-factors on this bone (commitment, effectiveness and staff capability) were assumed to be constant also, but the last sub-factor, the existence or otherwise of an integrated test strategy can be considered a potential variant owing to the different professional backgrounds in the group. This and the potential for variance in the related “test case design” attribute are discussed in more detail in the results analysis section.

But the research in this paper is really focussed on the implementation factors and the test suite - or more specifically we can say that this paper is concerned with effects of changes in the source code factors in one on the structure of the test cases in the other. Using Lo & Shi’s definition this study is about faults rather than failures. It is concerned solely with the testability of code, and specifically the ease of creation of the tests which will prove the validity of that code; and which will fail when the code fails.
We have equated the testability of the code with the simplicity or ease of the design of the process used to test it. Bruntink and van Deursen in their 2006 paper [17] also focused on the source code factors of Binder's fish-bone. They equated testability for object oriented code with metrics calculated from the test code. They identified two categories of source code factors, “test case generation factors” being factors which influence the no. of test cases required and “test case execution factors” related to the complexity of the test cases themselves. While noting that the testing criterion will have a bearing on the no. of test cases required, for the purposes of their study they also assume the test suite to be complete. They measure the factors as exhibited in the test code as measurements of the no. assert statements (no. of test cases) used against a class and as the count of lines of test code per class. They evaluated five systems and their test suites. Three of these had been developed using the same methodology and so were grouped and analysed together, so their results are shown against three systems. They found that the size factors in the source had a strong relationship with the test generation or size factors in the test suite. They found that the particular coupling metrics they used correlated well to their test suite complexity, measured through counting the lines of code per class and the no. of test cases (counting “assert” statements).

So we recognise the area of testability with which we are concerned to be related to Binder’s source code factors. The next question is around the identification and measurement of those specific factors which apply. Binder [25] did also identify a no. of metrics which he stated would affect some aspects of testability. He grouped them as Polymorphism, Inheritance and Encapsulation metrics and noted, for example that a high value for the LCOM metric (Lack of Cohesion of Methods) would indicate a lower level of testability, as it would mean that more states would need to be tested to prove the absence of side effects among methods. But his was more an observation than an attempt to prove the correlation between metrics and testability. There have been many studies since, like the Bruntink and van Deursen [17] work mentioned above where they investigated the use of software metrics as potential predictors of testability as evidenced through analysis of test suite properties. The study of software metrics and their accuracy at measuring properties for specific purposes is currently a very interesting and active field of study. It is plausible that research into the measurement of testability through source code analysis will drive further refinement and possible extension of the existing set of coupling, cohesion and other source code metrics.

3 Refactoring

It is accepted that the design or structure of code and the positioning of methods, classes etc... in a class hierarchy will have an effect on its maintainability. There is more than one aspect to this: there is the programmer’s style of coding, i.e. how the naming conventions used for variables or the placement of brackets, and there is the actual logical architecture which defines the execution flow. In corporate development houses a huge effort is often made to encourage adherence to coding standards and styles. Usually a casual assessment of code so forged especially if done so with readability and maintenance in mind will or should note a couple of characteristics; small short methods with short parameter lists and simple class definitions with logical groupings of related functionality and properties.
It transpires that there are official descriptions for these useful logical or architectural qualities which contribute to maintainability – cohesion and coupling. In general the desired qualities are high cohesion and low coupling. It is just common sense to plan for a maintainable code-base by trying to factor these qualities into the design? It is unlikely that there are many developers who do not strive to write highly cohesive code with the least occurrence of maintainability interdependencies. But there are many reasons why code cohesion may be lessened over time or why coupling between classes may be increased. Often it may have to do with schedules being squeezed at certain junctures in a project leading to rushed coding. This may happen also when features are tagged on quickly near the end of a project, or when bug fix patches or DOT releases with added functionality are handled by teams other than the original.

What we do know is that it certainly does happen; that what were once well structured, cohesive code-bases evolve over time to be much less so. If a product is likely to have a long shelf-life with on-going maintenance, then some scheduled attempts at assessing any loss in maintainability and returning the code-base to its formerly cohesive state will obviously help to reduce the cost of that maintenance. It is also possible that in a highly fluid development environment with frequent staff changes it may be quite tough to hold to standards and more frequent refactoring exercises may be required. Refactoring may be carried out to affect multiple source code properties. These may be stylistic or cosmetic to affect a uniform readability, but the more material refactoring exercises are to the code structure and usually have the aim of increasing cohesion and decreasing coupling, and as argued here, testability. There is obviously a plausible case for including refactoring phases in software projects. Martin Fowler, author of the influential book “Refactoring: Improving the Design of Existing Code” [8] defined refactoring as follows:

“A series of small steps, each of which changes the program’s internal structure without changing its external behaviour”

He calls out the need for solid tests to be in place before the refactoring is carried out, so as to validate afterwards that the external behaviour has not been changed. But if the refactoring causes dramatic enough changes to the code, then perhaps the original unit tests will no longer be completely valid. At the minimum, methods which were previously available for test in one class may have moved up or down the hierarchy to a more logical location and may at least require some refactoring of the test code also. So what is probably required is to ensure that the tests defined before the refactoring still pass afterwards, even if they themselves need some amount of refactoring to achieve this. Fowler [8] argues that refactoring should not be something for which a developer plans, or sets aside time, but which should be part of the on-going process, that almost every time you look at some code, whether to add functionality, fix a bug or just to understand it you could potentially perform some refactoring work. As such it is included as one of the main pillars of the XP “Extreme Programming” approach where pair programmers or the class user and the class designer review each other’s work and between them they apply refactoring changes. Murphy-Hill and Black [5], [5] use the phrase “Floss” refactoring to describe this frequent or habitual refactoring, done to maintain healthy code.

In keeping with the dental metaphor they also describe “root canal” [4], [5] refactoring which more closely describes the type undertaken in this project; where time is set aside for a refactoring job. The refactoring methodology used in this project is to analyse the whole class hierarchy at once for potential improvements, each improvement being one of Fowler’s [8] small steps.

Harry M. Sneed [9] in his work on reengineering noted that the size and complexity of software drives up test costs. He talks about refactoring to reduce complexity and said:
“Deeply nested code can be factored out into separate methods or procedures. This will not decrease the number of paths but it is easier to test smaller units than it is to test large, complex ones. So refactoring has a positive effect on testability. Besides it can be easily automated.” [9]

So while we certainly cannot reduce the specified or required complexity, we should aim to reduce the complexity which is not required. Sneed also states that:

“Experts claim that At least 33% of a system’s complexity is artificial. It is caused at the unit level by sloppy, unconsidered coding, at the component level by unnecessary and redundant functions and data, and at the system level by an over complicated architecture and overloaded user interfaces.”[9]

The importance of accurate refactoring to the industry, as evidenced by the body of work available on the subject, is testament to the simple fact that software code even if designed well, can and does grow less healthy throughout a products life cycle as bugs are fixed and features added. The potential cost-savings in reduced maintenance effort alone have often been justification enough for teams certainly to refactor habitually throughout the development process, but also to undertake “root canal” [4], [5] refactoring. There is also a cost associated with the development and maintenance of any test suite. If it can be shown that refactoring work which reduces unnecessary complexity in the system could simultaneously reduce the test creation and maintenance effort then the inclusion of a dedicated refactoring phase as standard in a development project may appear almost mandatory. At the very least, the test team may be added to the list of refactoring advocates.

Addendum to section

This author once worked as a contract or “journeyman” developer, moving from project to project and company to company. I remember well the importance of writing code which could be maintained or changed by others after I was gone. This would involve both adherence to the styles in force at the particular company and most importantly, more important than any documentation, being attentive to the readability of the code. And for me, this is where the two elements of style and logical design combine; leading me to paraphrase the old legal aphorism, “Not only must the code be executable; it must also be seen to be executable” [24]

4 Metrics and their application

So we can say that the goal of any real refactoring is to reduce complexity and size, thus increasing maintainability and therefore testability. Looking for known and identifiable issues or patterns and applying the accepted fixes for these is probably the safest way to do this automatically. But how do we measure the degree to which we may have changed the code? What metrics can we use prove the validity of a refactoring exercise. As we have said already, the process is one of reducing complexity in the individual code units and grouping only logically connected functionality, i.e. increasing cohesion and decreasing coupling. There are multiple metrics available to measure these properties to different levels of granularity

But there is a question over the existence of specific metrics that we can use to measure “testability”. Are there metrics which might help a test team to assess the scope of work involved in writing a unit test suite? In general, the cohesion of a class system is a function of the cohesion of its constituent parts; the classes and methods within those classes. So it does seem a reasonable assumption that changes which effect an improvement in cohesion are likely to lead to smaller
more specific classes and methods, and therefore smaller more specific test classes and tests. The research documented here looks at some of the metrics applied to the measurement of cohesion in an object-oriented code base, with the aim of finding one which reliably equates to testability as measured by the complexity of construction of unit tests.

There is a wealth of reference work available on the suitability of different metrics to the overall assessment of cohesion or the level of coupling in a code-base. A lot of the research work approaches the assessment of published metrics (or the creation of new ones) from the maintainability or extensibility angles. There is probably unanimous agreement on the attractiveness of the twin “grails” of high cohesion and, loose coupling. Development managers may sometimes seem to use these terms almost like a mantra, usually without conversing on the specifics of either (to the extent that one might sometimes wonder if they could actually explain them). Here we are concerned with the correlation between cohesion and testability. It will be easier to identify the requirements and write test cases for well organised code, but can we automatically organise badly organised code so that it reaches this point through cohesion analysis?

There is a smaller body of work which explores metrics specifically in relation to testability and some of the works referred to in this report use cohesion metrics and more specifically the “lack of cohesion” metrics to provide measurement which may correlate to testability. Bruce Lo and Haifeng Shi in 1998 [3], having observed that a high number of methods in a class is an indicators of lower testability, also studied the cohesion among methods in classes and stated “The lack of cohesion in methods results in lowering the testability of the class” [3]. Lo and Shi also looked at coupling and found that communication coupling through message passing can also have an adverse effect on testability, however the issue here is the length of the parameter list, which also effects cohesion values. Bruntink and van Deursen in their 2004 paper [17] did not find conclusive correlation between the values for the LCOM metric and testability for all of their test systems, and found that the coupling metrics they used actually provided for them a better indicator of the characteristics of the test suite required. But they did find the LCOM correlation was strong in one of the systems they analysed and that they felt they could explain its apparent lack of correlation in the other two systems in their study by way of anomalies they found in the relationship of the “Number of Fields” (NOF) metric to their measures of testability.

Badri, Badri and Toure [16] conducted a study in 2010 using two open sourced Java software applications for which suites of junit test cases existed. Their work [16] suggests a strong almost linear relationship between cohesion (measured as a lack of cohesion) and testability. The cohesion metrics they used were LCOM, LCOM* and LCD. They measured test case complexity in lines of code and counts of junit assert method calls. The Badri et al. paper was titled “Exploring Empirically the Relationship between Lack of Cohesion and Testability in Object-Oriented Systems” [16] and their research, as it explores potential correlation between lack of cohesion and code testability is very relevant to this project.

They managed to show that there is a “significant relationship between the lack of cohesion of classes and testability” [16]. Their research involved the analysis of test case suites which are available for two open source code-bases. They used the class based cohesion metrics, LCOM and LCOM* as measures of cohesion in the source code and then looked for correlation between these measurements and the test suite size. As inverse measures these metrics measure the lack of cohesion of methods. They do this by assessing the level of sharing of attributes between methods in classes. LCOM (Lack of Cohesion of Methods was first described by Chidamber & Kemerer [27] in 1994 while the LCOM* variation is set out in Henderson-Sellers 1996 paper [2].
Chidamber and Kemmerer’s original definition of LCOM [27]:

Consider a Class C, with methods \( M_1, M_2, \ldots, M_n \). Let \( \{I_i\} \) = set of instance variables used by method \( M_i \). There are \( n \) such sets \( \{I_i\}, \ldots, \{I_n\} \).

Let \( P = \{ (I_i, I_j) | I_i \cap I_j = \emptyset \} \) and \( Q = \{ (I_i, I_j) | I_i \cap I_j \neq \emptyset \} \). If all \( n \) sets \( \{I_i\}, \ldots, \{I_n\} \) are \( \emptyset \) then let \( P = \emptyset \)

\[
\text{LCOM} = \begin{cases} 
|P| - |Q|, & \text{if } |P| > |Q| \\
0, & \text{otherwise}
\end{cases}
\]

\( \text{LCOM} = \) The number of disjoint sets formed by the intersection of the \( n \) sets.

It is also very simply expressed in various other papers as being the count of the number of pairs of methods whose similarity is exactly zero.

Badri, Badri, Toure described it in their paper as:

“LCOM is defined as the number of pairs of methods in a class, having no common attributes, minus the number of pairs of methods having at least one common attribute” [16].

The other cohesion metric used in the Badri et al. study is a refinement of the LCOM metric proposed by Brian Henderson-Sellers in 1995 [2]. \( \text{LCOM}^* \) (also known as LCOM-HS or LCOM2) is a revised LCOM metric which normalises it for the total no. of methods and variables that are present in the class. So it measures cohesion as being proportional to the total number of variables that are referenced by the methods of the class.

The Henderson-Sellers definition [2] is:

\[
\text{Where the number of methods is } m \text{ and the number of instance variables} \ (\text{attributes}) \ \text{a set of } \{A_j\} \ (j=1, 2, \ldots, a). \ \text{Let } \mu(A_j) \text{ be the number of methods which access each datum.}
\]

\[
\text{LCOM}^* = \frac{\left( \frac{1}{a} \sum_{j=1}^{a} \mu(A_j) \right)^{m}}{1-m}
\]

The LCOM metrics have been critiqued, refined and extended in various scholarly articles since they were first published. Refined versions of the metrics have been made to take account of inherited attributes and methods and papers like Ezekiel Okike’s 2010 paper [6] show different levels of normalization can be applied to mitigate anomalies around the outlier classes. Hitz and Montazeri in 1996 [21] showed a number of weaknesses in the original LCOM metric and identified methods which scored well using Chidamber and Kemerer’s metric formula, but which displayed intuitive cohesion anomalies. They concluded that “Although it will most probably take much more time and effort until we have arrived at our goal, we are certain, that the metrics community is on the right way” [21].

A more recent body of work by Al Dallal and Briand (from 2010) [11] concentrates on the suitability of metrics to the support of refactoring work. Their paper “A Precise Method-Method
Interaction-Based Cohesion Metric for Object-Oriented Classes” [11] introduces a new low-level design class cohesion metric, “LSCC”. Part of their work involved a refactoring case study where they artificially moved methods so as to reduce Method-Method Interactions (MMI) cohesion in an open source, code-base which they knew to be regarded as well-structured and cohesive (JHotDraw 2010 [12]). Applying the LSCC metric allowed them to detect all of the artificial method moves, and this was concluded as being empirical evidence that LSCC is an appropriate cohesion metric to guide refactoring. They also conducted a comprehensive study into the accuracy of LSCC and ranked it against various other MMI cohesion metrics including a number of the LCOM variations (LCOM1-4). The data case study results in the paper support their claims that:

“LSCC is based on a precise MMI definition that satisfies widely accepted class cohesion properties and is useful as an indicator for restructuring weakly cohesive classes”. [11]

They demonstrate LSCC as a more mathematically “complete” metric when tested against the four mathematical properties of class cohesion metrics as defined by Briand et al. (1998). [15] They measure MMI as a “method attribute reference” (MAR), constructing a matrix of methods and attributes and measuring the average cohesion of all pairs of methods using the formula (for a class C consisting of “k” methods and “l” attributes):

\[
LSCC = \sum_{i=0}^{k} \sum_{j=i+1}^{k-1} \sum_{l=1}^{l} ns(i,j)
\]

This paper’s hypothesis required the programmatic refactoring of a code base to allow the evaluation of any increase or decrease in testability (as measured by the complexity required in test case design). LSCC was chosen as a modern, more evolved cohesion metric and because it’s designers had the purpose of refactoring in mind. So it was deemed to give a greater chance of effecting more dramatic or obvious differences between the two versions of the code-base. A number of arbitrary trials were also run initially on the sample code base using the other metric algorithms which are available in Code-Imp. Refactoring based on LSCC, especially when the inheritance hierarchy was included produced the greater resultant no. of changes.

5 Code-Imp and how it refactors

The latest version of the Code-Imp tool as provided by Iman Hemati Moghadam builds on the original work on this tool carried out by O’Keeffe and Ó Cinnéide [20]. It uses the RECODER framework [28] to analyse java code and to apply transformations to these sources. It calculates numeric score values for different cohesion and coupling metrics and refactors the code in a given java class hierarchy with the aim of improving the score for the given metric. To identify starting points for refactoring Code-Imp applies search techniques and in this regard it is described as a search-based refactoring tool [20]. Search-based software engineering has been defined as the application of search-based approaches in solving optimisation problems in software engineering. [18][20].
The list of recognised refactoring’s as originally defined by Martin Fowler [8] has been updated and extended over the years so that a recent glance at his web site shows a catalogue of 93 different “moves” which have supporting information [25].

Code-Imp has implemented a good subset of these and at the time of use for this study had a fairly comprehensive implementation of cohesion metrics. Many these could be viewed as what Simon, Steinbrückner, and Lewerentz in their 2001 paper [7] referred to as “Distance based cohesion”. They spoke of the principle of “Put together what belongs together”.

Code-Imp applies refactoring changes at the field, method and class level as follows:

5.1 Field-level Refactoring

“Pull Up” and “Push Down” - Simon, Steinbrückner, and Lewerentz [7] called this “Move Attribute” refactoring. If it can be ascertained programatically that a field is declared at the wrong level in a class hierarchy, then Code-Imp can move this field up and/or down the hierarchy. For example, if a field is declared in one superclass, not referenced by any of their immediate subclasses, but by a subclass further down the hierarchy then pushing the field down the hierarchy may increase cohesion by removing it from classes which do not reference it. So a “Pull Up” refactoring moves a field from a superclass to one or more subclasses that require or reference it.

Increase and Decrease Field Security - This refactoring change a fields access modifier; so increasing field security would change public to protected, while applying a decrease may change a default field or one with no access modifier specified to a protected field.

5.2 Method-level Refactoring

“Pull Up” and “Push Down” - This was described by Simon, Steinbrückner, and Lewerentz [7] as “Move Method” refactoring. Similar to the “Pull Up” and “Push Down” refactorings for fields, these implementations move methods up or down the class hierarchy. So a method can be moved closer to where it is called.

Increase and Decrease Method Security - The access modifier of a method can also be changed. Similar to the access to fields, increasing method security might change public to protected, and then applying a decrease may change it back again.

5.3 Class-level Refactoring

While the field and method refactoring’s described are simple and straightforward enough, class level refactoring at first seems like a daunting concept – this is where we may actually change the actual class hierarchy architecture. Can we really automate this? The answer is yes; as with the field and method level we just need to be sure that we are not changing any actual logic, and that all potential execution paths for the accessible methods remain the same.

The class level refactoring’s currently available in Cod-Imp are:

Collapse Hierarchy - In this type of refactoring we remove a class from an inheritance hierarchy. This is usually done if a superclass and its subclass do not appear to be very different. The “Collapse Hierarchy” refactoring effectively merges the classes.
Make Superclass Abstract - In cases where the super class is never explicitly instantiated, and has no constructor the refactoring declares it to be abstract. This can enable further refactoring later on where common fields or methods in the subclasses can then be “pulled up” to the super abstract class.

Make Superclass Concrete - This is the opposite of “Make Superclass Abstract”, in that it removes the explicit abstract declaration of an abstract class which does not have abstract methods.

Replace Inheritance with Delegation - Replaces an inheritance relationship between two classes with a delegation relationship; the former subclass will have a field of the type of the former superclass.

Replace Delegation with Inheritance - The reverse of the previous refactoring; the class which was referenced as a field is converted into a superclass and the original class then inherits from it.

Code-Imp has over 20 coupling and cohesion metric algorithms available and is constantly being extended. The method of use is to target specific metrics, either purely for evaluation or for refactoring. It then applies search-based refactoring as an iterative process, so it initially assesses the code base and calculates the metric, then looks for possible refactoring’s. At each stage it tests to see if any given refactoring will improve the value for the targeted metric and if it does, applies it to the code. This process is then continued until it can find no refactoring’s worth applying, or which could have a positive effect on the metric value.

6 The case study

6.1 The Plan
The basis for this work is the hypothesis that the testability of an application can be increased through automatic cohesion-driven refactoring. “Code-Imp” was adopted as the refactoring platform. It performs refactoring by repeated analysis of the metric values in the classes of the code base, specifically in our case the values for LSCC. For our case study the specific search technique adopted was first-ascent hill-climbing, described by O’Keeffe and Ó Cinnéide as

“A local search algorithm where the search examines neighbouring solutions until a higher quality solution is discovered. This neighbour then becomes the current solution” [20].

So the process starts by identifying random refactorings which improve the LSCC value and finishes refactoring when it can find no further changes to make which will improve it.

The project undertaken by Badri, Badri and Toure (2010) [16] assessed testability by analysis of existing test case suites and then correlated that data with an analysis of the cohesion values of the tested code base. Their hypothesis was basically that a more complex test suite would probably relate to less cohesive classes in the product code, and their results strongly support this.
This study, while related has a very different approach to testability assessment. As one of the code bases in question has been created by the refactoring in the case study itself the Badri et al. approach was not deemed as practical. Specifically there were not available two sets of unit tests for the two versions of the code base (before and after refactoring). If there had been then this study would have been able to concentrate on their differences and a correlation with the properties of their respective code bases. In their absence then, to assess the increase the differences in testability between the two versions, the help was requested of a panel of volunteers, all professional software engineers with varying experience levels of unit test creation. They were simply asked to write test cases for a sample set of methods both in their original and in their refactored versions and to then comment on the ease of this task for each. It is the correlation of these results which informs the analysis, conclusions and recommendations for further work.

6.2 The sample application

The java application used for this case study is a small program with some basic input, storage and retrieval functionality. It has a small no. of classes, 14 in all in the original non-refactored version including the basic UI (Windows Forms) classes. When run, it prompts the user for input by way of dialog boxes. The user can add some basic records with information about three types of people; teachers, students and managers. Apart from data input the only other functionality available is the serialization of these records to a simple text file, and the subsequent viewing of the saved records. The screens are very basic and simple, and the idea was that it would take a minimal effort on the volunteers part to get to grips with the application and it’s code structure and therefore allow them to form opinion on the class design and its effect on the ease of test case creation.

The standard input screen accepts details for a student and their enrolled courses as seen in the following screenshot:

![Figure 2.](image)

The original version has serious cohesion issues, and the fact that the design was not optimal is obvious from a glance at the hierarchy in UML (generated using AltoNova UModel 2011 [29]). A lot of the cohesion deficiencies are then detectable fairly quickly upon a cursory inspection of the class code and the placement of fields in the hierarchy. In Figure 3Figure 2 below, it can be seen that the class relationships certainly appear more complex than the described functionality would imply.
6.3 The Refactoring

First ascent hill-climbing can give a different result each time, so Code-Imp was run on the original version of the code-base 6 times, and the run that produced the greatest number of refactoring changes. This run produced 26 incremental refactoring changes in the code. While the majority of these were of the “Move Field” or “Move Method” variety, almost a third of the changes involved more structural refactoring; i.e. Extracting or Collapsing of part of the hierarchy – 7 Extracting and 1 Collapsing.

The full breakdown is:
7 x “Extract Hierarchy”, 1 x Collapse Hierarchy
3 x “Pull Up fields”, 5 x “Push Down” fields
3 x “Pull Up” methods, 2 x “Push Down” methods
3 x “Decrease Security”, 1 x “Increase Security”

For example, in the “before” version (or version “A”), the “ManagerForm_A” class had a method called “InputValidated()”. The automated refactoring process pulled this method up to the “SuperForm” class, which is a more logical hierarchical position. The Boolean field “isStoring” is also pulled up to the superclass, (“SuperForm”) in the same fashion. One of the more structural changes was the introduction of an interface class “InterIndustrialistPerson_B”. This interface was then implemented by the “B” version of the “Person” class, so the class “Person_B” then

Figure 3. – Version “A”
implement the methods of the interface. The interface takes care of these name getter and setter method signatures and removes the need for a tester to trace through the full inheritance hierarchy to understand and test this functionality. The interface code itself is concise:

```java
public interface InterIndustrialistPerson_B {
    public void setFirstName(String str);
    public void setLastName(String str);
    public String getFirstName();
    public String getLastName();
}
```

An interesting and somewhat surprising result was that even though the test application was very small and contained a very limited amount of functional code, the refactoring created more classes, albeit smaller and more cohesive ones. So where initially we had a simple hierarchy of:

**Student->Academic->Person**

Version B now has a longer inheritance path of:

**Student_B->Collegiate_B->Academic_B->Trainee_B->Person_B**

Some of the classes themselves then are extremely small and similar, if we look at Collegiate_B:

```java
public class Collegiate_B extends Academic_B {
    protected Collegiate_B(String fName, String lName)
    {
        super(fName, lName);
    }
}
```

And “Academic_B”

```java
public class Academic_B extends Trainee_B {
    public Academic_B(String fName, String lName)
    {
        super(fName, lName);
    }
}
```

We see two small, simple, very similar classes. So overall the refactoring resulted in a larger set of simpler classes. If we generate a UML diagram for the refactored version it initially looks quite different from the original. But by comparing the two we can recognise the class structure and the more logical relationships in the later diagram – see Figure 4.
Because of the nature of the First Ascent Hill Climbing algorithm used to select potential refactoring changes, none of the refactoring changes are final until the process is complete. So it is not unusual to see a change introduced early in the process being cancelled by a later refactoring, and even being applied then for a second time later on again. In our case, the class “Coordinator_B” is introduced between the classes “Industrialist_B” and “Manager_B” at step 2; so that “Manager_B” inherits from “Coordinator_B” which in turn inherits from “Industrialist”. However this change is reversed at step 21, only to be implemented afresh at step 23. This may at first seem like an illogical redundancy, but it must be remembered that the process is incremental and that at each of these steps, the metric value has improved thanks to its implementation. Similarly, the field “isStoring” is subject to a “Push Down” refactoring at step 4, but is then subjected to a “Pull Up” refactoring at steps 17 and 18, and by then the other refactoring changes have made this a logical move to improve the LSCC metric value. The actual full output table from the Code-Imp run is included in the appendix, but the sequence of changes was as follows:
6.4 The Survey exercise format

The exercise undertaken by the volunteer group of software engineers involved presenting both versions of the sample application code with no bias towards either. The goal was to have them assess testability and nothing else. To this end there was no mention of phrases like “refactoring”, “cohesion” or “robustness”. Nor was there any allusion to either version being an earlier or later version or of one being a refinement of the other. They were simply labelled versions “A” and “B” and the only discernable difference (aside from the code structure) was that the classes in version “A” had the “_A” appended to their names for clarity and ease of comparison between the two version should they be viewed side by side; so the case study code contained classes “Industrialist_A”, “Industrialist_B” etc… Six short exercises were set, each relating to a piece of “functionality” (some methods, some whole classes) which existed different formats in both versions. The volunteers were asked to write test cases for both versions and report on the comparative ease of doing so, by answering the following question with this rating scale for each exercise:

<table>
<thead>
<tr>
<th>Step</th>
<th>Refactoring</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Replace Inheritance with Delegation</td>
<td>“Person” becomes a field of “Industrialist”</td>
</tr>
<tr>
<td>2</td>
<td>Extract Hierarchy</td>
<td>“Coordinator” is Added as “Industrialist’s” child to hierarchy structure</td>
</tr>
<tr>
<td>3</td>
<td>PullUp Method</td>
<td>“getTextValue” is added to “Academic”</td>
</tr>
<tr>
<td>4</td>
<td>PushDown Field</td>
<td>“isStoring” is added to some of “SuperForm’s ” subclasses</td>
</tr>
<tr>
<td>5</td>
<td>Decrease Security: Field</td>
<td>The security of field “firstName” is decreased</td>
</tr>
<tr>
<td>6</td>
<td>PullUp Field</td>
<td>“college” is added to Person</td>
</tr>
<tr>
<td>7</td>
<td>PushDown Method</td>
<td>“setCollege” is added to some of “Academic’s” subclasses</td>
</tr>
<tr>
<td>8</td>
<td>PushDown Method</td>
<td>“getCollege” is added to some of “Academic’s” subclasses</td>
</tr>
<tr>
<td>9</td>
<td>Extract Hierarchy</td>
<td>“TeacherInputForm” is Added as “NewSubjectForm’s” child to hierarchy structure</td>
</tr>
<tr>
<td>10</td>
<td>PullUp Method</td>
<td>“InputValidated” is added to “SuperForm”</td>
</tr>
<tr>
<td>11</td>
<td>Extract Hierarchy</td>
<td>“SubjectChoiceForm” is Added as “SubjectForm’s” child to hierarchy structure</td>
</tr>
<tr>
<td>12</td>
<td>Increase Security: Field</td>
<td>The security of field “companyEmployer” is increased</td>
</tr>
<tr>
<td>13</td>
<td>Extract Hierarchy</td>
<td>SuperFormManagerForm” is Added as “SuperForm’s” child to hierarchy structure</td>
</tr>
<tr>
<td>14</td>
<td>Extract Hierarchy</td>
<td>“Trainee” is Added as “Person’s” child to hierarchy structure</td>
</tr>
<tr>
<td>15</td>
<td>PushDown Field</td>
<td>“college” is added to some of “Person’s ” subclasses</td>
</tr>
<tr>
<td>16</td>
<td>Extract Hierarchy</td>
<td>“Collegiate” is Added as “Academic’s” child to hierarchy structure</td>
</tr>
<tr>
<td>17</td>
<td>PullUp Field</td>
<td>“isStoring” is added to SuperFormManagerForm</td>
</tr>
<tr>
<td>18</td>
<td>PullUp Field</td>
<td>“isStoring” is added to SuperForm</td>
</tr>
<tr>
<td>19</td>
<td>PushDown Field</td>
<td>“college” is added to some of “Trainee’s” subclasses</td>
</tr>
<tr>
<td>20</td>
<td>Decrease Security: Field</td>
<td>The security of field “subjects” is decreased</td>
</tr>
<tr>
<td>21</td>
<td>Collapse Hierarchy</td>
<td>“Coordinator” is removed from the program scope</td>
</tr>
<tr>
<td>22</td>
<td>Decrease Security: Field</td>
<td>The security of field “_InterIndustrialistPerson” is decreased</td>
</tr>
<tr>
<td>23</td>
<td>Extract Hierarchy</td>
<td>“Coordinator” is Added as “Industrialist’s” child to hierarchy structure</td>
</tr>
<tr>
<td>24</td>
<td>PullUp Method</td>
<td>“getTextValue” is added to “Trainee’s”</td>
</tr>
<tr>
<td>25</td>
<td>PushDown Field</td>
<td>“college” is added to some of “Academic’s ” subclasses</td>
</tr>
<tr>
<td>26</td>
<td>PushDown Field</td>
<td>“college” is added to some of “Collegiate’s” subclasses</td>
</tr>
</tbody>
</table>

Figure 5.
“Which version would be easier to write test cases for (please tick 1)?

- Version A is much easier to test.
- Version A is moderately easier to test.
- Version A is slightly easier to test.
- Both are the same / I have no opinion.
- Version B is slightly easier to test.
- Version B is moderately easier to test.
- Version B is much easier to test.”

The full set of exercise questions are included as an appendix to this paper, but for example, two of the questions were:

“*In version A, the class Industrialist_A is a subclass of Person_A. In Version B, this inheritance relationship does not exist, but Person_B and Industrialist_B both implement the interface InterIndustrialistPerson_B.*”

“*Write one test case to test the constructor for the class Industrialist_A in version A, and one to test the constructor for the class Industrialist_B in version B.*“

The first two exercises asked the group to write unit tests for methods where the refactoring had introduced an interface where there had been an inheritance relationship. The expectation was that delegation would be preferred over inheritance and that the responses would strongly favour version B. Exercise 3 involved a method which had been “pulled up” the hierarchy and asked which position made it easier to test, or if this had any bearing. In Exercise 4 the code differences involve a delegation relationship with a concrete class being changed to one of an appropriate interface type. Exercise 5 involved a comparison between a method which had existed in a larger more complex class, but which had been moved up the hierarchy in version B to a simple clean class. The last exercise then invited comparison between the testability of the two version of the Industrialist class, where an inheritance relationship had been replaced by an interface implementation, with the expectation that the abstraction of the commonalities between class to the common interface would be preferred.

As mentioned previously the survey group consisted of 10 volunteers (actually 14 originally volunteered, but 10 actually responded to the survey) who were all working as professional software engineers. The survey preliminary asked some question to gauge the experience levels in the specific areas of software development and Unit testing and the averages worked out as:

Years in the software industry: 9
Years as a software developer: 9
Years using automated unit testing: 3

These questions were asked to establish that the survey group could reasonably be expected to quickly furnish a set of unit tests for the exercise code and could therefore give a useful opinion on whether or not the class or method structure of either version lent itself better to this task. The average reported time to complete the full set of 6 exercises was 40 minutes.
6.5 The results

<table>
<thead>
<tr>
<th>Respondents</th>
<th>Resp 1</th>
<th>Resp 2</th>
<th>Resp 3</th>
<th>Resp 4</th>
<th>Resp 5</th>
<th>Resp 6</th>
<th>Resp 7</th>
<th>Resp 8</th>
<th>Resp 9</th>
<th>Resp 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exercise 1</td>
<td>No Diff</td>
<td>A(Slight)</td>
<td>No Diff</td>
<td>B(Slight)</td>
<td>A(Slight)</td>
<td>No Diff</td>
<td>No Diff</td>
<td>B(Slight)</td>
<td>No Diff</td>
<td>B(Much)</td>
</tr>
<tr>
<td>Exercise 2</td>
<td>No Diff</td>
<td>A(Mod)</td>
<td>No Diff</td>
<td>No Diff</td>
<td>No Diff</td>
<td>No Diff</td>
<td>No Diff</td>
<td>B(Slight)</td>
<td>No Diff</td>
<td>B(Mod)</td>
</tr>
<tr>
<td>Exercise 3</td>
<td>No Diff</td>
<td>No Diff</td>
<td>No Diff</td>
<td>A(Much)</td>
<td>B(Slight)</td>
<td>No Diff</td>
<td>No Diff</td>
<td>B(Slight)</td>
<td>No Diff</td>
<td>B(Mod)</td>
</tr>
<tr>
<td>Exercise 4</td>
<td>A(Slight)</td>
<td>A(Mod)</td>
<td>No Diff</td>
<td>A(Much)</td>
<td>A(Slight)</td>
<td>No Diff</td>
<td>No Diff</td>
<td>A(Slight)</td>
<td>No Diff</td>
<td>B(Much)</td>
</tr>
<tr>
<td>Exercise 5</td>
<td>No Diff</td>
<td>No Diff</td>
<td>No Diff</td>
<td>B(Much)</td>
<td>No Diff</td>
<td>B(Much)</td>
<td>No Diff</td>
<td>N/A</td>
<td>No Diff</td>
<td>B(Much)</td>
</tr>
<tr>
<td>Exercise 6</td>
<td>A(Much)</td>
<td>A(Slight)</td>
<td>No Diff</td>
<td>A(Slight)</td>
<td>A(Mod)</td>
<td>A(Mod)</td>
<td>No Diff</td>
<td>N/A</td>
<td>A(Much)</td>
<td>A(Much)</td>
</tr>
</tbody>
</table>

Figure 6 shows the tabulated set of responses. The evaluation of the responses results in a marginal favouring of Program “A” from a testability point of view, i.e. a greater number of those volunteers who indicated preferences actually found it easier to write test cases for the original version of the code and more difficult to test the refactored version.

However some of the comments from respondents allude to the simplistic nature of the test program and how the introduction of an interface in Program B would help in a larger system. Some of the feedback justified the respondents “No Difference” responses based on the small size and the lack of complexity in the test code base. Some of these comments actually referred to the B versions of the code as being better structured, but justified the “No Difference” comment as the creation of test cases for both versions was a relatively trivial task. One respondent to exercise 4 commented that “Using an interface to abstract away from the type of person, makes no real difference to such a small test case”. This seems to have been a factor particularly in the samples where the InterIndustrialistPerson interface was introduced in the refactored version. As stated there were no references to robustness or cohesion in the survey paper, or any comments that could be construed as “leading” and it certainly appears that even when respondents obviously preferred the refactored version from a software design point of view, they did make any link between this and the ease of test creation for either design. But it does suggest that in a real industrial project they would at least be happier working with code which displayed the design attributes of version “B”. It is certainly plausible that had the test program been a larger more complex entity the design preferences may also have become testability preferences. Some of the comments suggested that the (automatic) introduction of extra classes into the hierarchy in the refactored version may have slowed down the writing of test cases as it may have taken slightly longer to comprehend the elongated hierarchy. However when students commented just on the complexity of the individual methods they were testing they agreed that the re-factored versions were easier to test. The introduction of the interface in version B also led to comments suggesting it took slightly longer to comprehend the impact of this and that each implementation of the interface potentially would require a new test case – however one comment pointed out that the use of parameterized test cases for the interface methods would make this a more beneficial refactoring.

The following are a sample of the comments from responses where the answer was either in favour of the non-refactored “A” version or “No Difference”:

"Using an interface to abstract away from the type of person makes no real difference to such a small test case."

Increasing Code Testability through Automated Refactoring

Dermot Boyle
“Modern IDE’s would hide the relationship complexities by showing all available methods and attributes regardless of relationship.”

“Because it takes just a little longer to understand version B (how does implementing the interface work for the two classes in question?)”

It is certainly possible that the use of the interface in the refactored version caused a little confusion as can be seen from the variance of the responses to Exercise 6. If there were unified opinion in favour of the use and testability of interfaces that there may have been unanimous verdict in favour of either version “A” or version “B” for exercise 6.

One interesting anomaly involves the method getTextValue() which started life in version A in the subclass Student, but finished after refactoring in the Trainee class in version B. Exercise 5 asked the volunteers to compare testing the method in each version. This particular method is not actually called or referenced by any other code in the application, so as one might expect the majority of the responses (6) are “No Difference”. One respondent gave no answer in this case, but three did respond with opinions. Two favoured version B; one said it was slightly easier to test and one said it was much easier to test. One respondent favoured version A, stating that it was much easier to write a test case for it. Their comments suggest that their opinions were based on personal preferences when it comes to reading and understanding code. The respondent who favoured version A commented that they did so because having detected that the method was not used anywhere in Version A, they found it took longer to verify that in Version B due to the method being much higher up in the class hierarchy. The comments on exercise 5 from the two who favoured version B support the argument for an increase in testability in that they refer to the simplicity of the class which contains the method, indicating that the refactoring has speeded up the unit test creation process.

There were also two respondents who saw no difference at all in the difficulty of writing unit tests between the two versions for any of the exercise samples. One of these used the exact same comment for all exercises; stating that the same test case, “with minor changes” could test either version. The other respondent commented for a couple of exercise’s to the effect that although they had answered “No Difference” they recognised that version B was potentially more easily tested, but that the simplicity of the code under test made the point moot in their opinion.

However the wide variance in the responses as regards testability (or ease of unit testing) for the majority of the 6 exercises does suggest that either the method and class versions presented were not explicitly different enough to provoke a strong reaction from the unit test writer, or perhaps that there are distinct personal tastes in play when an individual assesses a piece of code and attempts to bullet-proof it with another piece of code. The latter hypothesis is certainly outside the scope of this report, and probably not one which could easily be scientifically tested without at least some detailed research into developers and their practices. In computer science program code is a measureable, provable artefact and the very existence of the metrics used here are part of that measurement. Predictability is of course a requirement of production code and perhaps ironically this is a key justification for unit testing. So it would seem reasonable that applying the same test unit design methodology to the same application code should produce roughly similar test code. But we do not know if the respondents in the volunteer group follow the same practices when designing unit tests. Could a group of software engineers who have graduated from different colleges and work in different corporate environments have a shared approach to
something like unit test creation? While this is not impossible (and it is certainly desirable that they share an understanding of the more fundamental aspects of programming) it is of course not expected, or even likely. Although it is a core process in the software development life cycle, and they may even possibly have taken similar courses in this area, in their jobs they will probably have adopted the standards and practises dictated by their employers. An interesting property of the result set in Figure 6 is the clustering of A or B preferences, that is to say that it appears that some respondents were just somehow more disposed to answer in favour of A or B; that perhaps that design of one or the other somehow suited them more. However, over 50% of the responses given indicated that there was no effect on the ease of test case creation, indicating that most of the time the differences between the versions (as regards test case creation) were not dramatic enough to provoke a “hard” reaction.

7 Conclusions & Potential future work

So overall this case study delivered an inconclusive result. However it has certainly not produced any strong evidence against the case for the feasibility of testability improvement through automated cohesion based refactoring. The cohesion properties were improved through the refactoring and it is probably a likely as not that, given a different set of volunteers, the results may have agreed with the hypothesis. So the hypothesis is still a valid, but unproven one.

In general the opinions expressed tend to suggest that this particular refactoring exercise did not massively improve (or reduce) the testability. It is possible that the sample program and the case study exercise may not have been of an adequate scope. It is evident also that the measurements gathered, based on opinions of highly skilled, but also individually diverse professionals was not an ideally method of evaluation. If the volunteers were all veterans of the same recent project and could be reliably predicted to approach the exercises in the same fashion then the results may or may not have been more conclusive, but they may certainly have been more consistent. Ideally we would find a way of empirically proving testability, even of assessing testability, and therefore any increase or decrease in same programmatically. As mentioned in the metrics section, there are other studies into the relationship between specific software metrics and the measurement of testability. The question is how we can measure testability, especially in the absence of a test case suite on which to perform analysis. Even then, the quality of the design of the test suite itself will be a factor in the properties derived from any such analysis. Perhaps the ideal case study would involve two test suites, one for the original and one for the refactored versions of the code base. But the study would need to mitigate against any outlying factors which could potentially skew the results. How can we be confident that two test suites are equal relevant to two different versions of a code base? Equally how can we be sure that we have employed the correct software metric(s) on which to base our refactoring in the first place? And then how do we measure the test suites complexity?

Badri et al. in their 2010 paper [16] used two open source Java software systems (ANT [1] and JFREECHART [12]) to look for a relationship between a lack of cohesion and low testability characteristics. They measured the testability of these systems by measuring the complexity of the existing test cases and they found evidence of a relationship between cohesion metrics like LCOM and the testability of a class structure, in that they could show LCOM values as predictors for things like the lines of code count (LOC) of test classes. The versions of these open source systems which they tested had over 700 classes, with between 5000 and 8000 methods. To run an exercise like ours on a data set like that would not be feasible without a much elongated timescale (and it would probably need to be a funded project).
It is certainly plausible that with further work on larger examples it could be possible to show that automated refactoring can reduce the complexity of the required test suite for an application’s code-base. It does seem that the respondents in this case did recognise an improvement in the refactored code’s design. Survey respondents who had no preference as regards testability did not comment in favour of version A”, but there were comments favouring version “B” from such respondents. Of the 45% of answers which actually showed a preference, nearly half (22%) favoured version “B”, but if we add to these the neutral responses which had comments favouring the design of version “B” there would then appear to be a majority of the respondents who showed a preference either through the testability questions or through their comments showing a preference for the design in the refactored version, if not for its testability. In any case it has been shown by O’Keeffe and Ó Cinnéide that automated refactoring can and does improve the quality and structure of code in their papers on the use of the refactoring tools “Dearthóir” [19] and Code-Imp [20].

One interesting further study might be to determine which cohesion metric will give the best refactoring from a testability perspective or if a no. of refactoring exercises ought to be carried out in a certain order. The choice of LSCC as a metric on which to base the refactoring work referred to in this report was not a scientific proof of “the best class cohesion metric for refactoring”. Rather it was based on informed opinion derived from other research work along with the observance of the size of the effect of refactoring with Code-Imp using other cohesion metrics. The refactoring changes made based on the LSCC metric did certainly simplify the classes and would seem to have increased readability (and by inference maintainability). However, the other refactoring runs done with the sample code-base using other cohesion metrics did produce a subset of the same changes, so it is of course possible that LSCC is the optimum choice. But it is a project proposal nonetheless, to do a comparison of the effects of a set a set of metrics applied to a code-base using Code-Imp or a similar tool and to search for an optimum set and order of metrics, perhaps something a Pareto-optimal algorithm.

But the question of the measurability of improvements or otherwise of testability remains largely unanswered, and based on the experience with this project, and the collation of opinions as measured results it might be preferable in a future work to experiment with more other empirical measurements such as the approach used by Badri et al. [16] where they equated the testability of code with the complexity of the test code. This would of course require the availability of two test suites, one for each version of the program under test. There would be a challenge in how to design these so that there are no outlying complexity factors introduced by two different approaches to their design. It may be possible that by taking the test suite for the original program, and making only the minimum changes to the test code structure so that as far as is possible the same tests are re-used (they are testing the same functionality) we may only be manually refactoring the test suite to align with the refactored program code. However this may be an impossible task and there may be some research required in this area too before we could be confident in any test suite comparison.

It could be a very interesting initial exercise to refactor one of the open source systems used by Badri, et al. [16] using Code-Imp driven with the LSCC metric, and to refactor the existing associated test code. As with their work, this new study could assess the complexities of the test suites using the Bruntink and van Duersen metrics [17]. Another approach might be to rely completely on automated test case generation for both suites, but the project proposal remains to apply the tools used in this study to the sources used by the Badri et al. study [16] and compare the results.
Another potentially area for further study then is in the metrics used to measure testability. A test suite is also body of code and it is a reasonable assumption that most evaluations that can be applied to source code can probably be applied on some level to test code also. Badri et al. used the Bruntink [17] metrics TNbLOC (a measure of the number of lines of code of the test class) and TNbOfAssert (based on the number of invocations of JUnit assert in the code of a test class). But an evaluation of the correlations between different metrics in code and the corresponding test code could be a potentially useful component of a further study or perhaps it might merit a separate paper.

8 Acknowledgments

Thanks to Iman Hemati Moghadam in UCD for the version Code-Imp tool used in this study, and also for his strong support with the relevant metric formulae and their implementation in code. Thanks to the 10 volunteers from the MSc ASE class of 2011 for taking the time to review the code versions, write the test cases and give the feedback. Much appreciation is also due to Mel Ó Cinnéide, my supervisor for his support, guidance and patience throughout as well for the opportunity to collaborate on the case study and the initial use of these results in a submission paper for the RefTest (REFactoring & TESTing) Berlin Workshop 2011 [22].

9 References

1. ANT (www.apache.org): a Java-based build tool


5. Emerson Murphy-Hill and Andrew P. Black. Refactoring Tools: Fitness for Purpose Department of Computer Science, Portland State University, Portland, Oregon, May 7, 2008


12. JFREECHART (http://www.jfree.org/jfreechart): a free chart library


24. R v Sussex Justices; Ex parte McCarthy [1924] 1 KB 256 at 259.


### 10 Appendices

#### 10.1 The raw refactoring output

The complete list of 26 refactoring steps which Code-Imp applied to the code the version “A” code to produce version “B”, listed in chronological order.

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Element Name</th>
<th>Source Class Name</th>
<th>Target Class Name</th>
<th>Comment</th>
<th>Metric Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Replace Inheritance with Delegation</td>
<td>Industrialist</td>
<td></td>
<td></td>
<td>&quot;Industrialist&quot; is not more a child of &quot;Person&quot;</td>
<td>0.0420168</td>
</tr>
<tr>
<td>2 Extract Hierarchy</td>
<td>&quot;EXH_IndustrialistManager&quot;</td>
<td></td>
<td></td>
<td>&quot;EXH_IndustrialistManager&quot; is Added as &quot;Industrialist's&quot; child to hierarchy structure</td>
<td>0.0420918</td>
</tr>
<tr>
<td>3 PullUp Method</td>
<td>getTextValue</td>
<td>Student</td>
<td>Academic</td>
<td>&quot;getTextValue&quot; is added to &quot;Academic&quot;</td>
<td>0.0431655</td>
</tr>
<tr>
<td>4 PushDown Field</td>
<td>isStoring</td>
<td>SuperForm</td>
<td>ManageForm, SubjectForm,</td>
<td>&quot;isStoring&quot; is added to some of &quot;SuperForm's &quot; subclasses</td>
<td>0.0451275</td>
</tr>
<tr>
<td>5 Decrease Security: Field</td>
<td>firstName</td>
<td>Person</td>
<td>Person</td>
<td>The security of field &quot;firstName&quot; is decreased</td>
<td>0.0451613</td>
</tr>
<tr>
<td>6 PullUp Field</td>
<td>College</td>
<td>Academic</td>
<td>Person</td>
<td>&quot;college&quot; is added to Person</td>
<td>0.0501012</td>
</tr>
<tr>
<td>7 PushDown Method</td>
<td>setCollege</td>
<td>Academic</td>
<td>Student,</td>
<td>&quot;setCollege&quot; is added to some of &quot;Academic's&quot; subclasses</td>
<td>0.0522703</td>
</tr>
<tr>
<td>8 PushDown Method</td>
<td>getCollege</td>
<td>Academic</td>
<td>Student,</td>
<td>&quot;getCollege&quot; is added to some of &quot;Academic's&quot; subclasses</td>
<td>0.0544255</td>
</tr>
<tr>
<td>9 Extract</td>
<td>&quot;EXH_New SubjectFor&quot;</td>
<td></td>
<td></td>
<td>&quot;EXH_NewSubjectFormTeacherForm&quot; is Added as</td>
<td>0.0547129</td>
</tr>
<tr>
<td>ID</td>
<td>Action</td>
<td>Component</td>
<td>Outcome</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>----------------</td>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Pull-Up Method</td>
<td>InputValidated</td>
<td>&quot;InputValidated&quot; is added to &quot;SuperForm&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Extract Hierarchy</td>
<td>&quot;EXH_SubjectFormNewSubjectForm&quot;</td>
<td>&quot;EXH_SubjectFormNewSubjectForm&quot; is added as &quot;SubjectForm's&quot; child to hierarchy structure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Increase Security: Field</td>
<td>companyEmployer</td>
<td>Industrialist</td>
<td>The security of field &quot;companyEmployer&quot; is increased</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Extract Hierarchy</td>
<td>&quot;EXH_SuperFormSubjectForm&quot;</td>
<td>&quot;EXH_SuperFormSubjectForm&quot; is Added as &quot;SuperForm's&quot; child to hierarchy structure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Extract Hierarchy</td>
<td>&quot;EXH_PersonAcademic&quot;</td>
<td>&quot;EXH_PersonAcademic&quot; is Added as &quot;Person's&quot; child to hierarchy structure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Push-Down Field</td>
<td>college</td>
<td>Person</td>
<td>&quot;college&quot; is added to some of &quot;Person's &quot; subclasses</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Extract Hierarchy</td>
<td>&quot;EXH_AcademicStudent&quot;</td>
<td>&quot;EXH_AcademicStudent&quot; is Added as &quot;Academic's&quot; child to hierarchy structure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Pull-Up Field</td>
<td>isStoring</td>
<td>Manager Form</td>
<td>&quot;isStoring&quot; is added to EXH_SuperFormSubjectForm</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Pull-Up Field</td>
<td>isStoring</td>
<td>EXH_SuperForm SubjectForm</td>
<td>&quot;isStoring&quot; is added to SuperForm</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Push-Down</td>
<td>college</td>
<td>EXH_PersonAcademic</td>
<td>&quot;college&quot; is added to some of</td>
<td></td>
</tr>
</tbody>
</table>

Increasing Code Testability through Automated Refactoring

Dermot Boyle
<table>
<thead>
<tr>
<th></th>
<th>Field</th>
<th>emic</th>
<th>ic,</th>
<th>&quot;EXH_PersonAcademic's &quot; subclasses</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Decrease Security: Field</td>
<td>subjects</td>
<td>SubjectForm</td>
<td>Subject Form</td>
</tr>
<tr>
<td>21</td>
<td>Collapse Hierarchy</td>
<td>EXH_IndustrialistManager</td>
<td>IndustrialistPerson</td>
<td>Industrialist</td>
</tr>
<tr>
<td>22</td>
<td>Decrease Security: Field</td>
<td>_InterIndustrialistPerson</td>
<td>Industrialist</td>
<td>Industrialist</td>
</tr>
<tr>
<td>23</td>
<td>Extract Hierarchy</td>
<td>&quot;EXH_IndustrialistManager&quot;</td>
<td>Academic</td>
<td>EXH_PersonAcademic</td>
</tr>
<tr>
<td>24</td>
<td>PullUp Method</td>
<td>getTextValue</td>
<td>Academic</td>
<td>EXH_PersonAcademic</td>
</tr>
<tr>
<td>25</td>
<td>PushDown Field</td>
<td>college</td>
<td>Academic</td>
<td>EXH_AcademicStudent</td>
</tr>
<tr>
<td>26</td>
<td>PushDown Field</td>
<td>college</td>
<td>EXH_AcademicStudent</td>
<td>Student</td>
</tr>
</tbody>
</table>

**Table 1:** Changes in Security and Hierarchy Structure

|   | Decrease Security: Field    | subjects | SubjectForm | Subject Form | The security of field "subjects" is decreased |
|---|-----------------------------|-------|-----|-----------------------------------|
| 20 | Decrease Security: Field    | subjects | SubjectForm | Subject Form | The security of field "subjects" is decreased |
| 21 | Collapse Hierarchy          | EXH_IndustrialistManager | IndustrialistPerson | Industrialist | "EXH_IndustrialistManager" is removed from the program scope |
| 22 | Decrease Security: Field    | _InterIndustrialistPerson | Industrialist | Industrialist | The security of field "_InterIndustrialistPerson" is decreased |
| 23 | Extract Hierarchy           | "EXH_IndustrialistManager" | Academic | EXH_PersonAcademic | "EXH_IndustrialistManager" is Added as "Industrialist's" child to hierarchy structure |
| 24 | PullUp Method               | getTextValue | Academic | EXH_PersonAcademic | "getTextValue" is added to "EXH_PersonAcademic" |
| 25 | PushDown Field              | college | Academic | EXH_AcademicStudent | "college" is added to some of "Academic's " subclasses |
| 26 | PushDown Field              | college | EXH_AcademicStudent | Student | "college" is added to some of "EXH_AcademicStudent's " subclasses |

**Table 2:** Changes in Security and Hierarchy Structure
10.2 The survey taken by the volunteer group

The following is the full text of the document as it was used to survey the volunteer groups opinions on the increases or decreases in testability resulting from the automated refactoring.

Instructions

The attached zip files contain two Java programs, version A and version B. They are models of the same domain.

Unzip both versions and open them side-by-side in whichever IDE you are comfortable with. It will be necessary to look at files from both versions in order to compare them. To avoid confusion when viewing the code, all classes in version A have “_A” appended and all classes in version B have “_B” appended.

There is no need to spend time familiarising yourself with the two applications. The exercises you will be asked to do are very focussed and don’t require an overall understanding of the applications.

In each of the six exercises you are asked to compare the two versions in terms of their testability, i.e., how easy it is to write the requested test cases. You can write these test cases using JUnit, or just sketch them in a file. They key issue is to form an opinion on which, if either, of the examples is easier to write test cases for, even if both cases seem easy.

When you are finished, please email this form to Dermot Boyle at dboyle@microsoft.com and cc mel.ocinneide@ucd.ie. There is no need to email the test cases you write.

Thank you again for the time and effort in taking part in this experiment. Your help is very much appreciated.
Preliminaries

For how long have you worked in the software industry?
___ years ___ months

For how long have you worked as a software developer?
___ years ___ months

For how long have you used automated unit testing (JUnit, etc)?
___ years ___ months

Please time how long it takes you (approximately) to complete the 6 exercises and write the number here:
___ minutes

If you wish to elaborate, please do so here:
Exercise 1

The class Industrialist_A in version A and the class Industrialist_B in version B both provide the functionality to set and get the industrialist's name. Write two test cases: one to test this functionality in version A and one to test it in version B.

Which version is easier to test (please tick 1)?

- Version A is much easier to test.
- Version A is moderately easier to test.
- Version A is slightly easier to test.
- Both are the same / I have no opinion.
- Version B is slightly easier to test.
- Version B is moderately easier to test.
- Version B is much easier to test.

Please provide a brief explanation (1 line is sufficient) of your answer.
Exercise 2

Write one test case to test the constructor for the class `Industrialist_A` in version A, and one to test the constructor for the class `Industrialist_B` in version B.

Which version is easier to test (please tick 1)?

- Version A is much easier to test.
- Version A is moderately easier to test.
- Version A is slightly easier to test.
- Both are the same / I have no opinion.
- Version B is slightly easier to test.
- Version B is moderately easier to test.
- Version B is much easier to test.

Please provide a brief explanation (1 line is sufficient) of your answer.
Exercise 3

In Version A, the ManagerForm_A class has a method called InputValidated. In Version B, this method is in the SuperForm_B class. Write two test cases: one to test the InputValidated method in version A and one to test it in version B.

Which version is easier to test (please tick 1)?

- Version A is much easier to test.
- Version A is moderately easier to test.
- Version A is slightly easier to test.
- Both are the same / I have no opinion.
- Version B is slightly easier to test.
- Version B is moderately easier to test.
- Version B is much easier to test.

Please provide a brief explanation (1 line is sufficient) of your answer.
Exercise 4

The class Company_A in version A and the class Company_B in version B both provide the functionality to set and get the company's boss. Write two test cases: one to test this functionality in version A and one to test it in version B.

Which version is easier to test (please tick 1)?

- Version A is much easier to test.
- Version A is moderately easier to test.
- Version A is slightly easier to test.
- Both are the same / I have no opinion.
- Version B is slightly easier to test.
- Version B is moderately easier to test.
- Version B is much easier to test.

Please provide a brief explanation (1 line is sufficient) of your answer.
Exercise 5

In Version A the class Student_A has a method getTextValue (see footnote^). In Version B, this method is in Trainee_B. Write two test cases: one to test this functionality in version A and one to test it in version B.

Which version is easier to test (please tick 1)?

☐ Version A is much easier to test.
☐ Version A is moderately easier to test.
☐ Version A is slightly easier to test.
☐ Both are the same / I have no opinion.
☐ Version B is slightly easier to test.
☐ Version B is moderately easier to test.
☐ Version B is much easier to test.

Please provide a brief explanation (1 line is sufficient) of your answer.

---

^getTextValue is private in both Student_A and Trainee_B. This is simply a bug. Please regard them both as being public.
Exercise 6

In version A, the class Industrialist_A is a subclass of Person_A. In Version B, this inheritance relationship does not exist, but Person_B and Industrialist_B both implement the interface InterIndustrialistPerson_B.

Which version would be easier to write test cases for (please tick 1)?

🔹 Version A is much easier to test.
🔹 Version A is moderately easier to test.
🔹 Version A is slightly easier to test.
🔹 Both are the same / I have no opinion.
🔹 Version B is slightly easier to test.
🔹 Version B is moderately easier to test.
🔹 Version B is much easier to test.

Please provide a brief explanation (1 line is sufficient) of your answer.

Thanks again for your time.
10.3 The class code definitions from version “A” of the application - before refactoring.

Some classes not affected by the refactoring process are omitted for brevity

```java
public class Academic_A extends Person_A {
    private String college;

    public Academic_A(String fName, String lName, String college) {
        super(fName, lName);
    }

    public void setCollege(String str) {
        college = str;
    }

    public String getCollege() {
        return college;
    }
}

public class Company_A {
    private String name;

    public enum ActivityType {
        Manufacturing, Service
    }

    private ActivityType coBusiness;
    private Person_A theBoss;

    public Company_A(String coName, ActivityType actType, Person_A boss) {
        SetCompanyName(coName);
        SetCoBusiness(actType);
        SetTheBoss(boss);
    }

    public void SetCompanyName(String coName) {
        name = coName;
    }

    public String GetCompanyName() {
        return name;
    }

    public void SetCoBusiness(ActivityType Bus) {
        coBusiness = Bus;
    }

    public ActivityType GetCoBusiness() {
        return coBusiness;
    }

    public void SetTheBoss(Person_A boss) {
        theBoss = boss;
    }

    public Person_A GetTheBoss() {
        return theBoss;
    }
}
```
public class Industrialist_A extends Person_A {
    Company_A companyEmployer;
    private ArrayList<String> skills;

    public Industrialist_A(String fName, String lName, Company_A employer) {
        super(fName, lName);
        setCompany(employer);
        skills = new ArrayList<String>();
    }

    public void setCompany(Company_A comp) {
        companyEmployer = comp;
    }

    public Company_A getCompany() {
        return companyEmployer;
    }

    public boolean AddSkill(String skill) {
        if (!skill.contains(skill)) {
            skills.add(skill);
            return true;
        } else {
            return false;
        }
    }
}

public class MainForm_A extends JFrame {
    private static final long serialVersionUID = 1L;
    private File recordsFile;

    private JButton viewfile;
    private JButton addstudent;
    private JButton addmanager;
    private JButton addteacher;

    // public List<Person> folk;

    public MainForm_A() {
        super("People Manager");
        setLayout(new FlowLayout());
        // folk = new ArrayList<Person>();

        viewfile = new JButton("ViewFile");
        addstudent = new JButton("AddStudent");
        addmanager = new JButton("AddManager");
        addteacher = new JButton("AddTeacher");

        add(viewfile);
        add(addstudent);
        add(addmanager);
        add(addteacher);

        ButtonHandler handler = new ButtonHandler();
viewfile.addActionListener(handler);
addstudent.addActionListener(handler);
addmanager.addActionListener(handler);
addteacher.addActionListener(handler);
}

private class ButtonHandler implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        if (event.getActionCommand() == "ViewFile") {
            JFileChooser fc = new JFileChooser();
            fc.setFileSelectionMode(JFileChooser.OPEN_DIALOG);
            fc.setFileSelectionMode(JFileChooser.FILES_ONLY);
            ExampleFileFilter_A filter = new ExampleFileFilter_A();
            filter.addExtension("xml");
            filter.setDescription("XML Files");
            fc.setFileFilter(filter);

            int ret = fc.showSaveDialog(MainForm_A.this);

            if (ret == JFileChooser.APPROVE_OPTION) {
                recordsFile = fc.getSelectedFile();
                if (recordsFile == null) {
                    JOptionPane.showMessageDialog(null, "No File Selected!");
                } else {
                    ShowRecordsFile(recordsFile);
                }
            }
        }
        return;
    }
    if (event.getActionCommand() == "AddStudent") {
        SubjectForm_A sf = new NewSubjectForm_A(true);
        sf.setSize(220, 500);
        sf.setVisible(true);
        return;
    }
    if (event.getActionCommand() == "AddManager") {
        ManagerForm_A mf = new ManagerForm_A(true);
        mf.setSize(220, 500);
        mf.setVisible(true);
        return;
    }
    if (event.getActionCommand() == "AddTeacher") {
        TeacherForm_A tf = new TeacherForm_A(true);
        tf.setSize(220, 500);
        tf.setVisible(true);
        return;
    }
}

public void ShowRecordsFile(File theFile) {
    Document doc;
    try {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();

        if (!theFile.exists()) {
            System.out.println("Failed to Find: ");
        }
    } catch (IllegalArgumentException e) {
        e.printStackTrace();
    } catch (ParserConfigurationException e) {
        e.printStackTrace();
    } catch (SAXException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
public class ManagerForm_A extends SuperForm_A {
    public ManagerForm_A(boolean storing) {
        super();
        setLayout(new FlowLayout());
        isStoring = storing;
        submit.setActionCommand("SubmitManager");
        ManagerButtonHandler handler = new ManagerButtonHandler();
        submit.addActionListener(handler);
    }
    public ManagerForm_A() {
        super();
    }
    private class ManagerButtonHandler implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            if (event.getActionCommand() == "SubmitManager") {
                SaveManager();
            }
        }
    }
    public boolean InputValidated() {
        if ((firstName.getText().length() == 0)
            || (lastName.getText().length() == 0)
            || (new String(passWord.getPassword()).length() == 0)) {
            JOptionPane.showMessageDialog(null, "Missing field data!");
            return false;
        }
        return true;
    }
}
public void SaveManager() {
    if (!InputValidated()) {
        return;
    }
    Company_A co = new Company_A("ACME", ActivityType.Service,
        new Person_A("Jemmy", "Ned");
    
    Manager_A manager = new Manager_A(firstName.getText(),
        lastName.getText(), new String(passWord.getPassword()), co);
    
    // JOptionPane.showMessageDialog(null, manager.toString());
    if (isStoring) {
        manager.Serialize();
    }
    ManagerForm_A.this.dispose();
}

class Manager_A extends Industrialist_A {
    private String passWord;
    private boolean subject[];
    private String subjects[];

    public Manager_A(String fName, String lName, String pWord, String skills[],
        Company_A office) {
        super(fName, lName, office);
        setPassWord(pWord);
        for (int i = 0; i < skills.length; i++) {
            super.AddSkill(skills[i]);
        }
    }

    public Manager_A(String fName, String lName, String pWord, Company_A office) {
        super(fName, lName, office);
        setPassWord(pWord);
    }

    public void ChangeFirstName(String fName) {
        super.setFirstName(fName);
    }

    public void ChangeLastName(String lName) {
        super.setLastName(lName);
    }

    public void setPassWord(String str) {
        passWord = str;
    }

    public String getPassWord() {
        return passWord;
    }

    public String toString() {
        String str = "";
        str += getFirstName() + " " + getLastName() + "\n Modules: ";
        for (int i = 0; i < subjects.length; i++)
if (isSubject(i))
    str += subjects[i] + "\n";
return str;

public void setSubject(int i, boolean bool) {
    subject[i] = bool;
}

public boolean isSubject(int i) {
    return subject[i];
}

public boolean Serialize() {
    try {
        if (AddToXML()) {
            return true;
        } else {
            return false;
        }
    } catch (Exception ex) {
        System.out.println("ERROR: " + ex.getMessage());
        return false;
    }
}

public boolean AddToXML() {
    Element rootElement;
    Document doc;
    int i = 0;
    try {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        File dataFile = new File("People.xml");
        if (!dataFile.exists()) {
            System.out.println("Failed to Find People.xml");
            doc = builder.newDocument();
            rootElement = doc.createElement("PEOPLE");
        } else {
            doc = builder.parse(dataFile);
            rootElement = doc.getDocumentElement();
            System.out.println("Root element: " + rootElement.getNodeName());
        }
        Element newManagerEl = doc.createElement("Manager");
        newManagerEl.setAttribute("Firstname", this.getFirstName());
        newManagerEl.setAttribute("Lastname", this.getLastName());
        newManagerEl.setAttribute("pwd", this.getPassWord());
        NodeList peopleSets = rootElement.getChildNodes();
        Node setNode = null;
        for (i = 0; i < peopleSets.getLength(); i++) {
            }
if (peopleSets.item(i).getNodeName() == "Managers") {
    setNode = peopleSets.item(i);
}

if (setNode == null) {
    Element ManagerSet = doc.createElement("Managers");
    ManagerSet.appendChild(newManagerEl);
    rootElement.appendChild(ManagerSet);
} else {
    setNode.appendChild(newManagerEl);
}

XMLSerializer serializer = new XMLSerializer();
serializer.setOutputCharStream(new java.io.FileWriter("People.xml"));
serializer.serialize(doc);
}
}

public class NewSubjectForm_A extends SubjectForm_A {
    public NewSubjectForm_A(boolean storing) {
        super();
        isStoring = storing;
        submit.setActionCommand("SubmitStudent");
    }

    public NewSubjectForm_A() {
        super();
    }
}

public class Person_A {
    private String firstName;
    private String lastName;

    public Person_A(String fName, String lName) {
        setFirstName(fName);
        setLastName(lName);
    }

    public void setFirstName(String str) {
        firstName = str;
    }

    public void setLastName(String str) {
        lastName = str;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }
}

Increasing Code Testability through Automated Refactoring

Dermot Boyle
public class Student_A extends Academic_A {

  private String passWord;
  private boolean subject[];
  private String subjects[];

  public Student_A(String fName, String lName, String pWord, String subs[],
                     String college) {
    super(fName, lName, college);
    this.setFirstName(fName);
    this.setLastName(lName);
    this.setCollege(college);
    setPassWord(pWord);
    subjects = subs;
    subject = new boolean[subjects.length];
  }

  public void ChangeFirstName(String fName) {
    super.setFirstName(fName);
  }

  public void ChangeLastName(String lName) {
    super.setLastName(lName);
  }

  public void setPassWord(String str) {
    passWord = str;
  }

  public String getPassWord() {
    return passWord;
  }

  public String toString() {
    String str = "";
    str += getFirstName() + " " + getLastName() + "\n Modules: ";
    for (int i = 0; i < subjects.length; i++)
    {
      if (isSubject(i))
        str += subjects[i] + "\n";
    }
    return str;
  }

  public void setSubject(int i, boolean bool) {
    subject[i] = bool;
  }

  public boolean isSubject(int i) {
    return subject[i];
  }

  public String getLastName() {
    return lastName;
  }

  public class Student_A extends Academic_A {

    private String passWord;
    private boolean subject[];
    private String subjects[];

    public Student_A(String fName, String lName, String pWord, String subs[],
                     String college) {
      super(fName, lName, college);
      this.setFirstName(fName);
      this.setLastName(lName);
      this.setCollege(college);
      setPassWord(pWord);
      subjects = subs;
      subject = new boolean[subjects.length];
    }

    public void ChangeFirstName(String fName) {
      super.setFirstName(fName);
    }

    public void ChangeLastName(String lName) {
      super.setLastName(lName);
    }

    public void setPassWord(String str) {
      passWord = str;
    }

    public String getPassWord() {
      return passWord;
    }

    public String toString() {
      String str = "";
      str += getFirstName() + " " + getLastName() + "\n Modules: ";
      for (int i = 0; i < subjects.length; i++)
      {
        if (isSubject(i))
          str += subjects[i] + "\n";
      }
      return str;
    }

    public void setSubject(int i, boolean bool) {
      subject[i] = bool;
    }

    public boolean isSubject(int i) {
      return subject[i];
    }

});

Increasing Code Testability through Automated Refactoring

Dermot Boyle
public boolean Serialize() {
    try {
        if (AddToXML()) {
            return true;
        } else {
            return false;
        }
    } catch (Exception ex) {
        System.out.println("ERROR: " + ex.getMessage());
        return false;
    }
}

public boolean AddToXML() {
    Element rootElement;
    Document doc;
    int i = 0;
    try {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        File dataFile = new File("People.xml");
        if (!dataFile.exists()) {
            System.out.println("Failed to Find People.xml");
            doc = builder.newDocument();
            rootElement = doc.createElement("PEOPLE");
            Element studentEl = doc.createElement("Students");
            rootElement.appendChild(studentEl);
            doc.appendChild(rootElement);
        } else {
            doc = builder.parse(dataFile);
            rootElement = doc.getDocumentElement();
            System.out.println("Root element: " + rootElement.getNodeName());
        }
        Element newStudentEl = doc.createElement("Student");
        newStudentEl.setAttribute("Firstname", this.getFirstName());
        newStudentEl.setAttribute("Lastname", this.getLastName());
        newStudentEl.setAttribute("pwd", this.getPassWord());
        Element newSubjects = doc.createElement("Subjects");
        for (i = 0; i < this.subjects.length; i++) {
            Element newSub = doc.createElement("Subject");
            newSub.setAttribute("Name", this.subjects[i]);
            newSub.setAttribute("Selected", new Boolean(this.subjects[i]).toString());
            newSubjects.appendChild(newSub);
        }
        newStudentEl.appendChild(newSubjects);
        NodeList Peoples = rootElement.getElementsByTagName("Student");
        for (i = 0; i < Peoples.getLength(); i++) {
            System.out.println(Peoples.item(i).getLocalName());
            System.out.println(Peoples.item(i).getNodeName());
            System.out.println(Peoples.item(i).getNodeValue());
    }
System.out.println(Peoples.item(i).getTextContent());
if (Peoples.item(i).hasAttributes()) {
    NamedNodeMap aMap = Peoples.item(i).getAttributes();
    System.out.println(aMap.getNamedItem("Firstname"));
}
Node Set = rootElement.getFirstChild();
if (Set.getNodeName().toString() == "Students") {
    Set.appendChild(newStudentEl);
}
XMLSerializer serializer = new XMLSerializer();
serializer.setOutputCharStream(new java.io.FileWriter("People.xml"));
serializer.serialize(doc);
} catch (Exception ex) {
    System.out.println("ERROR: " + ex.getMessage());
    return false;
} return true;
}
private String getTextValue(Element ele, String tagName) {
    String textVal = null;
    NodeList nl = ele.getElementsByTagName(tagName);
    if (nl != null && nl.getLength() > 0) {
        Element el = (Element) nl.item(0);
        textVal = el.getFirstChild().getNodeValue();
    }
    return textVal;
}
public class SubjectForm_A extends SuperForm_A {
    public JCheckBox[] subject;
    private String subjects[] = { "Programming3", "Systems Analysis",
                                  "Project Management", "Data Modelling",
                                  "Internet Systems Development" };
    private Student_A student;
    public SubjectForm_A() {
        subject = new JCheckBox[5];
        for (int i = 0; i < 5; i++) {
            subject[i] = new JCheckBox(subjects[i]);
            add(subject[i]);
        }
        add(submit);
        ButtonHandler handler = new ButtonHandler();
        submit.addActionListener(handler);
    }
    private class ButtonHandler implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            if (event.getActionCommand() == "SubmitStudent") {
                SaveSubject();
            }
        }
    }

Increasing Code Testability through Automated Refactoring

Dermot Boyle
if (event.getActionCommand() == "SubmitTeacher") {
    SaveTeacher();
}

public boolean InputValidated() {
    if ((firstName.getText().length() == 0)
        || (lastName.getText().length() == 0)
        || (new String{passWord.getPassword()}.length() == 0))
    {
        JOptionPane.showMessageDialog(null, "Missing field data!");
        return false;
    }
    return true;
}

public void SaveSubject() {
    if (!InputValidated()) {
        return;
    }
    student = new Student_A(firstName.getText(), lastName.getText(),
        new String{passWord.getPassword()}, subjects, "Trinity");
    for (int i = 0; i < subjects.length; i++) {
        student.setSubject(i, subject[i].isSelected());
    }
    JOptionPane.showMessageDialog(null, student.toString());
    if (isStoring) {
        student.Serialize();
    }
    SubjectForm_A.this.dispose();
}

public void SaveTeacher() {
    if (!InputValidated()) {
        return;
    }
    Teacher_A teacher = new Teacher_A(firstName.getText(),
        lastName.getText(), new String{passWord.getPassword()},
        subjects, "Trinity");
    for (int i = 0; i < subjects.length; i++) {
        teacher.setSubject(i, subject[i].isSelected());
    }
    JOptionPane.showMessageDialog(null, teacher.toString());
    if (isStoring) {
        teacher.Serialize();
    }
    SubjectForm_A.this.dispose();
}

Increasing Code Testability through Automated Refactoring

Dermot Boyle
public class SuperForm_A extends JFrame {
    public boolean isStoring = false;

    public JLabel firstNameL;
    public JLabel lastNameL;
    public JLabel passWordL;
    public JTextField firstName;
    public JTextField lastName;
    public JPasswordField passWord;

    // public JCheckBox[] subject;
    public JButton submit;

    public SuperForm_A() {
        super("Address Book");
        setLayout(new FlowLayout());

        firstNameL = new JLabel("First Name", SwingConstants.RIGHT);
        lastNameL = new JLabel("Last Name", SwingConstants.RIGHT);
        passWordL = new JLabel("Password", SwingConstants.RIGHT);

        firstName = new JTextField(12);
        lastName = new JTextField(12);
        passWord = new JPasswordField(12);

        add(firstNameL);
        add(firstName);
        add(lastNameL);
        add(lastName);
        add(passWordL);
        add(passWord);

        submit = new JButton("Submit");
        add(submit);
    }
}

public class TeacherForm_A extends NewSubjectForm_A {
    public TeacherForm_A(boolean storing) {
        super();
        isStoring = storing;
        submit.setActionCommand("SubmitTeacher");
    }

    public TeacherForm_A() {
        super();
    }
}

public class Teacher_A extends Academic_A {
    private String passWord;
    private boolean subject[];
    private String subjects[];

    public Teacher_A(String fName, String lName, String pWord, String subs[],
                      String college) {
        super(fName, lName, college);
        setPassWord(pWord);
    }
}

Increasing Code Testability through Automated Refactoring

Dermot Boyle
subjects = subs;
subject = new boolean[subjects.length];
}

public void ChangeFirstName(String fName) {
    super.setFirstName(fName);
}

public void ChangeLastName(String lName) {
    super.setLastName(lName);
}

public void setPassWord(String str) {
    passWord = str;
}

public String getPassWord() {
    return passWord;
}

public String toString() {
    String str = ""
    str += getFirstName() + " " + getLastName() + "\n Modules: ";
    for (int i = 0; i < subjects.length; i++)
    {
        if (isSubject(i))
            str += subjects[i] + "\n";
    }
    return str;
}

public void setSubject(int i, boolean bool) {
    subject[i] = bool;
}

public boolean isSubject(int i) {
    return subject[i];
}

public boolean Serialize() {
    try {
        if (AddToXML()) {
            return true;
        } else {
            return false;
        }
    } catch (Exception ex) {
        System.out.println("ERROR: " + ex.getMessage());
        return false;
    }
}

public boolean AddToXML() {
    Element rootElement;
    Document doc;
    int i = 0;

try {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder builder = factory.newDocumentBuilder();

    File dataFile = new File("People.xml");
    if (!dataFile.exists()) {
        System.out.println("Failed to Find People.xml");
        doc = builder.newDocument();
        rootElement = doc.createElement("PEOPLE");
    } else {
        doc = builder.parse(dataFile);
        rootElement = doc.getDocumentElement();
        System.out.println("Root element: " + rootElement.getNodeName());
    }

    Element newStudentEl = doc.createElement("Teacher");
    newStudentEl.setAttribute("FirstName", this.getFirstName());
    newStudentEl.setAttribute("LastName", this.getLastName());
    newStudentEl.setAttribute("pwd", this.getPassWord());
    Element newSubjects = doc.createElement("Subjects");
    for (i = 0; i < this.subjects.length; i++) {
        Element newSub = doc.createElement("Subject");
        newSub.setAttribute("Name", this.subjects[i]);
        newSub.setAttribute("Selected", new Boolean(this.subjects[i]).toString());
        newSubjects.appendChild(newSub);
    }
    newStudentEl.appendChild(newSubjects);
    NodeList peopleSets = rootElement.getChildNodes();
    Node setNode = null;
    for (i = 0; i < peopleSets.getLength(); i++) {
        if (peopleSets.item(i).getNodeName() == "Teachers") {
            setNode = peopleSets.item(i);
        }
    }
    if (setNode == null) {
        Element teacherSet = doc.createElement("Teachers");
        teacherSet.appendChild(newStudentEl);
        rootElement.appendChild(teacherSet);
    } else {
        setNode.appendChild(newStudentEl);
    }

    XMLSerializer serializer = new XMLSerializer();
    serializer.setOutputCharStream(new java.io.FileWriter("People.xml"));
    serializer.serialize(doc);
} catch (Exception ex) {
    System.out.println("ERROR: " + ex.getMessage());
    return false;
} return true;
}
10.4 The class code definitions from version “B” of the application - after refactoring has been applied

Some classes not affected by the refactoring process are omitted for brevity

```java
public class Academic_B extends Trainee_B {
    public Academic_B(String fName, String lName) {
        super(fName, lName);
    }
}

public class BaseChoiceForm_B extends SuperForm_B {
    protected BaseChoiceForm_B() {
        super();
    }
}

public class Collegiate_B extends Academic_B {
    protected Collegiate_B(String fName, String lName) {
        super(fName, lName);
    }
}

public class Company_B {
    private String name;
    public enum ActivityType {
        Manufacturing, Service;
    }
    private ActivityType coBusiness;
    private InterIndustrialistPerson_B theBoss;
    public Company_B(String coName, ActivityType actType, InterIndustrialistPerson_B boss) {
        SetCompanyName(coName);
        SetCoBusiness(actType);
        SetTheBoss(boss);
    }
    public void SetCompanyName(String coName) {
        name = coName;
    }
    public String GetCompanyName() {
        return name;
    }
    public void SetCoBusiness(ActivityType Bus) {
        coBusiness = Bus;
    }
    public ActivityType GetCoBusiness() {
        return coBusiness;
    }
    public void SetTheBoss(InterIndustrialistPerson_B boss) {
```
```java
theBoss = boss;
}

public InterIndustrialistPerson_B GetTheBoss() {
    return theBoss;
}

public class Coordinator_B extends Industrialist_B {

    protected Coordinator_B(String fName, String lName, Company_B employer) {
        super(fName, lName, employer);
    }
}

public class Industrialist_B implements InterIndustrialistPerson_B {

    InterIndustrialistPerson_B _InterIndustrialistPerson;
    private Company_B companyEmployer;
    private ArrayList<String> skills;

    public Industrialist_B(String fName, String lName, Company_B employer) {
        _InterIndustrialistPerson = new Person_B(fName, lName);
        setCompany(employer);
        skills = new ArrayList<String>();
    }

    public void setCompany(Company_B comp) {
        companyEmployer = comp;
    }

    public Company_B getCompany() {
        return companyEmployer;
    }

    public boolean AddSkill(String skill) {
        if (!skill.contains(skill)) {
            skills.add(skill);
            return true;
        } else {
            return false;
        }
    }

    public void setFirstName(String str) {
        _InterIndustrialistPerson.setFirstName(str);
    }

    public void setLastName(String str) {
        _InterIndustrialistPerson.setLastName(str);
    }

    public String getFirstName() {
        return _InterIndustrialistPerson.getFirstName();
    }

    public String getLastName() {
        return _InterIndustrialistPerson.getLastName();
    }
}
```
public interface InterIndustrialistPerson_B {
    public void setFirstName(String str);
    public void setLastName(String str);
    public String getFirstName();
    public String getLastName();
}

public class MainForm_B extends JFrame {
    private static final long serialVersionUID = 1L;
    private File recordsFile;
    private JButton viewfile;
    private JButton addstudent;
    private JButton addmanager;
    private JButton addteacher;
    public MainForm_B() {
        super("People Manager");
        setLayout(new FlowLayout());
        viewfile = new JButton("ViewFile");
        addstudent = new JButton("AddStudent");
        addmanager = new JButton("AddManager");
        addteacher = new JButton("AddTeacher");
        add(viewfile);
        add(addstudent);
        add(addmanager);
        add(ateacher);
        ButtonHandler handler = new ButtonHandler();
        viewfile.addActionListener(handler);
        addstudent.addActionListener(handler);
        addmanager.addActionListener(handler);
        addteacher.addActionListener(handler);
    }

    private class ButtonHandler implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            if (event.getActionCommand() == "ViewFile") {
                JFileChooser fc = new JFileChooser();
                fc.setFileSelectionMode(JFileChooser.OPEN_DIALOG);
                fc.setFileSelectionMode(JFileChooser.FILES_ONLY);
                ExampleFileFilter_B filter = new ExampleFileFilter_B();
                filter.addExtension("xml");
                filter.setDescription("XML Files");
                fc.setFileFilter(filter);
                int ret = fc.showSaveDialog(MainForm_B.this);
                if (ret == JFileChooser.APPROVE_OPTION) {
                    recordsFile = fc.getSelectedFile();
                    if (recordsFile == null) {
                        JOptionPane.showMessageDialog(MainForm_B.this,
                        "File selection canceled.");
                    }
                }
            }
        }
    }
}
showMessageDialog(null, "No File Selected!");

if (event.getActionCommand() == "AddStudent") {
    SubjectForm_B sf = new NewSubjectForm_B(true);
    sf.setSize(220, 500);
    sf.setVisible(true);
    return;
}

if (event.getActionCommand() == "AddManager") {
    ManagerForm_B mf = new ManagerForm_B(true);
    mf.setSize(220, 500);
    mf.setVisible(true);
    return;
}

if (event.getActionCommand() == "AddTeacher") {
    TeacherForm_B tf = new TeacherForm_B(true);
    tf.setSize(220, 500);
    tf.setVisible(true);
    return;
}

pubic static void main(String args[]) {
    MainForm_B mf = new MainForm_B();
    mf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    mf.setSize(200, 300);
    mf.setVisible(true);
}

public void ShowRecordsFile(File theFile) {
    Document doc;
    try {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        if (!theFile.exists()) {
            System.out.println("Failed to Find: "+ theFile.getAbsolutePath());
        } else {
            doc = builder.parse(theFile);
            Node rootNode = doc.getFirstChild();
            if (rootNode.getNodeName() != "PEOPLE") {
                JOptionPane.showMessageDialog(null,
                "File is not a valid PEOPLE record");
                return;
            }
            JFrame viewFrame = new JFrame();
            viewFrame.setSize(400, 300);
            viewFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
            XmlTreeModel model = new XmlTreeModel(rootNode);
            JTree IJTree = new JTree();
            IJTree.setModel(model);
            JScrollPane viewPane = new JScrollPane(IJTree);
            viewPane.setPreferredSize(new Dimension(400, 300));
            viewFrame.add(viewPane);
            viewFrame.pack();
            viewFrame.setVisible(true);
        }
    }
    return;
}
viewFrame.add(viewPane, BorderLayout.CENTER);
viewFrame.setVisible(true);
}
} catch (Exception ex) {
    System.out.println("Error reading file: "+ ex.getMessage());
}

public class ManagerForm_B extends BaseChoiceForm_B {
    public ManagerForm_B(boolean storing) {
        super();
        setLayout(new FlowLayout());
        isStoring = storing;
        submit.setActionCommand("SubmitManager");
        ManagerButtonHandler handler = new ManagerButtonHandler();
        submit.addActionListener(handler);
    }
    public ManagerForm_B() {
        super();
    }
    private class ManagerButtonHandler implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            if (event.getActionCommand() == "SubmitManager") {
                SaveManager();
            }
        }
    }
    public void SaveManager() {
        if (!InputValidated()) { return;
        } Company_B co = new Company_B("ACME", ActivityType.Service,
            new Person_B("Jemmy", "Ned");
        Manager_B manager = new Manager_B(firstName.getText(),
            lastName.getText(), new String(passWord.getPassword()), co);
        // JOptionPane.showMessageDialog(null, manager.toString());
        if (isStoring) { manager.Serialize();
        } ManagerForm_B.this.dispose();
    }
}

public class Manager_B extends Coordinator_B {
    private String passWord;
    private boolean subject[];
    private String subjects[];
    public Manager_B(String fName, String lName, String pWord, String skills[],
        Company_B office)
    {
        super(fName, lName, office);
        setPassWord(pWord);
        for (int i = 0; i < skills.length; i++) {
            subject[i] = (skills[i].equals("English") ? true : false);
            subjects[i] = skills[i];
        }
    }
}
super.AddSkill(skills[i]);
}

public Manager_B(String fName, String lName, String pWord, Company_B office) {
    super(fName, lName, office);
    setPassWord(pWord);
}

public void ChangeFirstName(String fName) {
    super.setFirstName(fName);
}

public void ChangeLastName(String lName) {
    super.setLastName(lName);
}

public void setPassWord(String str) {
    passWord = str;
}

public String getPassWord() {
    return passWord;
}

public String toString() {
    String str = "";
    str += getFirstName() + " " + getLastName() + "\n Modules: ";
    for (int i = 0; i < subjects.length; i++)
    {
        if (isSubject(i))
            str += subjects[i] + "\n";
    }
    return str;
}

public void setSubject(int i, boolean bool) {
    subject[i] = bool;
}

public boolean isSubject(int i) {
    return subject[i];
}

public boolean Serialize() {
    try {
        if (AddToXML()) {
            return true;
        } else {
            return false;
        }
    } catch (Exception ex) {
        System.out.println("ERROR: " + ex.getMessage());
        return false;
    }
}

public boolean AddToXML() {
    Element rootElement;
    Document doc;

    Element module = doc.createElement("Module");
    Element name = doc.createElement("Name");
    name.setTextContent("name");
    module.appendChild(name);
    rootElement.appendChild(module);
    return true;
}
int i = 0;

try {
    DocumentBuilderFactory factory = DocumentBuilderFactory
        .newInstance();
    DocumentBuilder builder = factory.newDocumentBuilder();

    File dataFile = new File("People.xml");
    if (!dataFile.exists()) {
        System.out.println("Failed to Find People.xml");
        doc = builder.newDocument();
        rootElement = doc.createElement("PEOPLE");
    } else {
        doc = builder.parse(dataFile);
        rootElement = doc.getDocumentElement();
        System.out
            .println("Root element: " + rootElement.getNodeName());
    }

    Element newManagerEl = doc.createElement("Manager");
    newManagerEl.setAttribute("Firstname", this.getFirstName());
    newManagerEl.setAttribute("Lastname", this.getLastNa
        Name());
    newManagerEl.setAttribute("pwd", this.getPassWord());

    NodeList peopleSets = rootElement.getChildNodes();
    Node setNode = null;

    for (i = 0; i < peopleSets.getLength(); i++) {
        if (peopleSets.item(i).getNodeName() == "Managers") {
            setNode = peopleSets.item(i);
        }
    }

    if (setNode == null) {
        Element ManagerSet = doc.createElement("Managers");
        ManagerSet.appendChild(newManagerEl);
        rootElement.appendChild(ManagerSet);
    } else {
        setNode.appendChild(newManagerEl);
    }

    XMLSerializer serializer = new XMLSerializer();
    serializer
        .setOutputCharStream(new java.io.FileWriter("People.xml")));
    serializer.serialize(doc);
} catch (Exception ex) {
    System.out.println("ERROR: " + ex.getMessage());
    return false;
}
return true;
};

public class NewSubjectForm_B extends SubjectChoiceForm_B {
    public NewSubjectForm_B(boolean storing) {
        super();
        isStoring = storing;
        submit.setActionCommand("SubmitStudent");
    }
}
public NewSubjectForm_B() {
    super();
}

public class Person_B implements InterIndustrialistPerson_B {
    String firstName;
    private String lastName;

    public Person_B(String fName, String lName) {
        setFirstName(fName);
        setLastName(lName);
    }

    public void setFirstName(String str) {
        firstName = str;
    }

    public void setLastName(String str) {
        lastName = str;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }
}

public class Student_B extends Collegiate_B {

    private String passWord;
    private boolean subject[];
    private String subjects[];
    String college;

    public Student_B(String fName, String lName, String pWord, String subs[], String college) {
        super(fName, lName);
        this.setFirstName(fName);
        this.setLastName(lName);
        this.setCollege(college);
        setPassWord(pWord);
        subjects = subs;
        subject = new boolean[subjects.length];
    }

    public void ChangeFirstName(String fName) {
        super.setFirstName(fName);
    }

    public void ChangeLastName(String lName) {
        super.setLastName(lName);
    }

    public void setPassWord(String str) {
        passWord = str;
public String getPassWord() {
    return passWord;
}

public String toString() {
    String str = "";
    str += getFirstName() + " " + getLastName() + "\n Modules: "
    for (int i = 0; i < subjects.length; i++)
    {
        if (isSubject(i))
            str += subjects[i] + "\n";
    }
    return str;
}

public void setSubject(int i, boolean bool) {
    subject[i] = bool;
}

public boolean isSubject(int i) {
    return subject[i];
}

public boolean Serialize() {
    try {
        if (AddToXML()) {
            return true;
        } else {
            return false;
        }
    } catch (Exception ex) {
        System.out.println("ERROR: " + ex.getMessage());
        return false;
    }
}

public boolean AddToXML() {
    Element rootElement;
    Document doc;
    int i = 0;

    try {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        File dataFile = new File("People.xml");
        if (!dataFile.exists()) {
            System.out.println("Failed to Find People.xml");
            doc = builder.newDocument();
            rootElement = doc.createElement("PEOPLE");
            Element studentEl = doc.createElement("Students");
            rootElement.appendChild(studentEl);
        } else {
            doc = builder.parse(dataFile);
            rootElement = doc.getDocumentElement();
            Element studentEl = doc.createElement("Students");
            rootElement.appendChild(studentEl);
        } else {
            doc = builder.parse(dataFile);
            rootElement = doc.getDocumentElement();
        }
    }
System.out.println("Root element: " + rootElement.getNodeName());

Element newStudentEl = doc.createElement("Student");
newStudentEl.setAttribute("Firstname", this.getFirstName());
newStudentEl.setAttribute("Lastname", this.getLastName());
newStudentEl.setAttribute("pwd", this.getPassWord());
Element newSubjects = doc.createElement("Subjects");
for (i = 0; i < this.subjects.length; i++) {
    Element newSub = doc.createElement("Subject");
    newSub.setAttribute("Name", this.subjects[i]);
    newSub.setAttribute("Selected", new Boolean(this.subjects[i]).toString());
    newSubjects.appendChild(newSub);
}
newStudentEl.appendChild(newSubjects);

NodeList Peoples = rootElement.getElementsByTagName("Student");
for (i = 0; i < Peoples.getLength(); i++) {
    System.out.println(Peoples.item(i).getLocalName());
    System.out.println(Peoples.item(i).getNodeName());
    System.out.println(Peoples.item(i).getNodeValue());
    System.out.println(Peoples.item(i).getTextContent());
    if (Peoples.item(i).hasAttributes()) {
        NamedNodeMap aMap = Peoples.item(i).getAttributes();
        System.out.println(aMap.getNamedItem("Firstname"));
    }
}

Node Set = rootElement.getFirstChild();
if (Set.getNodeName().toString() == "Students") {
    Set.appendChild(newStudentEl);
}

XMLSerializer serializer = new XMLSerializer();
serializer.setOutputCharStream(new java.io.FileWriter("People.xml"));
serializer.serialize(doc);
} catch (Exception ex) {
    System.out.println("ERROR: " + ex.getMessage());
    return false;
}
return true;
}

public void setCollege(String str) {
    college = str;
}

public String getCollege() {
    return college;
}

public class SubjectChoiceForm_B extends SubjectForm_B {
    protected SubjectChoiceForm_B() {

public class SubjectForm_B extends BaseChoiceForm_B {
    public JCheckBox[] subject;
    private Student_B student;
    public boolean isStoring = false;

    public SubjectForm_B() {
        subject = new JCheckBox[5];
        for (int i = 0; i < 5; i++) {
            subject[i] = new JCheckBox(subjects[i]);
            add(subject[i]);
        }
        add(submit);
        ButtonHandler handler = new ButtonHandler();
        submit.addActionListener(handler);
    }

    private class ButtonHandler implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            if (event.getActionCommand() == "SubmitStudent") {
                SaveSubject();
            }
            if (event.getActionCommand() == "SubmitTeacher") {
                SaveTeacher();
            }
        }
    }

    public boolean InputValidated() {
        if ((firstName.getText().length() == 0) || (lastName.getText().length() == 0) || (new String(passWord.getPassword()).length() == 0)) {
            JOptionPane.showMessageDialog(null, "Missing field data!");
            return false;
        }
        return true;
    }

    public void SaveSubject() {
        if (!InputValidated()) {
            return;
        }
        student = new Student_B(firstName.getText(), lastName.getText(), new String(passWord.getPassword()), subjects, "Trinity");
        for (int i = 0; i < subjects.length; i++) {
            student.setSubject(i, subject[i].isSelected());
        }
        JOptionPane.showMessageDialog(null, student.toString());
        if (isStoring) {
            // Additional code here for storing the student information
        }
    }
}
public void SaveTeacher() {
    if (!InputValidated()) {
        return;
    }
    Teacher_B teacher = new Teacher_B(firstName.getText(),
                                        lastName.getText(), new String(passWord.getPassword()),
                                        subjects, "Trinity");
    for (int i = 0; i < subjects.length; i++) {
        teacher.setSubject(i, subject[i].isSelected());
    }
    JOptionPane.showMessageDialog(null, teacher.toString());
    if (isStoring) {
        teacher.Serialize();
    }
    SubjectForm_B.this.dispose();
}

public class SuperFormManagerForm_B extends SuperForm_B {
    protected SuperFormManagerForm_B() {
        super();
    }
};

public class SuperForm_B extends JFrame {
    public JLabel firstNameL;
    public JLabel lastNameL;
    public JLabel passWordL;
    public JTextField firstName;
    public JTextField lastName;
    public JPasswordField passWord;
    public SuperForm_B() {
        super("Address Book");
        setLayout(new FlowLayout());
        firstNameL = new JLabel("First Name", SwingConstants.RIGHT);
        lastNameL = new JLabel("Last Name", SwingConstants.RIGHT);
        passWordL = new JLabel("Password", SwingConstants.RIGHT);
        firstName = new JTextField(12);
        lastName = new JPasswordField(12);
        passWord = new JPasswordField(12);
        add(firstNameL);
        add(firstName);
        add(lastNameL);
add(lastName);
add(passWordL);
add(passWord);

submit = new JButton("Submit");
add(submit);

public boolean InputValidated() {
    if ((firstName.getText().length() == 0)
        || (lastName.getText().length() == 0)
        || (new String(passWord.getPassword()).length() == 0))
    {
        JOptionPane.showMessageDialog(null, "Missing field data!");
        return false;
    }
    return true;
};

public class TeacherForm_B extends TeacherInputForm_B {

    public TeacherForm_B(boolean storing) {
        super();
        isStoring = storing;
        submit.setActionCommand("SubmitTeacher");
    }

    public TeacherForm_B() {
        super();
    }
};

public class TeacherInputForm_B extends NewSubjectForm_B {

    protected TeacherInputForm_B(boolean storing) {
        super(storing);
    }

    protected TeacherInputForm_B() {
        super();
    }
};

public class Teacher_B extends Collegiate_B {

    private String passWord;
    private boolean subject[];
    private String subjects[];

    public Teacher_B(String fName, String lName, String pWord, String subs[], String college) {
        super(fName, lName);
        setPassWord(pWord);
        subjects = subs;
        subject = new boolean[subjects.length];
    }

    public void ChangeFirstName(String fName) {
        super.setFirstName(fName);
    }
}
public void ChangeLastName(String lName) {
    super.setLastName(lName);
}

public void setPassWord(String str) {
    passWord = str;
}

public String getPassWord() {
    return passWord;
}

public String toString() {
    String str = ""
    str += getFirstName() + " " + getLastName() + "\n Modules: ";
    for (int i = 0; i < subjects.length; i++)
    {
        if (isSubject(i))
            str += subjects[i] + "\n";
    }
    return str;
}

public void setSubject(int i, boolean bool) {
    subject[i] = bool;
}

public boolean isSubject(int i) {
    return subject[i];
}

public boolean Serialize() {
    try {
        if (AddToXML()) {
            return true;
        } else {
            return false;
        }
    } catch (Exception ex) {
        System.out.println("ERROR: "+ ex.getMessage());
        return false;
    }
}

public boolean AddToXML() {
    Element rootElement;
    Document doc;
    int i = 0;
    try {
        DocumentBuilderFactory factory = DocumentBuilderFactory
            .newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        File dataFile = new File("People.xml");
        if (!dataFile.exists()) {
            System.out.println("Failed to Find People.xml");
            doc = builder.newDocument();
        } else {
            rootElement = builder.parse(dataFile);
        }
        doc.appendChild(rootElement);
    } catch (Exception ex) {
        System.out.println("ERROR: "+ ex.getMessage());
        return false;
    }
}
rootElement = doc.createElement("PEOPLE");
} else {
    doc = builder.parse(dataFile);
    rootElement = doc.getDocumentElement();
    System.out
        .println("Root element: " +
        rootElement.getNodeName());
}

Element newStudentEl = doc.createElement("Teacher");
newStudentEl.setAttribute("Firstname", this.getFirstName());
newStudentEl.setAttribute("Lastname", this.getLastName());
newStudentEl.setAttribute("pwd", this.getPassWord());
Element newSubjects = doc.createElement("Subjects");
for (i = 0; i < this.subjects.length; i++) {
    Element newSub = doc.createElement("Subject");
    newSub.setAttribute("Name", this.subjects[i]);
    newSub.setAttribute("Selected",
        new Boolean(this.subjects[i]).toString());
    newSubjects.appendChild(newSub);
}
newStudentEl.appendChild(newSubjects);
// = rootElement.getElementsByTagName("Teacher");
NodeList peopleSets = rootElement.getChildNodes();
Node setNode = null;
for (i = 0; i < peopleSets.getLength(); i++) {
    if (peopleSets.item(i).getNodeName() == "Teachers") {
        setNode = peopleSets.item(i);
    }
}
if (setNode == null) {
    Element teacherSet = doc.createElement("Teachers");
    teacherSet.appendChild(newStudentEl);
    rootElement.appendChild(teacherSet);
} else {
    setNode.appendChild(newStudentEl);
}
XMLSerializer serializer = new XMLSerializer();
serializer
    .setOutputCharStream(new java.io.FileWriter("People.xml"));
    serializer.serialize(doc);
} catch (Exception ex) {
    System.out.println("ERROR: " + ex.getMessage());
    return false;
}
return true;
;

public class Trainee_B extends Person_B {

    protected Trainee_B(String fName, String lName) {
        super(fName, lName);
    }

    private String getTextValue(Element ele, String tagNamae) {


String textVal = null;
NodeList nl = ele.getElementsByTagName(tagName);
if (nl != null && nl.getLength() > 0) {
    Element el = (Element) nl.item(0);
    textVal = el.getFirstChild().getNodeValue();
}
return textVal;