Applying Case Based Reasoning and Structural Similarity for Effective Retrieval of Expert Knowledge from Software Designs

Markus Adrian Wolf

A thesis submitted in partial fulfilment of the requirements of the University of Greenwich for the degree of Doctor of Philosophy

September 2012

School of Computing and Mathematical Sciences
University of Greenwich
30 Park Row, Greenwich, SE10 9LS
London, England

Dedication

In loving memory of my father, Gunnar Morgenstern-Wolf, and my grandfather,

Dr Erhard Wolf, both of whom would have been very proud of this achievement.

Abstract

Due to the proliferation of object-oriented software development, UML software designs are ubiquitous. The creation of software designs already enjoys wide software support through CASE (Computer-Aided Software Engineering) tools. However, there has been limited application of computer reasoning to software designs in other areas. Yet there is expert knowledge embedded in software design artefacts which could be useful if it were successfully retrieved. Thus, there is a need for automated support for expert knowledge retrieval from software design artefacts.

A software design is an abstract representation of a software product and, in the case of a class diagram, contains information about its structure. It is therefore possible to extract knowledge about a software application from its design. For a human expert an important aspect of a class diagram are the semantic tags associated with each composing element, as these provide a link to the concept each element represents. For implemented code, however, the semantic tags have no bearing. The focus of this research has been on the question of whether is it possible to retrieve knowledge from class diagrams in the absence of semantic information.

This thesis formulates an approach which combines case-based reasoning with graph matching to retrieve knowledge from class diagrams using only structural information. The practical applicability of this research has been demonstrated in the areas of cost estimation and plagiarism detection. It was shown that by applying case-based reasoning and graph matching to measure similarity between class diagrams it is possible to identify properties of an implementation not encoded within the actual diagram, such as the domain, programming language, quality and implementation cost.

Abstract

An approach for increasing users' confidence in automatic class diagram matching by providing explanation is also presented.

The findings show that the technique applied here can contribute to industry and academia alike in obtaining solutions from class diagrams where semantic information is lacking.

The approach presented here, as well as its evaluation, were automated through the development of the UMLSimilator software tool.

Acknowledgements

The work presented here would not be in its current shape without the help and support I received from various people.

Firstly, I would like to thank my supervisor Prof Miltos Petridis for all of his support, trust, unceasing motivation, encouragement and for never losing his positive attitude. Without him, I would not have embarked on this research.

I would like to thank Prof Brian Knight for his support and advice throughout this journey and for making this research possible in the first place.

Further, I would like to express my gratitude to Prof Liz Bacon for her interest and encouragement and my second supervisor Dr Jixin Ma for his support.

A very big thank you to Irena for the many fruitful discussions and brainstorming sessions which helped me resolve problems encountered along the way and whose expert input has been very valuable.

Finally, I want to thank all my family and friends for their support and sincere interest in a foreign subject. A special thank you to my grandmother Helein, Geli, Christoph, Stephan, Jan, Marta and Jassie (for keeping my lap warm on occasion while working on this research).

Contents

List of Figures

List of Tables

List of Equations

Chapter 1

1 Introduction

The entire object-oriented paradigm revolves around reuse [Lewis et al., 1990]. Classes are created, which define blueprints for generating any number of objects. Reuse happens at an intra-class level, where one can factor code into methods which can be reused within the class. Inheritance enables us to reuse existing classes through extension and specialisation. Classes can be packaged as components, which can be reused. However, these are all examples of code reuse.

Design plays an important role in the creation of software artefacts, as it enables visualising the structure and distribution of responsibility, within a planned application and it is at the design stage that ideas start taking shape. Reuse in object-oriented development is not limited to the reuse of code, although this is the kind of reuse which has been applied at a practical level for a long time. Traditionally, less importance has been given to the reuse of software design, but this has changed, especially with the emergence of design patterns [Gamma et al., 1995].

The way a software artefact is structured is important for a developer. A sound design enables the creation of reusable modules and more manageable code that can be more easily understood by fellow programmers, which is maintainable, flexible and easily extendible [Victor et al., 1996]. However, a question posed in this research is whether the design is only important to a developer who is writing code or whether there are other circumstances in which the design of a software artefact is of importance. The functionality of a software application is determined by the choice of code constructs and how the code is laid out (its design). As there are numerous ways of structuring

code to achieve the same functionality, it is not possible to determine the exact functionality of an application just by its design. It is possible, however, to compare designs within a particular context. This research focuses on the use of knowledge that can be retrieved from software designs.

Another consideration is the impact of a software artefact's design on its performance at runtime, a topic which is not addressed in this research, but considered in research into areas such as refactoring [Demeyer, 2005], binary refactoring [Tilevich and Smaragdakis, 2005] and compiler optimisation [Su and Lipasti, 2006]. According to Abreu and Melo [Abreu and Melo, 1996], quality of software can be influenced by object-oriented design mechanisms, such as inheritance and polymorphism.

This research addresses whether it is possible to effectively automate the retrieval of knowledge from software engineering structures.

1.1 Motivation

There is no exact science dictating how functional and non-functional requirements of a software application are translated into a software design. Software design is an experience-driven process and the structure of a software artefact stems from a developer's or systems architect's experience, personal preference, understanding of the business requirements, target technologies or may even be reused in part from existing designs. Giving the same non-trivial requirements to a number of developers or architects, would result, without doubt, in different software designs. This is obvious when setting software design assignments to a class of students.

For a person comparing software designs depicting applications with similar functionality, a prime indicator for identifying similarity would be the semantic relationship between the different elements. The problem with automating this comparison is that the names chosen for elements composing a software design could be in different languages, abbreviations, words joined together or be meaningful only to the designer. It is partly due to this limitation that this research ignores the semantic aspect of software artefacts and focuses entirely on their structure.

As has been established above, it is not possible to determine the functionality of an application or system strictly based on its design structure. If each composing element is stripped of its semantic description, it can represent anything and is therefore stripped of much of its meaning. However, there is meaning in how composing elements are related to one another and how they are structured internally. Comparing software designs merely from a structural perspective, within a context, can provide valuable information.

There are contexts within which the structural similarity of software designs is more relevant than the information provided by semantic comparison. One such example is cost estimation of software development. In software engineering, cost or effort estimation is the process of predicting the effort required in order to develop a piece of software. Cost estimation of software is considered more difficult than in other industries, as it generally involves creating new products [Briand and Wieczorek, 2002]. As outlined in [Jones, 2007], different approaches exist to software cost estimation; some are based on formal models, such as parametric or size-based models, while others rely on expert estimation. A major motivation of this research is to determine whether cost estimation can be automated effectively by comparing software designs. Given a collection of software designs and corresponding cost it took to implement each design,

e.g. in lines of code, it may be possible to predict the effort necessary to implement a new target design, provided that the implementation shares some common context, such as, complexity of the functionality, similar programming languages and technologies, etc.

There have been a number of attempts to automate software cost estimation using approaches as varied as fuzzy decision trees, neural networks, rule induction and case-based reasoning. A comparison of various automated software effort estimation techniques has been undertaken by Finney et al. [Finnie, et al., 1997] whose findings were that while regression models performed poorly, neural networks and case-based reasoning both have value for software development cost estimation models. In the case of case-based reasoning, it seems particularly appealing due to similarity to expert judgment approaches and as an expert assistant in support of human judgement.

Using case-based reasoning for cost estimation means that when a new project is provided for estimation, the most similar projects from the case-base are selected in order to predict the cost of the new project. Much of the research for software cost estimation using case-based reasoning is based on the Desharnais data set [Desharnais, 1989] with cases containing, among other, data such as project details, project length, team experience, programming language and entities [Huang, 2009; Li, et al., 2009]. Other research involving case-based reasoning relates to web development [Mendes, et al., 2002; Mendes, et al., 2003]. None of this research, however, takes into consideration the software design artefacts. Given that a software design artefact depicts the intended structure of the software product, it can contain information useful to the estimation of its development.

Another context within which the structural similarity of software designs is more relevant than the information provided by semantic comparison is plagiarism detection. Within software development, a person committing plagiarism may intentionally change the semantic values of elements in order to avoid detection. By measuring similarity between software designs purely on a structural level, copying someone's code and changing the names or order of classes, operations and attributes, would still be detected.

Much of the research into plagiarism detection focuses on free text and source code [Mozgovoy, 2006; Lukashenko, et al., 2007; Jadalla and Elnagar, 2008]. Nevertheless, one may not always be in possession of the source code when comparing two software structures. As the subjects of comparison are software designs, there may not be any source code if the comparison occurs prior to implementation.

A software design is defined by its composing elements, which in the case of a UML (Unified Modelling Language) class diagram are its classes and interfaces. A class/interface is further defined by its internal members, which could be operations, constructors and/or attributes. Beyond the composing elements, an essential aspect of defining a software design is the relationships between the elements.

By reducing structurally complex data, as present in a software design, to a graph-based notation, it is possible to compare two software designs taking into account not just the sum of its composing elements, but also their relationships.

Representing software architectures as graphs has been discussed by Le Métayer [Le Métayer, 1996] and Fahmy [Fahmy, et al., 1997] where hierarchical systems are represented in graph-notation. While the research was not specifically applied to UML in these cases, the software architecture or architectural designs included components,

such as files or procedures which could be expressed diagrammatically. As long as a diagrammatic representation of a software system has a hierarchical structure it may be represented as a graph, where nodes represent components and arcs represent relationships between the components.

Much of the research on UML and graphs appears to be related to the problem of how to visually present diagrams, such as, visualising designs [Gutwenger, 2003; Eichelberger, 2003] and aesthetics for domain-specific layout [Purchase, et al., 2001].

Once a software design is reduced to graph-based notation it is possible to apply graph matching algorithms. In this research, the similarity is measured based on the maximum common subgraph, which is defined by Chartrand [Chartrand, 1989] as follows: "A graph *G* without isolated vertices is a greatest common subgraph of a set *S* of graphs, all having the same size, if *G* is a graph of maximum size that is isomorphic to a subgraph of every graph in *S*".

Graph matching and case-based reasoning has been successfully combined by [Petridis, et al., 2007a; Petridis, et al., 2007b; Mileman, et al., 2000; Mileman, et al., 2002], with the purpose of aiding the design of metal castings.

In this research, case-based reasoning and graph matching are combined in order to measure similarity between software designs so that this information can be applied to automated cost estimation, plagiarism detection and identification of properties, such as implementation quality, programming language and functionality.

1.2 Research Question

Software designs are an essential element of current software development. Yet it is difficult to automate the retrieval of knowledge from software designs. Software design is a process driven by personal experience and preferences, and a software engineer encodes semantic and structural information in the design artefacts. The semantic information is essential to convey how the design reflects the application or system it represents, but structural information is just as important. It provides the internal structure of elements, as well as the relationships between elements which will dictate the overall structure of the application/system.

Very broadly defined, the aim of this research can be summarised in the following question:

- Is it possible to effectively retrieve expert knowledge from structural software engineering artefacts?

The above aim is very general, so more specific aims are:

- In the absence of semantic information, is it possible to extract meaningful knowledge merely from the structural information, and can this retrieval process be automated?
- Can case-based reasoning be applied to software designs to identify software systems from the same domain based on their structure?
- Can structural similarity and case-based reasoning be combined to estimate software development cost?
- A user's trust in an automated system is reinforced if the system provides explanation of the results it provides. A system which applies case-based reasoning, measures similarities of cases to reach a result, which makes it even

more important to provide explanation. Thus a further aim of the research is for the developed system to explain why it reached a particular result.

1.3 Research Methodology

To validate the research findings and to warrant that the research question has been answered appropriately, the following measures are taken:

- Development of a software application to automate the algorithms and procedures outlined in the research
- Use of a case base with a considerable amount of diverse cases. There are over one hundred cases in the case base. The cases are based on results from a number of different teaching assignments, which ensures that there is a varied mix of cases. Having cases based on a set of teaching assignments also ensures that a number of cases represent software solutions with identical requirements (same assignment), making it possible to assess the identification of cases from the same domain and measure quality.
- Controlled and planned experiments to test various aspects of algorithms and methods implemented in the software application. Extensive tests are carried out comparing only elements (without graphs), graphs using maximum common subgraph and graphs with optimised weighting.
- Application of cross-validation to reinforce trust in the algorithms and their results.
- Use of the software's ability to explain results, tracing how a particular result was obtained, to evaluate the outcomes against human expert opinion.

1.4 Thesis Outline

This thesis comprises six chapters.

Chapter I: **Introduction**. This chapter gives an introduction to the thesis by providing the motivation for the research, stating the research question, outlining the research methodology applied in the research and finally presenting the structure of this thesis.

Chapter II: **Literature Review and Background**. This chapter provides a literature survey of the state of research in the various areas which are relevant to this work, namely, case-based reasoning and graph similarity, retrieval of software knowledge, software engineering, cost estimation and explanation. The amount of publications in these subject areas is vast, thus the review is concentrated on the research contributions which are most closely related to this research or from which it follows. Surveying related research publications places this research into context, demonstrating that the research question is of value to the research community.

Chapter III: **Using Case-Based Reasoning for Retrieval of Knowledge from Software Design Artefacts**. This chapter presents the methodologies and techniques that were applied, as well as providing justification for their application. The importance of structural similarity within software engineering artefacts is highlighted, in comparison to semantic similarity. The chapter also discusses the reasoning for applying graph modelling.

Chapter IV: **UMLSimilator**. This chapter presents UMLSimilator, which is the software tool developed to evaluate the research. It explains how the algorithms are implemented and provides examples to illustrate.

Chapter V: **Experiments and Evaluation**. This chapter describes the experiments performed in order to assess the performance of the applied algorithms and techniques. It delineates the results and provides an analysis of the results compared to alternative approaches. The contribution to knowledge of each experiment is demonstrated.

Chapter VI: **Conclusion**. This chapter concludes the thesis by outlining the main contribution of this research and discusses new directions in which this work could evolve.

1.5 Publications

The following publications were completed for this research, all of which were co-authored with Prof Miltos Petridis:

- Wolf, M. Petridis, M. (2003) Similarity Metrics for Reuse of Software Design using CBR, in Proceedings of the 8th UK Workshop on Case Based Reasoning, Cambridge
- Wolf, M. Petridis, M. (2004) Applying CBR to Measure Similarity of Software Design Structures, Expert Update (regular journal of the Specialist Group on Artificial Intelligence - SGAI)
- Wolf, M. Petridis, M. (2008) Measuring Similarity of Software Designs using Graph Matching for CBR, in Proceedings of the Artificial Intelligence Techniques in Software Engineering Workshop, 18th European Conference on Artificial Intelligence

1.6 Conclusion

Software designs are an essential element of current software development, but the automated retrieval of knowledge from the designs poses a problem. Software design artefacts can be reduced to a hierarchical structure, made up of their composing elements and the relationships between these elements. Thus, a software design can be represented as a graph. Reducing a design to a graph of a hierarchical structure makes it possible to automate the comparison of designs using defined similarity metrics. Knowledge retrieved in this way can be effectively applied in a number of fields, including cost estimation and plagiarism detection.

In the next chapter, a review of the current state of the field in subject areas relevant to this research is presented.

Chapter 2

2 Literature Review and Background

The previous chapter set the context of this research and defined the research question that the work addresses.

The goal of this chapter is to present the current state of the research in the main subject areas that relate to this work. The chapter starts by describing core areas of software engineering and Case-Based Reasoning (CBR), given that these are central to the research. Software engineering as the discipline dedicated to the design and implementation of software, resulting in software design artefacts, and Case-Based Reasoning as a methodology for extracting and using knowledge. Here, software designs are the case representations in the case-base of the CBR system. A key aspect of CBR is determining similarity of cases. Thus, analogical reasoning is discussed, leading to the discussion of structural similarity, as well as, the concepts of graph representation and graph similarity. This chapter then continues to explore the retrieval of software knowledge, looking at work which influenced this research, as well as considering alternative approaches. Given that the research is applied to cost estimation, this area is also discussed. The final section of this chapter covers the area of explanation.

2.1 Software Engineering

The term software engineering stems from the 1968 NATO Software Engineering Conference [Naur and Randell, 2004] where it was applied with the intention of adopting theoretical foundations and practical disciplines in place for established branches of engineering design, to the creation of software. Software engineering is defined by the IEEE as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software" [Abran and Moore, 2004] and as such, it relates to the entire process of designing, implementing and maintaining software.

The application of a systematic approach to software development has led to the emergence of multiple software engineering methodologies, such as structured analysis and design [Yourdon, 1979] (SSADM – Structured Systems Analysis and Design Methodology [Eva, 1994]), object-oriented development [Booch, 2007], agile software development [Beck, et al., 2001], etc. In all of these methodologies, design features prominently.

According to Tong and Sriram [Tong and Sriram, 1992], design is the process of generating a structural description that complies with a set of functional specifications and constraints. The process of designing requires knowledge of the functional and behavioural requirements which are analysed, synthesised and represented in a structural description. The creation of a visual design model aids the development of software by considering the entire set or a significant subset of the requirements and conceptualising them into an architectural blueprint for a software system. This provides an overview which aids understanding of the system to be implemented by visualising its structure. Further, it facilitates sharing a common perception of a system among multiple people.

In recent years, object-oriented programming languages have consistently dominated the top positions of the TIOBE programming community index [TIOBE, 2012], with the most recent rating at 57.1% (July 2012 – see Table 1). This clearly shows the market dominance of object-oriented software development.

Category	Ratings July 2012	Delta July 2011
Object-Oriented Languages	57.1%	+1.5%
Procedural Languages	37.4%	-0.5%
Functional Languages	3.6%	-1.3%
Logical Languages	1.9%	+0.3%

Table 1 - Programming Paradigms according to TIOBE Index

2.1.1 Unified Modelling Language and Meta-Object Facility

The most common modelling approach for object-oriented software development is the Unified Modelling Language (UML) [Jacobson, et al., 1998], which is standardised and managed by the Object Management Group (OMG). UML was created in 1997 by unifying modelling approaches by the Three Amigos (James Rumbaugh, Grady Booch and Ivar Jacobson).

UML has fourteen sets of diagrams divided into two types, namely structural and behavioural views of a system model [OMG, 2009]. The structural diagrams are used to depict the architecture of a software system, while the behavioural diagrams emphasise the functionality of a system and how different composing elements interact. The most commonly used diagram by programmers is the class diagram, which describes the structure of a system including its classes, their attributes and the relationships between the classes.

UML diagrams are visual representations and to automate the extraction of knowledge from them requires representing them in a data format that can easily be manipulated by a software application. It is essential to the research presented here that the visual representation is converted to a format that can be stored persistently, used to effectively extract information from and compare design models. Fortunately, UML is in fact a metamodel, defined by the Meta-Object Facility (MOF). MOF is a meta-modelling architecture, which in the case of UML uses four layers [OMG, 2006] (see Figure 1). The top layer, called M3 provides a meta-metamodel which defines the language used to build metamodels in the layer below (M2) and actually uses the modelling framework and notation of UML. Layer M2 is a metamodel which describes UML itself, defining what concepts exist in UML and how they relate to one another – e.g. a class has attributes and attributes have a type, a name, a set of valid modifiers, etc. Models in layer M1 are described by layer M2 and represent concrete models, such as a particular class diagram. Finally, layer M0 embodies real-world objects, which in the case of a class modelled in a UML class diagram would be an object of this class's type. In short, an object is modelled by a UML model, which is an instance of the UML metamodel, which is described by the MOF meta-metamodel, which uses a subset of UML object modelling constructs to define itself (e.g. types, containers).

A software design model is situated on layer M1, but in order to correctly represent the model in a system, be able to persist it and manipulate it, one needs to be aware of the structure of the model as defined by its metamodel on layer M2.

An OMG standard which enables persisting and exchanging metadata information via XML (Extensible Markup Language) is XMI (XML Metadata Interchange) [OMG, 2007]. XMI documents are XML documents which contain metadata of a model expressed in MOF, which includes UML models. This is made possible by a mapping,

which expresses how MOF elements should be mapped to XMI-compliant XML. While XMI documents can be used to encode the composing elements in a UML model, they don't include diagram information (e.g. the graphical layout of a particular diagram). It was in order to overcome this limitation of XMI, that XMI[DI] (XMI Diagram Interchange) was created by OMG [OMG, 2006b]. Diagram Interchange uses the underlying concept of modelling the contents of UML diagrams as graphs, where every element is represented as a node or an edge.

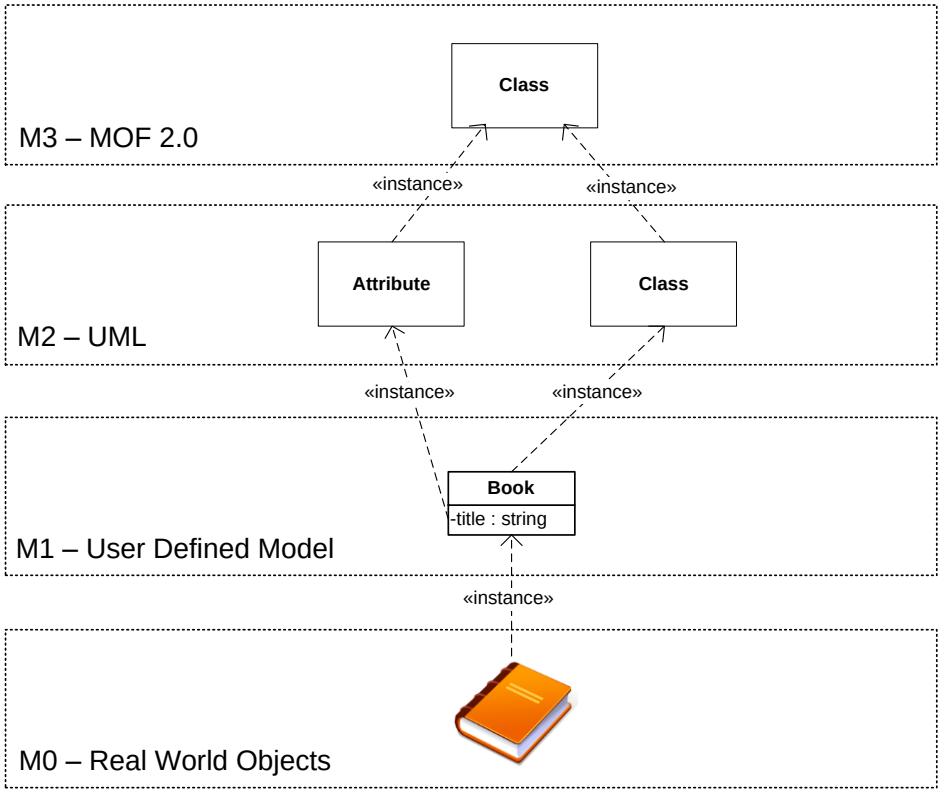

Figure 1 - MOF Layers

Treating UML models as graphs and manipulating the graphs using XMI has been applied in order to create model transformations using graph rewriting systems [Gelhausen, 2008].

Lack of consistent support for the XMI and XMI[DI] standards means that there are interoperability problems when exchanging UML models between different CASE (Computer Aided Software Engineering) tools. However, an understanding of the MOF metamodel architecture makes it possible to dissect a UML model into a hierarchical structure, which is essential to enable comparisons of UML models as it exposes, not only the composing elements of a UML model, but also the kinds of relationships that exist between these elements.

2.1.2 CASE Tools and Reverse Engineering

CASE tools were used in the 1980s and 1990s with the aim of reducing software process costs by automating some process activities and providing information about software during the development life cycle and, according to Huff [Huff, 1992], this has resulted in improvements in the order of 40%.

CASE (Computer Aided Software Engineering) tools are commonly used to depict and manipulate UML software designs. While the UMLSimilator developed to evaluate this research is not a fully-fledged CASE tool, it does visually present class diagrams and applies a feature supported by many CASE tools, which is reverse engineering. Chikofsky and Cross [Chikofsky and Cross, 1990] define reverse engineering as being "the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction". Using reverse engineering it is thus possible to take compiled existing software and abstract it into the higher-level design from which it originated. Existing tools for reverse engineering code have been discussed by Gorton and Zhu [Gorton and Zhu, 2005].

A major driver for reverse engineering is the understanding and maintenance of legacy code, especially where appropriate design documentation is lacking. Even if original designs are available, systems change over time as transformations are applied in response to new requirements or identified bugs, and the transformations may not have been captured in design documentation. Once designs have been recovered, it is possible to apply new transformations to the existing code [Baxter and Mehlich, 1997].

The process of understanding and changing legacy systems is referred to as re-engineering. The functionality of the software is not normally changed during the re-engineering process and even the system architecture generally stays the same [Sommerville, 2004]. A key advantage of re-engineering is the reduction of cost in comparison to developing a new system. Ulrich [Ulrich, 1990] presents an example of a system whose estimate for re-implementation was $50 million and which was successfully re-engineered for $12 million.

However, reverse engineering is not only relevant to maintaining legacy code. It can also be used to obtain knowledge about compiled code, e.g. understanding web site content and structure [Tilley and Huang, 2001], creating UML sequence diagrams [Merdes and Dorsch, 2006], obtaining graphical user interface layouts [Ramon, et al., 2010] or verification of forward engineering to automatically determine whether an implementation is consistent with the original design [Cooper, et al., 2004].

When a software artefact has been reverse engineered into a hierarchical design structure, it is then possible to extract knowledge from this design. One way in which this can be achieved is through the use of CBR.

2.2 Case-Based Reasoning

Case-Based reasoning (CBR) [Kolodner, 1993] is at the centre of the work presented here. In essence, CBR is the process of solving new problems which are based on the solution of similar problems while learning from the new cases, if possible.

The use of CBR in the area of Artificial Intelligence (AI) is founded on work by Roger Schank in 1982 [Schank, 1982], who developed a theory of learning and reminding based on preserving experience in a dynamic and evolving memory structure.

The CBR process typically involves four steps as outlined by Aamodt and Plaza [Aamodt and Plaza, 1994] (see Figure 2):

1. Retrieval of most similar cases
2. Reuse / adaptation of retrieved cases to attempt to solve the problem
3. Revision of proposed solution, if necessary
4. Retention of the new case in the case base

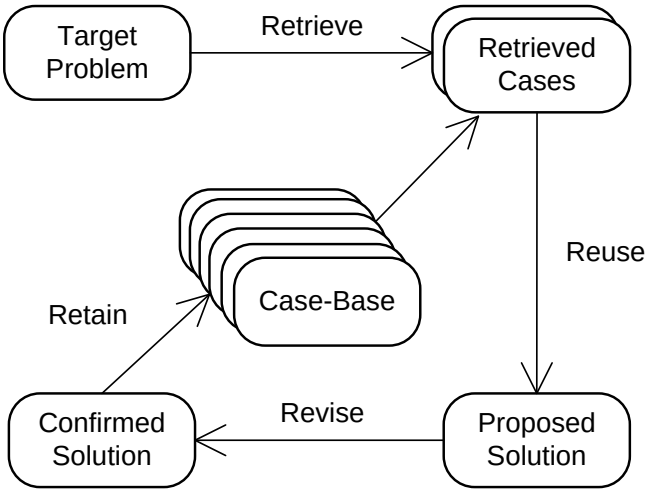

Figure 2 - CBR Cycle - adapted from [Aamodt and Plaza, 1994]

Due to the nature of this research, its emphasis is on the retrieval and reuse steps, as no new cases are added to the case-base as part of the evaluation.

2.2.1 Case Representation and Case-Base

In CBR a case represents specific knowledge tied to a context. The representation of a case has to be in a format that can be used by the CBR system, but it can take many different forms. Different types of case representation are discussed by Bergmann, Kolodner and Plaza [Bergmann, et al., 2005], who describe traditional approaches, feature vector representation, structured representation and textual representation, as well as sophisticated approaches using hierarchical representations and generalised cases.

In practice a case can store many types of data, ranging from attribute/value pairs, textual representations or even multimedia representations. When dealing with object-oriented design models, case representations which are of particular interest include object-oriented case representation and graph representation [Bergmann, 2002].

Key to any CBR system is the case-base or case library. This is the repository which stores the cases. The case-base structure should reflect the conceptual view of the case. There are two main academic case-memory models which address the knowledge representation problem in CBR, namely, the dynamic memory model [Riesbeck and Schank, 1989] and the category exemplar model [Bareiss, 1988]. These models are intended to simplify the access and retrieval of relevant cases. In practice, however, many case-bases are stored using flat file structures or relational databases. The case-base in this research was stored using a relational database as the cases are complex hierarchical structures which can be easily represented using a relational structure representing the meta-data that makes up a software design.

2.2.2 Indexing and Retrieval

When case-bases contain large amounts of cases, indexing becomes essential in order to speed up the retrieval of cases. An index is a data tag which is associated with a case and could consist of one or more features of the case or an abstraction inferred from the case. A lot of research has been carried out aiming at automating and improving case indexing methods, including query sphere algorithm [Stéphane, et al., 2010], semantic indexing [Recio-Garcia and Wiratunga, 2010], explanation-based indexing [Barletta and Mark, 1988] and many more.

The retrieval phase within the CBR cycle involves identifying the most suitable cases from the case-base and ranking them. The initial retrieval of relevant cases can be achieved using indices. The ranking of the retrieved set is usually achieved by using a similarity metric which determines the distance between the target case and each retrieved case. In some cases a CBR system may not have the identification of relevant cases and ranking as two distinct tasks, but apply the ranking on all cases in the case-base. This evidently removes the need for indexing and only works with a relatively small case-base. The results in this case would be more accurate as the ranking is done on the entire case-base rather than applying it just to the set identified by the indices. However, this is computationally more demanding and could be very time consuming. In this research, the size of the case-base was sufficiently small to allow ranking across the entire case-base. Indexing was therefore not necessary. Should the case-base increase in the future, applying some form of indexing would become necessary.

The two most common techniques for retrieval are nearest-neighbour retrieval and inductive retrieval. Nearest neighbour is a simple technique which applies the K-Nearest Neighbour (KNN) algorithm [Dudani, 1976] which applies a similarity metric to measure the distance between the target and source case. In most cases, inductive

retrieval uses the ID3 induction algorithm [Quinlan, 1986], which constructs decision trees from past data. Nearest neighbour retrieval is less sensitive to missing data, but its main drawback is that it is computationally intensive. The two techniques can be combined by using inductive indexing to retrieve a set of matching cases and apply nearest neighbour to rank them.

2.2.3 Clustering

A positive aspect of the CBR cycle is the fact that it can easily be complemented with other techniques in order to improve performance or accuracy at different stages of the cycle. Clustering is an area of data mining which has been combined with CBR in a number of ways. It refers to the assignment of objects into groups (clusters) such that objects in the same cluster are more similar. Given that case retrieval in CBR is all about measuring similarity, it invites the use of clustering. Clustering can be applied at different stages of the retrieval process. For instance, when dealing with large scale case-bases, clustering of features can make retrieval more efficient by enabling indexing of representative attributes from the clusters [Hong and Liou, 2008]. It can also be applied to group cases from the case-base [Tsatsoulis and Amthauer, 2003]. This research uses the latter approach to generate clusters of the cases post retrieval. This approach is used as part of the experiments in order to validate the results. Clustering of features to enable indexing, as applied by [Hong and Liou, 2008] could be applied to this research in future with the introduction of indexing.

An advantage of CBR is that it works well with complex domains, where there are many different ways in which to generalise a case. As the target case is already provided, it is not necessary to perform an exact calculation and obtain the one true result, but to identify the case that is closest to the target – the case that is the most similar. CBR is also very useful when there is no algorithmic method available for

evaluating a solution, making it possible to work with unknowns, as the target case is evaluated in the context of other cases in the case-base. This makes CBR well suited for working with software design artefacts as it is not possible to easily calculate their similarity using other methods.

In order to effectively retrieve knowledge associated with software designs it is essential to be able to compare designs and measure similarity of designs. The comparison of software designs requires establishing what composing elements of the design are being used to determine similarity. A key objective of this research is to determine whether structural similarity can be used effectively to retrieve knowledge from software design artefacts.

2.3 Structural Similarity

The creation of software using the object-oriented paradigm requires the "decomposition of a domain into noteworthy concepts or objects" [Larman, 2005]. As discussed in the section on Software Engineering, UML class diagrams are used to depict the structure of object-oriented software by illustrating its classes, interfaces and the associations between these (see Figure 3 for an example). Class diagrams are used for static object modelling; they capture the static structure of the object-oriented software application. This is opposed to other types of UML diagrams which relate to behavioural aspects (state transition, sequence/collaboration diagrams) or functional description (use-case diagrams) of a software application.

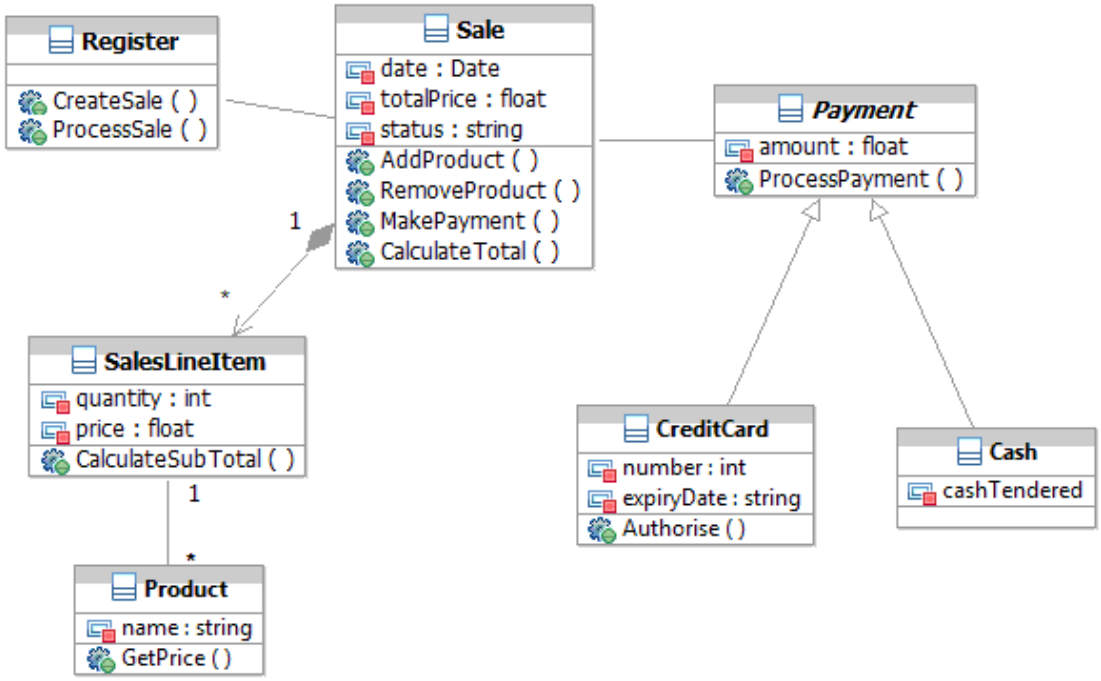

Figure 3 – A UML Class Diagram

2.3.1 Intra-Class Similarity and Hierarchical Structure

For each composing class/interface, a class diagram also captures class-specific attributes[1] and operations, which capture the state and behaviour of the class. In the object-oriented paradigm, attributes and operations are referred to as members. The classes/interfaces and their members collectively make up the structure of the software design. Attributes are also associated with a data type, while operations have a return type and possibly one or more parameters/arguments. All members, as well as the actual classes/interfaces can be associated with one or more modifiers. Modifiers qualify the element they are associated with, which in an object-oriented programming language provides instructions to the compiler on how to treat that element.

[1] Interfaces don't normally contain attributes, however some object-oriented programming languages do allow the use of public static attributes

The fact that a software design can be broken down into its composing elements, makes it possible to view its structure as a hierarchy of elements. The hierarchical structure of class diagrams can be utilised as is evident from the work by Egyed [Egyed, 2002] who automates the abstraction of class diagrams by creating tree structures.

If one creates a tree-like representation of a class diagram, classes and interfaces, as the main composing elements, would be placed at the top of the hierarchy. Members and modifiers would create the lower levels of the hierarchy.

The hierarchical structure outlined in Figure 4, shows how a class can be broken down into its composing elements. Different levels of the hierarchy are shown in different colours and elements shown in italics are defined by their composing elements.

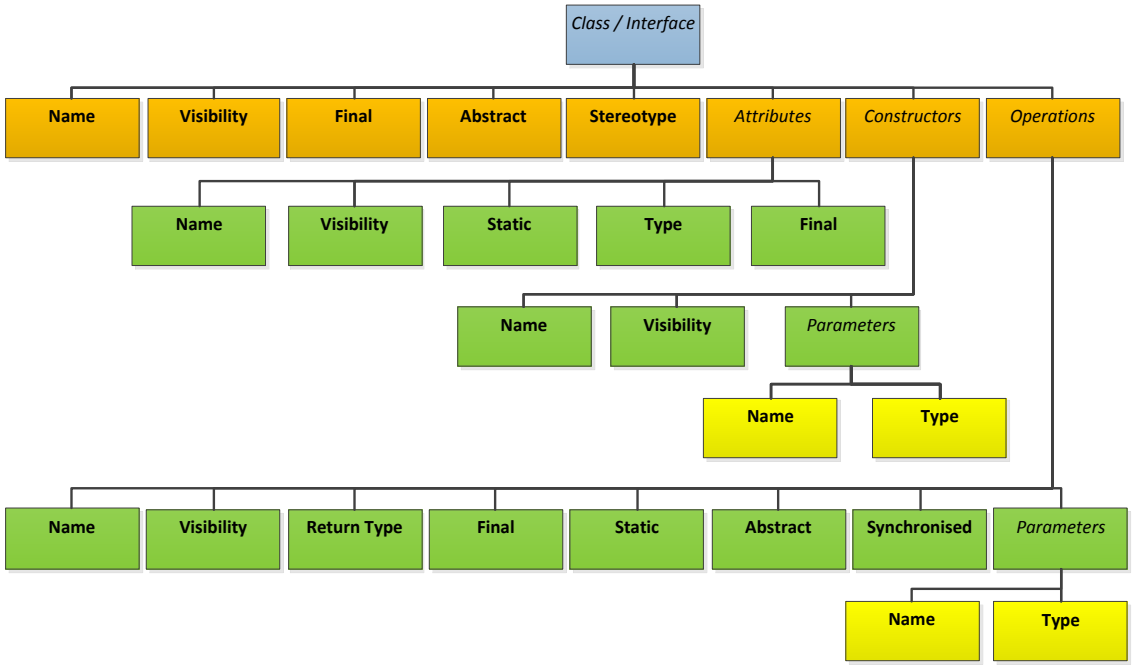

Figure 4 - Hierarchical Structure of Composing Elements of a Class

The one element which is common to classes, attributes, constructors, operations and parameters alike, is the name. What distinguishes the name from all the other elements

is the fact that it is just an identifier, but doesn't have any other operational value or implication. That is not to say that the name is irrelevant. In an object-oriented design, classes represent concepts from the domain which is being modelled and the names would identify which concept is represented by a particular class or interface. Likewise, the name of an attribute would identify what piece of information it stores. So for a person who looks at a software design, the names would be some of the most important features.

As is evident from the hierarchical structure presented above, software designs are complex structures, and there is no preset benchmark for comparing or defining similarity between software designs or elements of software designs. This makes calculating the similarity between software design models a somewhat difficult and complex process to automate. It is possible for a human expert to identify similarities between given designs using a heuristic approach, thus the key lies in classifying the characteristics that would make a human expert identify the similarity.

As with any complex structure, partitioning into substructures or composing elements is required in order to be able to compare software models. In order to define the overall similarity of the entire structure, these elements are compared against each other.

2.3.2 Feature Weighting and Genetic Algorithms

By dividing a software design into its composing elements, a hierarchy of elements is created. Every element in the hierarchy of a software design can then be compared to the corresponding elements from another design, contributing to the overall similarity of the two designs. As each element contributes to the overall similarity in uneven shares, a weighting scheme is used to assign the weights proportionally. This way the weights can be adapted individually to reflect the reasoning process of the human expert.

Feature weighting is employed heavily when using the k-nearest neighbour algorithm in CBR and several weighting methods have been reviewed and evaluated by Wettschereck, Aha and Mohri [Wettschereck, et al., 1997].

Different approaches can be taken when determining the weights that should be applied to each feature. Weight settings could be obtained from an expert human, who knows the importance of each feature and to what extent it should contribute to the overall similarity. It is also possible to obtain weights empirically by running experiments with different weight settings and identifying those which work best and to automate the process, as is evident from the research of Kelly and Davis [Kelly and Davis, 1991], who compared results obtained using a hybrid genetic algorithm to unweighted k-NN. They used the genetic algorithm and a training data set to learn real-valued weights associated with individual attributes in the data set and used the obtained weights in the k-NN algorithm. Their findings were that using weights yielded better results.

Genetic algorithms are optimisation and machine learning techniques in the area of AI, which are loosely based on the process of natural evolution. Genetic algorithms have been made very popular through the work of John Holland [Holland, 1975]. Being based on natural evolution, genetic algorithms have adopted terms used in this field, such as *population*, *chromosome*, *mutation* and *crossover*. Genetic algorithms were employed in this research in an attempt to optimise the weight settings applied across all features in the hierarchical structure.

A chromosome represents a candidate solution to a problem and each chromosome has a fitness, which is related to the success of the chromosome at solving that problem. A population is the initial set of all chromosomes and genetic algorithms solve an optimisation problem by manipulating the population of chromosomes. The genetic

algorithm repeatedly selects chromosomes from the population which act as parents and who are replaced in the current population with new, potentially modified, copies. The selection of parents is not random, but is biased in function of its fitness, which means that better chromosomes have a higher probability of being selected – this is referred to as fitness proportionate selection. The reproduction phase during which the new chromosomes are formed can employ mutation and crossover operators.

The crossover operator generates children which combine chromosomal matter from both parents and different crossover techniques exist for use with different data structures. The mutation operator introduces localised change, thus creating differences between the child chromosome and its parents. The mutation type will depend on the chromosome. The process of reproduction and replacement is continued until a termination criterion is met. This could be a predetermined length of time, number of replacements or when a solution is found which satisfies a threshold criterion.

Genetic algorithms have been applied successfully to generate attribute weights as is shown in Özşen and Güneş [Özşen and Güneş, 2009] or the research by Beddoe and Petrovic [Beddoe and Petrovic, 2006], which used the generated weights in a CBR system.

Within the set of features that are compared within a class, some can only be measured in direct matches. For example when measuring the similarity of the static modifiers of two attributes, it will either be a 100% match or a 0% match. However, many of the features' similarity can be matched in part by establishing distances between individual values within the domain of possible values for a particular feature. The integer and float data types, for instance, both share the characteristic of being numerical. While it

is not an exact match, it is conceptually closer than a numerical data type and a string data type.

The idea of measuring structural similarity between object-oriented models using data types has also been used by Meditskos and Bassiliades [Meditskos and Bassiliades, 2007] for semantic web service matchmaking.

While all of this makes it possible to measure the intra-class similarity of individual classes or even a set of classes, it ignores the relationships between classes, which are also captured in UML class diagrams. To include the relationships in the similarity metric, one can employ graph matching.

2.4 Graph Similarity

As discussed earlier, UML class diagrams contain the classes/interfaces, as well as the relationships between these. There are three main types of relationships: *association*, *generalisation* and *interface realisation* (implementation). Generalisation establishes an *is-a* relationship between a parent (base) and a child (sub) class. This type of relationship is very strong as all of the visible members of the parent are inherited by the child. Interface realisation is also an *is-a* relationship, but is specific to an interface and a class. As interfaces have no implemented members, there is no inheritance, but the interface specifies members which have to be implemented by the class. Associations establish a *has-a* relationship, which represents a conceptual relationship between two classes. An association merely means that a class knows of another and communicates with it at runtime. This is by far the most common relationship in object-oriented designs. In Figure 3, associations connect the *Register*, *Sale*, *SalesLineItem*, *Product* and *Payment* classes as these merely communicate with each other in order to

operate, while inheritance is used to show that the *CreditCard* and *Cash* classes are types of *Payment* and therefore inherit common payment-related members from the abstract *Payment* class.

In their research, Sanders, et al. [Sanders, et al., 1997] discuss benefits of graph-structured case representations and the potential difficulty of encoding cases as graphs in a CBR system. A UML class diagram already records different relationships between classes or interfaces and it is therefore a straight-forward process to treat a software design as a graph composed of classes/interfaces (nodes) and relationships (arcs). This makes it possible to view one case as a single unit, rather than a set of unrelated classes.

The work on similarity measures for object-oriented case representations by Bergmann and Stahl [Bergmann and Stahl, 1998] defined the similarity between two objects as a combination of the intra-class similarity and the inter-class similarity. The intra-class similarity refers to the similarity between the composing features of a class. Measuring the intra-class similarity is also applied in this research whenever two classes are being compared. The inter-class similarity measures the similarity between two objects based on their positions in the hierarchy. The approach taken here does consider the relationships between classes, but doesn't place the classes themselves in a hierarchy. Instead it treats them as a graph. By representing software designs as graphs, it is possible to compare a design with regards to its composing elements, using the structural hierarchy presented earlier, as well as their relationships. This requires comparing graphs, and as the use of graph-based representations has increased in the last decade [Raveaux, et al., 2010] there has been much research into alternative approaches to graph matching algorithms. Previous research at the University of Greenwich [Knight, et al., 2001; Woon, et al., 2001] has shown that competent case

retrieval based on the structural similarity between cases can be achieved using graph matching techniques.

2.4.1 Graph Matching Algorithms

Graph matching is a computationally expensive process given that the number of nodes and arcs in graphs provide a multitude of combinations in which elements from the source and target graph can be matched. This is aggravated further if graphs are allowed to have circular links and if links are bi-directional, as is the case with UML class diagrams.

A number of algorithms have been developed to address the problem of graph matching. Zhao et al. apply the eigen-decomposition approach to weighted graph matching problems [Zhao, et al., 2007]. A limitation of this approach is that it only guarantees to work well in cases where both graphs to be matched must be nearly isomorphic. As this is not the case with software designs, this approach was not feasible for this research.

Genetic algorithms have also been applied to graph matching [Krcmar and Dhawan, 1994; Auwatanamongkol , 2005]. Auwatanamongkol's approach is interesting as indirect links within a graph may be replaced with direct ones in the matching process. While this approach would work well with relatively small graphs, computing graph similarity with complex graphs would be difficult, as the number of potential combinations is increased. This method could be applied to software designs, but would need to be combined with other techniques in order to make it feasible. While it provides interesting ideas for future research, it was decided to limit graphs to direct links initially and defer the use of indirect links for future research.

Another alternative would have been to use spanning trees to compare the complexity of the graphs. Spanning trees have been successfully applied in textual CBR by creating minimum spanning trees of cases within a cluster based on their semantic similarity [Patterson, et al., 2008].

An algorithm which has been successfully applied to graph matching and CBR [Petridis, et al., 2007b] is based on the correct identification of the *Maximum Common Subgraph* (MCS).

The purpose of this algorithm is that given two graphs, G_1 and G_2, it identifies the largest induced subgraph of G_1 which is isomorphic to an induced subgraph of G_2 (see figure 5).

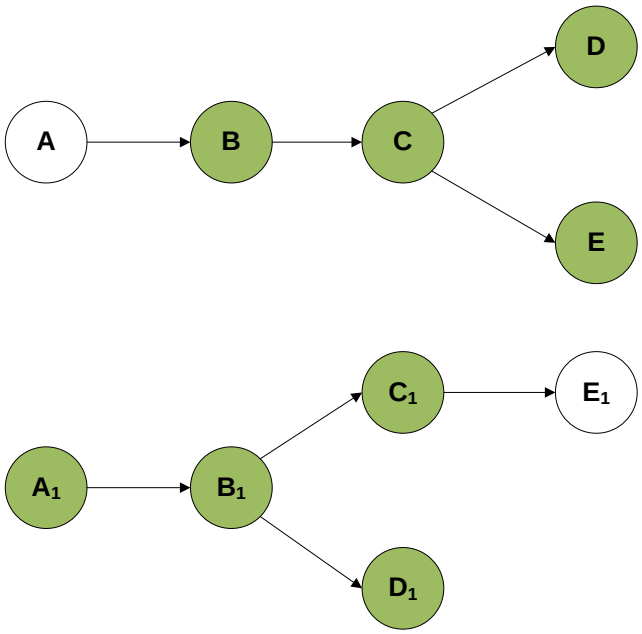

Figure 5 - Example of a maximum common subgraph

The maximum common subgraph and CBR have been successfully combined in the area of metal casting by Petridis et al. [Petridis, et al., 2007a; Petridis, et al., 2007b].

2.4.2 Maximum Common Subgraph in Software Designs

In the case of a UML class diagram, the graph matching algorithm maps the maximum common connected subgraph based on matches between individual classes of each graph, where a minimum threshold similarity value is satisfied.

MCS is known to be NP-hard. Identifying the maximum common subgraph requires an exhaustive search which matches all possible combinations within two graphs to identify the largest isomorphic subgraphs. This can be a time consuming process and graph matching may require the use of a greedy algorithm to be feasible [Champin and Solnon, 2003]. A key difference between the greedy algorithm of Champin and Solnon and this research is that in their algorithm a vertex in a graph may be associated with several vertices of the other graph. Research has also been carried out to speed up subgraph isomorphism detection [Weber, et al., 2011].

In his research of subgraph matching, Ullmann [Ullmann, 1976] proposes an algorithm which reduces the number of successor nodes. This idea is applied in this research by using a minimum threshold between nodes in the graph.

An alternative approach would have been to identify a set of all matching subgraphs above an established number of nodes. This method would be very computationally demanding and has not been explored in this research.

Combining structural similarity of classes and graph similarity of relationships between classes, it may be possible to measure similarity between UML class diagrams and thereby retrieve knowledge, which is what this research is attempting to achieve.

2.5 Retrieval of Knowledge from Software Designs

The main driving element of this research is the retrieval of expert knowledge from software designs. In order to maximise the potential for knowledge retrieval, it is advantageous to have not just a set of software designs, but to have the actual implementations of the designs. This has two main advantages:

1. The additional information can be used to verify findings
2. The designs are realistic as they reflect real software solutions

The main disadvantage is that the software designs may include elements which did not form part of the original design, but were added during the implementation process, such as automatically generated code or existing integrated components. Furthermore, ad-hoc implementations, especially of less-experienced developers, may not have a very meaningful underlying design. For this work, it was decided that overall the advantages of reverse-engineering existing implementations outweigh the disadvantages.

To reconstruct accurate software design representations based on existing implementations, it is necessary to consider the set of constructs available in UML class diagrams [Vinita, et al., 2008] and ensure that the algorithm used in the reverse engineering process correctly captures the classes, interfaces, relationships and the internal structure of classes (operations, constructors, attributes, modifiers). Reverse engineering of existing code is most easily achieved using reflection, which is supported by many object-oriented programming languages. Reflection enables a computer program to examine the structure of a class at runtime. As a matter of fact, it even enables instantiation and invocation of a class, but this is not relevant in this case, as for this research only obtaining the structure was of interest. Reverse engineering has not been limited to UML class diagrams, but has been used to generate UML sequence

diagrams from compiled code [Ziadi, et al., 2011] and the automatic detection of design patterns [Lee, et al., 2007].

Design patterns are an important aspect of object-oriented software design. A "pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same twice" [Alexander, et al., 1977]. A design pattern in software is thus a conceptual solution for common design problems. It is not a concrete design description and it doesn't involve implementation or coding details, but it is rather an abstract description providing a general plan on how to solve the problem. Design patterns represent reuse at design, rather than at code level.

The use of CBR and software reuse has been approached from different angles. Tautz and Althoff have drawn comparisons between CBR and software knowledge reuse models and proposed the extension of the CBR cycle to include organisational issues, which would further support software knowledge reuse [Tautz and Althoff, 1997]. This research considers reuse at conceptual knowledge level and considers organisational structure, but it has also been considered at a much lower level. It has been successfully applied to support reuse in an object-oriented environment [Fernández-Chamizo, et al., 1996] and more specifically to the reuse of design patterns [Tessem, et al., 1998; Shi and Olsson, 2006]. Grabert and Bridge combined CBR and semantic information to reuse code snippets (examplets) [Grabert and Bridge, 2003], which works with the retrieval of information at code level as opposed to design level, as is applied in the research presented here.

2.5.1 Structural vs. Semantic Similarity

As discussed earlier in this chapter, the process of designing software requires knowledge of the functional and behavioural requirements which are then represented in a structural description. To aid understanding of the design and clarify how the requirements are translated to structural elements requires semantic information which is encoded in the names chosen by the designer.

When comparing software designs, one important factor for a human expert is certainly semantic similarity. This allows an expert to recognise what type of system a particular design depicts. Research has been undertaken by Gomes et al. [Gomes, et al., 2007] into CBR and software design similarities using lexical similarity. This research makes use of WordNet, an electronic lexical database, which allows conceptual, as well as, lexical searches to measure semantic similarities. The aim of their research is the reuse of software design by identifying the most similar software design to a given specification and reusing relevant parts of the design.

Another interesting approach is that of Bjornestad [Bjornestad , 2003], which uses analogical reasoning to measure similarity between object-oriented specifications. Emphasis is placed on the role of an object within the context of the software design.

Working with semantic similarity makes it is possible to apply analogical reasoning to relate textual terms. In fact, the work by Gomes et al. uses analogical reasoning in order to transfer knowledge from candidate diagrams to new diagrams based on a query diagram [Gomes, et al., 2002; Gomes, et al., 2002].

Analogical reasoning [Gentner, 1983] is an area of research which shares similarity with CBR. As in CBR, analogical reasoning is a process of comparing similarities between a source and a target. The key difference between the two is that CBR uses intra-domain

reasoning as cases generally represent concepts from the same domain, whereas analogical reasoning is inter-domain, whereby similarities are compared between elements which originate from unrelated domains – e.g. comparing a software design to a blueprint of a house.

Zaremski and Wing [Zaremski and Wing, 1997] use a different notion of semantic similarity in their work on specification matching of software components. Their approach requires metadata in the form of the specification of a component's behaviour. While not measuring semantic similarity of the names used, their use of specifications as a set of pre- and post-conditions provides semantic description of a component.

Interesting research on the reverse engineering of design patterns using a semantic approach has been carried out by Alnusair and Zhao [Alnusair and Zhao, 2009; Alnusair and Zhao, 2010]. Their reasoning is that the structure of a design pattern may take different forms when implemented, which complicates the detection during the reverse engineering process. Their solution was to build an ontology for each design pattern describing its essential participants and their properties, collaborations and role restrictions. This is similar to the approach taken by Gomes et al. [Gomes, et al., 2007], who define the specific participants, application conditions and actions for a specific design pattern.

The merit of semantic similarity is evident from the research of Gomes et al. [Gomes, et al., 2003; Gomes, et al., 2004; Gomes, et al., 2007], who achieved good results creating a CASE tool that helps software developers reuse previously developed software in the creation of new designs. More recently, Robles et al. [Robles, et al., 2012] have also applied semantics when working with software designs, but their approach is based on the creation of a domain ontology and doesn't apply CBR. Nevertheless, important

information is also encoded in the structural similarity. In the area of analogical reasoning, Crean and O'Donoghue [Crean and O'Donoghue, 2002] argue that structural similarity is more important for the identification of good analogical candidates than semantic similarity.

There is no claim here that structural information is more important than semantic information when dealing with software designs. Nevertheless, the structural information may be able to convey knowledge that is not captured by the semantics. The most complete perspective obviously combines structural and semantic information. Gomes et al. complement the semantic similarity with some structural similarity.

The approach taken in this research steps away from both, semantic and role similarities, and starts from the assumption that all information from a given software design model may be obfuscated[2]. Thus, there may be no information about the design other than structural. The question this work is trying to address is whether the structural information on its own can convey knowledge. For example, it is important to distinguish between the structure and functionality of a software design, but does the structure contain within itself enough information to identify its functionality? At least to the extent that software designs with similar functionality can be found to have similar structures and behaviour. Or is there other knowledge which can be extracted from measuring structural similarity between software designs? The areas which are explored in this research include plagiarism detection, identification of properties such as quality of implementation, implementation language, functionality and cost estimation.

[2] In software development, obfuscation refers to the act of making code difficult to understand by humans

2.6 Cost Estimation

In project management it is critical to accurately estimate the size, cost and time required in order to develop a software product. Should the effort be underestimated, as is commonly the place, it leads to time pressure, which could compromise the quality of the final product due to lack of functionality or poor implementation, and in some cases could lead to financial loss due to breach of contract. Overestimating the effort required can too be problematic, as it could lead to quotes which are not competitive. According to a 2011 enterprise resource planning survey by Panorama Consulting, 61.1% of projects take longer than expected and 74.1% are over budget [Krigsman, 2011].

Software cost estimating first began to be discussed as a technology in the late 1960s. In 1969, Joel Aron gave a presentation at a NATO conference on the topic [Aron, 1970].

As outlined in [Jones, 2007], different approaches exist to software cost estimation; some are based on formal models, such as parametric or size-based models, while others rely on expert estimation. According to Boehm et al. [Boehm, et al., 2000a] all classes of techniques are challenged by the rapid pace of change in software technology. The most common approaches are formal estimation models, such as COCOMO (COnstructive COst MOdel), established by Boehm [Boehm, 1981], which apply formulas derived from historical data.

The COCOMO model requires the classification of cost drivers using a scale of ratings, which must be performed by a human expert. However, a human expert remains prone to human errors and biases [Valerdi, 2007]. A major motivation of this research is to determine whether cost estimation can be automated effectively by comparing software designs. Given a collection of software designs and corresponding cost it took to implement each design, e.g. in lines of code, it may be possible to predict the effort

necessary to implement a new target design, provided that the implementation shares some common context, such as, complexity of the functionality, similar programming languages and technologies, etc.

2.6.1 Automating Software Cost Estimation

Attempts have been made to automate software cost estimation using a number of approaches, such as fuzzy decision trees [Andreou and Papatheocharous, 2008; Huang, et al., 2007], neural networks [Tadayon, 2005; Kumar, et al., 2008; Tronto, et al., 2008], rule induction [Shepperd, 1996] and CBR [Huang, 2009; Mendes, et al., 2002; Mendes, et al., 2003; Li, et al., 2009]. A comparison of various automated software effort estimation techniques has been undertaken by Finney et al. [Finnie, et al., 1997] whose findings were that while regression models performed poorly, neural networks and CBR both have value for software development cost estimation models. Further, the research suggests that CBR is particularly appealing due to similarity to expert judgment approaches and as an expert assistant in support of human judgement.

Using CBR for cost estimation means that when a new project is provided for estimation, the most similar projects from the case base are selected in order to predict the cost of the new project. Much of the research for software cost estimation using CBR is based on the Desharnais data set [Desharnais, 1989] with cases containing, among other, data such as project details, project length, team experience, programming language and entities [Huang, 2009; Li, et al., 2009]. Other research involving CBR relates to web development [Mendes, et al., 2002; Mendes, et al., 2003]. None of this research, however, takes into consideration the software design artefacts. Given that a software design artefact depicts the intended structure of the software product, it can contain information useful to the estimation of its development.

2.7 Explanation

It has already been established that there is no existing benchmark for measuring similarity between two software designs. As an analogy, consider architectural blueprints for two houses. It may be difficult for an architect to provide an exact numerical similarity between both as there is no precise algorithm for obtaining this value. However, an architect would be able to select the most similar blueprint to a given target from a set of ten sources. The target blueprint is evaluated in the context of the sources. In the absence of an exact formula, the question would be what features are considered by the architect in order to perform the match. It could be the size of the house, the number of floors, number of rooms, building material, etc.

In the case of software designs the features which could influence an expert to determine similarity could be the number of classes, their relationships or internal class structures.

If one was to ask the architect why he/she selected a particular blueprint from the set as the best match for a target, he/she would probably be able to provide us with an explanation. When a particular software design is selected as the most similar from the case-base, a user's confidence in the system increases if it too can provide an explanation of how the result was achieved.

2.7.1 Explanation in Case-Based Reasoning

The case-based reasoning (CBR) approach is well suited for explanation, because it can use retrieved cases in order to explain prediction. According to Leake [Leake, 1996], "neural network systems cannot provide explanations of their decisions and rule-based systems must explain their decisions by reference to their rules, which the user may not fully understand or accept. On the other hand, the results of CBR systems are based on

actual prior cases that can be presented to the user to provide compelling support for the system's conclusions." The first exploration of CBR and explanation was undertaken by Schank, who proposed an approach based on "explanation patterns" [Schank, 1986], which are generalised patterns of explanation events.

In their research on case-based explanation, Cunningham et al. [Cunningham, et al., 2003] have shown how case-based explanation increased trust more than rule-based or no explanation. Their system shows target cases of individuals' alcohol consumption. It provides predictions of whether an individual is over the blood alcohol limit and also explains the reasoning. It is interesting to note that the rule-based explanation was not as convincing as the case-based one, but as Cunningham et al. point out, the results may differ depending on the subjects' insight into the underlying mechanisms within the domain.

An in-depth overview of different theories of explanation in CBR has been presented by Sormo et al. [Sormo, et al., 2005]. While surveying theories of explanation in the areas of philosophy of science, linguistic and cognitive sciences, it also considers explanation goals and techniques in CBR. The explanation goals are transparency, justification, relevance, conceptualisation and learning. Of these goals, the ones most applicable in this research is transparency, which explains to a user how the system reached a particular answer, and justification, which increases the confidence in the solution offered by the system by providing support for the conclusion it provides.

The explanation technique applied will depend on what the CBR system is trying to explain. In its simplest form, just displaying the most similar case or cases can provide an explanation. This is the approach used by Cunningham et al. [Cunningham, et al., 2003] who along with the target case, also display the most similar case as explanation

of the solution reached. This is very efficient, as it allows the user to make a direct comparison – if person *A* drank *X* amount and was over the blood alcohol limit, then person *B*, who drank a very similar amount was probably also over the limit. Other explanation techniques include visualisation, explanation models, concept maps, and more. This research deals with software designs, which are visual artefacts. It is therefore sensible to employ visualisation in the explanation process. The identification of similar cases is based on the maximum common subgraph, so displaying this to the user can justify the similarity result calculated by the system. To strengthen the trust, this can be complemented with a complete breakdown of the structural similarity between two cases.

2.7.2 Case Provenance

The competence of a CBR system lies in its collection of cases. While good quality cases will obviously improve the accuracy of a CBR system, sometimes it is also relevant to know where a case came from. Case provenance relates to the value of knowing the source of a case and its importance has been discussed by Leake and Whitehead [Leake and Whitehead, 2007]. Provenance can be used in explanation, as the result obtained by the CBR system may be explained in light of the source from where the matching case(s) originated. Using provenance to support explanation has been used by Murdock et al. [Murdock, et al., 2006] in the area of Semantic Web.

Provenance is relevant to this research as the case-base consists of software designs obtained from a set of assessments using different requirements, programming languages and from different levels. Provenance is also significant in cost estimation, where the cost of an implementation is predicted based on existing cases. If cases come from very different sources, the cost may be more difficult to predict accurately.

2.8 Conclusion

As is evident from this literature survey, many of the areas relevant to this research are of importance to academia, as well as industry and are attracting a lot of high quality research. The number of publications in these subject areas is extensive, thus the review concentrated on the research contributions which are most closely related to this work or from which it follows. Surveying the related research publications has placed this research into context, demonstrating that the research question is of value to the research community.

This next chapter presents the methodologies and techniques that were applied, as well as providing justification for their application

Chapter 3

3 Using CBR for Retrieval of Knowledge from Software Designs

The previous chapter presented the current state of the research of the main subject areas that relate to this work.

This chapter presents the methodologies and techniques that were applied in this research, as well as providing justification for their application. As is evidenced in the literature review, the work draws from a number of diverse areas of research. Different approaches and techniques are combined and applied in order to successfully extract knowledge based on the structure of software design artefacts. The case-based reasoning paradigm is at the centre of the comparison process and is discussed at the beginning of the chapter. A key factor which differentiates this research from other approaches introduced in the literature review is the focus on structural information which is reinforced in the next section.

The approach taken in this research achieves retrieval of expert knowledge based on the contextualisation of software designs. It is by relating designs to each other that one can extract knowledge. Viewing a design in the context of others makes it necessary to compare them. The chapter therefore continues with the discussion of complex structures and how these can be compared. Improvements to measuring similarity are considered using weight optimisation and genetic algorithms. This is taken further in the section on graph modelling, which is followed by an explanation of how the solution cases are clustered. This approach to cost estimation is then compared to COCOMO. The chapter concludes with a section on CBR explanation.

3.1 Case-Based Approach for Measuring Similarity of UML Class Diagrams

Case-Based Reasoning was chosen as the paradigm for extracting knowledge from the class diagrams as it is a method which works with complex domains and when no algorithmic method is available for evaluation.

Case-bases with large amounts of cases rely on indexing to speed up the retrieval of cases. However, the case-base used contains just over one hundred cases, which made it feasible to combine identification and ranking of cases. However, this means that a target case is ranked against every existing case in the case-base. As all cases are ranked, the results are more accurate than when ranking is applied only to a set of cases identified using indices.

The two most common techniques for case retrieval are nearest-neighbour retrieval and inductive retrieval. Inductive retrieval creates a decision tree based on past data and requires pre-indexing as the decision tree is used in the retrieval process. This is not appropriate in this case as the class diagrams are complex structures containing many possible features for indexing and it is not known how they should contribute to overall similarity of the diagrams. Depending on the knowledge one is attempting to retrieve or based on the provenance of cases, a feature's importance could change. In this case, inductive reasoning would require recreating the decision tree. Inductive reasoning is also sensitive to missing data and a class diagram is a complex structure containing many optional elements, which may or may not be present. It was for these two reasons that nearest-neighbour retrieval was adopted.

Nearest-neighbour retrieval typically applies the K-Nearest Neighbour (KNN) algorithm [Dudani, 1976], which applies a similarity metric to measure the distance between a

target and source case. The distance could be measured using a Euclidean distance function or a weighted sum of differences. The approach taken here uses the weighted sum of differences.

In principle, when cases share the same structure (contain the same features), the process of measuring the similarity between two cases is straightforward. (1) Each feature in the source case is matched to the corresponding feature in the target case; (2) the degree of match is computed and (3) multiplied by a coefficient which represents the importance of the feature. (4) The results are added to obtain the overall match score. This process is expressed in the following formula, which shows the similarity between a target (C_t) case and a source (C_s):

$$\boldsymbol{\sigma(C_t, C_s) = \frac{\sum_{i=1}^{n} \omega_i * \sigma(f_i^t, f_i^s)}{\sum_{i=1}^{n} \omega_i}} \quad \textbf{(1)}$$

Where f_i is value for the feature and ω_i is the weight (importance) attributed to the feature. In this formula, the result is normalised.

Due to the fact that the cases are represented as a complex hierarchical structure, measuring the similarity between two cases becomes much more complicated. This is discussed in more detail later in this chapter in the sections on structural similarity and graph matching.

3.2 Structural vs. Semantic Information

Creating object-oriented software designs requires an understanding of the tools available for creating designs, as well as of the object-oriented paradigm. The process of designing software doesn't use an exact scientific approach for translating requirements into a software design. This process is experience-driven and often requires an element of creativity and ability for abstraction, especially when a design is trying to solve a problem new to the designer. The quality of a software application from a user's perspective is not synonymous with the quality of the underlying design. Two software applications can have identical user interfaces and support identical functionality and yet have very different designs. This demonstrates that the purpose of a good design is not entirely to create an application or system which implements the specified requirements. Sometimes, features that distinguish a good from a bad design are its flexibility, maintainability and intelligibility. Particularly with large-scale projects, a robust and extensible design is essential to ensure that the end product meets the requirements, can evolve in response to future requirements and is error free. Design can also have an impact on performance. A good design is not necessarily the most efficient in terms of performance. For example, an application where all code is placed in a single class may execute faster at runtime, but would be more difficult to understand and maintain than a design which assigns the same responsibility to a set of cooperating classes.

The functionality of a software application is determined by the choice of code constructs and how the code is laid out (its design), but as there are numerous ways of structuring code to achieve the same functionality, it is not possible to determine the exact functionality of an application just by its design. A UML class diagram is an abstract representation of a software application. The functionality is added when a

design is transformed into an implementation. Yet, an expert who looks at a design can usually recognise what a design represents and how it will work. This information is conveyed through a combination of the object-oriented structure and the selection of labels for the various elements within this structure.

If one was to look at the UML class diagram shown in Figure 6, it would be safe to presume that this represents some kind of retail application. What this software diagram conveys is the structure of the application. However, understanding of the structure is aided by the semantic information and data types.

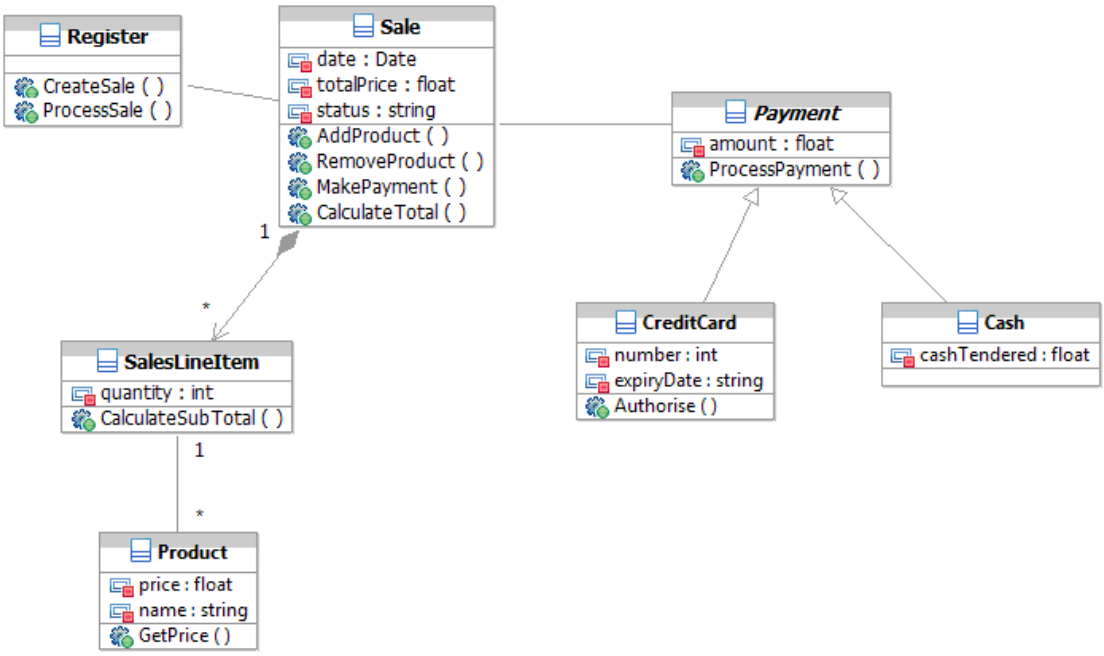

Figure 6 – UML Class Diagram for a Retail Application

If the same diagram were to be obfuscated (see Figure 7), the structure would still be clear. One can see that there will be seven classes; that one is abstract and inherited from by two other classes; the operations and attributes within each class; the relationships between classes. However, it would be impossible to determine what domain this diagram represents. At best, one could make educated guesses.

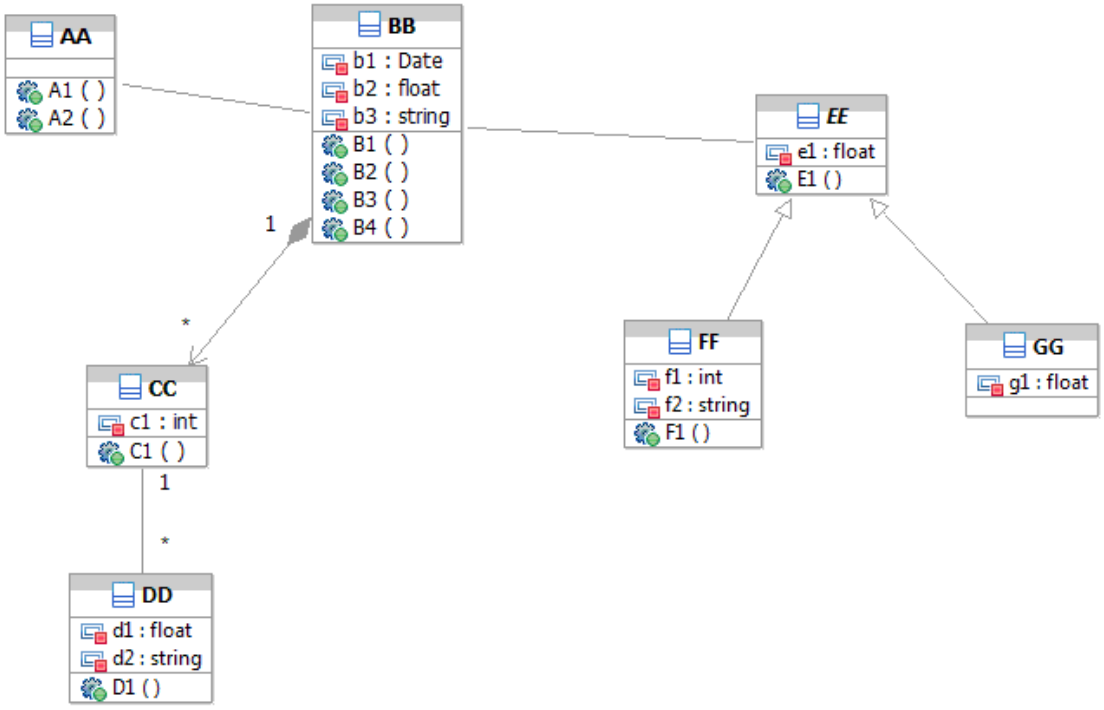

Figure 7 - Obfuscated UML Class Diagram for a Retail Application

This example clearly demonstrates the importance of semantic information when examining a diagram. For a human expert, the semantic information would normally also influence or guide the comparison process of two different diagrams. The semantic information is relevant only to the cognitive process and has no bearing on the functionality, assembly or execution of a software application, which is why it is possible to have obfuscated code.

The question is not whether a semantic approach to knowledge retrieval is useful, this has been verified by much research in the field [Gomes, et al., 2007; Robles, et al., 2012], but whether effective retrieval of information from software designs can be achieved when semantic information is not available; for instance when a developer uses another language to name members or a design is reverse-engineered from

obfuscated code. It is for this reason that the approach taken in this research was to focus entirely on structural information.

3.3 Complex Structural Similarity

It has been established that only the structural information will be considered in order to compare diagrams. So what structural data is available?

Class diagrams are used in object-oriented software design to depict the classes and interfaces of a software design and the way these relate to one another. Apart from the classes and their relationships, class diagrams often show additional information, such as certain class-specific properties, attributes and operations. Every feature of a class diagram can be seen as being fully contained in another. For instance, a class diagram contains a class, the class contains an operation, the operation contains a parameter and the parameter contains a data type. This concept of containment makes it possible to regard a class diagram as a hierarchical tree-like structure. When a class diagram is regarded as a hierarchical structure, features can be located at different levels within the hierarchy. Features at the same level are siblings and every container feature (one which is composed of others) creates a new lower level. If one thinks of a class diagram as a box, then anything that can be seen at a glance when opening the box would form the top layer in the hierarchy. In this case this would be the classes and interfaces, which are the top element in the hierarchy of components of a software design.

Current object-oriented programming languages, such as Java and Visual C#, follow a standard model for determining the elements of the class structure. The class structure may contain modifiers, which state a class's visibility and whether it is final or abstract. It could also contain attributes, operations (methods) and constructors. The attributes,

operations and constructors will in turn have defining properties themselves. This way the hierarchy of features is created, as can be seen in Figure 8 (different colours denote different levels of the hierarchy and features in italic denote a container which is composed of sub-features). This structure differs slightly from that introduced in the literature review, as any names of classes, attributes, etc. have been removed, because the semantic information is omitted from the comparison.

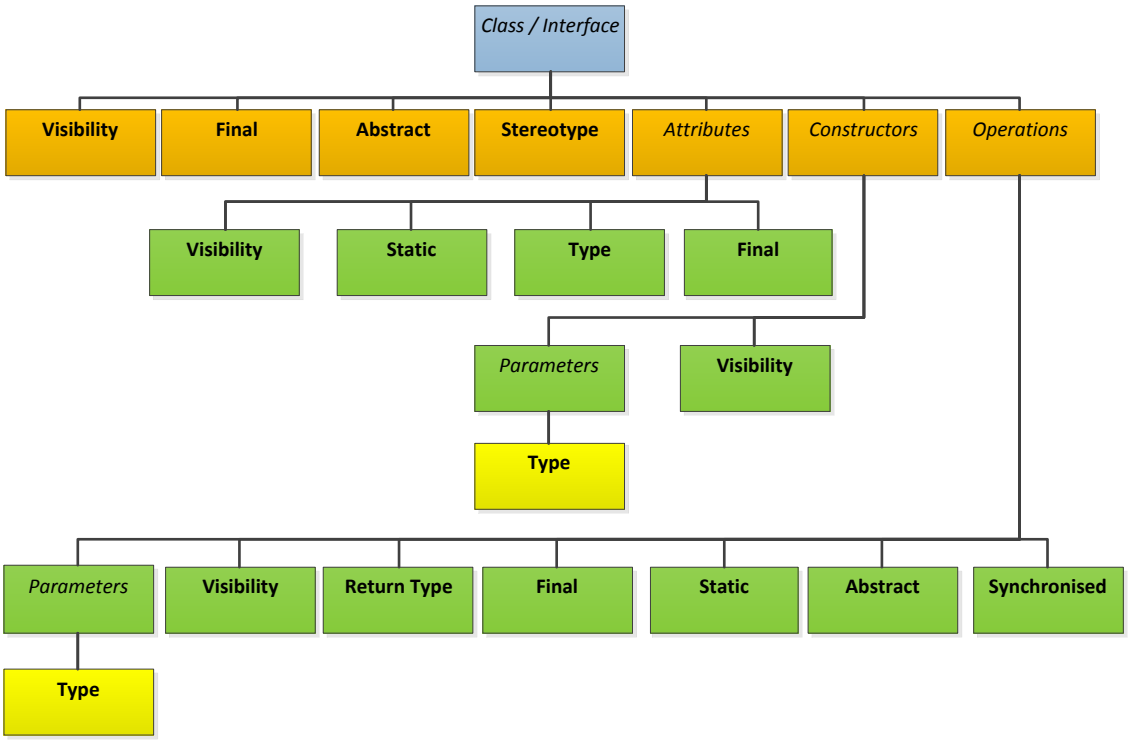

Figure 8 - Hierarchical structure of composing elements of a class (names omitted)

Any leaf feature (one which has no sub-feature) can only occur once in its containing feature (e.g. an attribute can have only one type or an operation can have only one visibility. However, any of the containers can be repeated (e.g. a class can have multiple attributes or an operation can have multiple parameters). The values of leaf features state something about their containing feature overall, for instance that a class is abstract and has public visibility. However, another relevant aspect of a class's similarity to

other classes is its complexity. An indication of this is the multiplicity of features; the number of attributes, constructors and operations, and, at a lower level, the number of constructor and operation parameters.

In addition to the internal hierarchical structure of a class presented above, there are further elements obtained from a class diagram, which can be used for measuring software design similarity based on structure. These are all related to how a class relates to other classes in the class diagram:

- Inheritance: a class's super/base class
- Interface realisation (implementation): the number of interfaces a class implements
- Associations: the number of other classes that a class cooperates with

Many features are compared in order to establish the overall similarity between two classes. It is quite possible that different features contribute to the overall similarity in uneven shares. To account for uneven contribution, a weighting scheme is used to assign weights proportionally. This way, the weights can be adapted individually to reflect the importance of the feature.

The following equation shows how the similarity between two classes (target and source) is calculated.

$$\sigma(C_t, C_s) = \omega_{hl}\sigma_{hl}(C_t, C_s) + \omega_{ll}\sigma_{ll}(C_t, C_s) \quad \textbf{(2)}$$

The overall similarity combines the low-level and the high-level similarity between the two given classes, where ω_{hl} is the weight coefficient applied to the high-level similarity and ω_{ll} is the weight applied to the low-level similarity

The high-level similarity between two classes is based on the defining characteristics of a class (its modifiers) and the number of attributes, operations, constructors and relationships with other classes (superclass, number of associations and interface realisations). These are properties of a class which can be obtained without having to look at the detailed low-level structure of the class. In essence it looks at anything that is at the first level of a class's hierarchical structure.

The high-level similarity between two classes is obtained as follows:

$$\begin{aligned}\boldsymbol{\sigma_{hl}(C_t, C_s)} = &\sum_{\substack{\boldsymbol{all} \\ \boldsymbol{mods}}} \boldsymbol{\omega_{mod}\sigma_{mod}(C_t, C_s)} + \boldsymbol{\omega_{super}\sigma_{super}(C_t, C_s)} \\ &+ \boldsymbol{\omega_{numAttr}\sigma_{numAttr}(C_t, C_s)} \\ &+ \boldsymbol{\omega_{numOpr}\sigma_{numOpr}(C_t, C_s)} \\ &+ \boldsymbol{\omega_{numConstr}\sigma_{numConstr}(C_t, C_s)} \\ &+ \boldsymbol{\omega_{numAssc}\sigma_{numAssc}(C_t, C_s)} \\ &+ \boldsymbol{\omega_{numImpl}\sigma_{numImpl}(C_t, C_s)}\end{aligned} \quad \textbf{(3)}$$

where *all mods* are the modifiers of a class, $\omega_{mod}$is the weight applied to the similarity of each of these modifiers, ω_{super} is the weight applied to the similarity between the super/base class, $\omega_{numAttr}$ is the weight assigned to the number of attributes and similarly ω_{numOpr}, $\omega_{numConstr}$, $\omega_{numAssc}$ and $\omega_{numImpl}$ refer to the number of operations, constructors, associations and interface realisations. The modifiers of a class are visibility, abstract, final and stereotype (can be interface or enumeration).

The low-level similarity is concerned with the internal structure of a class (attributes, operations and constructors), as stated in equation (4), where ω_{attr} is the weight applied to the sum of all matched attributes' similarity, $n_{attrMatches}$ refers to the total number

of attribute matches and A_x, A_{bmx} refers to an attribute and its best match (bmx). Similarly, ω_{opr} refers to operations and ω_{constr} to constructors.

$$\begin{aligned}
&\boldsymbol{\sigma_{ll}(C_t, C_s)} \\
&= \boldsymbol{\omega_{attr}} \frac{\mathbf{1}}{\boldsymbol{n_{attrMatches}}} \sum_{\substack{\boldsymbol{all} \\ \boldsymbol{matched} \\ \boldsymbol{attrs}}} \boldsymbol{\sigma_{attr}(A_x, A_{bmx})} \\
&+ \boldsymbol{\omega_{opr}} \frac{\mathbf{1}}{\boldsymbol{n_{oprMatches}}} \sum_{\substack{\boldsymbol{all} \\ \boldsymbol{matched} \\ \boldsymbol{oprs}}} \boldsymbol{\sigma_{opr}(O_x, O_{bmx})} \\
&+ \boldsymbol{\omega_{constr}} \frac{\mathbf{1}}{\boldsymbol{n_{constrMatches}}} \sum_{\substack{\boldsymbol{all} \\ \boldsymbol{matched} \\ \boldsymbol{constrs}}} \boldsymbol{\sigma_{constr}(Ct_x, Ct_{bmx})}
\end{aligned} \tag{4}$$

The similarity between two attributes is defined as follows:

$$\begin{aligned}
\boldsymbol{\sigma_{attr}(A_t, A_s)} = &\sum_{\substack{\boldsymbol{all} \\ \boldsymbol{mods}}} \boldsymbol{\omega_{mod} \sigma_{mod}(A_t, A_s)} \\
&+ \boldsymbol{\omega_{type} \sigma_{type}(A_t, A_s)}
\end{aligned} \tag{5}$$

The modifiers in this case are static, final and visibility. ω_{type} refers to the weight assigned to the data type of an attribute.

When calculating the similarity between two operations, the modifiers specific to each operation, the number of parameters and the similarity between these parameters are all taken into consideration:

$$\begin{aligned}
&\boldsymbol{\sigma_{opr}(O_t, O_s)} \\
&= \sum_{\substack{\boldsymbol{all} \\ \boldsymbol{mods}}} \boldsymbol{\omega_{mod}\sigma_{mod}(O_t, O_s)} + \boldsymbol{\omega_{retType}\sigma_{retType}(O_t, O_s)} \\
&+ \boldsymbol{\omega_{par}} \frac{\mathbf{1}}{\boldsymbol{n_{parMatches}}} \left(\sum_{\substack{\boldsymbol{all} \\ \boldsymbol{matched} \\ \boldsymbol{pars}}} \boldsymbol{\sigma_{par}(Par_x, Par_{bmx})} \right) \\
&+ \boldsymbol{\omega_{numPar}\sigma_{numPar}(O_t, O_s)}
\end{aligned} \tag{6}$$

For operations the modifiers are visibility, static, synchronized, abstract and final. *retType* is the return type of a method, $n_{parMatches}$ is the number of parameter matches and *numPar* is the number of parameters.

Similarity metric (7) is applied to constructors. It is very similar to the one for operations, but has only a single modifier (visibility).

$$\begin{aligned}
&\boldsymbol{\sigma_{constr}(Ct_t, Ct_s)} \\
&= \boldsymbol{\omega_{vis}\sigma_{vis}(Ct_t, Ct_s)} \\
&+ \boldsymbol{\omega_{par}} \frac{\mathbf{1}}{\boldsymbol{n_{parMatches}}} \sum_{\substack{\boldsymbol{all} \\ \boldsymbol{matched} \\ \boldsymbol{pars}}} \boldsymbol{\sigma_{par}(Par_x, Par_{bmx})} \\
&+ \boldsymbol{\omega_{numPar}\sigma_{numPar}(Ct_t, Ct_s)}
\end{aligned} \tag{7}$$

For operation and constructor parameters, similarity is defined simply as:

$$\boldsymbol{\sigma_{par}(Par_t, Par_s) = \omega_{type}\sigma_{type}(Par_t, Par_s)} \tag{8}$$

For most modifiers the similarity is easy to calculate as they are either present or absent, this means that it will be either a 100% or a 0% match. This applies to:

- abstract
- final
- static
- synchronised

Visibility (access level) modifiers control what members (attributes or operations) can be accessed by other members. The accessibility of members will depend on the relative position of the class containing them and the class trying to access them. For instance, whether both members are in the same class, the accessing member is in a class that inherits from the class containing the accessed member or whether both classes are in the same namespace/package. The different modifiers apply different restrictions with private being the most restrictive and public the least. This makes it possible to use expert knowledge to establish distances between the different visibility modifiers and these distances determine the similarity. Table 2 outlines the similarities used between the various visibility modifiers:

	Public	Protected	Internal	Private
Public	100%			
Protected	60%	100%		
Internal	50%	80%	100%	
Private	0%	20%	40%	100%

Table 2 – Similarity of Visibility Modifiers

As with visibility modifiers, data type similarity can be matched in part by establishing distances between various data types. For instance, the integer and float data types both

share the characteristic of being numerical. While it is not an exact match, it is conceptually closer than a numerical data type and a string data type. Table 3 shows a matrix with the similarities of data types and return types. The similarity values were established based on expert opinion.

	Bool	Byte	Short	Int	Long	Float	Double	Char	String	Object	Array	Collect.	Void
Bool	100%												
Byte	10%	100%											
Short	10%	80%	100%										
Int	10%	70%	80%	100%									
Long	10%	60%	70%	80%	100%								
Float	5%	30%	40%	50%	50%	100%							
Double	5%	30%	40%	50%	50%	80%	100%						
Char	5%	5%	5%	5%	5%	5%	5%	100%					
String	10%	10%	10%	10%	10%	10%	10%	60%	100%				
Object	0%	0%	0%	0%	0%	0%	0%	0%	20%	100%			
Array	0%	0%	0%	0%	0%	0%	0%	0%	10%	30%	100%		
Collect.	0%	0%	0%	0%	0%	0%	0%	0%	0%	30%	80%	100%	
Void	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%

Table 3 – Similarity of Data Types and Return Types

The similarity between an interface and an abstract class was set at 50% as both contain abstract operations.

To measure the similarity between two numerical values, such as the number of attributes, operations, parameters, etc., the metric used was:

$$\sigma(x, y) = \frac{x}{y} \quad (9)$$

where x is the smaller and y is the larger of the two numbers.

The following rule is applied when comparing features of different classes: *the lack of features in both classes being compared would denote a 100% similarity for that feature, while the absence in merely one class would result in a 0% similarity*. For

example, if neither class has any attributes, the overall attribute similarity is 100%, while if one class contains attributes, but the other does not, 0% similarity would be applied.

Not all of the modifiers available in different object-oriented languages are included in the feature set used in this approach. The focus was on those features that are included in a class diagram and can be extracted from it. This meant that modifiers such as transient, volatile or native were not included. Furthermore, only features which could be obtained using reverse-engineering from a class diagram are used. So labels on associations, relationship multiplicities, aggregation, composition and constraints are not included.

The structure created to represent a class is quite extensive and includes even small internal structural features. An alternative approach to this could have been to include only the most obvious features when measuring similarity. The system implemented to evaluate the research therefore allows individual features and even entire levels of the hierarchy to be excluded from the similarity metrics.

3.4 Weight Optimisation & Genetic Algorithms

Expert input is being used in the weight optimisation process at three levels of granularity.

1. Determine individual weights for every feature in the similarity metric
2. Set desired class matches within a diagram by setting scores – weights are determined by applying a genetic algorithm to obtain the highest possible overall score
3. Set desired diagram matches by selecting the best-matching diagram - weights are determined by applying a genetic algorithm to obtain the highest possible match between the given diagrams

As discussed earlier, each feature within a class may contribute to the overall similarity between two classes in uneven shares. This requires a weighting scheme to be in place in order to assign the weights proportionally to a feature's importance in terms of the overall class similarity. The features of a class are arranged in a hierarchical structure, which is reflected by the weights, creating different levels of weights.

A key issue in calculating the class similarity effectively is to identify what the weight setting should be in order to successfully match up the correct classes. There is no established norm or convention for measuring similarity between classes. This is an abstract activity and even stating whether a match is good or bad is not always straightforward. A human expert could identify similarities between given designs using a heuristic approach, thus the key lies in classifying the characteristics that would make a human expert identify the similarity and adjust the weights for those features accordingly. However, this could vary from person to person or even diagram to

diagram. It is also possible that a human expert would measure some features intuitively without being able to express them in rules.

In order to solve this problem this research employs weight optimisation. This process can be automated by training a system to automatically identify optimal weight settings, given the desired matching results. An expert assigns values from a predefined scale to class matches (similar to a grading scheme).

Given a set of desired/undesired matches, the system can apply different weight settings, run comparisons between all the different classes and keep a score of the points obtained from matching the classes against one another. An analysis of the scores from these comparisons makes it possible to adopt the best weight settings. This weight optimisation algorithm is outlined in Figure 9.

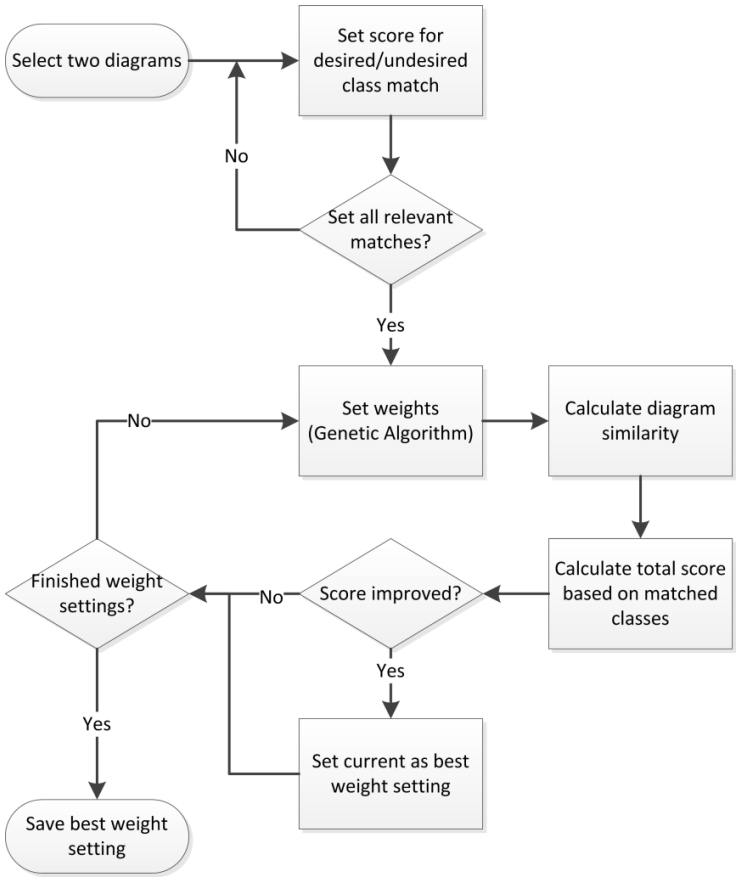

Figure 9 - Weight Optimisation Algorithm using Class Scores

A slightly different approach to this weight optimisation problem asks the expert to identify the most similar diagrams from a set (see Figure 10). The system applies different weight settings and adopts the setting which yielded the highest result.

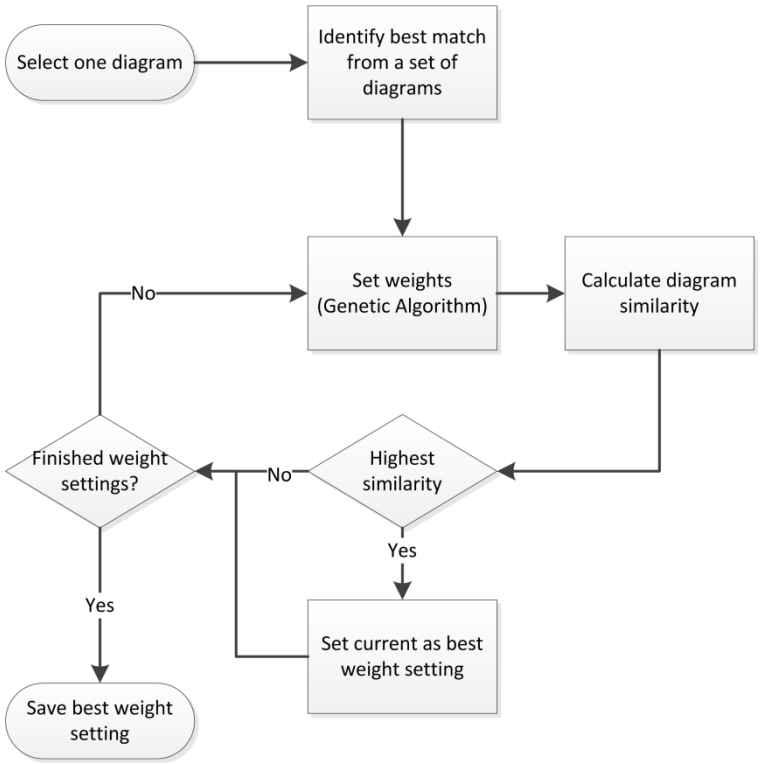

Figure 10 - Weight Optimisation Algorithm using Diagram Matching

An important question is to what extent the optimisation of weights can be generalised. Can a weight setting obtained from a particular set of designs be applied to other cases?

According to [Mitchell, 1990] it is not possible to use a weight optimisation method that would obtain an optimal weight setting which could be used for all tasks, since each task requires a different bias for optimal performance. A possible solution to this would be the creation of several different profiles according to the specific characteristics of different designs. This issue will be discussed in more detail in chapter 5.

Rather than randomly changing weights and hoping to chance upon a good setting, it would be more efficient to try to improve a setting by incrementally manipulating the values, verifying the results and making adjustments accordingly. A way of achieving this is through means of a genetic algorithm (see Figure 11).

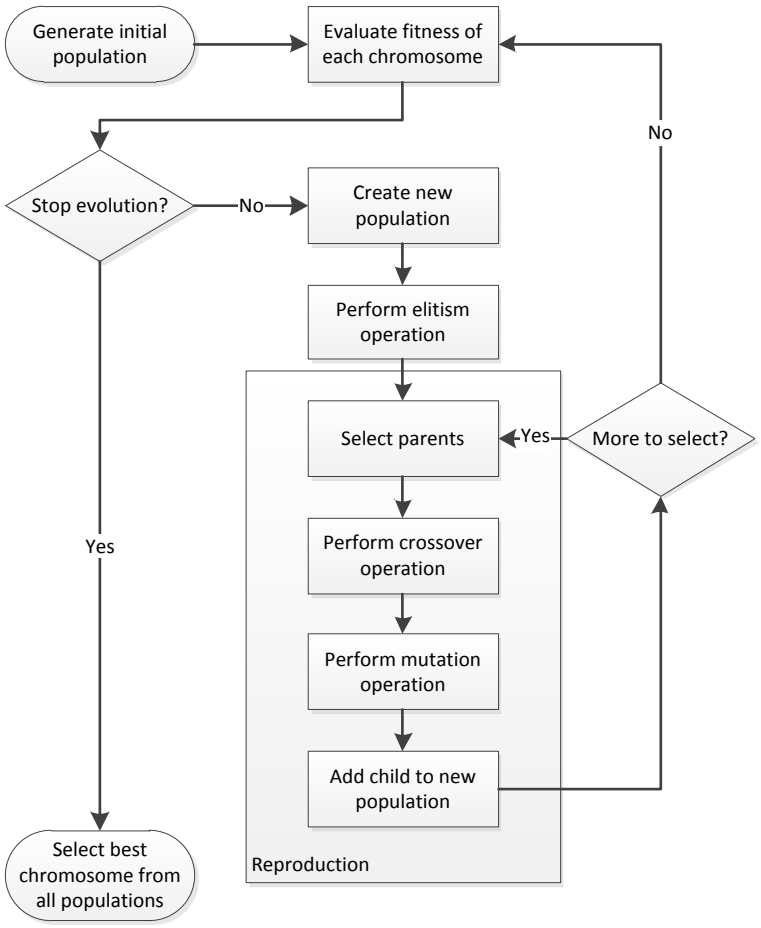

Figure 11 - Genetic Algorithm

The genetic algorithm is used in the *Set Weights* phase of the weight optimisation algorithms (Figure 9 and Figure 10), effectively automating the weight setting.

A population contains a set of chromosomes and in this case each chromosome represents a weight setting. A chromosome's fitness shows the similarity or score achieved when using this particular set of weights. The number of elite chromosomes

which are automatically selected for the next population is determined by an elitism rate. The crossover and mutation operations are not performed in every reproduction cycle, but their occurrence is dictated by a random function taking into consideration a set rate for each. During the reproduction phase, the selection of parents is not completely random, but is biased in function of a chromosome's fitness. This means that better chromosomes have a higher probability of being selected – which is referred to as fitness proportionate selection.

The probability of a chromosome being selected is defined as:

$$\boldsymbol{p_i = \frac{f_i}{\sum_{j=1}^{N} fj}} \quad \textbf{(10)}$$

Where f_i is the fitness of a chromosome and *N* is the number of chromosomes in the population.

The weight settings obtained from optimisation are cached for later retrieval so that they don't have to be computed at query time.

3.5 Graph Matching

To measure the similarity between two class diagrams, one can simply measure the similarity between the number of classes in both diagrams and sum of the similarities of all matched classes (11).

$$\begin{aligned}&\boldsymbol{\sigma(CD_t, CD_s)}\\&\boldsymbol{= \omega_{attr} \frac{1}{n_{classMatches}} \sum_{\substack{all\\matched\\classes}} \sigma_{class}(C_x, C_{bmx})}\\&\boldsymbol{+ \omega_{numClass} \sigma_{numClass}(CD_t, CD_s)}\end{aligned} \quad \textbf{(11)}$$

Where C_x is a class from CD_t and C_{bmx} is the best match for that class in CD_s.

However, a great deal of the knowledge associated with UML models is encoded in the links between these elements. Typical of this are the associations between classes in a UML class diagram and the message passing in interaction diagrams. In fact it can be argued that most of the practical reuse of design and code by software engineering practitioners is associated with design patterns that are related to patterns of interaction between objects.

As a UML class diagram consists of classes which are connected using relationships, they can easily be represented as graphs of nodes (classes) and arcs (relationships). To measure similarity between class diagrams, represented as graphs, requires graph matching. A full search graph matching algorithm has been adapted to be applied to UML class diagrams. Given the graph representations of two UML class diagrams, CD_t and CD_s, the algorithm returns the Maximum Common Subgraph $MCS(CD_t,CD_s)$ present in both graphs.

The algorithm attempts to find the best matching elements in the two graphs based on the metrics presented in the previous sections. The similarity metric between arcs in the diagram is based on a simple classification of association types in terms of the nature of their relationship as follows:

- association
- generalisation
- interface realisation (implementation)

The similarity between associations of the same type is defined as:

$$\sigma(As_t, As_s) = \begin{cases} 1 & type(As_t) = type(As_s) \\ 0 & type(As_t) \neq type(As_s) \end{cases} \quad \textbf{(12)}$$

The graph matching algorithm maps the maximum common connected subgraph based on matches between individual elements of each graph, where a minimum threshold similarity value is satisfied.

The overall similarity between the two case graphs (G_t, G_s) is then defined as:

$$\sigma(G_t, G_s) = \frac{(\sum_{\substack{matches \\ C_t, C_s \\ in \\ MCS}} \sigma(C_t, C_s))^2}{count(G_t).count(G_s)} \quad \textbf{(13)}$$

where $count(G_t)$ represents the number of nodes (classes) in graph G_t.

The sum of results is squared to emphasise the differences between graphs and make it easier to visualise results. This formula for graph matching has been used successfully

in previous research at the University of Greenwich [Petridis, et al., 2007a; Petridis, et al., 2007b].

Figure 12 shows an example application of the algorithm and similarity measure. At a similarity threshold set to 0.4, three connected nodes in each graph have been picked up by the algorithm. A further match between the classes "Customer" and "POS" is rejected for having similarity value less than the threshold. Potential matches between "Supplier" and "Saleable" are not considered as the connecting associations are of different types (association vs. interface realisation).

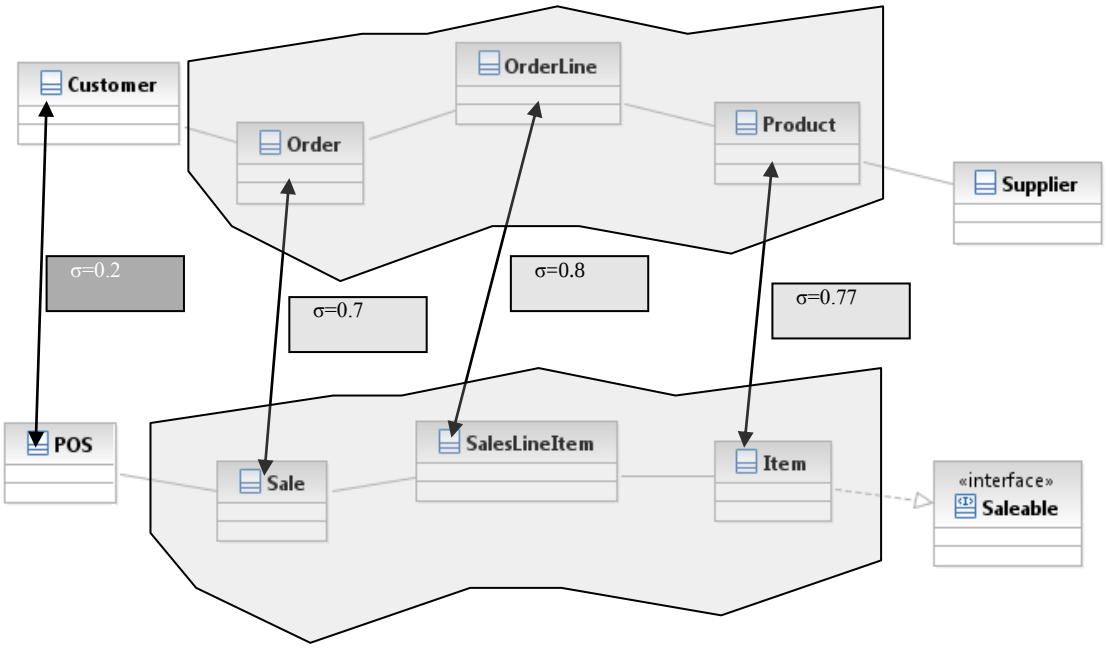

Figure 12 - Maximum Common Subgraph Example

In this example, the overall graph similarity between the two class diagrams would be:

$(0.7+0.8+0.77)^2 / (5*5) = 0.206$

The algorithm used in this research is based on the full recursive search of all elements in the graph representations where both target and source diagrams are being compared. The algorithm attempts to match individual classes from the source and target diagram

where the similarity is greater than a predefined threshold. For each successful match, the algorithm attempts to match all respective matches of neighbours whose similarity measure is greater than the threshold. The algorithm finishes when no additional neighbour matches can be found. This algorithm has been extended from previous research [Petridis, et al., 2007b] to follow links representing only similar types of associations as defined by equation (12) above.

Figure 13 outlines a simplified version of the algorithm used to identify the MCS between two graph representations of corresponding class diagrams.

The graph matching algorithm has been used as an alternative to the simpler class diagram matching algorithm presented at the beginning of this section (11).

```
FORALL possible matches (x,y)
   mcs ← CALL match(x,y)
ENDFOR


match(x,y):
   calculate struct similarity sim(x,y)
      IF sim(x,y) > threshold THEN
         find all matching neighbour(x_n,y_n)
         FOR each (x_n,y_n)
               mcs ← Append(mcs,match(x_n,y_n))
               IF mcs best so far THEN
                 retain mcs
               ENDIF
         ENDFOR
      ENDIF
      RETURN mcs
```

Figure 13 - Maximum Common Subgraph Matching Algorithm

3.6 Clustering

Given that clustering makes it possible to group objects that are similar, it can be applied in order to evaluate whether the similarity metrics applied in this research are working. Clustering is used as part of the experiment stage in order to validate the results, whereby clusters of the cases are generated post retrieval and evaluated against facts which are known, but don't form part of the similarity metrics. For instance, clustering is being used to determine whether cases are grouped according to the source, programming language, size or quality of the software designs.

Clustering is an area of data mining which consists of the assignment of objects into groups (clusters) such that objects in the same cluster are more similar. Given that the essence of CBR is measuring similarities, it is sensible to combine the two areas. It is possible to find applications for clustering at various stages in the CBR cycle, but in this case it is applied to the set of retrieved and ranked cases.

Agglomerative hierarchical clustering takes as input a set of objects each of which is placed in its own cluster. The clustering algorithm then repeatedly merges the closest pair of clusters.

The distances between clusters can be defined using:

- Single-link clustering (nearest neighbour) – merges clusters with the smallest minimum distance from any member of cluster *A* to any member of cluster *B*
- Complete-link clustering (farthest neighbour) – merges clusters with the smallest maximum distance from any member of cluster *A* to any member of cluster *B*
- Average-link clustering – merges clusters with the smallest average distance from any member of cluster *A* to any member of cluster *B*

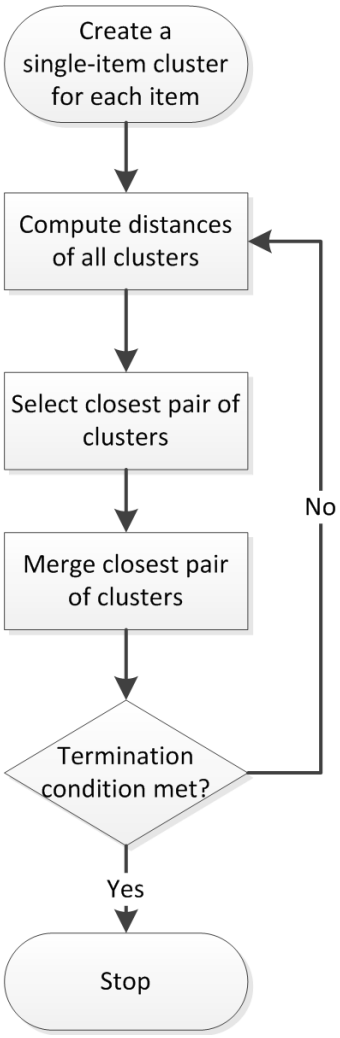

Figure 14 – Agglomerative Hierarchical Clustering Algorithm

The single-link clustering method is known to be unsuitable for isolating poorly separated clusters. In this research it was found that single-link clustering would result in one massive cluster containing the large majority of items and all remaining clusters each containing a single item. Better results were achieved using average-link clustering, which has also ranked well in evaluation studies [Sileshi and Gamback, 2009]. A disadvantage of the average-link clustering method is that it is computationally expensive for large collections of data, but it computed all clusters within a reasonable time.

The condition for terminating the clustering process could be when a predetermined number of clusters is reached or when a cluster fails its criteria of compactness (average distance between items in the clusters are too high). The predetermined number of clusters was used, which was based on a rule of thumb by Mardia et al. [Mardia, et al., 1980] and the number of different sources of the cases in the case-base.

Clustering is an active area of research in its own right and experiments with more complex clustering algorithms could have been carried out, such as the approach of Bai et al. [Bai, et al., 2011] who use multiple weights for attributes within clusters in order to identify clusters in subspaces. However, the focus of this research is not on clustering methods. Clustering is used as a means to evaluate the effectiveness of knowledge retrieval from software designs.

3.7 Cost Estimation

Software development projects have a notorious reputation for being late and over budget, yet much of the cost estimation for projects is done based on experience of project managers and estimates done on the "back of an envelope". The estimation process also needs to account for numerous factors, such as system size, complexity, experience of the development team, etc. There is also a tendency to create very optimistic estimates, as this makes it more probable for a project to be approved or attract funding.

As with any type of estimation, cost estimation works most efficiently when backed by historical and statistical information. Thus, a variety of different approaches exist to software cost estimation, which take into account historical and statistical information, or which use models which were created using historical data. Some are based on

formal models, such as parametric or size-based models, while others rely on expert estimation. The most common approaches are formal estimation models, such as COCOMO (COnstructive COst MOdel), established by Boehm [Boehm, 1981], which apply formulas derived from historical data. The successor to COCOMO is COCOMO II [Boehm, et al., 2000b] which is better suited for modern software development projects, as it considers the changes to the software development process since the first version was published.

COCOMO uses function point analysis to compute program size, development time and the number of people required. It requires the classification of cost drivers using a scale of ratings to determine information such as the class of project. It also requires setting complexity factors and weight assignment, all of which must be performed by a human expert. This means that the estimation is very much dependent on the judgement of the expert and remains prone to human errors and biases [Valerdi, 2007].

When using the object-oriented paradigm and UML models, the function point analysis uses use cases (use-case diagram) and classes (class diagram) in order to compute the effort in lines of code. The following formula is used to calculate function points for cost estimates:

$$\boldsymbol{FP = F_u * \omega_u + F_c * \omega_c} \quad \textbf{(14)}$$

where F_u is the number of use cases directly connected to an actor (i.e. use cases which are connected not solely to other use cases), ω_u is a weight between 4 and 7, F_c is the total number of classes in the model and ω_c is a weight between 7 and 15.

The function points obtained using this calculation are then adjusted using the formula:

$$\boldsymbol{FP_{adjusted} = FP * (0.65 + 0.01 * \sum f_i)} \quad \textbf{(15)}$$

where f_i are a set of fourteen factors, each of which is assigned a complexity coefficient between 0 (no influence) and 5 (essential). The technical complexity factors defined by COCOMO are:

1. System requires reliable backup and recovery
2. Data communications required
3. Distributed processing functions
4. Performance critical
5. System to run in existing, heavily utilized operational environment
6. Requires on-line data entry
7. On-Line data entry requires input transaction over multiple screens/transactions
8. Master files updated on-line
9. Inputs/outputs/queries complex
10. Internal processing complex
11. Code designed to be reusable
12. Conversion and installation included in design
13. System designed for multiple installations in different organisations
14. Application designed to facilitate change and ease of use by the user

Once the function points are calculated, it is possible to convert them to lines of code (LOC) using a mapping. The mappings for some programming language are given as:

Programming Language	LOC per FP
Assembly language	320
C	128
Cobol / Fortran	105
Pascal	90
PHP / Python	67
Java / C++ / C#	30
Code generators	15

Table 4 – Mapping of Function Point to Lines of Code

The technical complexity factors, function point mappings and formulas in COCOMO are all derived from the analysis of historical data.

A major motivation of this research is to determine whether cost estimation can be automated effectively by comparing software designs. This is an approach which uses estimation by analogy, as experience from previous projects is analysed to determine the cost for a new one. Given a collection of software designs and corresponding cost it took to implement each design, e.g. in lines of code, it may be possible to predict the effort necessary to implement a new target design, provided that the implementation shares some common context, such as, complexity of the functionality, similar programming languages and technologies.

In this research the results of COCOMO estimates for lines of code of a target software system are compared to the lines of code obtained from the nearest neighbours retrieved.

3.8 Explanation

Using the methods described in the previous sections of this chapter, it is possible to compute the similarity of two software designs and obtain a match expressed in percentage. This result would be calculated based on the maximum common subgraph, number of classes in each diagram, similarity of the internal hierarchical structure of the classes that form part of the maximum common subgraph and the weights attributed to all of the features. It is also possible to indicate which features were ignored in the comparison, if any. The reasoning used to obtain the results is remembered and can be shown to a user to explain the overall similarity match by breaking it down into each constituent element. It is also possible to show how groups of features have contributed, such as a particular level in the hierarchical structure (e.g. what the overall match for all attributes was for two particular classes).

The explanation is achieved by providing a complete breakdown of all similarities between all features of the two diagrams and is supported by the visualisation of the maximum common subgraph, showing exactly what classes were matched and how they were linked.

3.9 Conclusion

This chapter has presented the methodologies and techniques that were applied in this research. The research relates to a variety of subject areas and the chapter started by introducing the case-based reasoning paradigm and explained how it can be used in order to extract knowledge from software design artefacts.

Most of the research on applying case-based reasoning to software designs relies heavily on the semantic information encoded in the designs to measure similarity. While

the importance of semantic aspects has been clearly demonstrated, the significance of the hierarchical structure has been raised together with the question of whether the effective retrieval based solely on structural information was possible.

It has been demonstrated how class diagrams can be broken down into composing features resulting in a hierarchical tree structure with different levels. All the contributing structural features have been presented, along with their similarity metrics. As a consequence of the uneven contribution to overall similarity, feature weighting has been discussed and it was shown how a genetic algorithm can be employed to optimise feature weighting.

To take advantage of the knowledge determined by the relationships between classes, a class diagram can be regarded as a graph, which requires some form of graph matching to measure similarity. The method proposed for finding the maximum common subgraph and a similarity metric and algorithm were provided. It was shown how clustering can be used to group cases to evaluate the effectiveness of the knowledge retrieval. Retrieval of knowledge is applied in an attempt to perform software cost estimation. Existing methods for cost estimation were presented and it was revealed how this research will evaluate results against the COCOMO model for estimating lines of code. Finally, it was outlined how the system will provide explanation to its users.

The next chapter will present UMLSimilator, which is the software system developed to evaluate the research.

Chapter 4

4 UMLSimilator

The previous chapter outlined the techniques and methodologies used. This chapter explains how the algorithms were implemented and provides examples to illustrate. The UMLSimilator tool is presented, which is the software tool which was developed in order to evaluate the research. The tool is capable of importing cases, measuring similarity and applying all of the techniques discussed in the previous chapter.

This chapter provides a synopsis of work carried out and starts by presenting the case study which forms the basis of the work. The next section provides the overall architecture of the system, indicating the various modules it contains. The various modules are then presented in turn, describing their specific function within the knowledge retrieval process.

4.1 Case Study

A case-based reasoning system requires a case-base or case-library. As the purpose of the application is to retrieve knowledge from structural information of software design artefacts, the content of the case-base consists of representations of UML class diagrams. To ensure that there is enough data and variety within the case-base, just over one hundred cases were obtained and stored. Diversity was introduced by using results from a number of different teaching assignments, thus assuring a good assortment of cases. Another positive consequence of using a set of teaching assignments is the fact that it provides diverse implementations of identical requirements, making it possible to validate results and evaluate provenance. Finally, this approach provided a measure of

quality in form of the grade achieved. Having this quality indicator enables the investigation of the relationship between the structure and quality of a software design.

To maximise the potential for knowledge retrieval, it is advantageous to have not just a set of software designs, but to obtain the designs from actual implementations. This makes it possible to verify the findings of this work in relation to cost estimation and plagiarism detection. It also implies that the designs are realistic as they reflect real software solutions.

Application	Programming Language	Level	Number of Cases
Software catalogue application	.NET	6	25
Project management application	.NET	6	14
Application for a car repair shop	.NET	5	34
Stock management application	.NET / Java (optional)	7	21
Project bidding system	.NET	6	7

Table 5 – Teaching Assignments in the Case-Base

Table 5 shows details of the assignments which were used to populate the case-base. As can be seen, each assignment has different requirements and they are sourced from different levels. All assignments had to be completed using an object-oriented language.

For each assignment, the grade has been recorded, given that this is a measure of quality of the finished software product. A high grade doesn't necessarily guarantee a good design, as the grade is generally derived from a number of factors, which include, but are not limited to the design. These could include the graphical user interface design, functionality offered, validation, accompanying written report, etc.

The approach to cost estimation applied here computes the estimated cost (lines of code) by identifying the nearest neighbour designs and calculating the average cost of

their implementation. This requires knowing the cost of implementation for the designs in the case-base. To automate the calculation of implementation cost, a source code reader module was built into the UMLSimilator tool. The source code reader allows a user to map classes to source files and then automatically extracts the lines of code and number of characters for each class. The .NET suite of programming languages allow partial classes, where one class is spread over multiple files. The source code reader caters for this by allowing multiple files to be mapped to the same class.

4.2 System Architecture

The UMLSimilator tool implements the proposed approach and aims to support the entire knowledge retrieval process using CBR, including the management of the case-base. The tool comprises a variety of modules to support this functionality. An overview of the architecture of the tool and its main modules has been outlined in Figure 15. The modules include:

- *UI Forms:* Contains the majority of graphical user interfaces, as well as a controller and some file handling support
- *Visualiser:* A user interface component implemented using Windows Presentation Foundation to depict class diagrams and visualising maximum common subgraphs
- *Class Similarity Calculator:* Performs the internal class similarity calculations using the hierarchical structure, including data type similarity
- *Graph Matching:* Implements the most complex algorithm of the application which uses recursion to match the maximum common subgraph
- *Clustering:* Used for clustering cases based on their distances

- *Weight Optimiser:* Supports different types of weight optimisation based on desired settings from a human expert. Uses a genetic algorithm to improve the weight settings
- *.NET Reflector:* Reverse engineers class diagrams from compiled .NET CIL code in *.exe or *.dll format
- *Java Reflector:* Reverse engineers class diagrams from compiled Java byte code in *.class or *.jar format
- *Persistence:* Handles all persistence operations with the case base

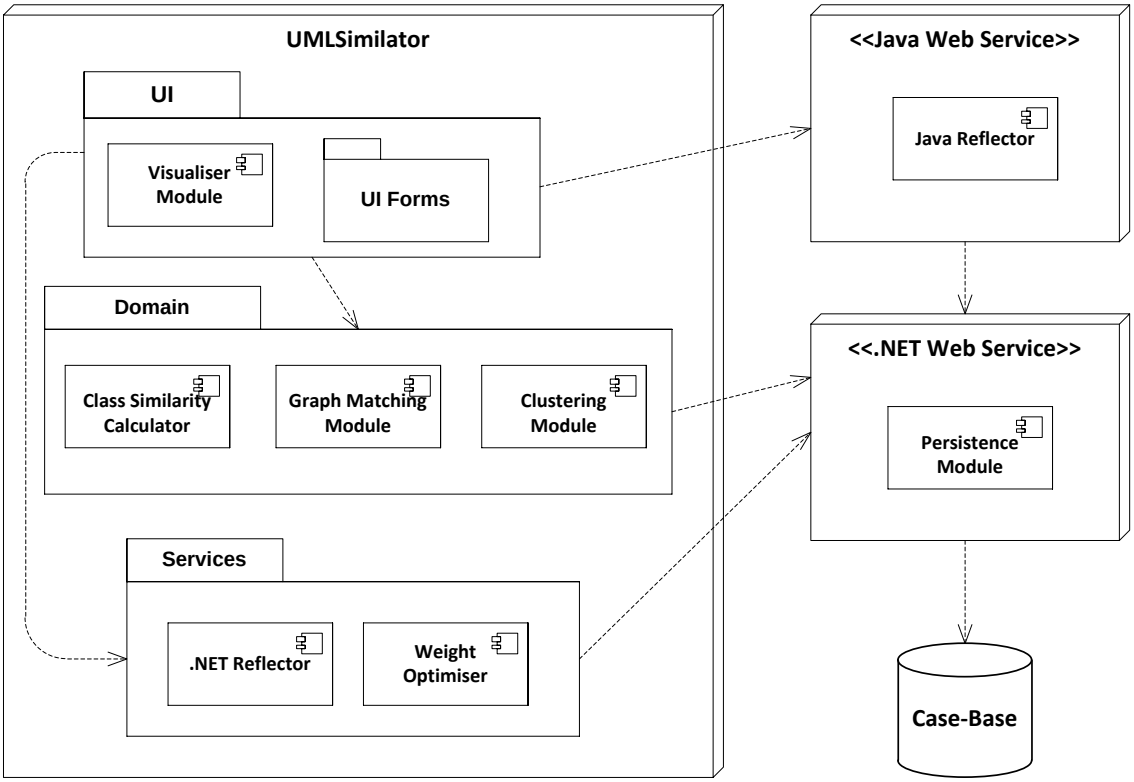

Figure 15 - UMLSimilator Architecture

The UMLSimilator tool also incorporates the case-base and its management. Initially XML (eXtensible Markup Language) was used as a storage mechanism for the case-base to provide flexibility. XML is well-structured and as a text-based format it is

widely supported and easily transferable, yet capable of representing complex data structures through containment. The use of XMI (XML Metadata Interchange) was considered in order to provide interoperability between UMLSimilator and popular CASE tools. XMI was discussed in chapter 2 with the MOF metamodel architecture. However, text processing is quite slow and it became necessary to speed up the manipulation of data models. A relational database was therefore created holding class representations of the UML metamodel M2 for class diagrams. The database contains 20 tables and the structure can be found in Appendix 1.

4.3 Reflector Module

As discussed earlier, existing implementations were used to populate the case-base. The cases were obtained by reverse engineering existing implementations.

The UMLSimilator contains a class structure reflecting the metadata of UML class diagrams, which makes it possible to work with in-memory representations of the class diagrams, thus speeding up processing. A class diagram showing the core domain classes can be found in Appendix 1. The persistence module deals with the serialisation and de-serialisation of the cases between in-memory representation and the case-base. However, the reflector modules make it possible to introduce the cases in the case-base.

The reflector modules convert classes from their compiled byte code into cases in the case-base. Reflector modules exist for Java, as well as .NET CLS-compliant languages (CLS - Common Language Specification). Both modules rely on the reflective ability of these programming languages, by means of which compiled classes are dynamically loaded at runtime and the features automatically extracted. Reflection enables inspection of classes at runtime, as it makes it possible to construct instances of classes

for which all one has is the name. Once instantiated, the details of all attributes, operations, constructors and parameters can be accessed, as well as metadata, such as annotations and assembly attributes. It is even possible to invoke methods on the object. Reflection makes it possible to obtain the complete hierarchical structure of a class, required to recreate the class diagram. The reverse-engineered designs can then be submitted to the case-base.

Applications written using the .NET suite of languages (e.g. C#, VB.NET) are compiled into an intermediate code called CIL (Common Intermediate Language). The compiled classes are bundled into assemblies, which for stand-alone applications are typically executable files (*.exe) or class libraries (*.dll). The reflector module loads any assembly it is passed, obtains a list of classes, loads each class and reflects on it to extract all of the required data.

The UMLSimilator tool is implemented in .NET, thus the .NET reflector module is directly integrated. Reflection of Java byte code, requires a Java runtime. While the file selection and case-base management is handled by part of the UMLSimilator implemented in .NET, the reflection of Java code is delegated to the Java reflector module, which is exposed as a web service. The case extraction process for .NET and Java code is outlined in Figure 16. The Java reflector module receives Java byte code and recreates files from it. It then loads the files and reflects on them. A custom implementation of the ClassLoader was used to make it possible to load linked classes from multiple nested JAR (Java Archive) files. The structure of the Java Reflector is available in Appendix 1.

Both reflector modules filter out inherited operations and extract only those implemented in the actual class. This corresponds to what would be displayed in a class

diagram. Inheritance and interface realisations can also be easily extracted using reflection. However, one aspect of a class diagram which cannot be obtained directly using reflection is the associations between classes. The reflector modules overcome this problem by identifying classes from the same diagram which are used either as return types of methods, parameter types or data types of declared attributes. Association relationships are then established based on this information.

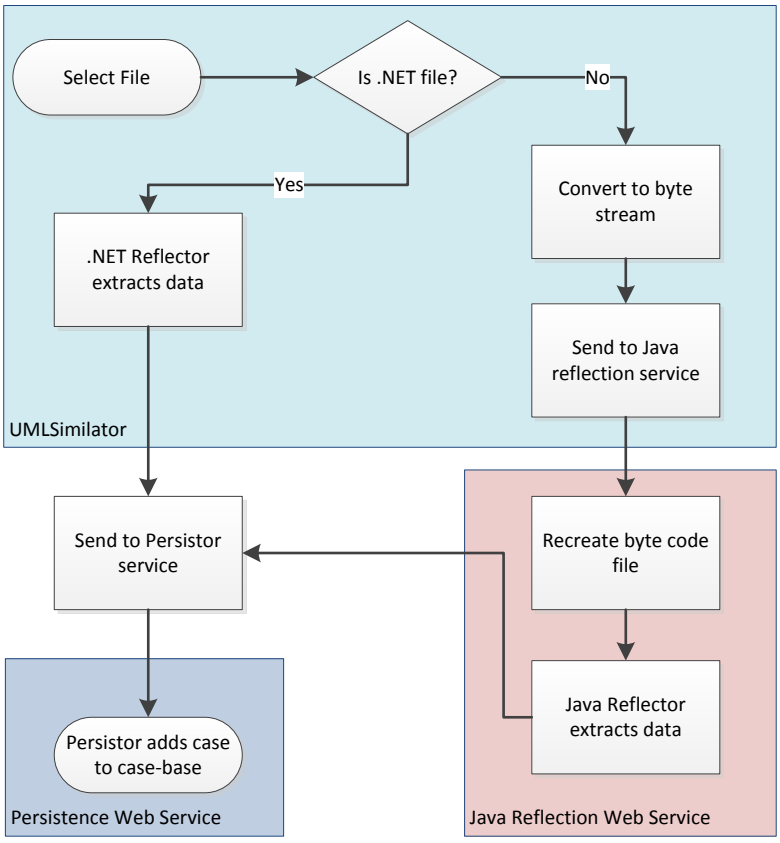

Figure 16 - Case Extraction Process

The main disadvantage of reverse-engineering implemented software solutions is that the software designs may include elements which did not form part of the original design, but were added during the implementation process, such as automatically generated code or existing integrated components. Furthermore, ad-hoc implementations, especially of less-experienced developers, may not have a very

meaningful underlying design. For this reason a manual checking step was added to the case extraction process. During this phase a human expert reviews the reverse-engineered solution and can do manual tidying (e.g. removing automatically generated classes, adding missing relationships, etc.).

Existing tools for reverse engineering code have been used by Gorton and Zhu [Gorton and Zhu, 2005] and Tessem et al. [Tessem, et al., 1998], who use Java's reflective ability combined with Case-Based Reasoning to "retrieve case-based components in a prototyping tool for the Java programming language." This tool is used to aid class retrieval and reuse, but on a code rather than design level and even though this research is based on existing implemented designs, it is currently only concerned with the design level. Their approach is also oriented towards semantic similarity similarly to Gomes et al. [Gomes, et al., 2004].

4.4 Class Similarity Module

At the core of the CBR approach is measuring the similarity between a target and source cases. As presented in the previous chapter, the proposed approach measures the similarity between classes by breaking them down into their composing elements and performing matches of similarity at all levels within the resulting hierarchical structure.

In order to calculate the similarity between two given classes, the UMLSimilator takes every element within the hierarchical structure of the first class and compares it to the equivalent element(s) from the second class. It proceeds until every element is matched and the overall similarity is obtained, not allowing any element to be matched more than once. A full search algorithm is applied which ensures a good overall match. The algorithms in place will repeatedly rematch all elements until satisfactory matching is

achieved. The results of the exhaustive matches are used to measure similarity between classes, thus determining which matches are considered in calculating the maximum common subgraph.

The process of exhaustively matching all elements to find the best matches uses the binomial coefficient $\binom{n}{k}$ which defines the number of ways of picking *k* unordered outcomes from *n* possibilities (see (16)).

$$\binom{\boldsymbol{n}}{\boldsymbol{k}} = \frac{\boldsymbol{n}!}{\boldsymbol{k}!\,(\boldsymbol{n}-\boldsymbol{k})!} \tag{16}$$

This equation uses factorial and is computationally demanding, especially as the number of elements increase. It is also applied on all levels of the element hierarchy, thus a trade-off was made by using a greedy algorithm.

A greedy algorithm follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum [Cormen, et al., 2009]. As applied to this problem, a greedy strategy does not produce an optimal solution, but a greedy heuristic may still yield locally optimal solutions that approximate a global optimal solution. The key is that it may do so in a reasonable amount of time. The algorithm is outlined in Figure 17.

While the greedy algorithm obtains a very good solution, it doesn't exhaust all possibilities and therefore does not guarantee optimum matching. This is due to the fact that by finding the best match for a particular element and then removing it from the pool of available elements, the overall similarity could actually be decreased. A slightly lower initial match could actually result in higher successive ones. However, the

algorithm required to guarantee optimum matching would be too computationally demanding to be feasible, while a heuristic algorithm still provides very good results.

To improve performance, the class similarity module uses caching. This avoids having to repeat the same calculations over and over. When comparing two classes for the first time, the result is calculated and stored in a dictionary. Any successive requests are read straight from the dictionary.

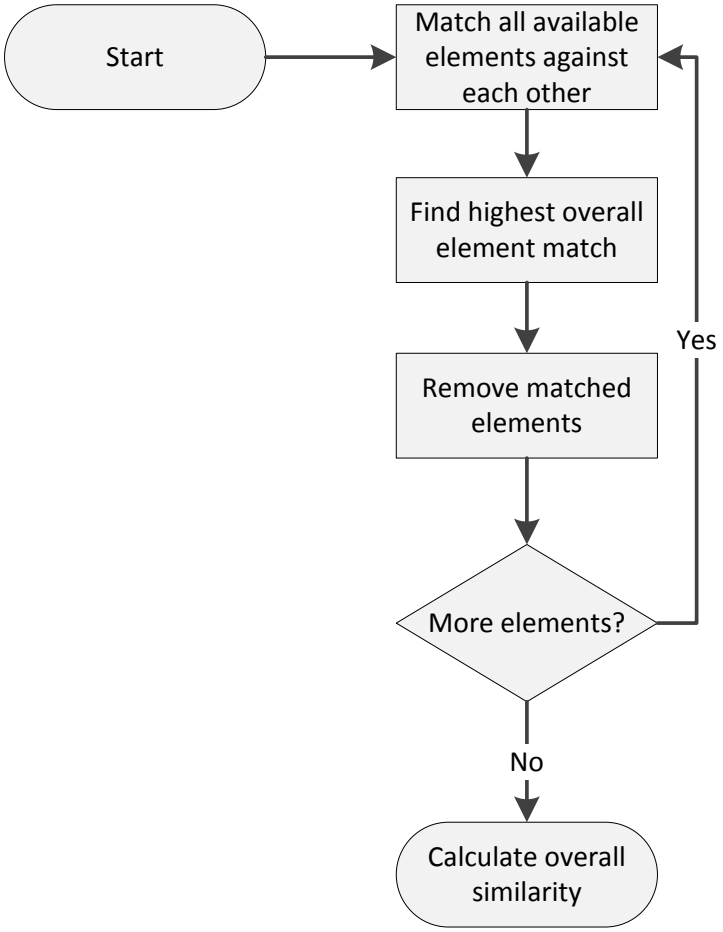

Figure 17 - Greedy Algorithm for Comparing Elements

The class similarity module obtains an overall match between two classes by comparing all elements within the hierarchical structure according to the current weight setting and using the similarity metrics defined in the previous chapter. The structure created to

represent a class is quite extensive and includes even detailed internal structural features. An alternative approach to this could have been to include only the most obvious features when measuring similarity. The UMLSimilator therefore allows individual features and even entire levels of the hierarchy to be excluded from the similarity calculation process.

Table 6 shows how the similarity between two classes (Bid and bid_tbl) is calculated.

Feature	Class Bid	Class bid_tbl	Similarity	Contribution
Inheritance	Object	Object	100%	7.69%
No. of Implementations	2	2	100%	7.69%
No. of Associations	0	2	0%	0.00%
No of Attributes	3	9	33.33%	2.56%
No. of Constructors	1	2	50%	3.85%
No. of Operations	3	18	16.67%	1.28%
Abstract (class)	False	False	100%	7.69%
Final (class)	False	False	100%	7.69%
Visibility (class)	Public	Public	100%	7.69%
Stereotype	None	None	100%	7.69%
Overall Attributes	3 attributes matched at 90%, 90% and 100%		93.33%	7.18%
Overall Constructors	1 constructor matched at 66.67%		66.67%	5.13%
Overall Operations	3 operations matched at 95%, 95% and 100%		96.67%	7.44%
Total				**73.58%**

Table 6 – Example match between class Bid and class bid_tbl

The example only shows the first level of features. The attributes, constructors and operations are broken down further into their constituent features. It also uses the same contribution for each feature. If optimised weights were used, then features would contribute in different proportions. More on this later in this chapter.

4.5 Graph Matching Module

The graph matching module implements the graph matching algorithm first presented in the previous chapter, to calculate the maximum common subgraph between two given class diagrams (*targetDiagram* and *sourceDiagram*). The implementation of this algorithm is quite complex and is broken down here into a number of smaller algorithms, which explain the different steps involved in identifying the maximum common subgraph. The first step of this process is presented here:

```
FindMaximumCommonSubgraph(targetDiagram, sourceDiagram) :
   mcs ← ∅
   bestGraphSimilarity ← 0
   graph ← ∅
   Class (cl_t) ∈ Classes WHERE Classes ⊂ targetDiagram
   FORALL Class (cl_t) IN Classes
      potentialMatches ← CALL FindMatchingClasses(cl_t, sourceDiagram)
      FORALL PotentialClass (pc_s) IN potentialMatches
         graph ← CALL FindBestGraph(targetDiagram, sourceDiagram, cl_t , pc_s)
         IF graph ≠ ∅ THEN
            graphSimilarity ← Get match percentage from graph
            IF graphSimilarity > bestGraphSimilarity THEN
               bestGraphSimilarity ← graphSimilarity
               mcs ← graph
            ENDIF
         ENDIF
      ENDFOR
   ENDFOR
   RETURN mcs
```

Figure 18 – Algorithm for Finding Maximum Common Subgraph

This algorithm ensures that subgraphs are created starting with every combination of classes from both diagrams. For example, if target diagram contains classes A_t, B_t and source diagram A_s, B_s, C_s, then subgraphs are created starting with the following combinations $\{A_t, A_s\}$, $\{A_t, B_s\}$, $\{A_t, C_s\}$, $\{B_t, A_s\}$, $\{B_t, B_s\}$, $\{B_t, C_s\}$. From this starting point, subgraphs are generated for every combination and the one that develops into the maximum common subgraph is selected.

This algorithm makes use of the *FindMatchingClasses* algorithm, which for a given class of the target class diagram identifies all of the classes in the source class diagram to which it could potentially be matched. Whether two classes can be matched is determined by the minimum class match threshold, which defines the lowest class similarity considered for allowing two classes to be added to the maximum common subgraph. The algorithm is defined in Figure 19.

```
FindMatchingClasses(cl_t, sourceDiagram) :
   matchingClasses ← ∅
   threshold ← Current minimum threshold
   Class (cl_s) ∈ Classes WHERE Classes ⊂ sourceDiagram
   FORALL Class (cl_s) IN Classes
      classSimilarity ← Compare Classes (cl_t ,cl_s)
      IF classSimilarity >= threshold THEN
         matchingClasses ← Add cl_s
      ENDIF
   ENDFOR
   RETURN matchingClasses
```

Figure 19 – Algorithm for Finding Potential Class Matches

The actual generation of subgraphs is achieved using the *FindBestGraph* (Figure 20) and *Match* (Figure 21) algorithms. It is important to remember that a node in the graph consists of a pair of classes which have been matched.

```
FindBestGraph(targetDiagram, sourceDiagram, cl_t , cl_s) :
   mcs ← Ø
   bestGraphSimilarity ← 0
   rootNode ← Create node containing cl_t and cl_s
   firstGraph ← Add rootNode
   graphs ← Add firstGraph
   CALL Match(graphs, cl_t, cl_s, firstGraph)
   FORALL Graph (g_i) IN graphs
      graphSimilarity ← Get match percentage from g_i
      IF graphSimilarity > bestGraphSimilarity THEN
         bestGraphSimilarity ← graphSimilarity
         mcs ← graph
      ENDIF
   ENDFOR
   RETURN mcs
```

Figure 20 – Algorithm for Calculating the Maximum Common Subgraph for a Combination of Classes

The *FindBestGraph* algorithm creates the first graph which contains as a root node the target (cl_t) and source (cl_s) classes provided. It starts the recursive creation of subgraphs by passing the first node to the *Match* algorithm, together with a reference to the collection of graphs. The recursive algorithm will add to the collection of graphs as it generates alternative subgraphs. As the *Match* algorithm is recursive, it will exhaust all possible combinations of subgraphs, so when it returns control to the *FindBestGraph* algorithm, all this has to do is iterate through the collection and identify the best match.

```
Match(graphs, cl_t, cl_s, currentGraph) :
   tempGraph ← Copy currentGraph
   targetRelations ← Get all links for cl_t
   sourceRelations ← Get all links for cl_s
   combinations ← CALL FindCombinations(targetRelations, sourceRelations)
   IF combinations ≠ Ø THEN
      FORALL Combination (cb_i) IN combinations
         tRel ← Get target relation from cb_i
         sRel ← Get source relation from cb_i
         IF tRel Is Same Type sRel AND Sim(tRel, sRel) > threshold THEN
            FORALL Graph (g_i) IN graphs
               IF tRel ∉ g_i AND sRel ∉ g_i THEN
                  node ← Create node containing tRel and sRel
                  g_i ← Add node
                  CALL Match(graphs, cl_t, cl_s, g_i)
                  nodeAdded ← TRUE
               ELSE
                  combinationExists ← TRUE
               ENDIF
            ENDFOR
            IF NOT nodeAdded AND NOT combinationExists THEN
               newGraph ← tempGraph
               graphs ← Add newGraph
               node ← Create node containing tRel and sRel
               newGraph ← Add node
               CALL Match(graphs, cl_t, cl_s, g_i)
            ENDIF
         ENDIF
      ENDFOR
   ENDIF
```

Figure 21 – Recursive Graph Matching Algorithm

The recursive graph matching algorithm starts by obtaining all possible combinations of relationships (links) starting from the given classes. Each possible combination is checked to ensure that the type of relationship matches and that the similarity between the classes which sit at the end of the relationships is over the established minimum threshold. If a combination can't be added to existing graphs and is not already used, then a new graph is created, which starts from a temporary copy made at the beginning of the recursive method and this is then built up using the recursive method.

The algorithm for obtaining all possible combinations of relationships between two classes uses an equation which applies factorial (see (17)).

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} \qquad \textbf{(17)}$$

where *n* and *k* are the sets of relations of a target and a source class. *n* being whichever of the two sets is larger.

A class diagram depicting the structure of the graph matching module can be found in the Appendix 1.

The problem of finding the maximum common induced subgraph of two graphs is NP-hard [Garey and Johnson, 1987]. This means that while this algorithm works for cases with relatively small numbers of classes and relationships, if the application were given more complex cases, the time required to compute them would increase drastically, making it unfeasible.

The previous section explained that the class similarity calculator module uses caching of results when measuring similarities between classes. This improves performance as the combinations of possible matches are high and the same combinations are checked

multiple times during class comparisons. The greedy algorithm used to perform matches (Figure 17) requires exhaustively comparing all features between two elements every time a best match is removed from the set. While the caching improves performance, it was found that the amount of comparison results kept in memory were resulting in the application draining all available memory on the heap (OutOfMemoryException). Optimisation techniques and analysis using a memory profiler were used to overcome this problem. Rather than individual objects tracking their own similarities, a set of static dictionaries were introduced, which are cleared when the algorithm moves between class diagrams.

4.6 Weight Optimiser Module

The UMLSimilator tool has a module that deals with weight attribution and optimisation. Every feature of the class diagram is being compared and its similarity contributes to establish the overall similarity between two classes. The weight associated with each feature controls the degree to which a feature contributes. The weight settings are applied at each level of the hierarchical structure representing a class. It is possible to compare classes in terms of a specific subset of features, e.g. to determine whether the closest matching class in terms of attributes is also the best match in terms of methods. The UMLSimilator allows the user to specify what elements should be taken into account when calculating the similarity between classes. It may not be possible for a user to easily identify individual weight settings as the process of comparing two designs is quite abstract and may involve intuition or reasoning that cannot be easily converted into rules.

When experts were asked to put weight settings on individual features, they found it difficult to abstract how much each feature would contribute to the overall structural similarity of two classes. It was easier to work with concrete examples and the most important factors in determining the similarity between class diagrams were the semantic information and the size/complexity of the designs.

In an attempt to convert an expert's choice of similarity matches into rules that can be applied by the UMLSimilator, a genetic algorithm was used to adjust the weights. Two approaches were introduced in chapter two:

- The expert defines matches between classes from two different diagrams, assigning each match a value between 0 and 10. The genetic algorithm evolves the weight settings and retests every new setting. The score of a weight setting is determined by the classes it matched out of the pairings identified by the expert. The weight setting with the highest score is adopted.
- The expert is given one diagram and merely identifies the closest match from a set of diagrams. The genetic algorithm evolves the weight settings and identifies the setting which gives the highest overall similarity between the two class diagrams.

The default settings used for the genetic algorithm are shown in Table 7, but these can be customised in the UMLSimilator tool.

The experiments showed that the number of runs could be kept relatively low, as results would rarely improve much with a high number of generations. The population size is a key parameter of a genetic algorithm and the observations confirmed the findings of Tsoy [Tsoy, 2003], that a large population with fewer runs yields better results than

small populations with more runs. If the population is small then it only covers a small search space and can therefore result in poor performance.

For crossover and mutation rates, the recommendations of De Jong [De Jong, 1975] and Schaffer et al. [Schaffer, et al., 1989] were followed, who state that crossover rates should be high and mutation rate should be very low. The mutation rate has to be kept quite low; otherwise it results in essentially random searches.

Setting	Value
Population size	100
Number of runs (generations)	20
Crossover rate	0.7
Mutation rate	0.05
Elitism rate	0.05

Table 7 – Default Settings for Genetic Algorithm

A chromosome in this genetic algorithm represents a set of weights for all features in a class and its fitness is the overall match achieved when applying these weights. When chromosomes are created, weights are generated randomly and then normalised to ensure that groups of weights add up to 1.0. To ensure that the genetic algorithm doesn't provide a solution that is worse than the default weight setting, a chromosome with the default weights is always included in the initial population and elitism ensures that a small percentage of the fittest chromosomes are automatically included in the next generation.

The crossover operation uses a single crossover point, which defines the index in the arrays of weights where the segments should be split. The mutation operation randomly selects weights, which are then changed to a new random value. This requires a renormalisation of the weights.

The structure of the weight optimiser module can be seen in a class diagram included in Appendix 1.

4.7 Clustering Module

The clustering module applies agglomerative hierarchical clustering to the set of retrieved and ranked cases. Using clustering, these are assigned into groups of similar objects.

The single-link clustering method was first implemented, which merges the clusters with the smallest minimum distance between any two members of both clusters. However, single-link clustering is not suitable for isolating poorly separated clusters and it was found that using this method would result in one massive cluster containing the large majority of cases and all remaining clusters each containing a single case. Better results were achieved using average-link clustering, which is generally attributed to Sokal and Michener [Sokal and Michener, 1958]. This merges the clusters with the smallest average distance of pairs of objects from both clusters.

This is computationally more demanding, but the number of cases in the case-base is small enough to warrant the use of this clustering method.

The termination condition used in this work is based on the number of clusters. Thus, starting with just over one hundred clusters, these are incrementally merged until the number is reduced to a small set of clusters. According to Mardia et al. [Mardia, et al., 1980], a good rule of thumb for determining the number of clusters *k* is:

$$k \approx \sqrt{\frac{n}{2}} \qquad (18)$$

where *n* is the total number of objects. With the current case-base, this rule of thumb would be 7.1, which was rounded up to 8 clusters to take into account the number of different assignments in the case-base.

An overview of the clustering module can be found in a class diagram in Appendix 1. The Cluster class and SimilarityResult class which form part of the clustering module both implement Comparable interface, but the comparison algorithm was reversed, so that results would be sorted from the highest to the lowest. This made it possible to work with sorted sets to achieve the clustering.

4.8 Visualisation Module

The visualisation module was developed to enable easy visualisation of the cases and comparison results. It is implemented using Windows Presentation Foundation and XAML and makes it possible to display the cases from the case-base in a class diagram-like fashion, by displaying the classes and their relationships.

The case-base contains descriptions of class diagram content, but no information about the graphical layout of a particular diagram. The visualisation module therefore applies an algorithm based on the square root of the number of classes in a diagram to lay out the classes evenly distributed. It was found that aligning classes in perfect grids made it difficult to clearly display the relationships, thus an offset was used to create a more uneven spread.

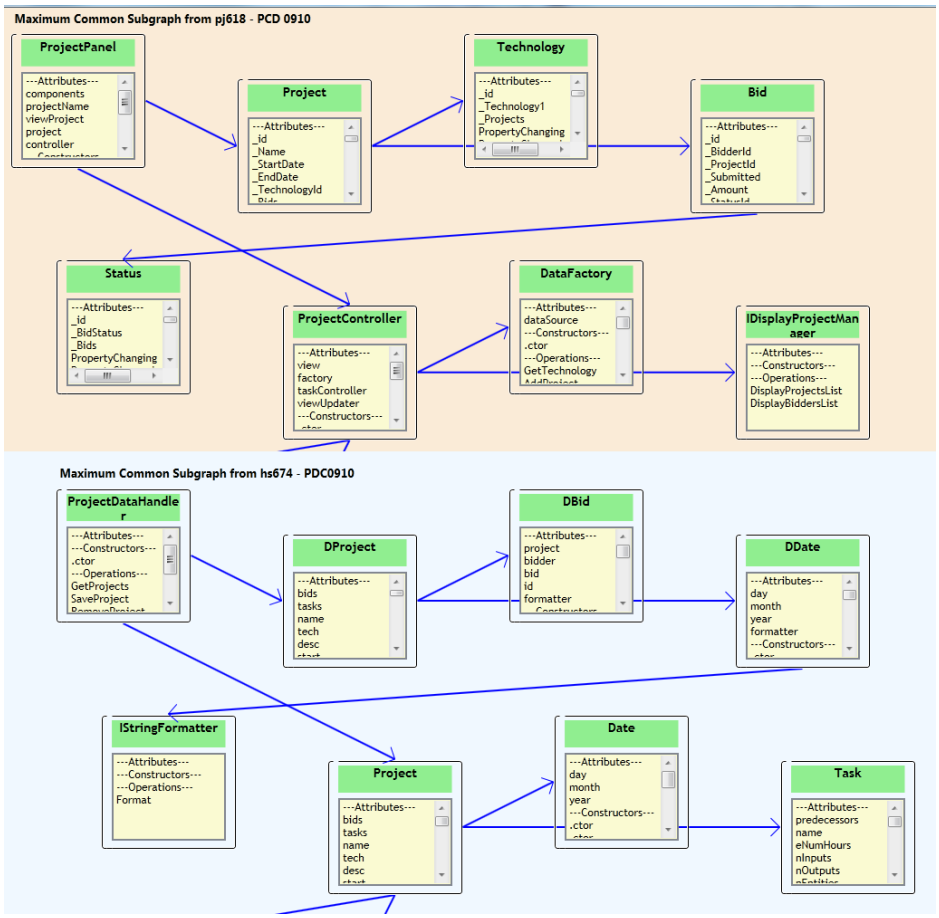

Figure 22 - Visualising the Maximum Common Subgraph

The ability to visually display a diagram aids in providing a better understanding to a user of how particular similarity results were obtained by the UMLSimilator tool, as one could view the diagrams being compared. The users of the system are also more likely to trust it when the system can explain how it has reached a particular answer [Ye and Johnson, 1995]. To provide users with an explanation of how results were achieved, the tool provides a complete breakdown of all similarities between all features of the two diagrams (see Figure 23).

To further enhance the explanation the system offers, visualisation of the maximum common subgraph was implemented. Figure 22 shows an example of how the maximum common subgraph is displayed in UMLSimilator.

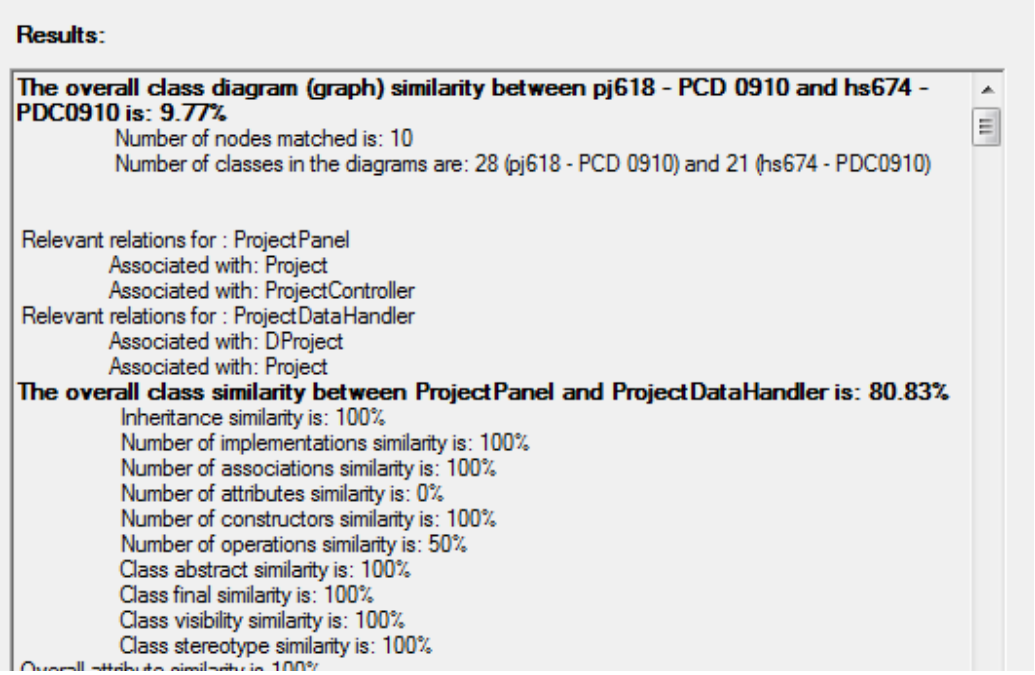

Figure 23 - Example Similarity Breakdown between Two Class Diagrams

By providing a trace of the reasoning the UMLSimilator has undergone in order to reach the answer, the system makes the reasoning process transparent to an expert user, who can use this information to validate the reasoning of the system.

4.9 Conclusion

This chapter provided details of how the various algorithms used in this work were applied in practice.

The case study was outlined, specifying the contents of the case-base, which consists of just over one hundred class diagrams which were obtained from five programming assignments of differing levels and technologies. The cases are therefore based on real software implementations, which made it possible to obtain the implementation cost in lines of code / numbers of characters and the grade each piece of work was awarded.

The structure of the UMLSimilator tool was presented, outlining the various modules and their integration.

The reflection process was discussed, which made it possible to reverse-engineer class diagrams from compiled .NET and Java code. This required the use of Web Services to integrate the various services required to use reflection with different technologies.

It was demonstrated how the similarity between two classes is measured by breaking them down into a hierarchical structure of composing elements and applying a greedy algorithm to find the best matches. While not guaranteeing the best possible solutions, this provides a near optimal solution, which is still computationally feasible.

The implementation of the graph matching algorithm was sketched out in detail, showing how the maximum common subgraph is calculated between two class diagrams. This requires a recursive algorithm, which creates subgraphs of classes above a specific minimum threshold. While this approach came close to reaching the limit of where it could be practicably applied, results would be calculated within an acceptable time limit with even the largest class diagrams in the case base.

It was explained how a genetic algorithm was applied in order to optimise the weights used to establish the similarity between two given classes. In an attempt to extract rules from experts' selection of similarity matches, two approaches to weight optimisation, using a genetic algorithm, were implemented.

Agglomerative hierarchical clustering was applied to all cases in the case-base to arrange them into groups of similar objects. The method used for merging clusters was average-link clustering, which performed better with this set of objects than single-link clustering.

Finally, the visualisation module was presented, which showed how the system can help explain the results of a class diagram comparison by visually displaying the maximum common subgraph, the original class diagrams and a complete breakdown of all similarity matches.

The next chapter will introduce the experiments which were performed in order to assess the performance of the algorithms and techniques applied in this work, as well as providing a detailed analysis and evaluation of the results obtained from the experiments.

Chapter 5

5 Experiments and Evaluation

In the previous chapter the UMLSimilator tool was introduced, which was built in order to evaluate whether effective knowledge retrieval from software design artefacts can be achieved using case-based reasoning when ignoring semantic information and using merely the structural one. Here the experiments carried out using the UMLSimilator are presented, along with an evaluation of the results. The goal is to make an assessment of the approach and the methodologies used and determine how these can be combined to obtain the best results. The reasoning mechanisms are tested and complemented with input from human expert evaluation.

Wherever possible the results of the experiments are compared against those achieved using alternative approaches. However, most of the experiments can't be compared against other tools, as there is no tool with enough similarity to UMLSimilator to enable comparative tests. Where no comparative results are available, the experiments are validated against expert opinion or using other means of validation. All experiments were validated using cross validation, with the exception of cost estimation and explanation which were validated against expert opinion.

5.1 Experiments

The UMLSimilator tool has been implemented in Visual C#.NET using Visual Studio. The case-base is stored in an Access database. All of the experiments were executed on the same computer, which was not used for any other work during the execution of the tests. The system specification of the computer is outlined in Table 8.

Detail	Specification
Processor	Intel Core i5 CPU 661 @3.33GHz
Installed memory (RAM)	8.00BG
System type	64-bit Operating System
Operating System	Windows 7 Enterprise

Table 8 – Test Computer System Specification

The case-base used for the experiments consists of 101 class diagrams obtained from five different coursework assignments. All of the assignments were implemented using an object-oriented language (.NET or Java). Using different assignments means that the cases come from five different domains:

- Software cataloguing application – (assessment at level 6 – high complexity)
- Project management application (assessment at level 6 – high complexity)
- Car repair shop application (assessment at level 5 – low complexity)
- Stock management application (assessment at level 7 – low complexity)
- Project bidding system (assessment at level 6 – high complexity)

The five assessments were from level 5, 6 and 7. As the level 6 assessments were sourced from an advanced programming course, the complexity of the requirements was higher than that from level 7.

Of the 101 class diagrams, 17 were implemented in Java and 84 in .NET.

The largest class diagram contains 40 classes, the smallest 3 classes and the average number of classes per class diagram across the entire case-base is 10.75. Each class has on average 13.06 attributes, 1.08 constructors and 8.76 operations.

The majority of class diagram comparisons executed very fast, with the average being 1.54 seconds. However, more complex diagrams could take a very long time to compute (up to an hour). Performing a run of all cases using a particular weight setting took several hours. Thus to keep the user informed of the progress during the lengthy operations and stay responsive, the UMLSimilator was implemented as a multi-threaded application. The entire set of 70 runs of experiments required over 184 hours of processing time to compute, the majority of which was taken up by the graph similarity calculations.

5.2 Methodology

The UMLSimilator tool automates the algorithms and techniques applied in this research. To ensure that these are appropriate requires controlled and planned experiments which test the various aspects of the algorithms and methods implemented by the software tool.

Extensive tests are carried out which use the structural information encoded in class diagrams, but ignore the semantic details. A suite of experiments are devised using:

1. Classes only (without graphs)
2. Graphs by calculating the maximum common subgraph
3. Classes only with optimised weights
4. Graphs with optimised weights

Comparing class diagrams is an abstract undertaking as there is no established procedure or approach for determining similarity between class diagrams. This poses the problem of how the competence of the work can be measured.

One solution applied here is to compare the results obtained from the experiments to human expert opinion. This is complemented using the software's ability to explain results, tracing how a particular result was obtained to reinforce trust in the approach. Other measures of competence can be obtained by using additional information available for the cases. The additional information available for the class diagrams in the case-base includes:

- *Domain of a case*: The domain of each case is known. For each class diagram, the system remembers from which of the five different assignments it originated. Having this information also makes it possible to determine the importance of provenance to this work, as the source of each case is known.
- *Grade achieved*: The grade can be regarded as a measure of quality. Although the grade awarded is generally not based solely on the structural details of a class diagram, but could include a written report, graphical user interface design, etc., having this quality indicator still enables an investigation of the correlation of marks to structural knowledge.
- *Cost of implementation*: By reverse-engineering real software implementations, it is possible to obtain the lines of code and number of characters it took to implement each class diagram.
- *Programming Language*: The programming language used to implement each solution is known.

To effectively use the above information to discriminate between cases, two methods are applied, namely cross validation and clustering. Both were applied to determine whether any of the above information can be identified, based solely on the structural details of class diagrams.

Cross-validation assesses how the results of the comparisons will generalise to an independent data set. The type of cross-validation used here is leave-one-out cross-validation, where a single item from the original sample is used as the validation data for testing the model and all the remaining items are used as training data. Given that the case-base is not very large, it was decided to repeat this process for a considerable amount of cases and apply the leave-one-out cross-validation instead of k-fold cross validation as it gives more accurate results. Thus, 50% of the cases were randomly selected from the case-base and used as the training data, one at a time. The sample was checked and contained diagrams from all five case domains.

Experiments were carried out using one, three and five KNN (k-nearest neighbours). The cross-validation for each KNN was measured for each set of results (with different weight settings, with and without graphs, with less weighting criteria, with different class thresholds). The application of cross-validation was used to evaluate the four criteria listed above (case domain, grade, implementation cost and programming language).

An alternative approach to evaluate these criteria is the use of clustering. As single-link clustering resulted in most clusters having only a single item, average-link clustering was the technique chosen for this work. Clustering was used in order to validate the results, whereby clusters of the cases were generated post retrieval and evaluated against facts which were known, but didn't form part of the similarity metrics. For

instance, clustering was used to determine whether cases were grouped according to the source, programming language, size or quality of the software designs.

Finally, it was known that two of the assessments in the case-base were found to be plagiarised. Thus it is possible to check how efficient the approach is at detecting plagiarism.

5.3 Structural Similarity

The first experiments were carried out using only class structure similarity, using the similarity metrics outlined in chapter 3.

Running a complete set of comparisons of every class diagram in the case-base against every other computed quite fast, taking an average of only 0.26 seconds per class diagram comparison and completing all 5050 comparisons in around 21 minutes.

Taking all groups of k-nearest neighbours together (one, three and five), using only the structural similarity, the experiments showed that the domain of the target case was matched in 74.16% of all cases. This means that for the majority of target cases, the nearest neighbours were selected from the same domain (assessment). Figure 24 shows a breakdown for all target cases

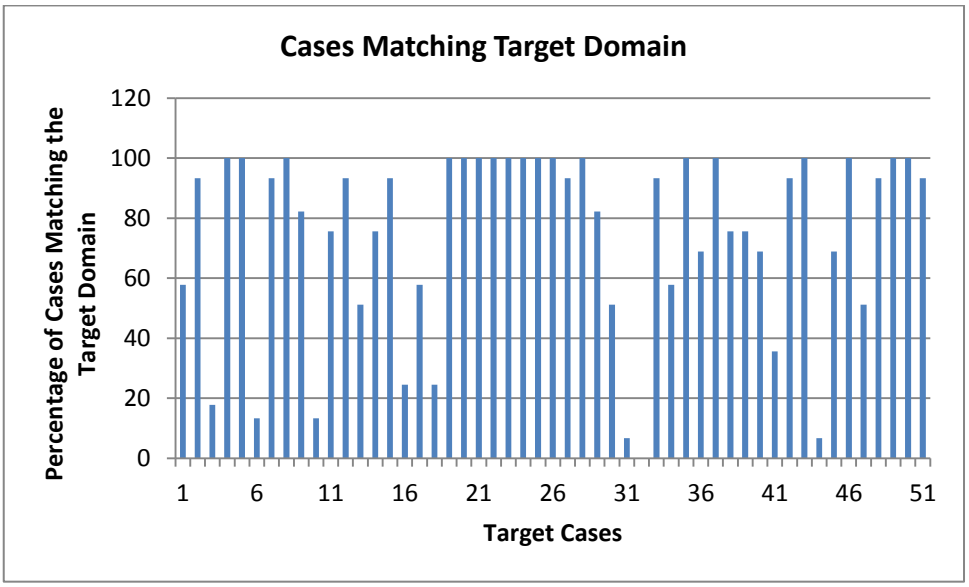

Figure 24 - Percentage of Nearest Neighbours Matching Target Domain

A similar breakdown based on the programming language shows even better results, with an average of 85.40% of nearest neighbours matching the programming language of the target case.

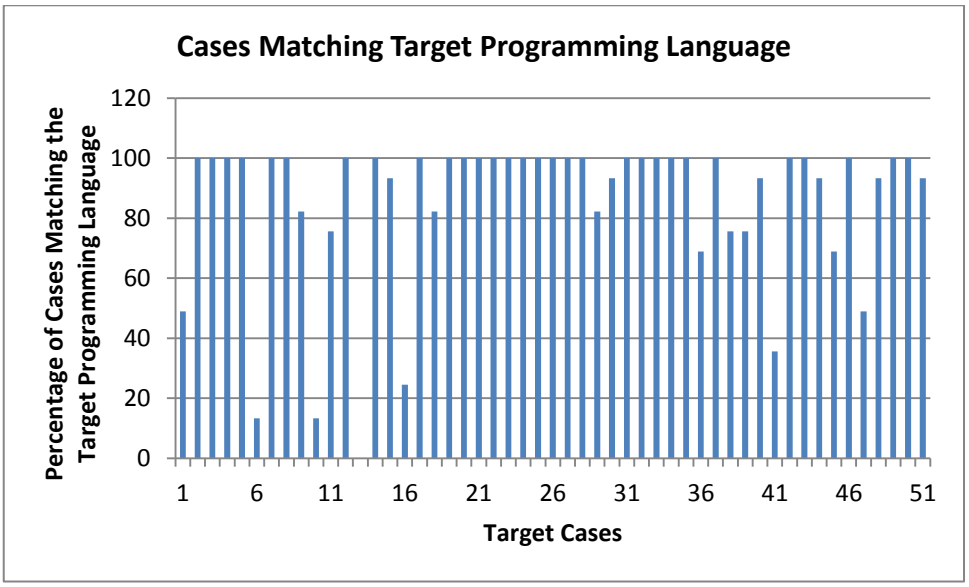

Figure 25 - Percentage of Nearest Neighbours Matching Target Programming Language

As the number of cases implemented in .NET is much higher (≈84%), it is statistically more likely that target cases and their nearest neighbours are implemented in the same programming language. Filtering the data by programming language, it was found that 93.89% of nearest neighbours for .NET target cases were .NET, but for Java it was only 32.06%. However, both of these figures show an improvement over the average (84% for .NET and 26% for Java).

The remaining two characteristics which can be measured using structural similarity are the grades and the implementation cost (lines of code/number of characters). These are numerical values, thus to evaluate how well the nearest neighbours fare, the standard deviation is calculated. The standard deviation of all cases in the case-base was also calculated to provide a measure of comparison. For grades, the standard deviation in the whole case-base is 20.77, while the average standard deviation of nearest neighbours and their target cases based on structural similarity is 11.57.

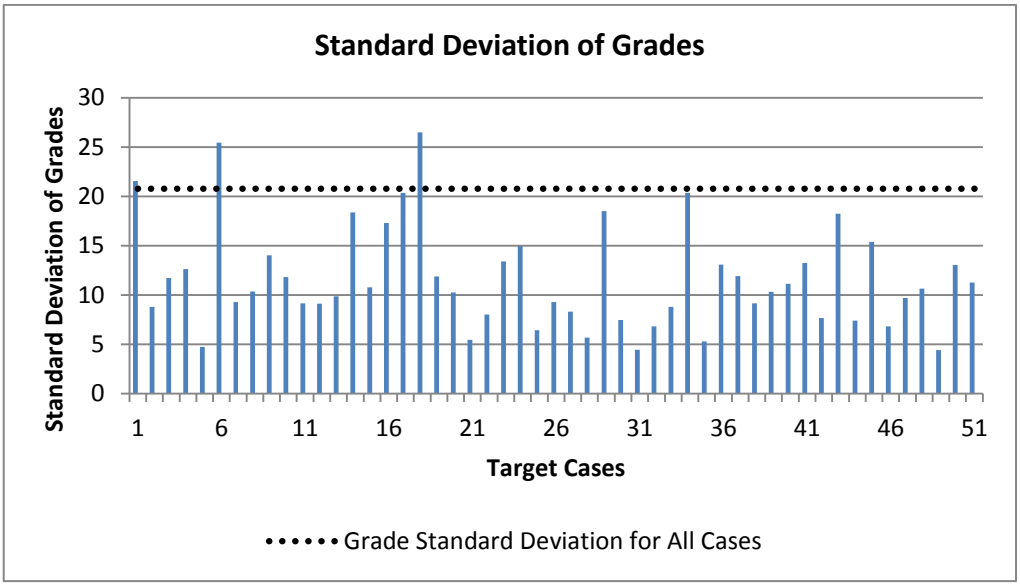

Figure 26 - Standard Deviation of Grades for Nearest Neighbours

As can be seen in the graph above, in the large majority of cases the standard deviation of grades improved. There is less dispersion from the average, so the grades are closer together. There are two cases (6 and 18) which are notable higher than the case-base average. Upon investigating these two cases, it was found that both were unusual, as they had low complexity (4 and 3 classes respectively), but had achieved very high grades (80% and 90%).

While the number of characters is a more accurate reflection of the effort it took to implement a software application, the lines of code are a measurement generally associated with specifying the size of an implementation. As can be seen in the two graphs below (Figure 27 and Figure 28), the standard deviation pattern for both is very similar. The remaining experiments will therefore use the lines of code.

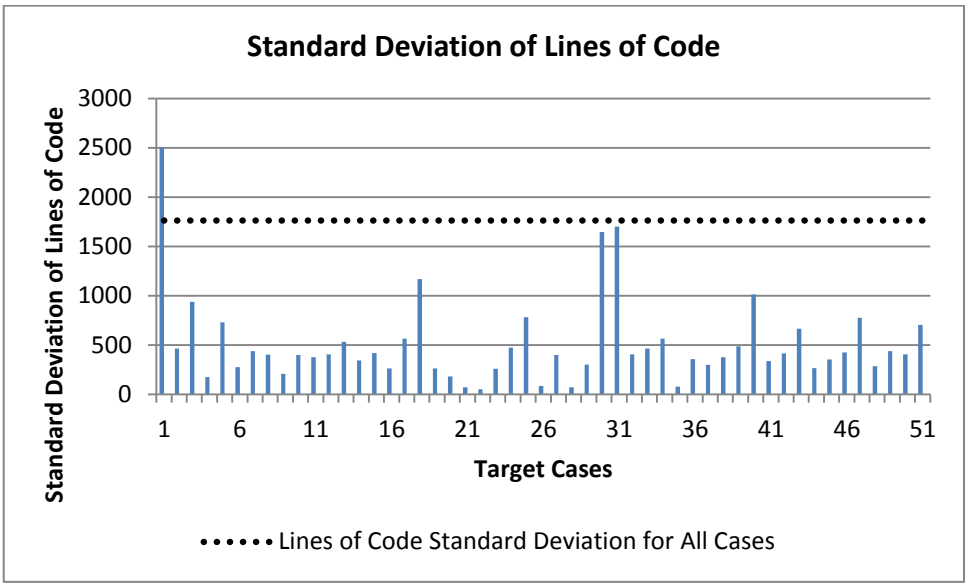

Figure 27 - Standard Deviation of Lines of Code for Nearest Neighbours

The standard deviation for all cases in the case-base was calculated to be 1762.68 and 86204.79 for lines of code and number of characters respectively. For both, the standard

deviation of the nearest neighbours was much better overall (509.87 and 26499.21) than the values measured for the entire case-base.

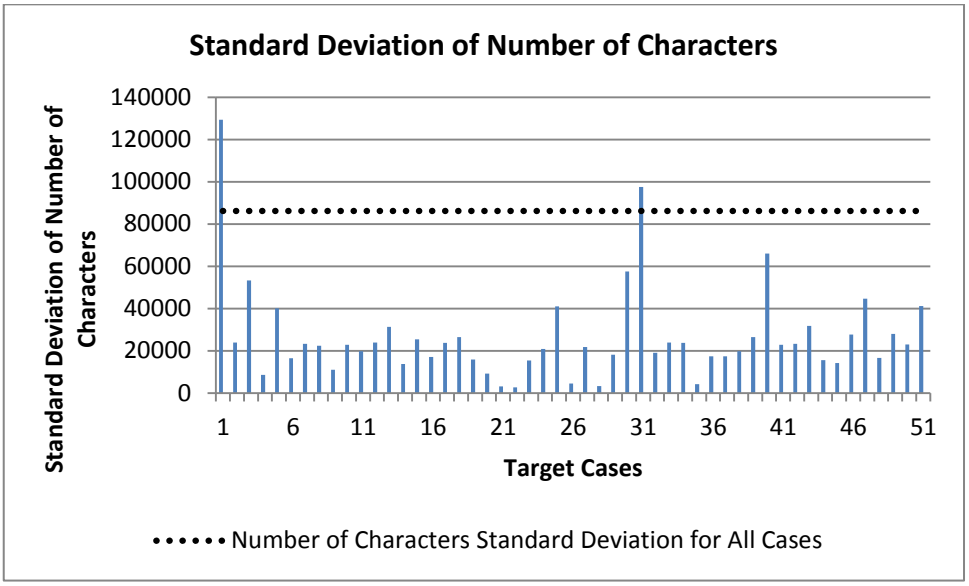

Figure 28 - Standard Deviation of Number of Characters for Nearest Neighbours

The graphs presented this far were showing averages for cases with one, three and five nearest neighbours (KNN) combined. An analysis of the separate sets revealed that the best results are achieved across the board when using a single nearest neighbour. The worse results with five nearest neighbours. The details can be seen in Table 9.

Number of nearest neighbours	% matching target domain	% matching target programming language	Standard deviation of grade	Standard deviation of lines of code
1	82.35	86.27	7.58	247.80
3	71.90	85.62	12.80	616.67
5	68.24	84.31	14.33	665.12

Table 9 – Average Results for One, Three and Five Nearest Neighbours

These findings suggest that the higher the number of nearest neighbours, the worse the results. This makes sense, because as the number of nearest neighbours increases, so does the average distance to the target case. The complete set of charts showing the performance for one, three and five nearest neighbours can be found in Appendix 2.

5.3.1 Structural Similarity and Provenance

All the previous results have been based on the nearest neighbours selected from the entire case-base. However, an interesting question is whether the results improve when the nearest neighbours are only selected from the same domain. Experiments have been carried out using the provenance of target cases to select cases only from the same source (assessment/domain). The results are outlined in the Table 10 – Average Results by Domain . The results are an average of using one, three and five nearest neighbours. To contextualise the results, the averages achieved without provenance have also been included.

Domain	% matching target domain	% matching target programming language	Standard deviation of grade	Standard deviation of lines of code
Software cataloguing	100	100	10.24	201.86
Project management	100	100	16.26	1007.66
Car repair shop	100	100	9.15	452.98
Stock management	100	56.77	13.95	536.60
Project bidding	100	100	11.87	1205.95
No Provenance	*74.16*	*85.40*	*11.57*	*509.87*

Table 10 – Average Results by Domain

Obviously the percentage of cases matching the domain of the target case is 100% and for four out of five domains the programming language is also 100%, as all of the assessments were developed using the same technology. In the Stock Management assignment, students had been given a choice of programming language and solutions were implemented in .NET or Java.

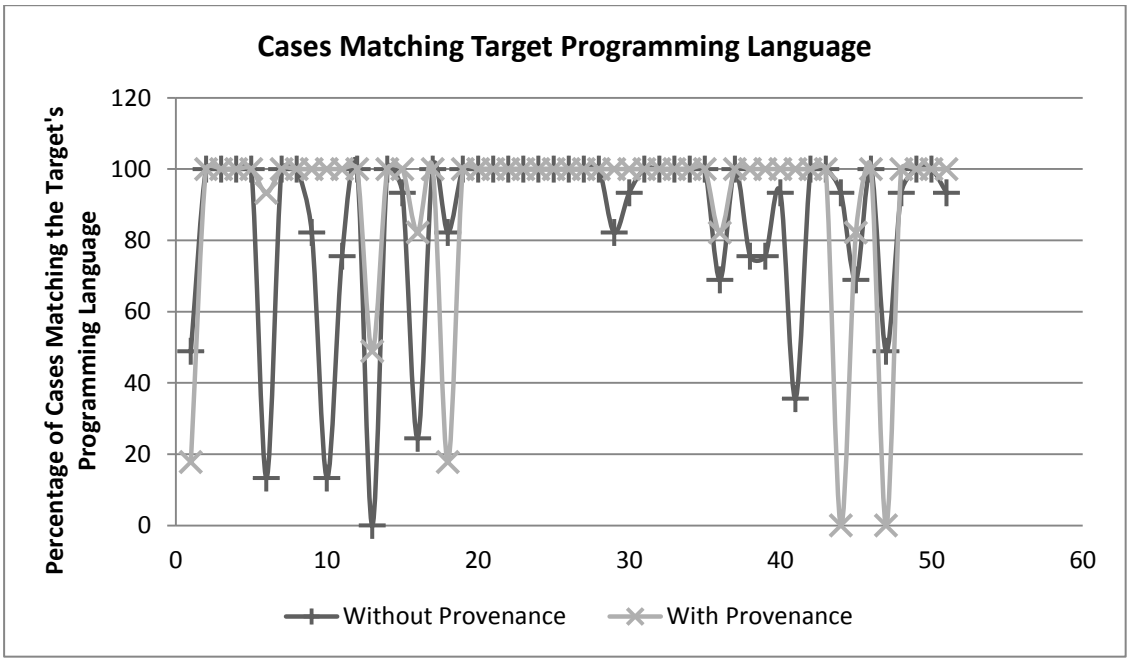

Figure 29 - Comparison of Nearest Neighbours Matching Target Programming Language With and Without Provenance

Overall, using provenance shows small improvements, but these are not very pronounced.

The graph on the previous page (Figure 29) shows the percentages of cases whose programming language matches that of the target case. For data with provenance, the target case would select nearest neighbours only from the same domain.

The average without provenance is 85.40% while using provenance increased the average to 90.68%. The results are similar for the standard deviation of grades and lines

of code. The overall standard deviation of grades was 11.57 and using provenance lowered this value very slightly to 11.20 (breakdown in Figure 30). The standard deviation for lines of code was lowered from 509.87 to 505.69 (breakdown in Figure 31).

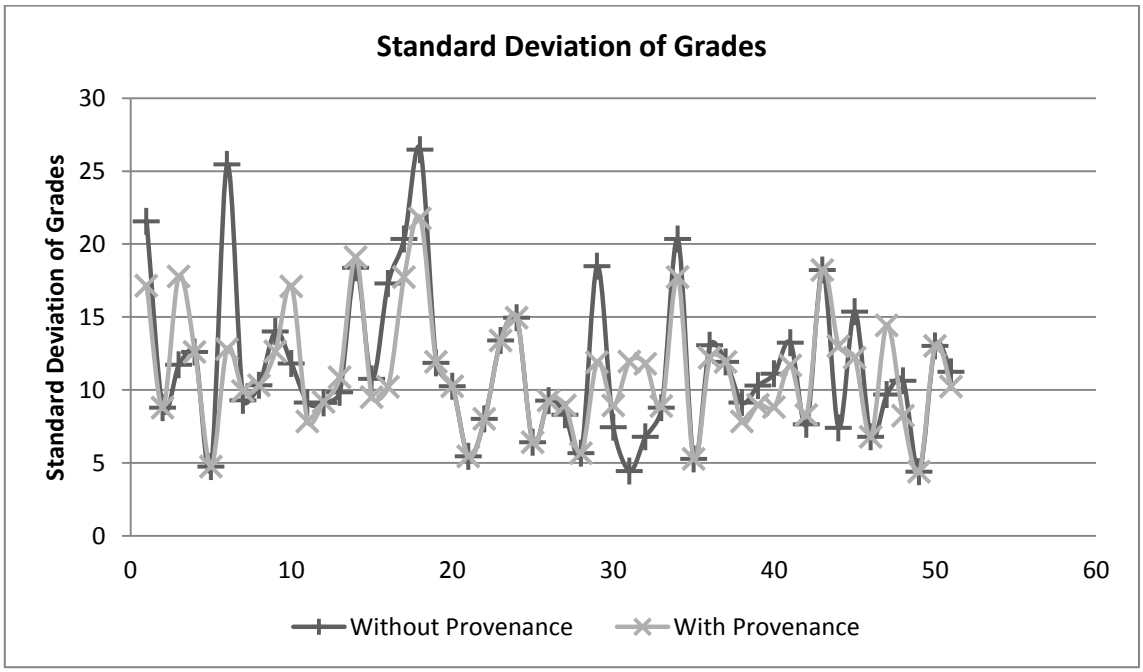

Figure 30 - Comparison of Standard Deviation of Grades With and Without Provenance

When using the class structure similarity without graphs, provenance had very little effect on the results. All results improved overall, but only marginally. The explanation for this is that cases were grouped by quality or complexity, rather than by domain. For instance, class diagram of similar quality or implementation cost were matched, but from across all domains. The next set of experiments will analyse the impact of applying graph matching.

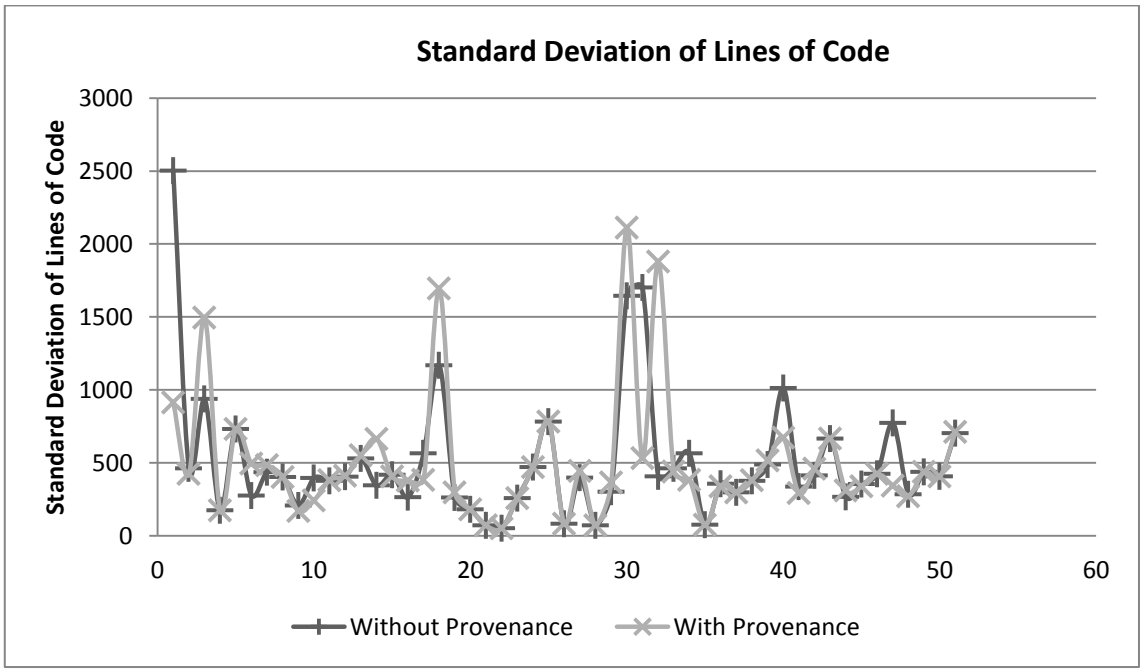

Figure 31 - Comparison of Standard Deviation of Lines of Code With and Without Provenance

5.4 Graph Similarity

The experiments in the previous section were calculating similarity of two class diagrams based on matching pairs of classes and the overall number of classes in each diagram (see equation (11) in section 3.5).

However, while the intra-class similarity takes into account the number of links of a class, the above equation doesn't explore how classes are connected. The next set of experiments takes this into account by treating the class diagrams as graphs and calculating the maximum common subgraph between them. The formula used to calculate the similarity shown below has been explained in chapter 3.

$$\sigma(G_t, G_s) = \frac{(\sum_{\substack{matches \\ C_t, C_s \\ in \\ MCS}} \sigma(C_t, C_s))^2}{count(G_t).count(G_s)} \quad \textbf{(19)}$$

The problem of finding the maximum common induced subgraph of two graphs is known to be NP-hard [Garey and Johnson, 1987]. The implication of this is that the algorithm doesn't scale to graphs having large numbers of nodes. Experiments with this algorithm found that cases with low complexity (small number of classes and relationships) computed quite fast, many of which computed in less than one second. With an increase of the complexity of the graphs, however, the time required to calculate the maximum common subgraph would increase drastically.

5.4.1 Computation Times

An analysis of the execution times clearly showed that the class diagrams with the largest number of classes would be among those taking longest to compute. However, it

is not just the number of classes, but also the number of relationships which influence the execution time. Even graphs with relatively low numbers of nodes can contain large numbers of possible graph combinations, if there are many arcs.

Class Diagram[3]	Number of Classes	Number of Relationships
Software cataloguing – 3394	17	24
Software cataloguing – 3401	18	21
Software cataloguing – 3418	12	17
Project management – 3425	24	42
Project management – 3433	20	27
Project management – 3435	40	60
Car repair shop – 3460	23	21
Stock management – 3483	9	21
Stock management – 3491	19	38
Project bidding – 3500	28	47
Average across all class diagrams	*10.59*	*13.23*

Table 11 – Number of Classes and Relationships for Diagrams Taking Longest to Compute

The time taken to compute the maximum common subgraph is also not dependent strictly on the number of classes and relationships, but also how they are arranged. For instance, when analysing the results in Table 11, two class diagrams which appear peculiar are: Software cataloguing – 3418 and Stock management – 3483. These have considerably lower number of classes than the remaining class diagrams. The reason they are involved in lengthy computations is due to the fact that they have similar distributions of classes and relationships to a very large diagram which could result in many matching combinations of possible subgraphs. If two diagrams have very different shapes, there is not much overlap and therefore a reduced set of potential subgraph combinations to evaluate (e.g. a class diagram arranged in star-shape and

[3] The number following the domain name is a unique identifier which makes it possible to easily distinguish between class diagrams

another as a long line). Both of the two diagrams share a similar layout to subgraphs of Project management – 3435, which is the largest and most complex class diagram in the case-base.

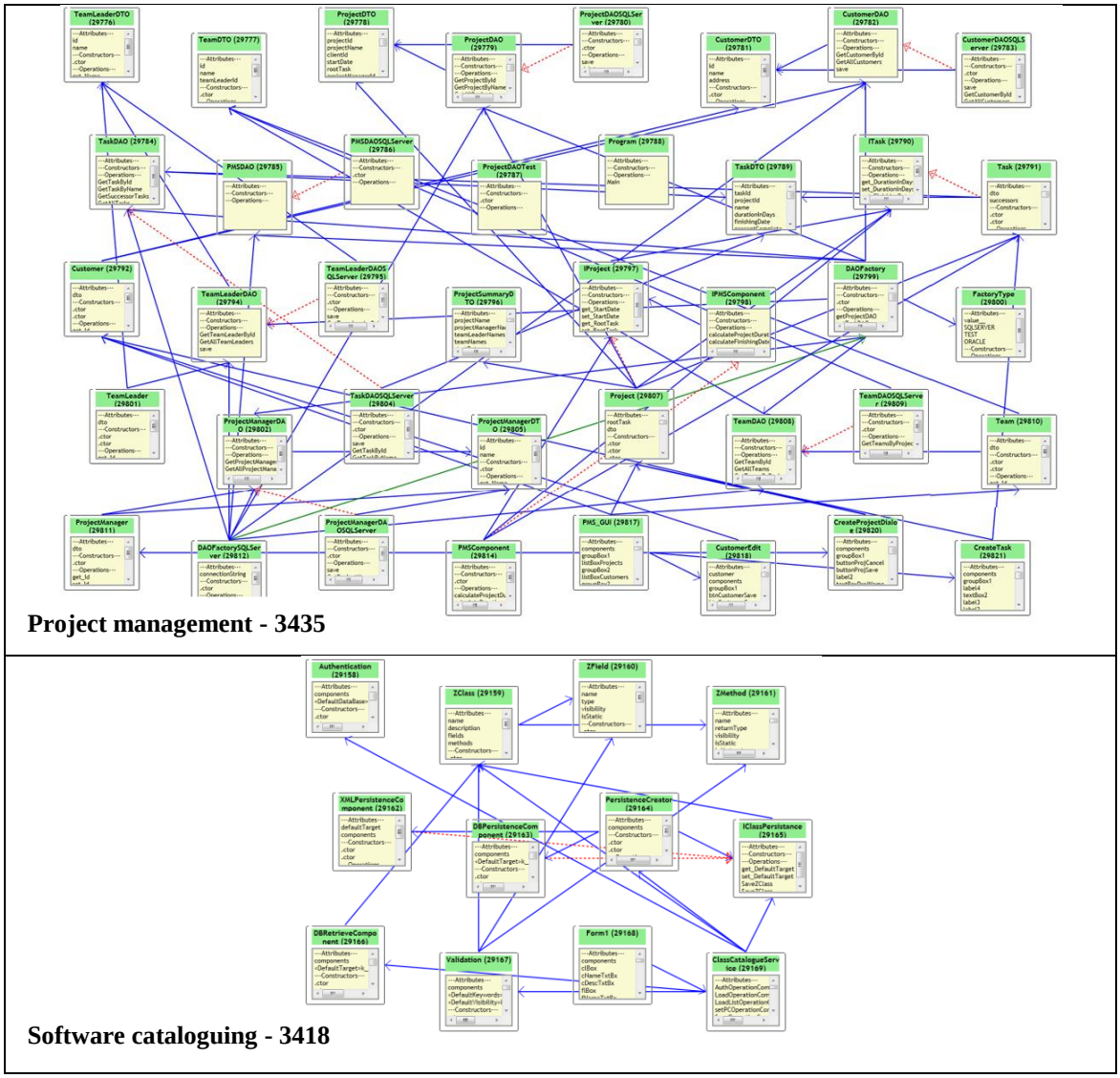

Figure 32 - Class Diagrams for 3435 and 3418

While at first sight, the two diagrams may appear very different (see Figure 32), their maximum common subgraph is actually quite large, encompassing all but one of the classes from 3418 (Figure 33). 3418 has only got twelve classes, but the maximum common subgraph between it and 3435 contains eleven out of these twelve classes.

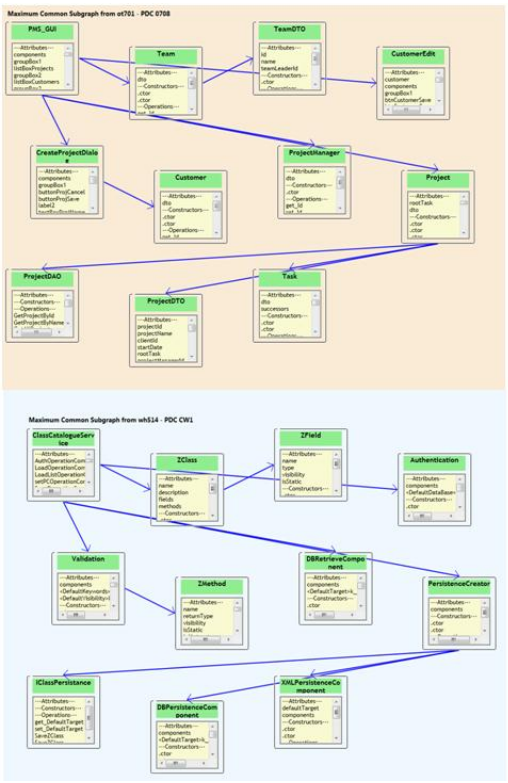

Figure 33 - Maximum Common Subgraph of 3435 and 3418

Apart from the number of classes and relationships, another factor influencing the time it takes to compute the maximum common subgraph is the minimum similarity threshold which must be met by two classes in order for them to be added to the graph. To measure graph similarity, experiments were carried out using a minimum threshold of 80%, 60%, 40% and 20%. Obviously, the higher the threshold, the smaller the maximum common subgraphs and the shorter it takes to compute. Calculating the maximum common subgraphs between each class diagram in the case-base and all other class diagrams took just over half an hour with a minimum class similarity threshold of 80%. At 60% it took over two hours and at 40% over four and a half hours to compute. With a minimum threshold of 20%, it took only 9 seconds longer to compute than 40%, so the difference was minimal. The reason for this is that very few class matches would result in similarities of less than 40%.

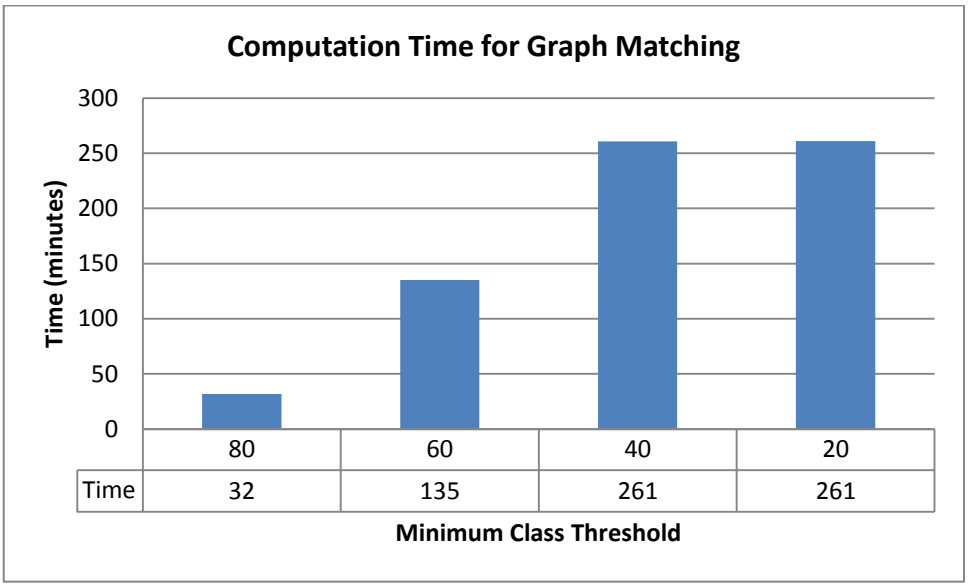

Figure 34 - Execution Times for Measuring Graph Similarity for All Diagrams

The increase in time required to calculate the similarity of complex graphs using the maximum common subgraph is exponential. While on average each graph would compute in between 0.38 and 3.1 seconds, depending on the minimum class threshold, the more complex graphs would take a disproportionate amount of time, with some requiring around three quarters of an hour to compute. Figure 35 shows the exponential distribution of computation times, ordered from the lowest to the highest. The curve is the same regardless of the minimum class threshold. This demonstrates that the current algorithm could not feasibly be applied if the complexity of class diagrams in the case-base were to increase.

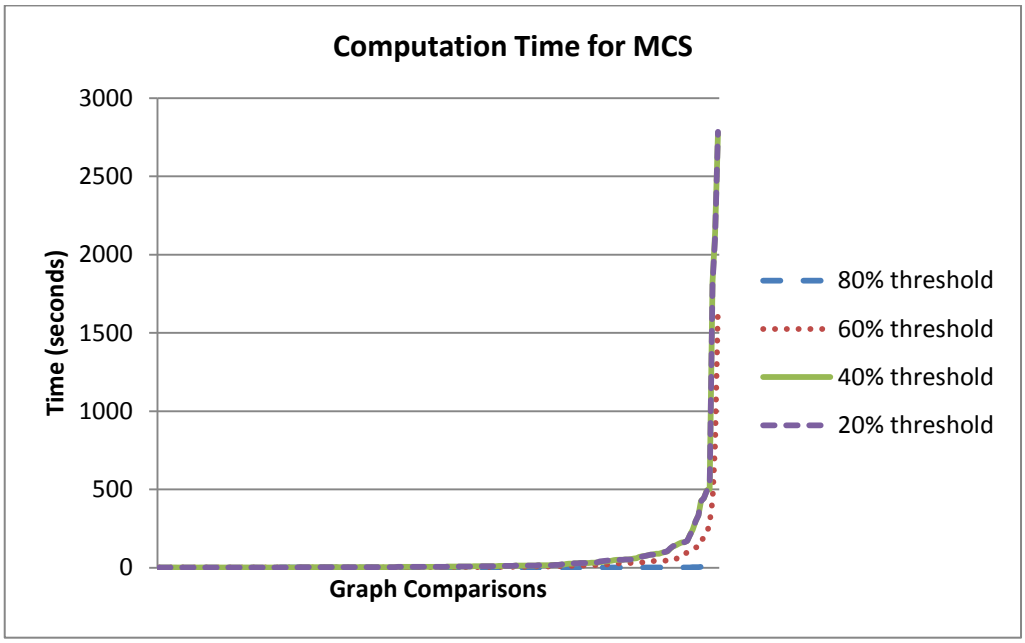

Figure 35 - Computation Times for Individual Graph Matches

5.4.2 Graph Similarity Results

The graph similarity for all cases was measured by calculating the maximum common subgraph using different minimum class similarity thresholds. Again, for all graphs the one, three and five nearest neighbours were obtained. Table 12 shows the average results.

Minimum Class Similarity Threshold	% matching target domain	% matching target programming language	Standard deviation of grade	Standard deviation of lines of code
80%	84.53	90.72	11.00	462.10
60%	85.51	93.16	10.84	470.94
40%	83.63	92.81	10.80	480.92
20%	83.25	92.81	10.79	481.03
Average	***84.90***	***92.37***	***10.86***	***473.75***

Table 12 – Average Results for Graph Similarity Using Different Minimum Class Similarity Thresholds

The results for the various categories differed depending on the threshold configuration. There was very little difference between the 40% and 20% threshold. The threshold setting which performed best overall was 60%. It achieved the best results for matching the target domain and programming language, was only 0.05 worse than the best grade standard deviation and was second best for the standard deviation of lines of code. It was the only threshold setting which was better than the average in every single category. Detailed graphs comparing the different threshold settings can be found in Appendix 3.

A comparison between the graph similarity experiments and the class structure similarity experiments discussed in the previous section shows that the graph similarity performed better in every category overall, but obviously not in every case.

Using graph matching, the average percentage of correctly identified target domains increased from 74.16%, obtained using class structure similarity, to 85.51%. The percentage of nearest neighbours with matching programming language also rose from 85.40% to 93.16%. See Figure 36 and Figure 37 for detailed comparisons.

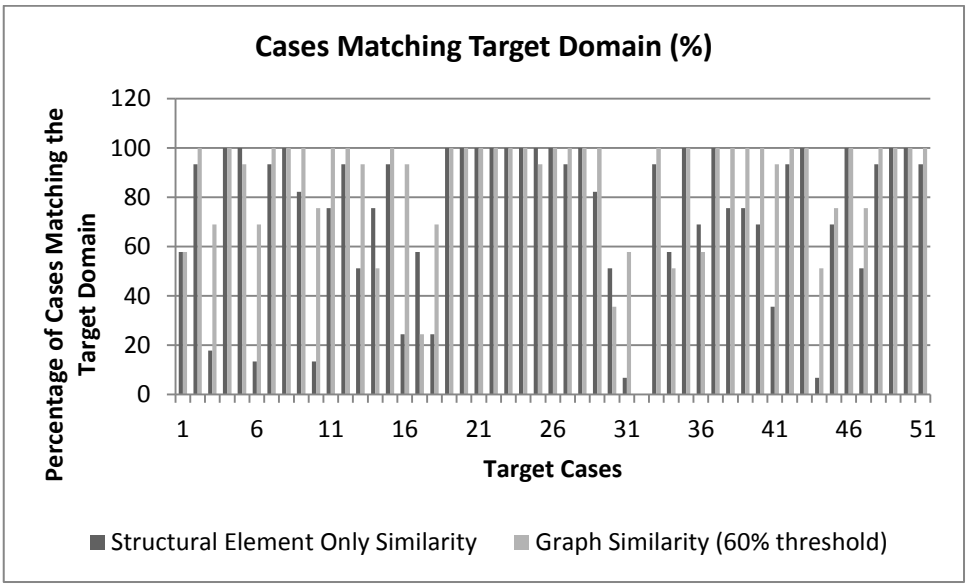

Figure 36 - Comparison of Structural and Graph Similarity for Measuring Matching Domain

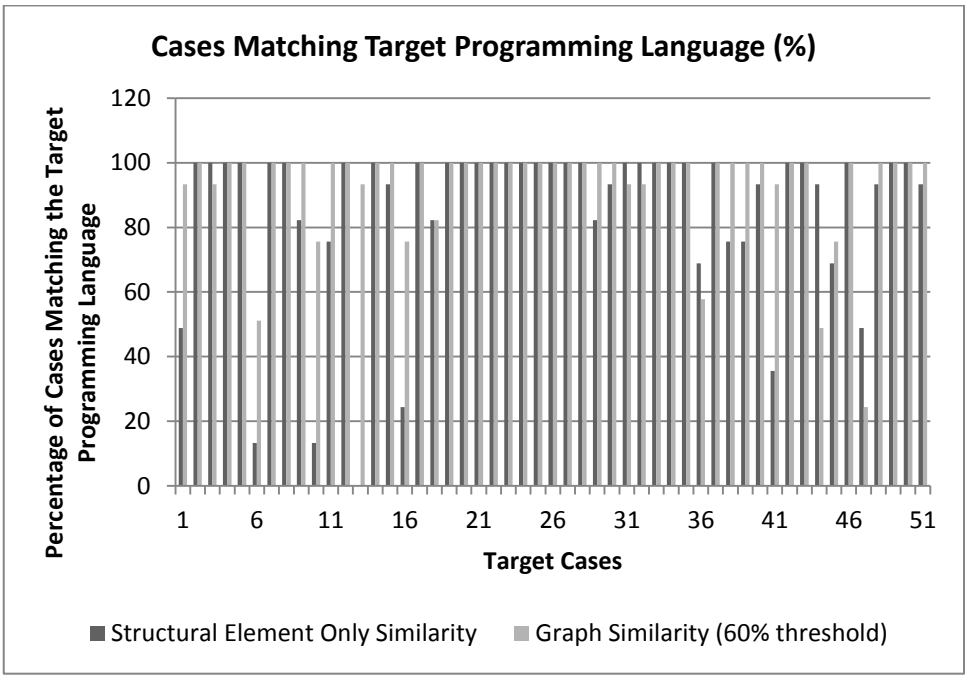

Figure 37 - Comparison of Structural and Graph Similarity for Measuring Matching Programming Language

The standard deviation for grades and lines of code decreased using graph similarity from 11.57 and 509.87 to 10.84 and 470.94. For complete breakdowns see Figure 38 and Figure 39.

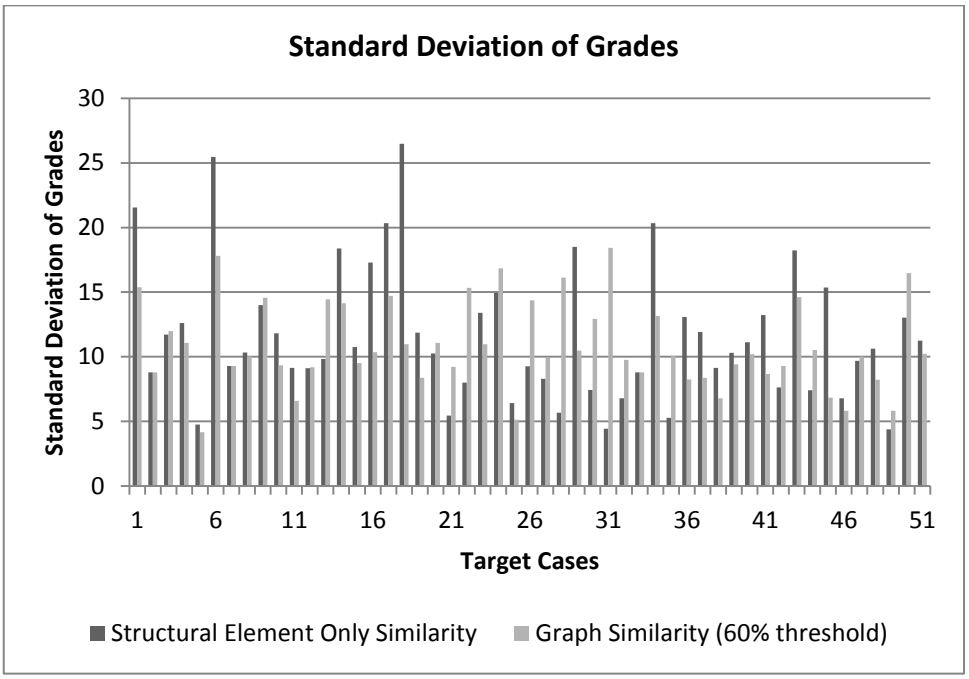

Figure 38 - Comparison of Structural and Graph Similarity for Measuring Standard Deviation of Grades

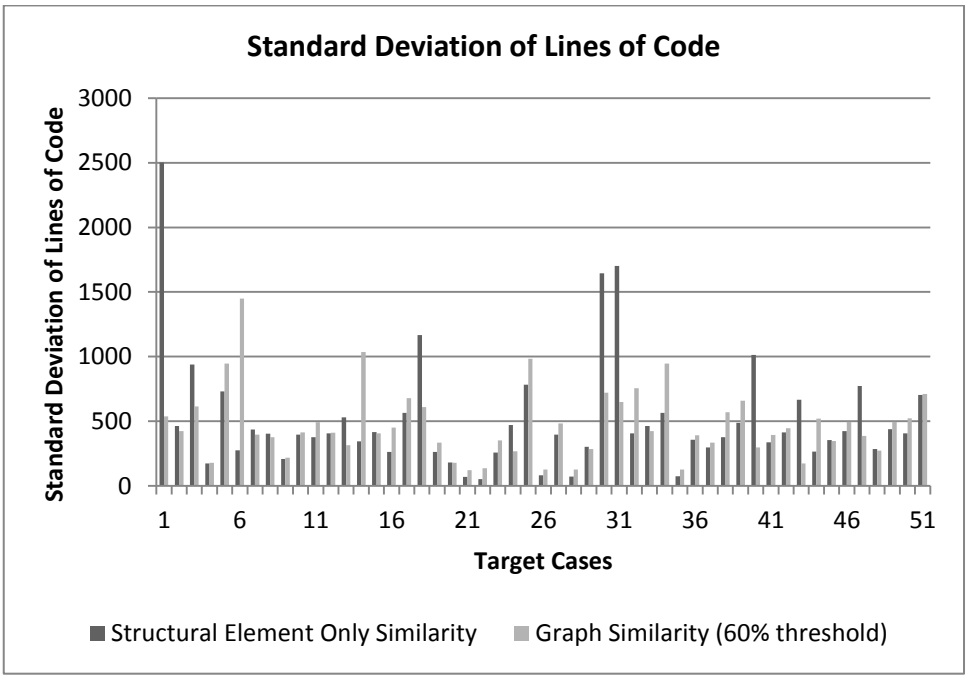

Figure 39 - Comparison of Structural and Graph Similarity for Measuring Standard Deviation of Lines of Code

As with the class structure experiments, the impact of using provenance was measured by selecting nearest neighbours for graphs only from the same domain.

The identification of cases with matching programming language improved very slightly when using slightly when using provenance. As can be seen in the graph on the next page (

Figure 40), the standard deviation also improved with the use of provenance, but only very marginally.

The standard deviation for lines of code, however, deteriorated by using provenance. It is still better than using structural similarity, but it decreased when compared to graph similarity without provenance. This shows that selecting cases only from the same domain doesn't guarantee that the results will be better in every category.

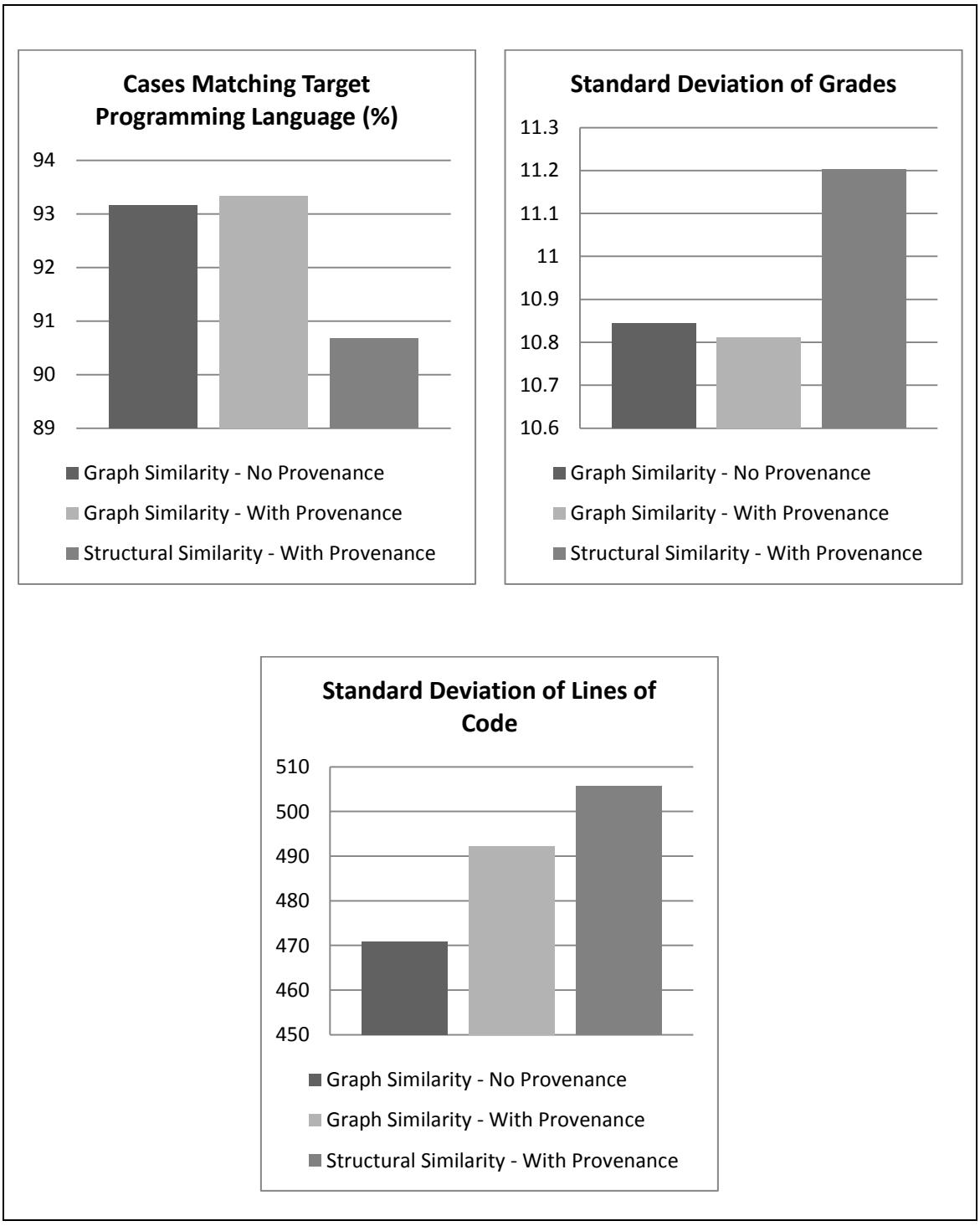

Figure 40 - Comparisons of the Effect of Provenance

A more detailed analysis of the results showed that the domains which performed worse for standard deviation of lines of code using provenance were the two with the highest standard deviation in the case-base, namely "Project management" and "Project

bidding" (see Table 13). With a widely dispersed set, it is more likely that the standard deviation of the nearest neighbours is dispersed.

Domain	Standard Deviation of Grades	Standard Deviation of Lines of Code
Software cataloguing	18.95	713.72
Project management	25.13	2184.33
Car repair shop	20.56	1959.67
Stock management	19.38	1101.73
Project bidding	19.17	2472.90
Overall Average	*20.77*	*1762.68*

Table 13 - Standard Deviation of Lines of Code and Grades by Domain

The experiments so far were all using a default weight setting, which meant that every feature is assumed to contribute equally within its level of the structural hierarchy. The next set of experiments introduces weight optimisation.

5.5 Weight Optimisation

The similarity metrics introduced in chapter 3 all include coefficients (ω_x) which determine the importance of a particular feature. By multiplying a measured similarity by the corresponding coefficient it is possible to determine to what extent the feature should contribute to the overall similarity of two classes. As the class structure is broken down into a hierarchy of features, the coefficients (weights) reflect this hierarchy.

A human expert in software engineering may be able to express what he/she would consider the most important features in determining similarity between class diagrams. The first step for weight optimisation was to ask the expert to identify how each feature should contribute to the overall similarity.

Table 14 shows a breakdown of the default weights and the weights determined by the expert. The default weights simply attribute the same value to each feature within a particular level of the structural hierarchy.

	Feature	Default Weight	Expert Determined Weight
	Final modifier	0.077	0.03
	Visibility modifier	0.077	0.03
	Abstract modifier	0.077	0.05
	Stereotype (enumeration/interface)	0.077	0.05
	Number of attributes	0.077	0.11
	Number of constructors	0.077	0.08
	Number of operations	0.077	0.15
	Superclass	0.077	0.03
	Number of implementations	0.077	0.06
	Number of associations	0.077	0.2
	Attributes (internal structure)	0.077	0.09
	Constructors (internal structure)	0.077	0.03
	Operations (internal structure)	0.077	0.09
Class Total		***1***	***1***
	Data type	0.25	0.4
	Final modifier	0.25	0.1
	Static modifier	0.25	0.2
	Visibility modifier	0.25	0.3
Attribute Total		***1***	***1***
	Visibility modifier	0.333	0.15
	Number of parameters	0.333	0.55
	Parameters (internal structure)	0.333	0.3
Constructor Total		***1***	***1***
	Visibility modifier	0.125	0.11
	Return type	0.125	0.28
	Final modifier	0.125	0.07
	Static modifier	0.125	0.11
	Abstract modifier	0.125	0.07
	Synchronised modifier	0.125	0.05
	Number of parameters	0.125	0.2
	Parameters (internal structure)	0.125	0.11
Operations Total		***1***	***1***

Table 14 – Default and Expert Weights

Comparing the results of graph similarity using the default weights and those determined by the expert show no clear improvements (Figure 41). While the percentages of matching domains and programming languages fell, the standard deviation of grades and lines of code improved.

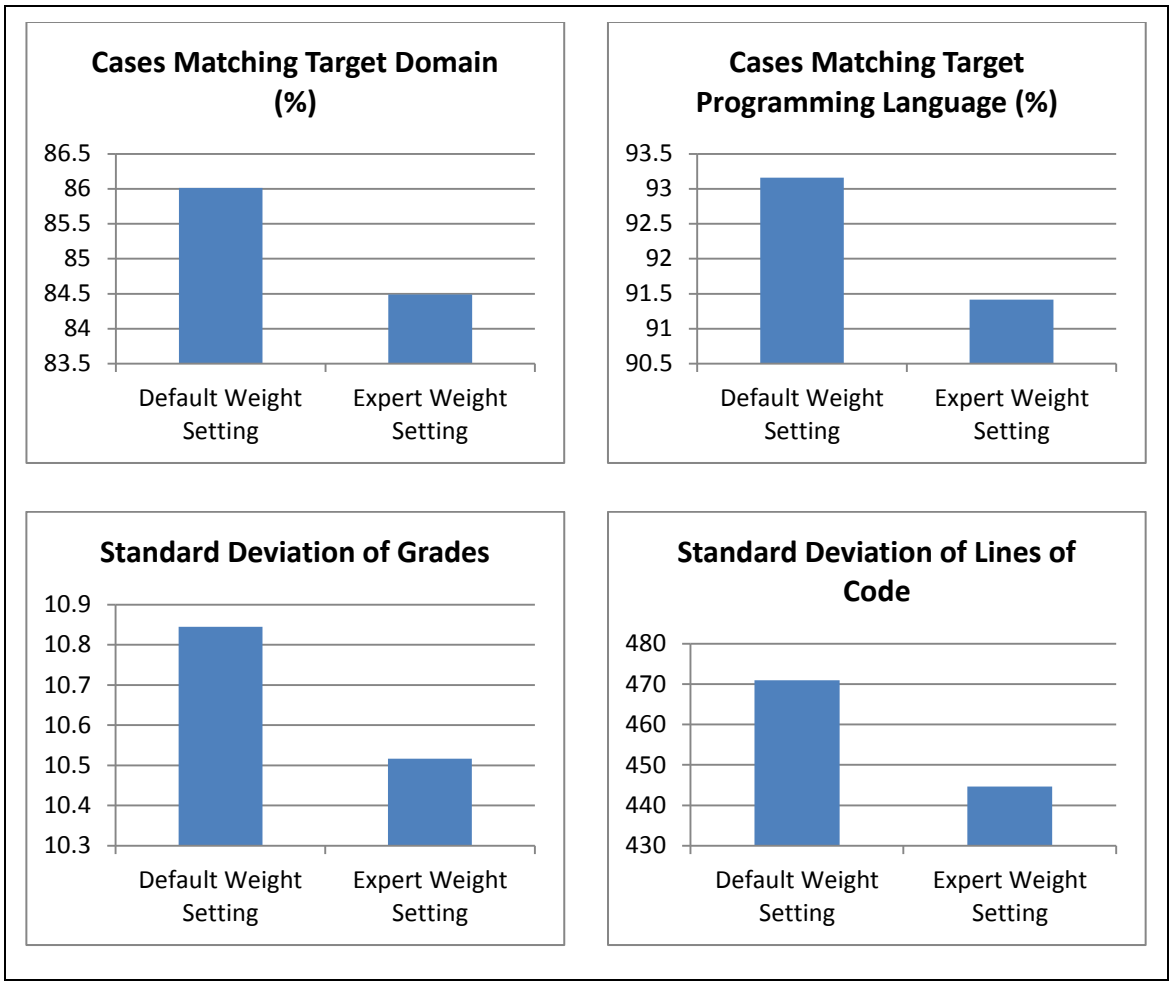

Figure 41 – Comparison of Graph Similarity Results Using Default and Expert Weight Settings

It is interesting to note that better results were achieved when comparing the class structure similarity (Figure 42), where the only category which didn't improve was the standard deviation of lines of code. This suggests that the expert weight setting performs better when measuring similarities between individual classes, but doesn't cater well for the relationships between classes.

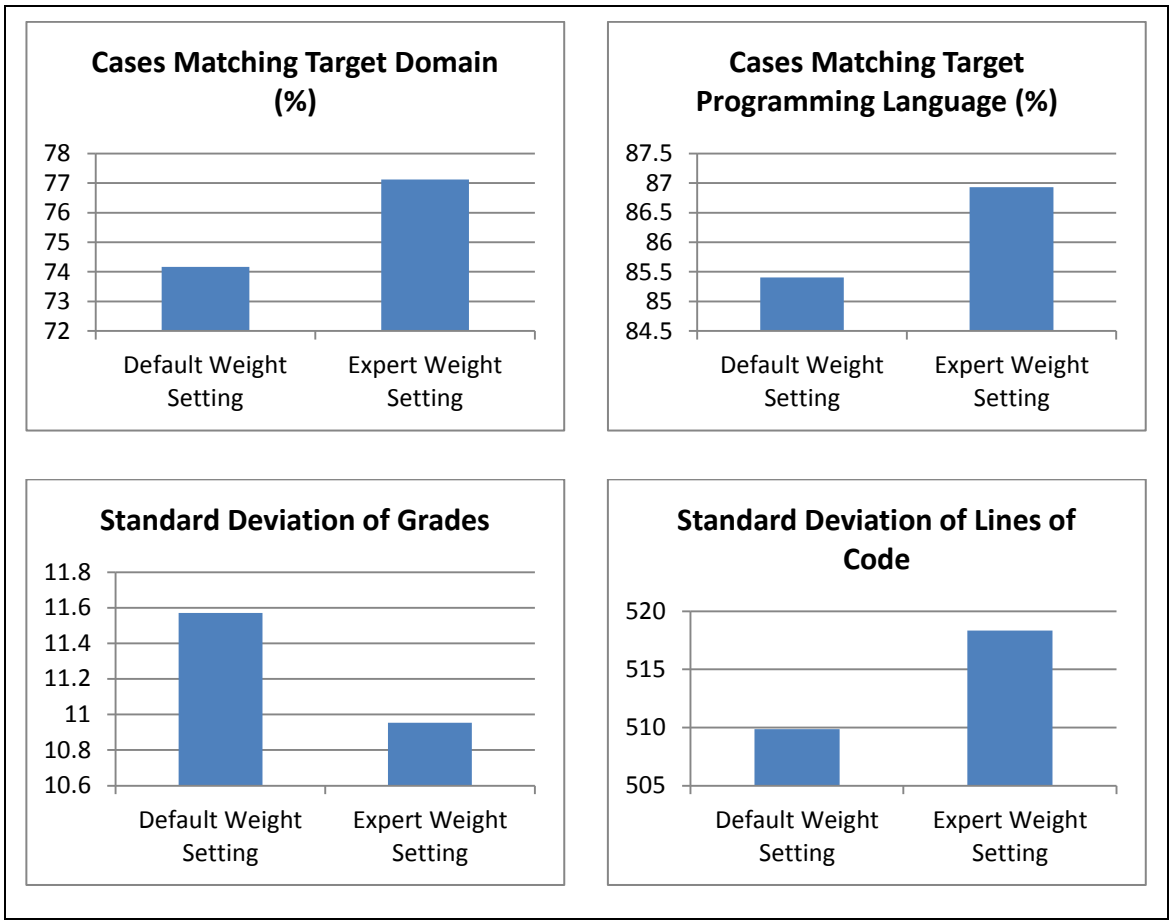

Figure 42 – Comparison of Structural Similarity Results Using Default and Expert Weight Settings

5.5.1 Automated Weight Optimisation

Chapter 3 outlined a different approach to weight optimisation, which automates the weight optimisation process by training the system to automatically identify the best weight settings. This is still based on expert knowledge, but rather than asking the expert to assign the individual weights, the expert works at a higher level of abstraction. The expert just assigns values from a predefined scale to set desired class matches. The system uses a genetic algorithm to test a variety of weight settings and selects those which achieve the highest score.

An important question is to what extent the optimisation of weights can be generalised and whether a weight setting obtained from a particular set of designs can be applied

successfully to other cases. To address this issue, the weight optimisation process was applied to each of the different domains to create different weight profiles.

The values used to set matches were: 2.5, 5, 7.5 and 10. The expert was asked to set desired class matches for five pairs of diagrams (one pair per case domain). Details of the matches defined by the expert can be found in Appendix 4.

Running the genetic algorithms to obtain the highest score did not identify a weight setting which would create higher scores than those obtained when using the default weights. While the different weight settings would result in different class similarity values, the granularity of the differences was not enough to modify the maximum common subgraph and thereby alter the score.

A variation of this approach was employed, which would still use input from a human expert to optimise weights, but would identify even small improvements in a weight setting. The expert was given a class diagram from each domain and asked to identify the most similar diagrams from a catalogue containing all class diagrams. The genetic algorithm manipulated and applied different weight settings, adopting the setting which yielded the highest similarity result. For details of the diagrams that the expert was given and which were chosen as the best matches, please look at Appendix 5.

Using this method, weight settings were obtained which improved the similarity between each pair of class diagrams. The complete breakdown of feature weights of all five weight settings can be found in Appendix 5.

A comparison of the results obtained using default weights and those obtained using the genetic algorithm is shown in Table 15.

Class Diagram Provided	Class Diagram Selected by Expert	Similarity using Default Weights	Highest Similarity Achieved using Genetic Algorithm
Software cataloguing - 3422	Software cataloguing - 3399	53.99%	56.22%
Project Management - 3427	Project Management - 3434	17.71%	22.23%
Car repair shop - 3453	Car repair shop - 3450	42.45%	47.71%
Stock management - 3488	Stock management - 3489	57.17%	72.06%
Project bidding - 3504	Project bidding - 3506	50.28%	57.56%

Table 15 – Comparison of Class Diagram Graph Similarities between Default and Generated Weights

Having generated five different profiles of weights, each one was applied in turn to measure similarity between all class diagrams in the case-base. In the first set of experiments, each weight setting was used on the entire training set of cases. This means that while a weight profile was optimised for one pair of class diagrams, it would be applied across the board. The reasoning behind this experiment was to determine whether weight profiles could be generalised, i.e. whether an optimised weight profile obtained from a particular experiment would yield good results from across the case-base. The results of these experiments can be seen in Figure 43, where the default weight profile and the five profiles obtained from the different domains were applied across all cases. When taken across the four categories that are being analysed, the default weight configuration outperformed the automatically generated weight profiles

overall, coming top in two of the four categories (cases matching target domain and standard deviation of lines of code).

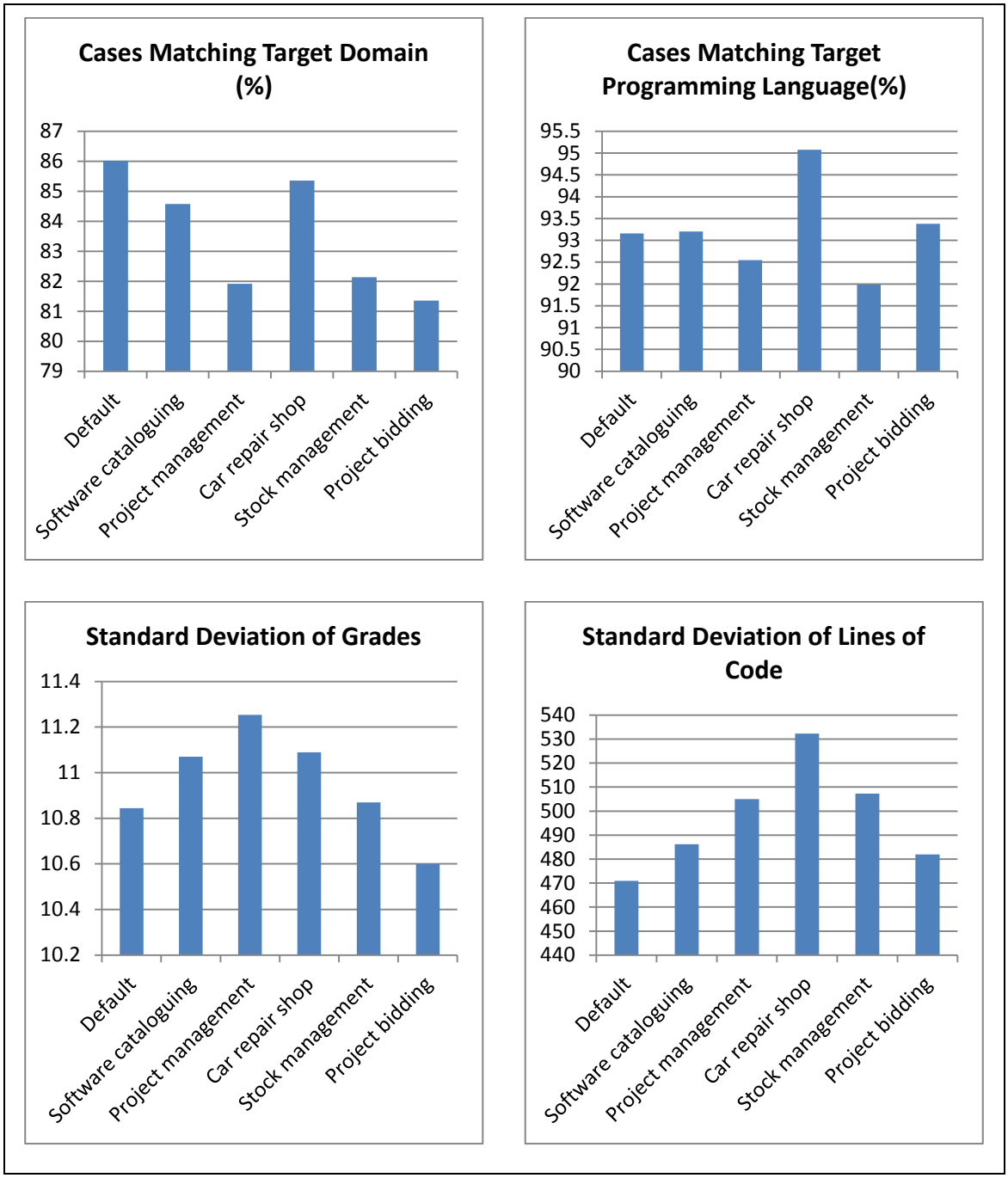

Figure 43 – Comparison of Graph Similarity Results Using Default and Generated Weight Settings

Given that the weight profiles were optimised by increasing the similarity of two class diagrams from the same domain, the next set of experiments was devised to verify whether a weight profile could be extended to other cases of the same domain.

Each weight profile was applied to identify the nearest neighbours from only the domain for which the weight profile was created. Table 16 shows the results of these experiments and detailed graphs of the comparisons per domain can be found in Appendix 6. Cases matching the target domain are not shown, as with provenance this value is always 100%.

Weights Applied	Cases Matching Target Programming Language (%)	Standard Deviation of Grades	Standard Deviation of Lines of Code
Software Cataloguing Domain			
Default	100	13.60	170.79
Software cataloguing	100	13.39	160.42
Project Management Domain			
Default	100	15.32	1137.36
Project management	100	15.23	1379.73
Car Repair Shop Domain			
Default	100	8.82	486.27
Car repair shop	100	9.14	503.99
Stock Management Domain			
Default	69.09	9.61	377.84
Stock management	70.10	8.86	387.97
Project Bidding Domain			
Default	100	15.12	1185.95
Project bidding	100	14.66	1167.28

Table 16 – Comparison of Class Diagram Graph Similarities between Default and Generated Weights using Provenance

Using the weight profiles did not always improve the results, but the overall picture is positive, as in over 63% the results improved. The category where the domain weight profiles performed best was the standard deviation of grades.

A problem with getting the system to automate the weight optimisation process is that due to the complexity of the model and the large number of parameters involved, it is prone to overfitting. This is due to the fact that the weights obtained through the genetic algorithm may describe noise in the form of elements which happen to be the same and therefore increase the overall match during the optimisation process, but which can't be generalised or don't describe the match in a meaningful way.

5.5.2 Disabling Features

The experiments carried out with weights so far all attempted to modify how similarity is measured by changing the values of the coefficients of the various features in the hierarchical structure. A further question considered was whether some of the features should not contribute to the overall similarity at all.

An expert in software engineering was asked to use expert judgement to identify the features which should be included in the similarity measurements. The expert identified the following features as relevant:

- Number of associations
- Number of operations
- Number of constructors
- Return type of operations
- Data type of attributes
- Number of parameters for constructors and operations
- Data types of parameters for constructors and operations

Experiments were carried out with all remaining features disabled. An interesting consequence of measuring the similarity based solely on the above features was that the overall similarities of class diagrams were lowered dramatically. The sizes of maximum common subgraphs were also lowered considerably and in some cases the system was even unable to identify any maximum common subgraphs for some of the class diagrams using higher minimum class thresholds (80% and 60%).

An investigation into this behaviour revealed that some of the features which were disabled were found to be identical for the majority of classes in the case-base. These features include the final and visibility modifiers for classes, synchronised and final modifiers for operations and the visibility modifier for constructors. Having these features matched in the majority of cases means that the similarities are increased overall. To cater for this, the minimum class similarity threshold was lowered to 20% for the experiments involving a reduced set of weights.

Applying a reduced set of weights didn't improve the results achieved with a default weight setting, which were lowered in three out of four categories (see Table 17).

Weights	% matching target domain	% matching target programming language	Standard deviation of grade	Standard deviation of lines of code
All weights	84.49	92.81	10.79	481.03
Subset of weights	86.97	91.81	10.80	504.64

Table 17 – Comparison of Results for Graph Similarity Using All and a Subset of Default Weights

When applied to the expert weight profile, however, the results improved in three out of four categories (see Figure 44). This demonstrates that the weight profile set by the expert is further improved by the expert not merely identifying the values of each

feature's weight, but also indicating which features should be ignored altogether. The system identified cases matching the same target domain better and also lowered the standard deviation for grades and lines of code, despite the fact that restricting the number of weights resulted in an average drop of 4.56% in the similarity results calculated between the training data and case-base.

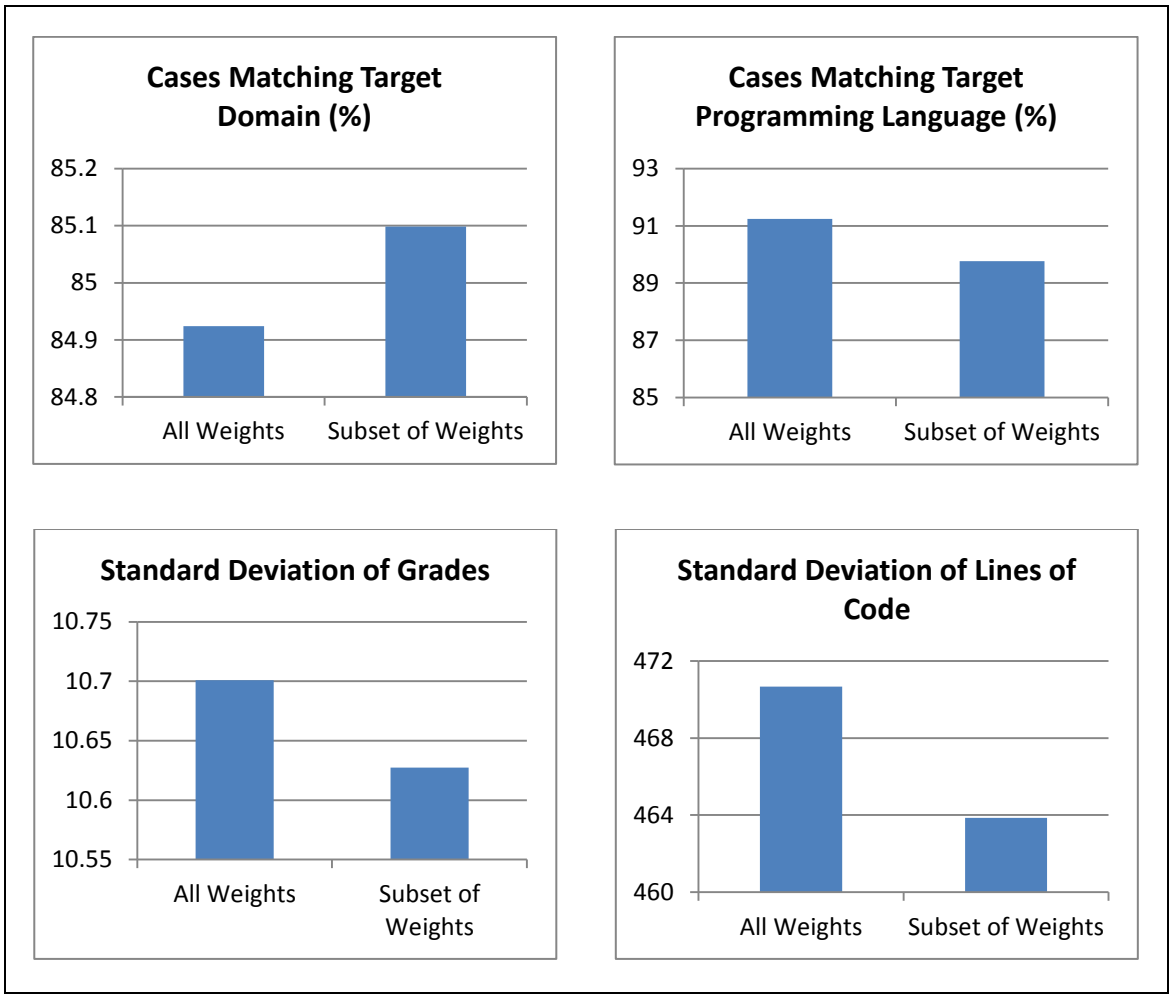

Figure 44 – Comparison of Results for Graph Similarity Using All and a Subset of Expert Weights

The experiments discussed thus far have been based on analysis of the nearest neighbours retrieved for the training data set. Another way to evaluate the results is by using clustering.

5.6 Clustering

Clustering assigns items into groups of similar objects, based on their distances. In this work, clustering is applied to the ranked cases and used to divide the entire case-base into a set of clusters. Initial tests with single-link clustering found that this method resulted in one cluster containing the large majority of all cases and the remaining clusters containing a single case each. Better results were achieved using average-link clustering, which merges the clusters with the smallest average distance of pairs of cases from both clusters. This resulted in a more even distribution of cases across the clusters.

As explained in chapter 4, the number of clusters for each experiment was fixed at 8.

The first set of clusters was created based on structural similarity. This resulted in a very bipolar distribution of cases, with two large clusters containing 48 and 45 of the 101 cases and the remaining 6 clusters together containing only 8 cases. An analysis of the clusters revealed that although there were only two relevant clusters, one contained class diagrams with an average of 3540 lines of code and an average grade of 70.38%, while the other contained an average of 1887 lines of code and average grade of 53.22%. This suggests that there was a trend to group the higher quality work into one cluster and the lower quality work into another.

When applied to all cases based on graph similarity the distribution became more even (Table 18).

While there were still two larger clusters, an analysis of their content showed that cluster 7 contained 32 of the *Car Repair Shop* class diagrams and cluster 6 contained 16 of the *Software Cataloguing* class diagrams. Even with other clusters there was a tendency to group class diagrams from the same domain. For instance, 70% of cluster 8 was made up of *Project Bidding* class diagrams.

Cluster	Structural Similarity Only	Graph Similarity
1	48	9
2	1	3
3	1	3
4	1	2
5	1	6
6	1	23
7	3	45
8	45	10

Table 18 – Number of Items per Cluster for Structural and Graph Similarity

Experiments were then carried out by creating different sets of clusters for each of the automated weight profiles to determine whether the weight profiles would improve the clustering of diagrams from the same domain. The results were as follows:

- *Software cataloguing weight profile:* The largest number of cases from the same domain in the same cluster dropped from 16 to 11. However, it generated a cluster of 14 cases which consisted to 79% of Software Cataloguing class diagrams.
- *Project management weight profile:* The largest number of cases from the same domain in the same cluster decreased. A small cluster was created containing only one case which was not from the Project management domain.
- *Car repair shop weight profile:* The largest number of cases from the same domain in the same cluster increased. Created a cluster containing all but one of the Car repair shop class diagrams.
- *Stock management weight profile:* The largest number of cases from the same domain in the same cluster increased and created a cluster consisting to 75% of

Stock management class diagrams and which also contained 41.18% of all diagrams implemented in Java.

- *Project bidding weight profile:* The largest number of cases from the same domain in the same cluster increased.

In three out of five cases the largest number of diagrams from the same domain in a single cluster increased. Even in the two cases where it didn't, applying the automatically generated weights for that domain created a more focussed cluster.

The final experiments with clustering were applying the expert weight profile and a reduced set of features (Figure 45).

For identification of the domain and programming language the expert weight profile with a reduced set of weights performed best, but this setting was the worse for standard deviation of lines of code, where the default settings fared better. The standard deviation of grades was worse for the default weight profile with all weights and very similar for the remaining three.

There was no clear trend across all four categories and different weight profiles performed differently in different categories.

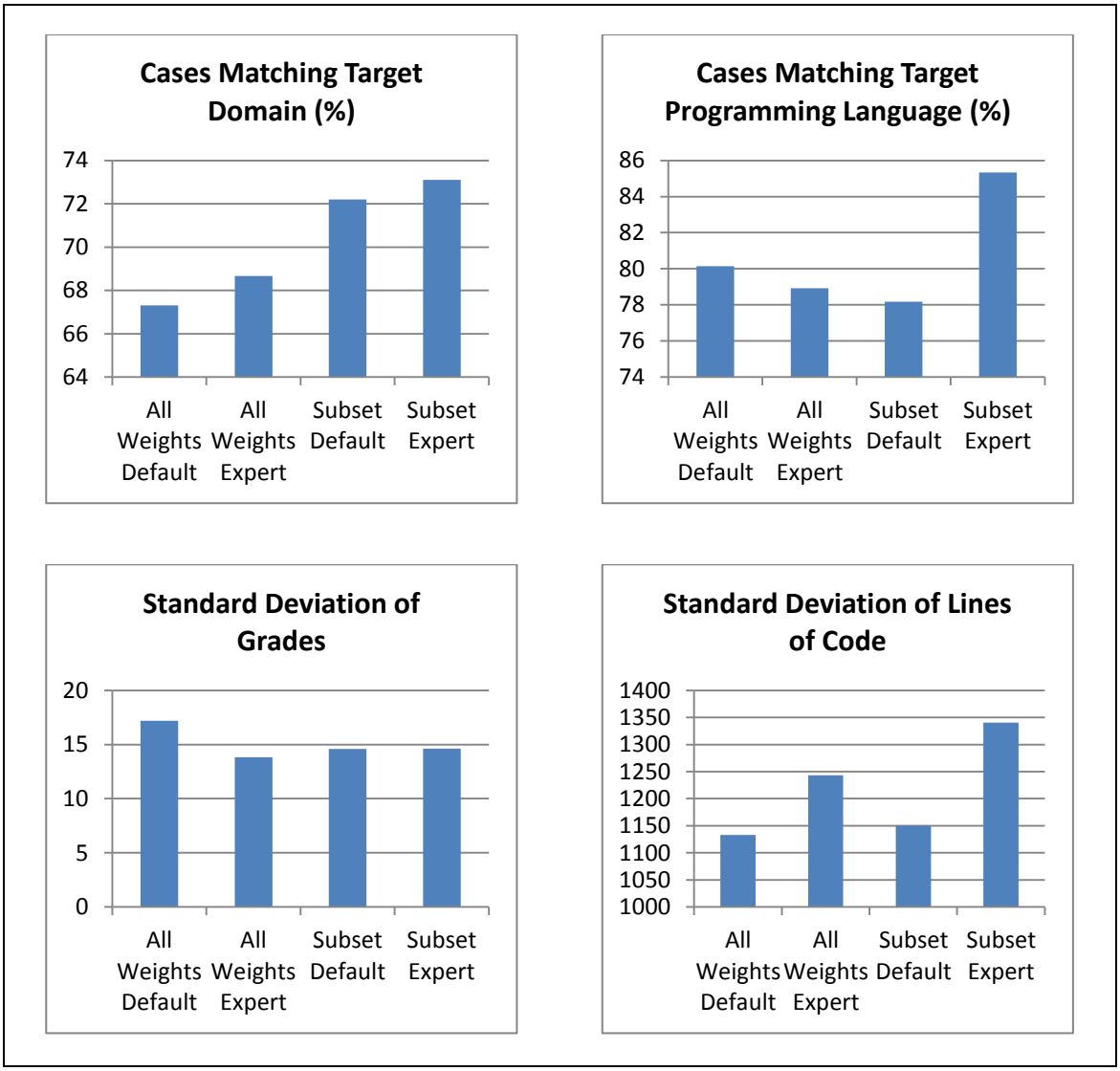

Figure 45 – Comparison of Clusters for Graph Similarity Using All and a Subset of Default and Expert Weights

5.7 Plagiarism

The class diagrams in the case-base all represent assignments completed by students. Two of the assignments from the *Software Cataloguing* assessment (3397 and 3416) were known to be plagiarised, as they were submitted for an assessment offence and found guilty.

The two applications from which the class diagrams were reverse-engineered were very different in the way the user interface looked and had some differences in the functionality implemented. The underlying classes had different names and different structures (Table 19).

Feature	3397	3416
Lines of Code	1179	1050
Number of Characters	40786	35350
Number of classes	8	8
Number of Relationships	10	9
Number of Attributes	56	51
Number of Constructors	9	9
Number of Operations	65	53
Number of Parameters	56	43

Table 19 – Comparison of Plagiarised Class Diagrams

The reason the two assignments were submitted for plagiarism was because some of the internal implementations of operations were found to be identical. Internal contents of operations and constructors cannot be obtained through reverse-engineering. This information is therefore not included in the class diagram structure recorded in the case-base. The similarity could be measured solely on the class structures and arrangement of classes.

The findings were that out of the 70 sets of experiments, using class structure similarity, graph similarity, seven different weight profiles and full or reduced set of weights, the two class diagrams were identified as the closest match in all but 7 cases.

Due to the different weight profiles and types of comparison, the actual similarity between the two cases varied between 30.01% and 99.7%, with the average being 91.57%. Nonetheless, even in the cases where the two class diagrams were not found as the highest match for each other, they would always rank between 2nd and 4th closest match and differ from the highest match by between 2.07% to 8.37%.

This demonstrates that the structural approach is well suited for detecting plagiarism in class diagram structures. A semantic approach may not have identified matches such as SQLPersistence and CatalogueComponent or SaveXML_Click and button4_Click. It is important to note, though, that this is the first stage of a two-stage process. The approach presented here can do filtering and flag high matches, but this would have to be verified by a human expert and potentially confirmed through a formal interview of the student suspected of committing the act of plagiarism.

5.8 Cost Estimation

The most common approaches for software cost estimation are formal estimation models. COCOMO (COnstructive COst MOdel) is one such model and was established by Boehm [Boehm, 1981]. It uses function point analysis to compute the application size, development time and the number of people required for a software development project. The calculation of the function points requires weight assignment and establishment of complexity factors which must be carried out by a human expert.

To calculate the lines of code required to implement a design, COCOMO uses the number of use-cases and number of classes in addition to complexity coefficients set by the expert. However, complexity can also be obtained from the internal structure of classes and using historical/statistical data. In fact, the formulas applied in COCOMO were created based on historical and statistical information.

Experiments were carried out in order to evaluate the suitability of measuring structural similarity between class diagrams to estimate cost and compare the results to COCOMO.

Five class diagrams were randomly selected (one from each domain). The estimated lines of code for implementing each diagram were calculated using COCOMO (see Table 20 for details).

	Software Cataloguing - 3419	**Project Management - 3427**	**Car Repair Shop - 3450**	**Stock Management - 3481**	**Project Bidding - 3504**
Number of use cases	2	6	8	5	3
Use case complexity coefficient	7	6	4	5	7
Number of classes	10	16	12	11	12
Class complexity coefficient	13	11	9	10	11
Function points	144	212	140	135	132
Sum of technical complexity factors	25	25	12	12	24
Adjusted function points	129.6	190.8	107.8	103.95	136.17
Estimated lines of code	**3888**	**5724**	**3234**	**3119**	**4085**

Table 20 – Lines of Code Calculation using COCOMO

Once the COCOMO estimates had been calculated, these were compared against the actual lines of code it took to implement each of the class diagrams (Figure 46).

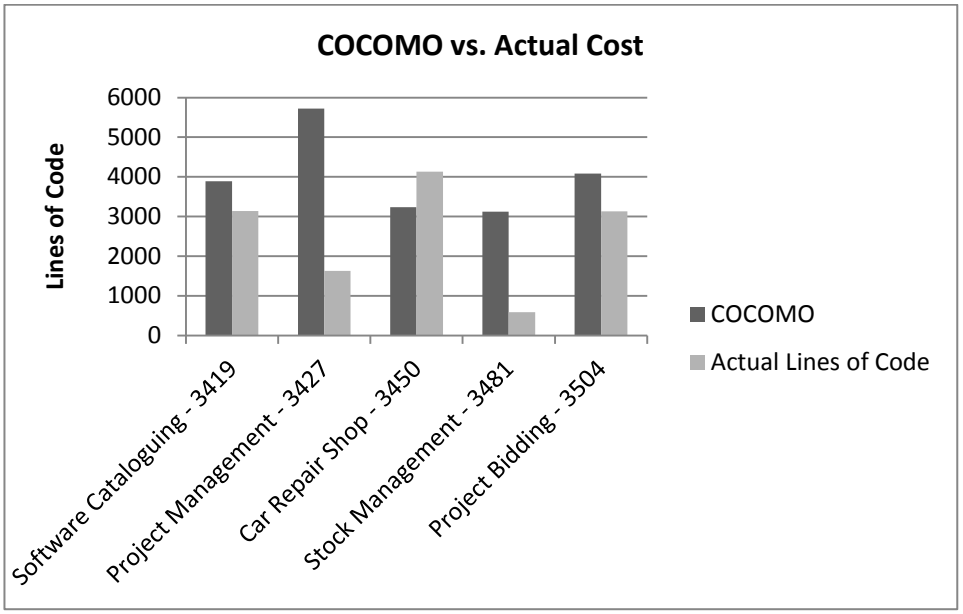

Figure 46 - Comparison of Actual Lines of Code and COCOMO Estimates

In three of the diagrams COCOMO estimates were quite accurate, but two were considerably higher than the actual implementation. Across all five diagrams the difference between actual implementation and the COCOMO estimates was 9238.

The COCOMO estimates were then compared to the lines of code obtained using a nearest neighbour retrieval applying only structural similarity (Figure 47).

The structural similarity estimates performed better than COCOMO for two of the diagrams (3427 and 3450), but COCOMO was more accurate for the remaining three (3419, 1481 and 3504). However, even with those three, COCOMO was only marginally better (127, 380 and 213 lines of code respectively) and the overall difference between the actual lines of code and those estimated was only 5796.

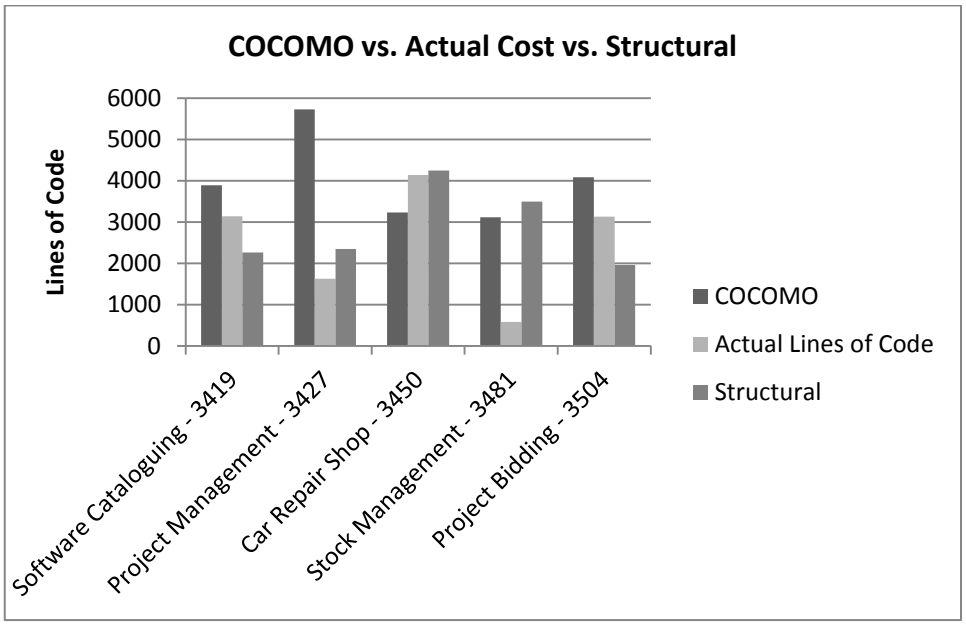

Figure 47 - Comparison of Actual Lines of Code, COCOMO and Structural Similarity Estimates

The next experiments compared COCOMO estimates to those achieved using graph similarity (Figure 48).

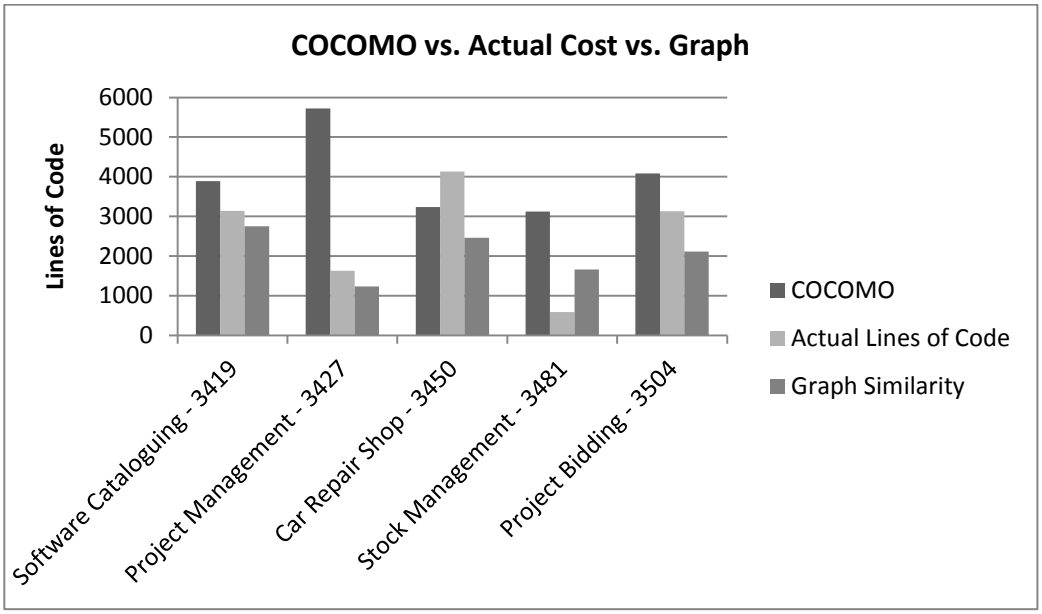

Figure 48 - Comparison of Actual Lines of Code, COCOMO and Graph Similarity Estimate

The graph similarity improved the results achieved using only class structure similarity. The estimates were more accurate than COCOMO for three diagrams (3419, 3427 and 3481) and the total difference of lines of code was further reduced to 4556. It is interesting to note that graph similarity performed worse than structural similarity for the Car Repair Shop. This was an assignment at level 5 and less complex, so it would be simpler and produce solutions based on more basic design skills.

The final set of experiments applied provenance, by choosing only the nearest neighbours from the same case domain (Figure 49).

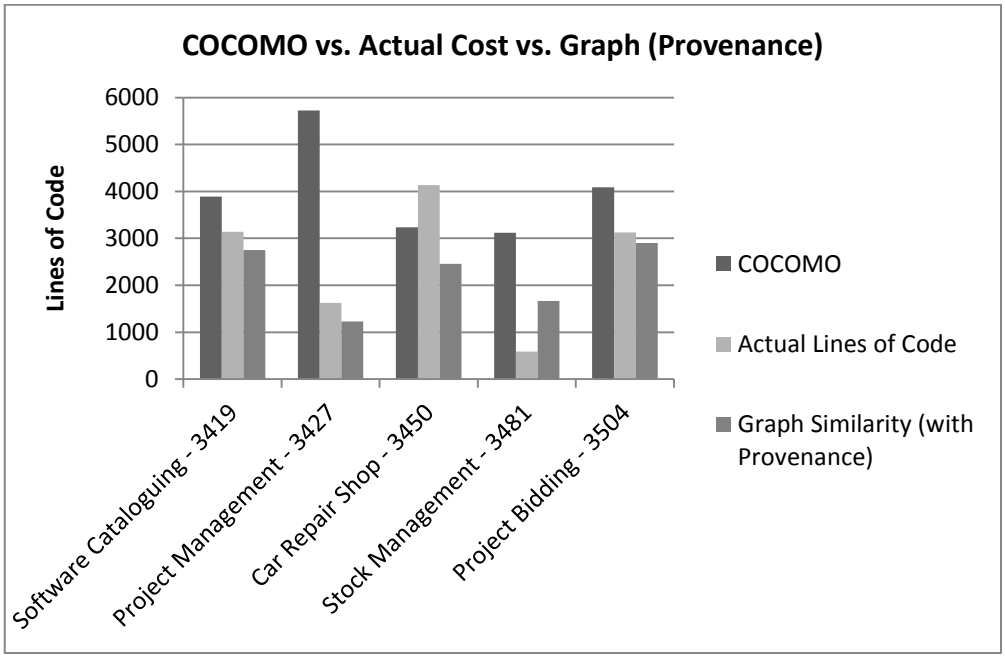

Figure 49 - Comparison of Actual Lines of Code, COCOMO and Graph Similarity Estimate using Provenance

By applying provenance, the lines of code are obtained only from cases that are from the same domain. The results are very similar to those achieved only with graph similarity, but using provenance improved the results further and COCOMO performed better in only one instance (3450). The total difference of estimated lines of code and

the actual lines of code was also reduced further to 3766 which is much lower than the 9238 lines of code difference with COCOMO.

In an industrial setting, the provenance approach could be used to discriminate between development teams, companies, technologies used in a software development project or even classification by types of applications.

These experiments have demonstrated that it is possible to perform cost estimation of the implementation of class diagrams by measuring the structural and graph similarity between the class diagrams. The results are comparable to COCOMO at worst and better when applying graph similarity and provenance.

5.9 Explanation

Given that there is no established standard or benchmark for measuring similarity between two class diagrams, it is important for the system to provide an explanation of how results were achieved.

Case-based reasoning is well-suited for explanation, as it can use retrieved cases in order to explain prediction. It is based on actual existing cases which can be presented to support the system's conclusions. The explanation goals which are most applicable to this work are *transparency* and *justification*.

Transparency requires an explanation of how the system reached a particular answer. In this case, it needs to convey an understanding to the user of how the system reached a particular similarity between two given diagrams. While it is important for the system to allow a user to examine the way it reasons to increase trust, the transparency was also very important during development to verify whether the reasoning process was

correctly implemented. Justification increases the confidence in the solution offered by the system by providing support for the conclusion it provides.

The reasoning used to obtain a result is remembered by the UMLSimilator system and it can therefore provide a user with a complete breakdown of how constituent elements have contributed to reach an overall similarity match. This makes the reasoning process transparent to a user who wishes to understand how the similarity between classes was created.

As class diagrams are visual artefacts and the similarity of two class diagrams is also based on the maximum common subgraph, UMLSimilator further allows a user to view the maximum common subgraph in order to increase trust in the conclusions of the system.

Experiments were devised to use explanation to verify whether it would aid trust in the system. The following steps were taken:

- Experts were asked to guess similarity between given diagrams
- The overall matches calculated by the system were provided and presented to the experts
- Experts were asked to confirm whether they trusted that the results produced were correct
- The overall matches were supported by a complete breakdown and the maximum common subgraph
- Experts were asked to re-confirm trust having seen the explanation outlining how the results were obtained

Details of the expert guesses and the results calculated by the system are shown in Table 21.

Class Diagram 1	Class Diagram 2	Expert 1 Guess	Expert 2 Guess	Measured Similarity[4]
Software cataloguing - 3388	Software cataloguing – 3392	50%	45%	42%
Project management - 3423	Project management – 3432	60%	60%	23%
Car repair shop - 3437	Car repair shop – 3466	70%	55%	51%
Stock management - 3478	Stock management – 3496	50%	45%	36%
Project bidding - 3502	Project bidding - 3506	40%	50%	50%

Table 21 – Expert Similarity Guesses and Similarity Measured by the System

An analysis of the results shows that the guesses of expert 2 were much closer to the results achieved by the system and were in the same ballpark region, with the exception of the similarity between 3423 and 3432, where both experts had a much higher figure than that provided by the system. The reason for the low match of 23% is explained by the fact that out of the seven possible matches, the maximum common subgraph only contained four. Matches between these four classes were very high (all over 90%).

Both experts agreed that providing a breakdown of the results and the visualisation of the maximum common subgraph increased the trust in the system. The low similarity calculated for 3423 and 3432 came as a surprise and would have been difficult to accept at face value, so this is where the additional explanation provided was perceived as most essential – to justify an unexpected result.

[4] Similarity was measured using graph matching and 60% minimum class threshold

Both experts were asked to identify on what they based their comparison and while both agreed that the semantic information was important, expert 2 stated that the overall size and structure of the class diagrams also played an important role. This could account for the reason why the predictions of expert 2 matched the similarities calculated by the system more closely. Expert 1 stated that having a maximum common subgraph where semantic terms were not matched correctly was a little confusing and that the formula used for measuring graph similarity could be shown more explicitly. For example class *AddCar* had been matched with class *addManufacturerFrm* and class *RequestOrder* was matched with *warehouseArgs*.

The additional information provided by the system regarding a similarity match increased the trust of both experts.

5.10 Conclusion

This chapter discussed the various experiments and analysed their outcomes. An assessment was made of the approach and the methodologies used and results were evaluated to determine their suitability. Combinations of techniques were also compared to verify whether they would improve results.

Part of the evaluation was made by comparing results from alternative approaches, but where necessary experiments were validated against expert opinion.

The setup for the experiments was described, providing details of the technical environment and of the data used for the experiments.

The approach followed in carrying out the experiments was explained, whereby a suite of extensive tests were devised to test structural similarity, graph matching and using optimised weighting.

Given that comparing class diagrams is an abstract process, this poses the question of how the competence of the results could be measured. As additional information was known about the class diagrams, this data could be used to measure whether the system identifies information such as the domain from which a case was sourced, the programming language it was built in, the grade it was attributed or the cost of the implementation. Furthermore, competence could be measured by comparison to known alternative methods, such as cost estimation using COCOMO, or through evaluation by human experts.

The first sets of experiments used only structural similarity, which confirmed that the target domain and programming language was correctly identified in 74% and 85% of cases. For grades and lines of code, the standard deviation was used to measure how focussed the nearest neighbours were – the lower the standard deviation, the better the results. In addition, the standard deviations of the nearest neighbours were compared to the standard deviation across the entire case-base and were found to be much lower in both instances. An analysis of the number of nearest neighbours used revealed that the lower the number, the better the results.

The impact of choosing cases from the same domain as the target case was verified (provenance). It was found that overall, this improved results, but only very marginally.

Moving from only class structure similarity to graph similarity, the computation times increased dramatically. Due to the complexity of calculating the maximum common

subgraph, the larger class diagrams came close to the limit of execution times considered acceptable for this research.

Graph similarity was measured using minimum thresholds of 20%, 40%, 60% and 80%, with the best overall results achieved using 60%. The results improved in every category over structural similarity. Using provenance improved the matching programming language and standard deviation for grades, but the standard deviation for lines of code deteriorated.

Many experiments with weight optimisation were carried out. Using weights determined by an expert didn't show a clear improvement with graph similarities, but interestingly improved the class structure results.

Using genetic algorithms and scoring of class matches did not work, so an alternative approach was taken which required an expert to identify the closest match for a given diagram from the same domain. The genetic algorithm would then run experiments with different weights and select the one giving the highest results. The weight profiles obtained using this technique, were first applied across the entire case-base, where they yielded poorer results than the default weights. However, when each weight profile was applied only to cases from its domain, the results improved in over 63% of cases.

Further experiments with weighting were using only a subset of features determined by an expert. The results using default weights were lowered, but when combined with the expert's weight profile, the results improved in three out of four categories.

Clustering was applied to further validate the results. The entire case-base was clustered into eight groups. Clustering showed that clusters generally had a motive. With class structure similarity the majority of cases were split between two clusters where one seemed to contain the class diagrams with higher lines of code and higher grades.

Clusters based on graph similarity were more evenly distributed and were much better at grouping class diagrams from the same domain. This was improved further within domains when using the weight profiles for each of the domains. Using subsets of weights showed no clear trends, improving in some categories, while declining in others.

A known case of plagiarism from the cases in the case-base made it possible to test the suitability of the system to detect plagiarised class diagrams which had been modified to avoid detection. The results were very good, with the plagiarised cases being paired as the highest matches in 90% of all the experiments.

A comparison between COCOMO and the system showed that while COCOMO was comparable to results obtained using class structure similarity, when applying graph similarity and especially graph similarity with provenance, UMLSimilator achieved better results with these cases than COCOMO.

Finally, a set of experiments were devised to check whether the system's ability to provide explanation would improve user's trust in its results and reasoning process. This was found to be the case, in particular with a case where the results provided by the system were unexpected.

The next chapter concludes the thesis by outlining the main contributions of this research and discussing potential future directions in which it could evolve.

Chapter 6

6 Conclusion

The results presented in the preceding chapter make it clear that it is possible to retrieve knowledge from class diagrams by applying case-based reasoning and using only structural information.

This chapter will outline the main contributions of this work and look at issues arising from it, as well as proposing how this research could be projected forward. To start with, the research objectives are presented to determine whether these have been met. The results of the experiments are summarised and the main contributions of this work highlighted. Finally, ideas of how this research can evolve further, but which are outside the scope of this work, are presented.

6.1 Review of Research Objectives

Software designs are an integral part of current software development. They depict the structure and distribution of responsibility within a planned software application. A software design captures functional and non-functional requirements, but there is no exact method dictating how these are translated into a design. Thus, there are many different shapes a software design can take, even if the requirements are well-established. A sound design would enable the creation of reusable modules, manageable and intelligible code and code which is maintainable, flexible and extendible. The design is therefore important for a developer as it guides the creation of the software artefact, but as it reflects the structure of the software artefact, it encodes information that can be used for other purposes. It conveys the complexity of a software application

and describes the domain of the application by depicting its concepts and how they relate. Concepts in the design are expressed through their structure, but also through the semantic tags used to name the classes and their constituent elements. While the semantic information is very important to facilitate human understanding, it is of no relevance to the runtime behaviour of the implemented application, which is determined solely by the structure of the code constructs. This suggests that the structural information is clearly significant and embodies knowledge about a software application. Automatically retrieving knowledge from software designs in the absence of semantic information has been the focus of this work.

A software design can be contextualised by comparing it to other designs, which is especially important when semantic information is not taken into consideration. The comparison of software designs is an abstract undertaking and no existing standard or benchmark exists for achieving it. In order to compare two class diagrams using structural similarity requires reducing them to hierarchical structures. Case-based reasoning is a good methodology to apply in this case, as it can contextualise design diagrams through their nearest neighbours. The target case is placed into context based on the cases that are found to be most similar.

Structural similarity between software diagrams is useful in domains where the structure is more important than the semantic information, such as cost estimation and plagiarism detection. With the availability of additional information about designs, one can also attempt to determine properties of an implementation such as the quality, its domain and the programming language in which it was implemented.

Chapter 1 defined the aim of the research as being summarised in the following question:

- Is it possible to effectively retrieve knowledge from structural software engineering artefacts?

This was then broken down into more specific aims:

- In the absence of semantic information, is it possible to extract meaningful knowledge merely from the structural information, and can this retrieval process be automated?
- Can case-based reasoning be applied to software designs to identify software systems from the same domain based on their structure?
- Can structural similarity and case-based reasoning be combined to estimate software development cost?
- A user's trust in an automated system is reinforced if the system provides explanation of the results it provides. A system which applies case-based reasoning, measures similarities of cases to reach a result, which makes it even more important to provide explanation. Thus a further aim of the research is for the developed system to explain why it reached a particular result.

The results of the experiments validate the conclusions that it is possible to retrieve meaningful knowledge from class diagrams even when semantic information is removed. A software application was developed to automate this process by implementing the proposed algorithms and procedures. With an automated process, a user's trust is reinforced if the system can provide an explanation of how the results were achieved, so this functionality was included in the system.

Controlled and planned experiments were carried out to test the various algorithms and methods. Cross-validation was applied to reinforce confidence in the results.

6.2 Conclusions from the Experiments

The research experiments have provided results which demonstrate that it is possible to retrieve knowledge from class diagrams based only on their structural information. Some of the techniques and approaches employed have been more successful than others, but overall the results achieved have been positive.

Experiments were carried out on 101 class diagrams obtained from five different coursework assignments, resulting in a varied case-base containing cases from five domains. A suite of experiments were devised to evaluate which techniques would provide efficient solutions. The experiments used:

1. Classes only (without graphs)
2. Graphs by calculating the maximum common subgraph
3. Classes only with optimised weights
4. Graphs with optimised weights

Validation of the results was achieved by carrying out cross-validation to measure the accuracy of each technique or combination of techniques in detecting the domain of a case, the implementation programming language, grade achieved and lines of code. Clustering was also used to further validate the results. Regarding cost estimation, the results were compared to COCOMO. Wherever necessary, input from human experts was used to aid validation.

6.2.1 Structural Similarity

Experiments using structural similarity were applying similarity metrics comparing the internal structure of classes, taking into consideration the number of relationships of a class, but not the actual distribution of classes (where the relationships lead).

The results obtained using the class similarity metrics were all better than the case-base average. The target domain was matched in 74.16% of cases and the programming language in 85.40% of cases.

To measure how well the case-based reasoning system predicts the quality of a given target case, the standard deviation of grades for all class diagrams was calculated to be 20.77. Using structural similarity, the average standard deviation for all target cases was reduced considerably to 11.57. The standard deviation for lines of code improved even more. The case-base average being 1762.68 and the average result measured from the nearest neighbours of all target cases measuring only 509.87.

These findings showed clearly that the structural similarity performed much better in predicting values in these categories than the case-base averages.

Further experiments combined the structural similarity metrics with provenance, by selecting only nearest neighbours from the same domain as the target case. This improved results even further across all categories, but only marginally - the detection of the programming language improved by 5.28%, the standard deviations of grades and lines of code improved by 3.2% and 0.82%.

6.2.2 Graph Similarity

The intra-class similarity was combined with graph matching. A maximum common subgraph algorithm was applied. This approach treated class diagrams as graphs and calculated the largest isomorphic subgraph between two given diagrams. Being an NP-

hard problem, the graph similarity algorithm stretched the UMLSimilator software tool to the limit of acceptable computation times as these increased drastically with large class diagrams. Comparing two larger class diagrams could take up to 45 minutes and over 184 hours of execution time was required just for performing similarity measurements across the various experiments.

A minimum class match threshold was used with the graph matching algorithm and experiments were carried out with 80%, 60%, 40% and 20% values. The best results were achieved with a 60% threshold.

The combination of intra-class similarity and the maximum common subgraph improved the results across the board.

	% matching target domain	**% matching target programming language**	**Standard deviation of grade**	**Standard deviation of lines of code**
Structural Similarity	74.16	85.40	11.57	509.87
Graph Similarity[5]	85.51	93.16	10.84	470.94

Table 22 – Comparison of Structural and Graph Similarity Results

Provenance was applied again, but with graph similarity the results were not conclusive. While some of the results improved, others deteriorated and no clear overall improvement was achieved by limiting cases to the same domain.

[5] Minimum Class Similarity Threshold set to 60%

6.2.3 Weight Optimisation

Every feature in the hierarchical class structure has an associated coefficient, which makes it possible to control the degree to which it will contribute to the overall class similarity. The first set of experiments with weighting applied coefficients determined by a human expert. Using these didn't show a clear improvement with graph similarities, but interestingly improved the class structure results, where all but the standard deviation of lines of code were better than using default weighting. This suggests that the expert weight setting performs better when measuring similarities between individual classes, but doesn't cater for the relationships between classes. This could be explained by the fact that the weight setting influences primarily the class similarity metrics and not the formula used to calculate similarity between graphs.

In an attempt to automate the weight optimisation process, an expert's judgement was used to identify desired/undesired class matches. This works at a level of abstraction which is higher than individual coefficients. A genetic algorithm was applied to test and evolve weight settings and adopt the weight profile with the highest score. Unfortunately the scoring of class matches did not work, as it found no higher scores than those achieved with default weights. The granularity of differences achieved by changing weights was enough to change the similarity matches between individual classes, but did not change these values sufficiently to considerably alter the maximum common subgraphs and thereby changing the scores.

Due to the shortcomings of the above scoring approach, an alternative method was used which required an expert to identify the closest match for a given diagram from the same domain. This worked at an even higher level of abstraction, as the expert didn't have to match individual classes. The genetic algorithm ran experiments with different weights and selected the setting giving the highest overall class diagram similarity for

the two selected class diagrams. The weight profiles obtained using this technique, were first applied across the entire case-base, where they yielded poorer results overall than the default weights and no domain weight profile stood out as being particularly good.

The picture changed, however, when each weight profile was applied only to cases from its own domain, the results improved in over 63% of cases.

The final experiments with weighting applied only a subset of features determined by an expert. All features not identified by the expert were disabled and excluded from the similarity metric. Applying a subset of weights to the default setting lowered results in three out of four categories, but when combined with the expert's weight profile, this picture was reversed with results improving in three out of four categories.

Overall, the weight optimisation did not have as much of a positive impact as initially expected. Domain weight profiles improved only within domains, and only marginally. A reduction of the number of weights applied yielded better results when applied to the expert weight setting, but they were only minor improvements.

6.2.4 Clustering

Clustering was applied to further validate the results. Having exhaustively calculated the similarity of every class diagram and all other diagrams in the entire case-base, all cases were assigned into eight clusters using average-link clustering. To begin with, the clustering algorithm was used with class structure similarity, which produced two main clusters. An analysis of the clusters revealed that these appeared to have been split according to quality. One cluster contained primarily class diagrams which had received a very good grade (70.38% average) and whose implementations were on average almost twice as large (3540 vs. 1887 lines of code).

Clusters based on graph similarity were more evenly distributed and class diagrams from the same domain were better grouped. This was improved further within domains when using the weight profiles for each of the domains, creating larger clusters of cases from the corresponding domain or creating more focused clusters.

6.2.5 Plagiarism

An area in which the structural similarity may be considered more important than semantic similarity is plagiarism detection. A person copying someone else's design or code may deliberately change the semantic tags in a class diagram or software implementation in order to avoid detection. He/she may also move elements around to change the order of attributes or operations. While this may avoid detection by a human, the similarity metrics are immune to some of these types of changes.

All cases in the case-based were reverse-engineered from real coursework assignment submissions and it was known that two of these had been found guilty of plagiarism. The two submissions were not identical and had been altered to be slightly different. They also included parts which were dissimilar altogether. This permitted validation of the suitability of this approach to plagiarism detection. The results were very good, with the plagiarised cases being paired as the highest matches in 90% of all 70 sets of experiments, even if the actual similarity measured between them fluctuated between 30.01% and 99.7%, depending on the type of experiment. These findings are very promising, but to fully evaluate the suitability of this approach for plagiarism detection, further experiments should be carried out, as the results obtained in this research are based solely on the two assignments which were known to be plagiarised.

6.2.6 Cost Estimation

In software development, the most common approaches for cost estimation are formal estimation models, such as COCOMO. The approach to cost estimation using case-based reasoning and nearest neighbour retrieval based on structural and graph similarity applied in this work was compared to estimated implementation effort obtained through COCOMO. The results were validated against the actual implementation cost.

The initial set of experiments used the class structure similarity, which showed that while the total difference between the actual lines of code and the estimated value was much lower, COCOMO performed slightly better overall – in 60% of cases.

Using graph similarity the results were reversed, with the UMLSimilator system performing better in 60% of cases. The total difference between the actual lines of code and the estimated value was reduced further.

Method	Difference between Actual and Estimated Lines of Code
COCOMO	9238
Structural Similarity	5796
Graph Similarity[6]	4556
Graph Similarity[7] **using Provenance**	3766

Table 23 – Comparison of the Overall Difference between Actual and Estimated Implementation Cost

In an industrial setting, the weight assignment and establishment of complexity factors by a human expert is adjusted to the environment in which the development project will be carried out, along with other external factors. These factors could include the

[6] Minimum Class Similarity Threshold set to 60%
[7] Minimum Class Similarity Threshold set to 60%

experience of the development team, the technical resources available and whether similar projects have been carried out in the past. An estimate which is correct for one setting or company may not be appropriate for another. Similarly, obtaining cases only from the same domain as the target case would ensure that the results would originate from a similar environment. The final set of experiments therefore compared COCOMO to estimates obtained using graph similarity matching and provenance, which improved results even further, with 80% being more accurate than COCOMO.

As is evident from Table 23, the approach used in this research for estimating cost performed better than COCOMO with these cases.

6.2.7 Explanation

There is no established standard or benchmark for measuring similarity between two class diagrams. This work proposed a set of similarity metrics to determine the similarity, but these may not be clear to a human expert who is presented with similarity matches expressed in percentages. To ensure that a user trusts the results produced by the system requires that it explains the results by clarifying how they were calculated. The UMLSimilator tool provides explanation by offering a complete breakdown of all similarity matches within the hierarchical structure, as well as visually displaying the maximum common subgraph between class diagrams.

A set of experiments were devised which asked experts to estimate the similarity between given diagrams and relate their estimates to those produced by the system. They were asked to confirm their trust in the results before and after seeing the accompanying explanations and the outcome was that trust increased in all instances – especially in those where the result provided by the system differed substantially from their own estimates.

6.3 Contribution to Knowledge

There is an intrinsic relationship between software designs and their code implementation. However, the modular nature of object-oriented programming languages, combined with the variations of language constructs available in modern programming languages signifies that there are numerous ways of designing a specified application and even more ways of implementing a particular design. It is not possible to deduct a software product's exact functionality from its design. Yet, a software design is an abstract representation of a software product and, in the case of a class diagram, it contains information about its structure. The main aim of this research was to determine whether knowledge can be effectively retrieved from software designs using only structural information. This research has demonstrated that it can be achieved by applying case-based reasoning and graph matching.

The main contributions of this work are:

- A CBR framework for the retrieval of knowledge from class diagrams using only structural information. The approach of measuring similarity of hierarchical structures and graph matching could be abstracted and applied in other domains
- A mechanism for identifying software solutions which came from the same domain, of the same programming language or of similar quality or size by applying a graph matching algorithm
- A technique for estimating implementation cost of a new software product based on its design and an existing catalogue of software designs
- An approach for increasing users' confidence in automatic class diagram matching by providing explanation
- A software tool for automating the retrieval of knowledge using the framework and techniques described above

There are several areas of research which are relevant to this work and by placing this research into context of related work, it has been demonstrated that the research question is of value to the research community.

The findings related to cost estimation and plagiarism detection have shown that the technique applied here can contribute to industry and academia alike in obtaining solutions from class diagrams where semantic information is lacking.

6.4 Further Work

The work presented in this thesis combined a number of techniques and methodologies to improve the knowledge retrieval from class diagrams using structural information. During this work a number of issues were encountered, some of which were addressed as they were within the scope of this research, others however, were left for future work. There were also weaknesses which were tackled by this work, but which would benefit from further dedicated research. Some of the findings resulting from this work have also raised interesting new ideas of how this work could be taken further.

There are numerous directions in which this research could evolve, some of which are presented in the following subsections.

6.4.1 Structural and Semantic Similarity

The use of semantic information to determine the similarity between two class diagrams has been successfully applied by Gomes et al. [Gomes, et al., 2007] and Robles et al. [Robles, et al., 2012]. The work by Gomes et al. did involve some structural information to determine class complexity, which counted the number of methods and attributes. However, it does not go into the same depth of structural information or use of graphs.

The importance of semantic information has been highlighted not only in other research, but it was also reinforced by the findings of this work. In order to improve the results achieved here, one should use a hybrid approach which combines the semantic and structural approach. Through the use of coefficients, the contribution of semantics and structure could be balanced.

6.4.2 Stereotyping

Within class diagrams certain types of classes occur numerous times and could be defined using stereotypes, thus creating categories of types which could further define particular classes. Examples of these include *user interface*s, *web services*, *events/delegates*, *exceptions*, *roles within a design pattern*, *communication classes*, *database handlers*, etc.

Assigning classes particular stereotypes or trying to automatically determine stereotypes would aid in the comparison process, as only classes of the same stereotype would be matched. The overall similarity between class diagrams could also take into consideration the total number of matched stereotypes. The stereotypes are not dependent on the semantic tags or the structure, but define the kind of class.

6.4.3 Design Pattern Library

Design patterns are often encoded using general UML class diagrams. Although these also use semantics to tag the different elements of the design pattern, the more important aspect is generally the structure. By creating a library of design pattern structures, it would be possible to identify design patterns in existing software designs and tag graphs with patterns.

A design pattern is a solution to a commonly occurring problem, so by identifying a design pattern, one would know more about the application – such as, what type of application it is or something specific the application does.

It would also be interesting to investigate the classification and detection of design patterns within a particular domain.

An interesting twist of the idea of a design pattern library would be to research the possibility of automated identification of new design patterns, which could be fed back into the case-base.

6.4.4 Behavioural UML Diagrams

Within object-oriented development, software design is most commonly used to represent static structures. However, a design may not consist solely of a static model, but combine a number of different aspects of a software application or system using a range of appropriate diagrams, such as use case, state or interaction diagrams.

Observing the interaction of elements within a software design may also be important, due to difference in object granularity. Different software designers have different styles and practices, which could result in applications or systems designed for the same purpose having a considerably different number of objects, due to the designer's personal perception of object granularity. However, it is possible that the messages sent between objects show similarities of designs, which are not structurally similar.

A key issue is that a set of UML diagrams depicting a piece of software are related and complement each other. For instance, an interaction diagram should represent a particular use-case, but using elements and relationships from the class diagram. This makes is possible to treat a set of related diagrams as a unit representing a piece of software which can be compared to others.

Further research could be extended to state transition, collaboration/sequence and use-case diagrams individually, as well as combinations of diagrams.

6.4.5 Case-Base Maintenance

From the various phases of the case-based reasoning process, the emphasis of this research has been on the ranking. Little consideration has been given to adaptation and retention of new cases. However, it is natural for a case-base to evolve, so case-base maintenance would need to be considered.

Related to case-base maintenance is the idea of assisted automatic re-engineering. When implementing a new software product, a rough structure of a class diagram would be designed. This would be fed into the CBR system as a new problem; the most similar designs would be retrieved and one of these could be adapted and re-engineered to fit the requirements of the current problem. This process would require assistance from a human expert.

6.4.6 Performance Improvements

The approach for measuring class diagram similarity presented in this thesis employs methods which are computationally demanding. Exhaustive matching algorithms are used, as well as a recursive algorithm for calculating the maximum common subgraph. While these techniques worked with the current case-base, it was reaching the limit of where it could still be computed within an acceptable amount of time. If one were to increase the case-base significantly or include more complex cases, the current algorithms would need to be optimised to increase performance.

The major hurdle to performance is computing the maximum common subgraph, so an optimisation of this graph matching algorithm would be the best starting point. One could determine the applicability of alternative graph matching algorithms, such as

those discussed in the literature review. Another approach would be find the maximum clique as has been applied to molecule matching in the Durand-Pasari algorithm [Durand, et al., 1999] or other approaches to graph matching optimisation (for example the algorithm outlined by [Wang and Maple, 2005]).

If the case-base were to increase significantly, an appropriate indexing mechanism would also have to be introduced.

Finally, it would be valuable to field test the method for cost estimation presented in this work in a commercial setting.

7 References

[Aamodt and Plaza, 1994] Aamodt, A. Plaza, E. (1994) Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches, AI Communications, Volume 7, Issue 1, pp. 39-59, IOS Press

[Abran and Moore, 2004] Abran, A. Moore, J. (2004) Guide to the Software Engineering Body of Knowledge, IEEE Computer Society

[Abreu and Melo, 1996] Abreu, F. Melo, W. (1996) Evaluating the Impact of Object-Oriented Design on Software Quality, Proceedings of the 3rd International Symposium on Software Metrics: From Measurement to Empirical Results. METRICS. IEEE Computer Society, Washington, DC, 90

[Alexander, et al., 1977] Alexander, C. Ishikawa, S. Silverstein, M. Jacobson, M. Fiksdahl-King, I. Angel, S. (1977) A Pattern Language, Oxford University Press, New York

[Alnusair and Zhao, 2009] Alnusair, A. Zhao, T. (2009) Towards a model-driven approach for reverse engineering design patterns, in Proc. International Workshop on Transforming and Weaving Ontologies in MDE (TWOMDE'09) at MoDELS'09, Oct. 2009

[Alnusair and Zhao, 2010] Alnusair, A. Zhao, T. (2010) Using Semantic Inference for Software Understanding and Design Recovery. Seventh International Conference on Information Technology: New Generations (ITNG), pp.980-985

[Andreou and Papatheocharous, 2008] Andreou, A. S. Papatheocharous, E. (2008) Software Cost Estimation using Fuzzy Decision Trees. In Proceedings of the 2008 23rd IEEE/ACM international Conference on Automated Software Engineering, Automated Software Engineering. IEEE Computer Society, pp. 371-374, Washington, DC

[Aron, 1970] Aron, J. D. (1970) Estimating Resources for Large Programming Systems. In Software Engineering Techniques, NATO Conference Report, Rome, October 1969, pp. 68-84

[Auwatanamongkol, 2005] Auwatanamongkol, S. (2005) Inexact pattern matching using genetic algorithm. In Proceedings of the 2005 Conference on Genetic and Evolutionary Computation, pp. 1567-1568, GECCO '05. ACM, New York, NY

[Bai, et al., 2011] Bai, L. Liang, K. Dang, C. Cao, F. (2011) A novel attribute weighting algorithm for clustering high-dimensional categorical data, Pattern Recognition, Volume 44, Issue 12, December 2011, pp. 2843-2861

[Bareiss, 1988] Bareiss, R. (1988) PROTOS: A Unified Approach to Concept Representation, Classification and Learning. Technical report CS 88-83, University of Texas at Austin, Dep. of Computer Science, Nashville, TN

[Barletta and Mark, 1988] Barletta, R. Mark, W. (1988) Explanation-Based Indexing of Cases. In Proceedings of the Seventh National Conference on Artificial Intelligence. Palo Alto, CA

[Baxter and Mehlich, 1997] Baxter, I. D. Mehlich, M. (1997) Reverse engineering is reverse forward engineering, Proceedings of the Fourth Working Conference on Reverse Engineering, pp.104-113, 6-8 Oct

[Beck, et al., 2001] Beck, K. et al. (2001) Manifesto for Agile Software Development, Agile Alliance

[Beddoe and Petrovic, 2006] Beddoe, G. Petrovic, S. (2006) Determining feature weights using a genetic algorithm in a case-based reasoning approach to personnel rostering. European Journal of Operational Research, Vol. 175, Issue 2, pp. 649-671.

[Bergmann, 2002] Bergmann, R. (2002) Experience Management: Foundations, Development Methodology, and Internet-Based Applications. Berlin: Springer

[Bergmann and Stahl, 1998] Bergmann R., Stahl, A. (1998) Similarity Measures for Object-Oriented Case Representations, in 4th European Conference on Case-Based Reasoning, Springer, 1998

[Bergmann, et al., 2005] Bergmann, R. Kolodner, J. Plaza, E. (2005) Representation in case-based reasoning. Knowledge Engineering Review Vol. 20, Issue 3, 209-213, Cambridge University Press, UK

[Bjornestad, 2003] Bjornestad, S. (2003) Analogical Reasoning for Reuse of Object-Oriented Specifications, in Proceedings of the 5th International Conference on Case-Based Reasoning (ICCBR'03)

[Boehm, 1981]Boehm, B. (1981) Software Engineering Economics, Englewood Cliffs, N.J.

[Boehm, et al., 2000a] Boehm, B. Abts, C. Chulani, S. (2000) Software Development Cost Estimation Approaches – A Survey. Technical Report 2000-505, Uni. of California and IBM Research, Los Angeles, USA

[Boehm, et al., 2000b] Boehm, B. Abts, C. Brown, A. W. Chulani, S. Clark, B. K. Horowitz, E. Madachy, R. Reifer, D. J. Steece, B. (2000) Software Cost Estimation with COCOMO II, Englewood Cliffs, NJ:Prentice-Hall

[Booch, 2007] Booch, G. (2007) Object-Oriented Analysis and Design with Applications, Addison-Wesley Professional

[Briand, Wieczorek, 2002] Briand, L. Wieczorek, I (2002) Resource Estimation in Software Engineering, Encyclopedia of Software Engineering, J. J. Marcinak. New York, John Wiley & Sons: 1160-1196

[Champin and Solnon, 2003] Champin, P. Solnon, C. (2003) Measuring the Similarity of Labeled Graphs, Lecture Notes in Computer Science, Volume 2689, pp. 1066-1067, Springer Berlin / Heidelberg, Germany

[Chartrand, et al., 1989] Chartrand, G. Oellermann, O. Saba, F. Zou, H. (1989) Greatest common subgraphs with specified properties, Graphs and Combinations, Volume 5, Number 1, pp. 1-14, Springer Japan

[Chikofsky and Cross, 1990] Chikofsky, E. J. Cross, J. H. (1990) Reverse Engineering and Design Recovery: A Taxonomy, IEEE Software, vol. 7, no. 1

[Cooper, et al., 2004] Cooper, D. Khoo, B. von Konsky, B. R. Robey, M (2004) Java implementation verification using reverse engineering. In Proceedings of the 27th Australasian conference on Computer science - Volume 26 (ACSC '04), Estivill-Castro (Ed.), Vol. 26, 203-211, Australian Computer Society, Inc., Darlinghurst, Australia, Australia

[Cormen, et al., 2009] Cormen, T. H. Leiserson, C. E. Rivest, R. L. Stein, C. (2009) Introduction to Algorithms, MIT Press, USA

[Crean and O'Donoghue, 2002] Crean, B. O'Donoghue, D. (2002) RADAR: Finding Analogies using Attributes of Structure. In Proceedings of the 13th Irish Conference on Artificial Intelligence and Cognitive Science (AICS'02), pp. 20–27, Limerick, Ireland, Springer

[Cunningham, et al., 2003] Cunningham, P. Doyle, D. Loughrey, J. (2003) An Evaluation of the Usefulness of Case-Based Reasoning Explanation, In Proceedings of the 5th International Conference on Case-Based Reasoning, ICCBR 2003, pp. 122–130, Springer, Berlin Heidelberg, Germany

[De Jong, 1975] De Jong, K. A. (1975) An Analysis of the Behavior of a Class of Genetic Adaptive Systems, Ph.D. thesis, University of Michigan, Ann Arbor, USA

[Demeyer, 2005] Demeyer, S. (2005) Refactor Conditionals into Polymorphism: What's the Performance Cost of Introducing Virtual Calls? IEEE International Conference on Software Maintenance, pp. 627-630, 21st IEEE International Conference on Software Maintenance (ICSM'05)

[Desharnais, 1989] Desharnais, J. M. (1989) Analyse statistique de la productivitie des projets informatique a partie de la technique des point des fonction. University of Montreal

[Dudani, 1976] Dudani, S. A. (1976) The Distance-Weighted k-Nearest-Neighbor Rule. IEEE Transactions on Systems, Man and Cybernetics, Vol.SMC-6, no.4, pp.325-327

[Durand, et al., 1999] Durand, P. J. Pasari, R. Baker, J. W. Tsai, C. [1999] An Efficient Algorithm for Similarity Analysis of Molecules, Internet Journal of Chemistry, Vol. 2

[Egyed, 2002] Egyed, A. (2002). Automated abstraction of class diagrams. ACM Transactions on Software Engineering Methodology, Vol. 11, Issue 4, pp. 449-491, ACM, New York, NY

[Eichelberger, 2003] Eichelberger, H. (2003) Nice class diagrams admit good design? In Proceedings of the 2003 ACM Symposium on Software Visualization. SoftVis '03. ACM, New York, NY

[Eva, 1994] Eva, M. (1994) SSADM version 4: A user's guide, McGraw-Hill Education

[Fahmy, et al., 1997] Fahmy, H. Holt, R. Mancoridis, S. (1997) Repairing software style using graph grammars. In Proceedings of the 1997 Conference of the Centre For Advanced Studies on Collaborative Research, IBM Press, Canada

[Fernández-Chamizo, et al., 1996]Fernández-Chamizo, C. González-Calero, P. Gómez-Albarrán, M. Hernández-Yánes, L. (1996) Supporting Object Reuse Through Case-Based Reasoning, in Advances in Case-Based Reasoning, pp. 135-149. Springer-Verlag, Germany

[Finnie, 1997] Finnie, G. R. Wittig, G. E. Desharnais, J-M. (1997) A comparison of software effort estimation techniques: Using function points with neural networks, case-based reasoning and regression models, Journal of Systems and Software, Volume 39, Issue 3, pp. 281-289, Elsevier Science, New York, NY

[Gamma, et al., 1995] Gamma, E. Helm, R. Johnson, R. and Vlissides, J. (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading

[Garey and Johnson, 1987] Garey, M. R. Johnson, D. S. (1987) Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman

[Gelhausen, 2008] Gelhausen, T. Derre, B. Geiß, R. (2008) Customizing grgen.net for model transformation. In Proceedings of the Third international Workshop on Graph and Model Transformations, GRaMoT '08, ACM, New York, NY

[Gentner, 1983] Gentner, D (1983) Structure mapping - a theoretical framework for analogy. Cognitive Science, Vol.7. pp. 155-170

[Gomes, et al., 2001] Gomes, P. Pereira, F. C. Ferreira, J. L. Bento, C. (2001) Using Analogical Reasoning to Promote Creativity in Software Reuse. ICCBR'01 Workshop on Creative Systems, Vancouver, Canada

[Gomes, et al., 2002] Gomes, P. Pereira, F. C. Paiva, P. Seco, N. Carreiro, P. Ferreira, J. L. Bento, C. (2002) Experiments on Software Design Novelty Using Analogy. European Conference on Artificial Intelligence - ECAI'02, Workshop: 2nd Workshop on Creative Systems

[Gomes, et al., 2003] Gomes, P. Pereira, F. C. Paiva, P. Seco, N. Carreiro, P. Ferreira, J. L. Bento, C. (2003) A Selection and Reuse of Software Design Patterns Using CBR and WordNet. In Proceedings of the Fifteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'03)

[Gomes, et al., 2004] Gomes, P. Pereira, F. C. Paiva, P. Seco, N. Carreiro, P. Ferreira, J. L. Bento, C. (2004) Using WordNet for case-based retrieval of UML models, AI Communications, Vol. 1

[Gomes, et al., 2007] Gomes, P. Gandola, P. Cordeiro, J. (2007) Helping Software Engineers Reusing UML Class Diagrams, in Proceedings of the 7th International Conference on Base-Based Reasoning (ICCBR'07) pp. 449-462, Springer, 2007

[Gorton and Zhu, 2005] Gorton, I. Zhu, L. (2005) Tool support for just-in-time architecture reconstruction and evaluation: an experience report, in Proceedings 27th International Conference on Software Engineering, pp. 514 - 523

[Grabert and Bridge, 2003] Grabert, M. Bridge, D.G. (2003) Case-Based Reuse of Software Examplets, Journal of Universal Computer Science, Vol. 9, No. 7, pp. 627-641

[Gutwenger, 2003] Gutwenger, C. Jünger, M. Klein, K. Kupke, J. Leipert, S. Mutzel, P. (2003) A new approach for visualizing UML class diagrams. In Proceedings of the 2003 ACM Symposium on Software Visualization, pp. 179-188, SoftVis '03, ACM, New York, NY

[Holland, 1975] Holland, J. (1975) Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor

[Hong and Liou, 2008] Hong, T. Liou, Y. (2008) Case-Based Reasoning with feature clustering, 7th IEEE International Conference on Cognitive Informatics, ICCI 2008, pp.449-454

[Huang, et al., 2007] Huang, X. Ho, D. Ren, J. Capretz, L. F. (2007) Improving the COCOMO model using a neuro-fuzzy approach, Applied Soft Computing, Vol. 7, pp. 29-40

[Huang, 2009] Huang, Z. (2009) Cost Estimation of Software Project Development by Using Case-Based Reasoning Technology with Clustering Index Mechanism. In Proceedings of the 2009 Fourth international Conference on innovative Computing, information and Control, ICICIC. IEEE Computer Society, pp. 1049-1052, Washington, DC

[Huff, 1992] Huff, C. C. (1992) Elements of a realistic CASE tool adoption budget. Comm. ACM, 35(4), 45-54 (Ch. 4), New York, NY

[Jacobson, et al., 1998] Jacobson, I. Booch, G. Rimbaugh, J. (1998) The Unified Software Development Process, Addison Wesley Longman

[Jadalla and Elnagar, 2008] Jadalla, A. Elnagar, A. (2008) PDE4Java: Plagiarism Detection Engine for Java source code: a clustering approach, International Journal of Business Intelligence and Data Mining, Volume 3, Issue 2, pp. 121-135, Inderscience Publishers, Switzerland

[Jones, 2007] Jones, C. (2007) Estimating Software Costs: Bringing Realism to Estimating, McGraw Hill

[Kelly and Davis, 1991] Kelly, J. D. Davis, L. (1991) A hybrid genetic algorithm for classification. In Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, pp. 645–650, Sydney, Australia: Morgan Kaufmann.

[Knight, et al., 2001] Knight, B. Petridis, M. Mejasson, P. Norman, P. A. (2001) Intelligent design assistant (IDA): a Case Based Reasoning System for Materials Design, in Journal of Materials and Design 22 pp. 163-170

[Kolodner, 1993] Kolodner, J. (1993) Case-based Reasoning, Morgan Kaufmann

[Krcmar and Dhawan, 1994] Krcmar, M. Dhawan, A.P. (1994) Application of Genetic Algorithms in Graph Matching, In Proceedings of the International Conference on Neural Networks, Volume 6, pp. 3872-3876

[Krigsman, 2011] Krigsman, M - http://www.zdnet.com/blog/projectfailures/2011-erp-survey-new-it-failure-research-and-statistics/12486 (last accessed April 2012)

[Kumar, et al., 2008] Kumar, K. Ravi, V. Carr, M., Raj Kiran, N. (2008) Software Development Cost Estimation Using Wavelet Neural Networks. In Journal of Systems and Software, Volume 81 - Issue 11, pp. 1853-1867, Elsevier Science Inc, New York, NY

[Larman, 2005] Larman, C. (2005) Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development. Pearson Education, NJ, USA

[Leake, 1996] Leake, D. (1996) CBR in context: the present and future, In Case-Based Reasoning: Experiences, Lessons & Future Directions. Cambridge, MA: MIT Press, pp. 3-30.

[Leake and Whitehead, 2007] Leake, D. Whitehead, M. (2007) Case Provenance: The Value of Remembering Case Sources, 7th International Conference on Case-Based Reasoning: CBR Research and Development (Belfast, UK, August 13 - 16, 2007), LNAI, vol. 4626, pp. 194-208, Springer

[Lee, et al., 2007] Lee, H. Youn, H. Lee, E. (2007) Automatic Detection of Design Pattern for Reverse Engineering, 5th ACIS International Conference on Software Engineering Research, Management & Applications, 2007. SERA 2007, pp. 577-583

[Le Métayer, 1996] Le Métayer, D. (1996) Software architecture styles as graph grammars, ACM SIGSOFT Software Engineering Notes, Volume 21, Issue 6, pp. 15-23, ACM, New York, NY

[Lewis, et al., 1991] Lewis, J. Henry, S. Kafura, D. Schulman, R. (1991) An Empirical Study of the Object-Oriented Paradigm and Software Reuse, Proc. Conference on Object Oriented Programming Systems Languages and Applications 1991, pp. 184 – 196, ACM, New York

[Li, et al., 2009] Li, Y. F. Xie, M. Goh, T. N. (2009) A study of mutual information based feature selection for case based reasoning in software cost estimation. Expert Systems with Applications: An International Journal, Volume 36, Issue 3, pp. 5921-5931, Pergamon Press, Tarrytown, NY

[Lukashenko, et al., 2007] Lukashenko, R. Graudina, V. Grundspenkis, J. (2007) Computer-based plagiarism detection methods and tools: an overview. In Proceedings of the 2007 international Conference on Computer Systems and Technologies, CompSysTech '07, Volume 285. ACM, New York, NY

[Mardia, et al., 1980] Mardia, K. Kent, J. T. Bibby, J. M. (1980) Multivariate Analysis (Probability and Mathematical Statistics), Academic Press

[Meditskos and Bassiliades, 2007] Meditskos, G. Bassiliades, N. (2007) Object-Oriented Similarity Measures for Semantic Web Service Matchmaking, in Proceedings 5th IEEE European Conference on Web Services

[Mendes, et al. 2002] Mendes, E. Mosley, N. Counsell, S. (2002) The Application of Case-Based Reasoning to Early Web Project Cost Estimation. In Proceedings of the 26th international Computer Software and Applications Conference on Prolonging

Software Life: Development and Redevelopment COMPSAC. IEEE Computer Society, pp. 393-398, Washington, DC

[Mendes, et al., 2003] Mendes, E. Mosley, N. Counsell, S. (2003) Early Web Size Measures and Effort Prediction for Web Costimation. In Proceedings of the 9th international Symposium on Software Metrics, METRICS. IEEE Computer Society, Washington, DC

[Merdes and Dorsch, 2006] Merdes, M. Dorsch, D. (2006) Experiences with the development of a reverse engineering tool for UML sequence diagrams: a case study in modern Java development. In Proceedings of the 4th international symposium on Principles and practice of programming in Java (PPPJ '06). 125-134, ACM, New York, NY, USA

[Mileman, et al., 2002] Mileman, T. Knight, B. Petridis, M. Cowell, D. Ewer, J. (2002) Case-based retrieval of 3-dimensional shapes for the design of metal castings, Journal of Intelligent Manufacturing, Volume 13, Number 1, pp. 39-45, Springer

[Mileman, et al., 2000] Mileman, T. Knight, B. Petridis, M. Preddy, K. Mejasson, P. (2000) Maintenance of a Case-Base for the Retrieval of Rotationally Symmetric Shapes for the Design of Metal Castings, in Proceedings of Advances in Case-Based Reasoning 5th European Workshop, EWCBR 2000, pp. 351-401, Springer

[Mitchell, 1990] Mitchell, T. M. (1990) The need for biases in learning generalizations, In Readings in machine learning, San Mateo, CA, Morgan Kaufmann

[Mozgovoy, 2006] Mozgovoy, M. (2006) Desktop tools for offline plagiarism detection in computer programs. Informatics in education, Volume 5, Issue 1, pp. 97-112

[Murdock, et al., 2006] Murdock, J. W. McGuinness, D. L. da Silva, P. P. Welty, C. Ferrucci, D. (2006) Explaining conclusions from diverse knowledge sources. In Proceedings of the 5th international conference on The Semantic Web (ISWC'06), pp. 861-872, Springer-Verlag, Berlin, Heidelberg

[Naur and Randell, 1969] Naur, P. Randell, B. (1969) Software engineering: Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, Scientific Affairs Division, NATO

[OMG, 2006a] OMG (2006) Meta Object Facility (MOF) Core Specification, Version 2.0, Object Management Group

[OMG, 2006b] OMG (2006) Diagram Interchange, Version 1.0, Object Management Group

[OMG, 2007] OMG (2007) MOF 2.0/XMI Mapping, Version 2.1.1, Object Management Group

[OMG, 2009] OMG (2009) UML Superstructure Specification Version 2.2, Object Management Group

[Özşen and Güneş, 2009] Özşen, S. Güneş, S. (2009) Attribute weighting via genetic algorithms for attribute weighted artificial immune system (AWAIS) and its application to heart disease and liver disorders problems, Expert Systems with Applications, Vol. 36, Issue 1, pp. 386-392

[Patterson, et al., 2008] Patterson, D. Rooney, N. Galushka, M. Dobrynin, V. Smirnova, E. (2008) SOPHIA-TCBR: A knowledge discovery framework for textual case-based reasoning, Knowledge-Based Systems. 21, 5 (July 2008), 404-414.

[Petridis, et al., 2007a] Petridis, M. Saeed, S. Knight, B. (2007) A Generalised Approach for Similarity Metrics Between 3D Shapes to Assist the Design of Metal Castings using an Automated Case Based Reasoning System, in Proceedings of the 12th UK CBR workshop, Peterhouse, December 2007, CMS press, pp.19-29, UK

[Petridis, et al., 2007b] Petridis, M. Saeed, S. Knight, B. (2007) Refining Similarity Measures for Effective Reuse of Metal Casting Design Knowledge, Workshop Proceedings of 7th International Conference on Case-Based Reasoning, ICCBR-07, pp. 16-32, Springer

[Purchase, et al., 2001] Purchase, H. C. McGill, M. Colpoys, L. Carrington, D. (2001) Graph drawing aesthetics and the comprehension of UML class diagrams: an empirical study. In Proceedings of the 2001 Asia-Pacific Symposium on information Visualisation - Volume 9, ACM International Conference Proceeding Series, vol. 16, pp. 129-137 Australian Computer Society, Darlinghurst, Australia

[Quinlan, 1986] Quinlan, J. R. (1986) Induction of Decision Trees, Machine Learning, Vol. 1, Issue 1, pp.81-106, Kluwer Academic Publishers Hingham, MA, USA

[Ramon, et al., 2010] Ramon, O. S. Cuadrado, J. S. Molina, J. G. (2010). Model-driven reverse engineering of legacy graphical user interfaces. In Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE '10). 147-150, ACM, New York, NY, USA,

[Raveaux, et al., 2010] Raveaux, R. Burie, J. Ogier, J. (2010) A graph matching method and a graph matching distance based on subgraph assignments. Journal of Pattern Recognition Letters, Vol. 31, Issue 5, pp. 394-406, Elsevier Science Inc., New York, NY, USA

[Recio-Garcia and Wiratunga, 2010] Recio-Garcia, J. Wiratunga, N. (2010) Taxonomic Semantic Indexing for Textual Case-Based Reasoning, Lecture Notes in Computer Science, Volume 6176, pp. 302-316, Springer Berlin / Heidelberg, Germany

[Riesbeck and Schank, 1989] Riesbeck, C.K. Schank, R.S. (1989) Inside Case-Based Reasoning. Erlbaum, Northvale, NJ

[Robles, et al., 2012] Robles, K. Fraga, A. Morato, J. Llorens, J. (2012) Towards an ontology-based retrieval of UML Class Diagrams, Information and Software Technology, Vol. 54, Issue 1, January 2012, pp. 72-86, Elsevier

[Sanders, et al., 1997] Sanders, K. Kettler, B. Hendler, J. (1997) The case for graph-structured representations, Lecture Notes in Computer Science, Volume 1266, pp. 245-254, Springer Berlin / Heidelberg, Germany

[Schaffer, et al., 1989] Schaffer, J. D. Caruana, R. A. Eshelman, L. J. Das, R. (1989) A study of control parameters affecting online performance of genetic algorithms for function optimization, In Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kauffmann

[Schank, 1983] Schank, R. C. (1983) Dynamic Memory: A Theory of Reminding and Learning in Computers and People. Cambridge University Press, New York, NY, USA

[Schank, 1986] Schank, R. C. (1986) Explanation Patterns – Understanding Mechanically and Creatively, Lawrence Erlbaum, New York.

[Shepperd, et al., 1996] Shepperd, M. Schofield, C. Kitchenham, B. (1996) Effort estimation using analogy. In Proceedings of the 18th international Conference on Software Engineering, International Conference on Software Engineering. IEEE Computer Society, pp. 170-178, Washington, DC

[Shi and Olsson, 2006] Shi, N. Olsson R. A. (2006) Reverse engineering of design patterns from java source code, in Proc. IEEE/ACM International Conference on Automated Software Engineering, Sep. 2006, pp. 123–132

[Sileshi and Gamback, 2009] Sileshi, M. Gamback, B. (2009) Evaluating Clustering Algorithms: Cluster Quality and Feature Selection in Content-Based Image Clustering, In WRI World Congress on Computer Science and Information Engineering, 2009. Vol. 6, 2009, pp. 435–441

[Sokal and Michener, 1958] Sokal, R. Michener, C. (1958) A statistical method for evaluating systematic relationships, University of Kansas Science Bulletin 38, pp. 1409–1438

[Sommerville, 2004] Sommerville, I. (2004) Software Engineering, Addison Wesley

[Sormo, et al., 2005] Sormo, F. Cassens, J. Aamodt, A. (2005) Explanation in Case-Based Reasoning-Perspectives and Goals, Artificial Intelligence Review, Vol. 24, Issue 2 (October 2005), pp. 109-143, Kluwer Academic Publishers Norwell, MA, USA

[Stéphane, et al., 2010] Stéphane, N. Hector, R. Marc, L. L. J. (2010) Effective retrieval and new indexing method for case based reasoning: Application in chemical process design, Engineering Applications of Artificial Intelligence, Volume 23, Issue 6, September 2010, pp. 880-894, Elsevier

[Su and Lipasti, 2006] Su, L. Lipasti, M. (2006) Dynamic Class Hierarchy Mutation, pp. 98-110, International Symposium on Code Generation and Optimization (CGO'06)

[Tadayon, 2005] Tadayon, N. (2005) Neural Network Approach for Software Cost Estimation. In Proceedings of the International Conference on Information Technology: Coding and Computing (Itcc'05) - Volume 02, ITCC. IEEE Computer Society, pp. 815-818, Washington, DC

[Tautz and Althoff, 1997] Tautz, C. Althoff, K-D. (1997) Using case-based reasoning for reusing software knowledge, Lecture Notes in Computer Science, Volume 1266, pp. 156-165, Springer Berlin / Heidelberg, Germany

[Tessem, et al., 1998] Tessem, B. Whitehurst, A. Powell, C. (1998) Retrieval of Java Classes for Case-Based Reuse, in Procs. of the Fourth European Workshop on Case-Based Reasoning, LNAI 1488, pp.148-159, Springer

[Tilevich and Smaragdakis, 2005] Tilevich, E. Smaragdakis, Y. (2005) Binary Refactoring: Improving Code Behind the Scenes, Proc. 27th International Conference on Software Engineering, pp. 264 – 273, ACM, New York

[Tilley and Huang, 2001] Tilley, S. Huang, S. (2001) Evaluating the reverse engineering capabilities of Web tools for understanding site content and structure: a case study. In Proceedings of the 23rd International Conference on Software Engineering (ICSE '01). 514-523IEEE Computer Society, Washington, DC, USA

[TIOBE, 2012] TIOBE - http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (last accessed April 2012)

[Tong and Sriram, 1992] Tong, C. Sriram, D. (1992) Artificial Intelligence in Engineering Design, Volume 1, Academic Press

[Tronto, et al., 2008] Tronto, I. F. da Silva, J. D. Sant'Anna, N. (2008) An investigation of artificial neural networks based prediction systems in software project management, Journal of Systems and Software, Volume 81 – Issue 3, pp. 356-367, Elsevier Science, New York, NY

[Tsatsoulis and Amthauer, 2003] Tsatsoulis, C. Amthauer, H. A. (2003) Finding Clusters of Similar Events within Clinical Incident Reports: A Novel Methodology Combining Case Based Reasoning and Information Retrieval, Quality & Safety in Health Care 12 (Suppl II), pp. 24–32.

[Tsoy, 2003] Tsoy, Y. R. (2003) The influence of population size and search time limit on genetic algorithm, In Proceedings of The 7th Korea-Russia International Symposium on Science and Technology, Vol.3, pp. 181-187

[Ullmann, 1976] Ullmann, J. R. (1976) An Algorithm for Subgraph Isomorphism. Journal of the ACM, Vol. 23, Issue 1 (January 1976), pp. 31-42, ACM, New York, NY, USA

[Ulrich, 1990] Ulrich, W. M. (1990) The evolutionary growth of software reengineering and the decade ahead, American Programmer, Vol. 3, Issue 11, 14-20

[Valerdi, 2007] Valerdi, R. (2007) Cognitive Limits of Software Cost Estimation. In Proceedings of the First international Symposium on Empirical Software Engineering and Measurement, Empirical Software Engineering and Measurement. IEEE Computer Society, pp. 117-125, Washington, DC

[Victor, et al., 1996] Victor, R. Lionel, C. Walcélio, L. (1996) A Validation of Object-Oriented Design Metrics as Quality Indicators, IEEE Transactions On Software Engineering, Vol. 22, 10, pp. 751-761

[Vinita, et al., 2008] Vinita Jain, A. Tayal, D. K. (2008) On reverse engineering an object-oriented code into UML class diagrams incorporating extensible mechanisms. SIGSOFT Software Engineering Notes, Vol. 33, Issue 5, Article 9, ACM, New York, NY, USA

[Wang and Maple, 2005] Wang, Y. Maple, C. (2005) A novel efficient algorithm for determining maximum common subgraphs, In Proceedings of the Ninth International Conference on Information Visualisation, pp. 657- 663

[Weber, et al., 2011] Weber, M. Langenhan, C. Roth-Berghofer, T. Liwicki, M. Dengel, A. Petzold, F. (2011) Fast Subgraph Isomorphism Detection for Graph-Based Retrieval. Lecture Notes in Computer Science, Vol. 6880, pp. 319-333, Springer Berlin / Heidelberg

[Wettschereck, et al., 1997] Wettschereck, D. Aha, D. W. Mohri, T. (1997) A Review and Empirical Evaluation of Feature Weighting Methods for a Class of Lazy Learning Algorithms. Artificial Intelligence Review, Vol. 11, Issue 1, pp. 273-314, Springer, Netherlands

[Woon, et al., 2001] Woon, F. Knight, B. Petridis, M. (2001) A Case Based System to assist in the design process in the manufacture of furniture products, in Proceedings 6th UK Workshop on Case Based Reasoning, Cambridge 2001

[Ye and Johnson, 1995] Ye, L. R. Johnson, P. E. (1995) The impact of explanation facilities on user acceptance of expert systems advice, MIS Quarterly. 19, 2 (June 1995), pp. 157-172

[Yourdon, 1979] Yourdon, E. (1979) Structured design: Fundamentals of a discipline of computer program and system design, Prentice-Hall

[Zaremski and Wing, 1997] Zaremski, A. Wing, J. (1997) Specification Matching of Software Components, in ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 4, pp. 333 – 369

[Zhao, et al., 2007] Zhao, G. Luo, B. Tang, J. Ma, J. (2007) Using eigen-decomposition method for weighted graph matching. In Proceedings of the intelligent Computing 3rd international Conference on Advanced intelligent Computing theories and Applications, Lecture Notes In Computer Science, pp. 1283-1294, Springer-Verlag, Berlin / Heidelberg, Germany

[Ziadi, et al., 2011] Ziadi, T. da Silva, M.A.A. Hillah, L.M. Ziane, M. (2011) A Fully Dynamic Approach to the Reverse Engineering of UML Sequence Diagrams, 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), pp.107-116

8 Appendices

8.1 Appendix 1 – UMLSimilator System Diagrams

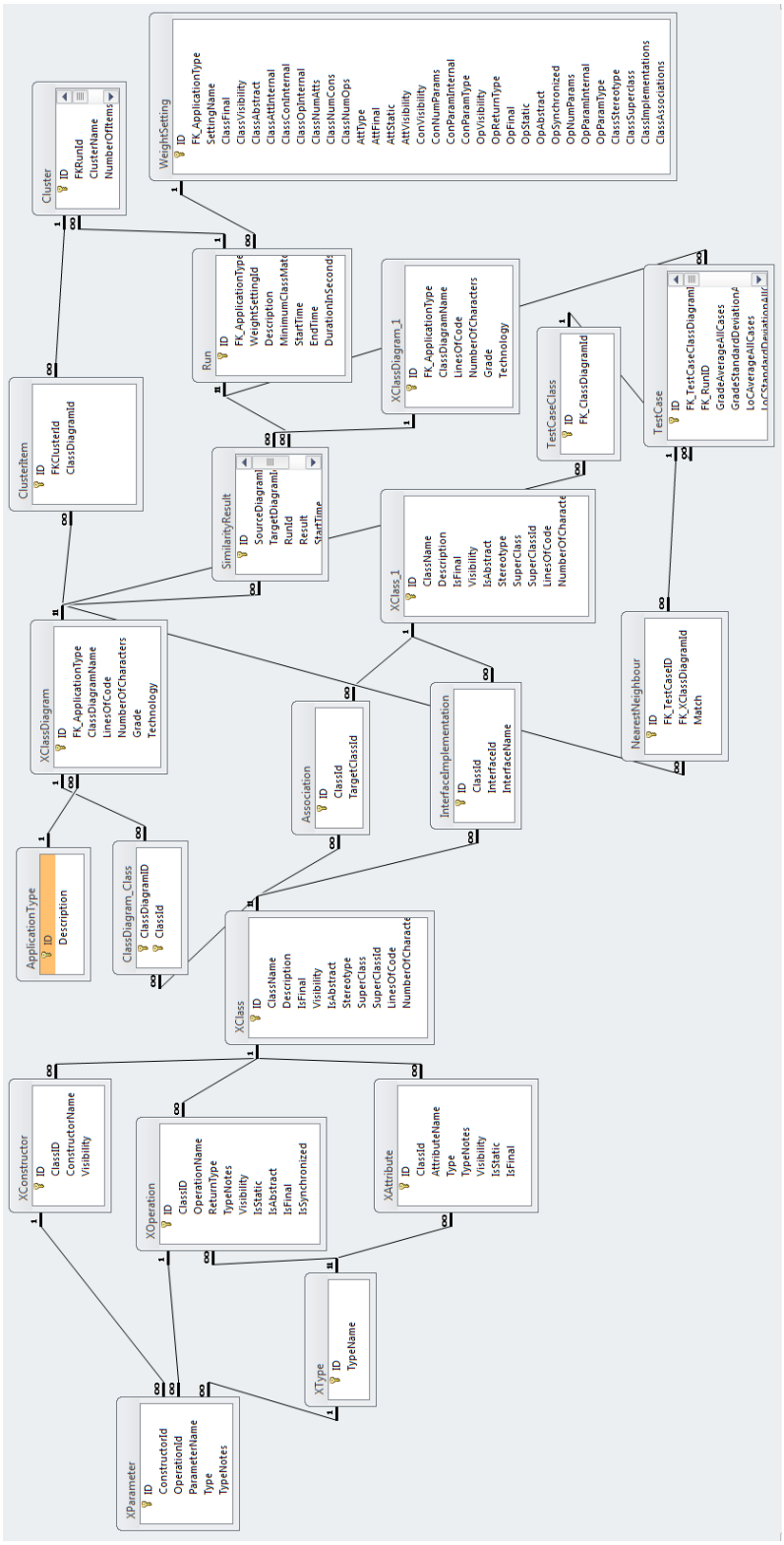

Figure 50 - Entity Relationship Diagram Depicting the Structure of the Case-Base

The entity relationship diagram on the previous page shows the database structure used to store the case-base, while the class diagram below shows the core domain classes of the UMLSimilator system.

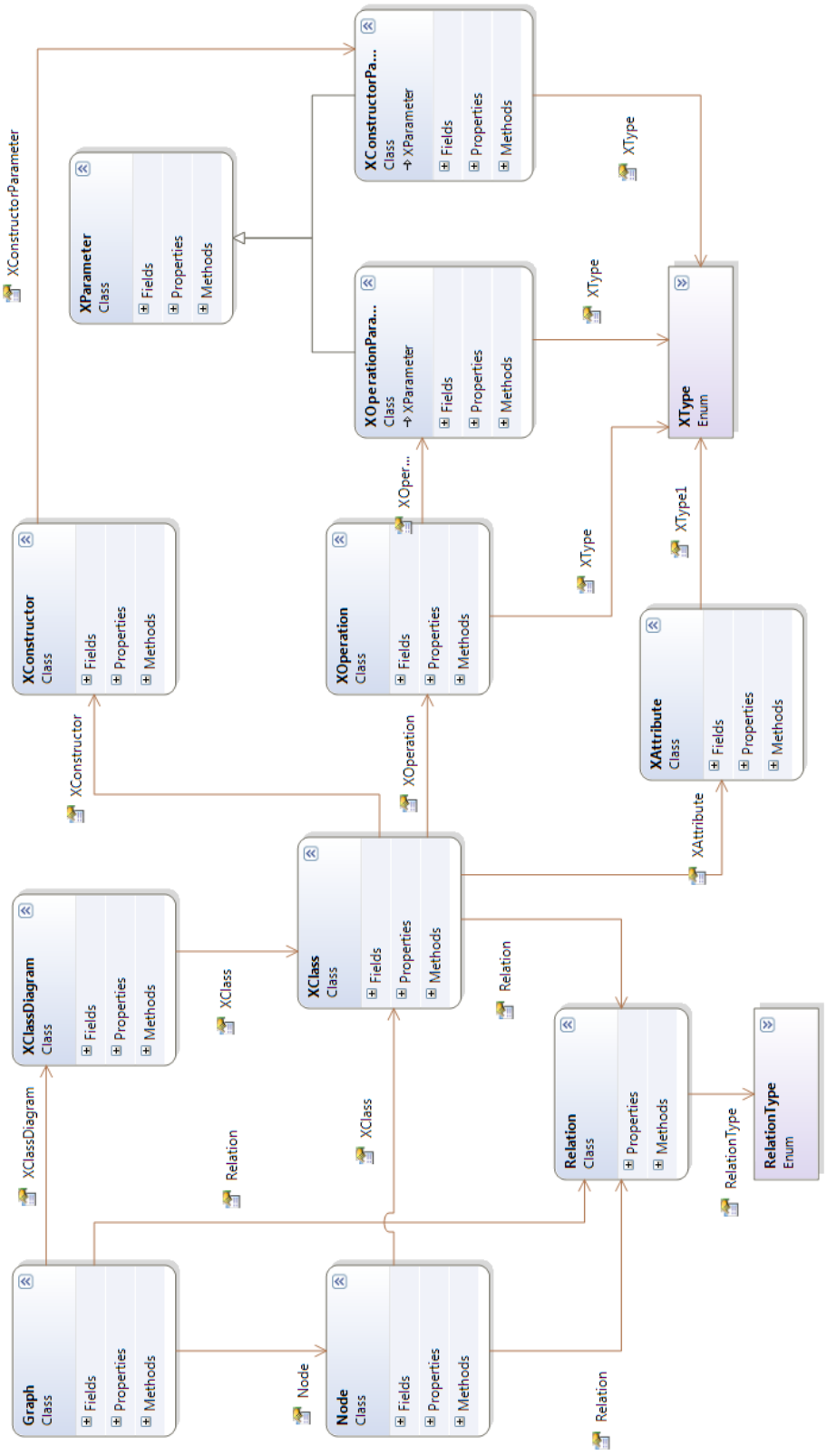

Figure 51 - Class Diagram showing Core Domain Classes

The Java Reflector is the only module of the UMLSimilator tool which is implemented in Java. It makes it possible to reverse-engineer compiled Java code from single class files, jar files or even nested jar files.

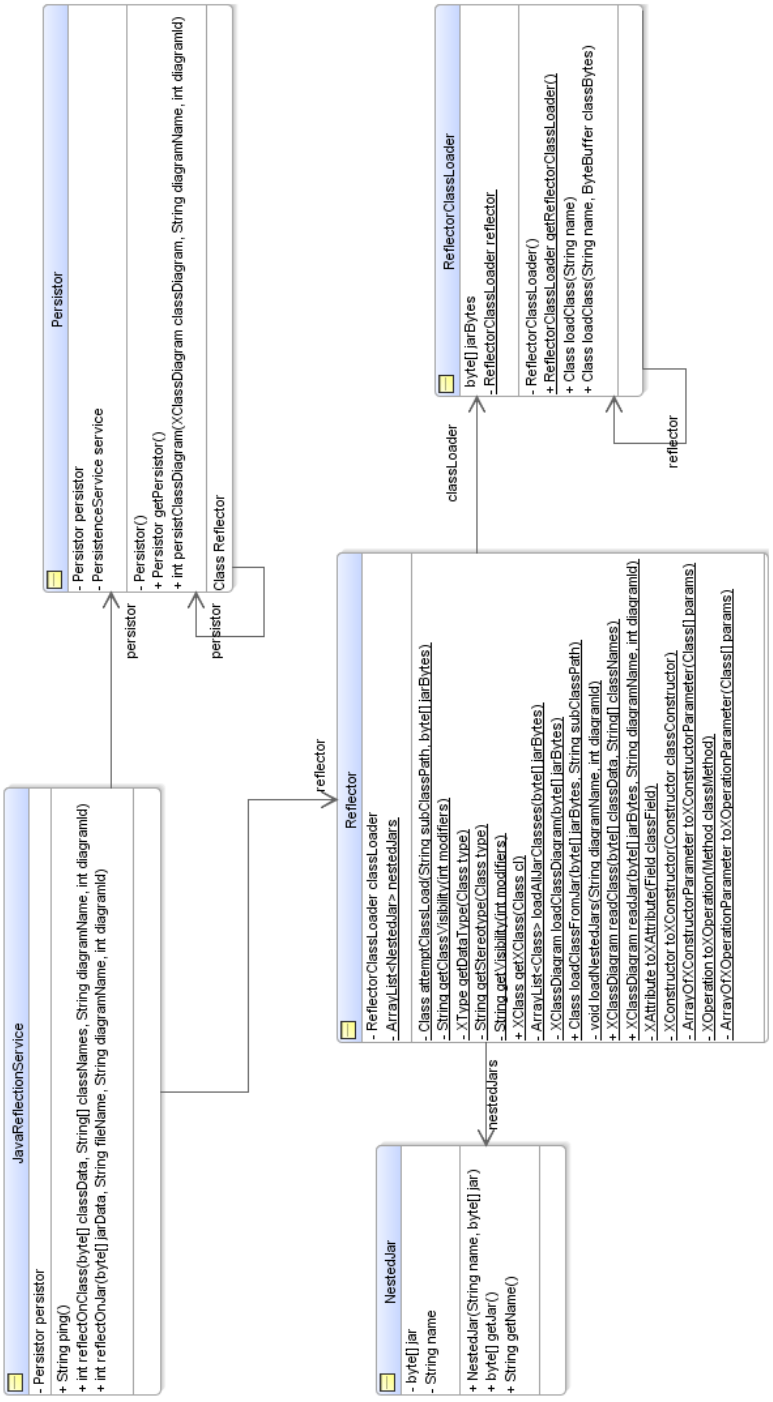

Figure 52 - Java Reflector Class Diagram

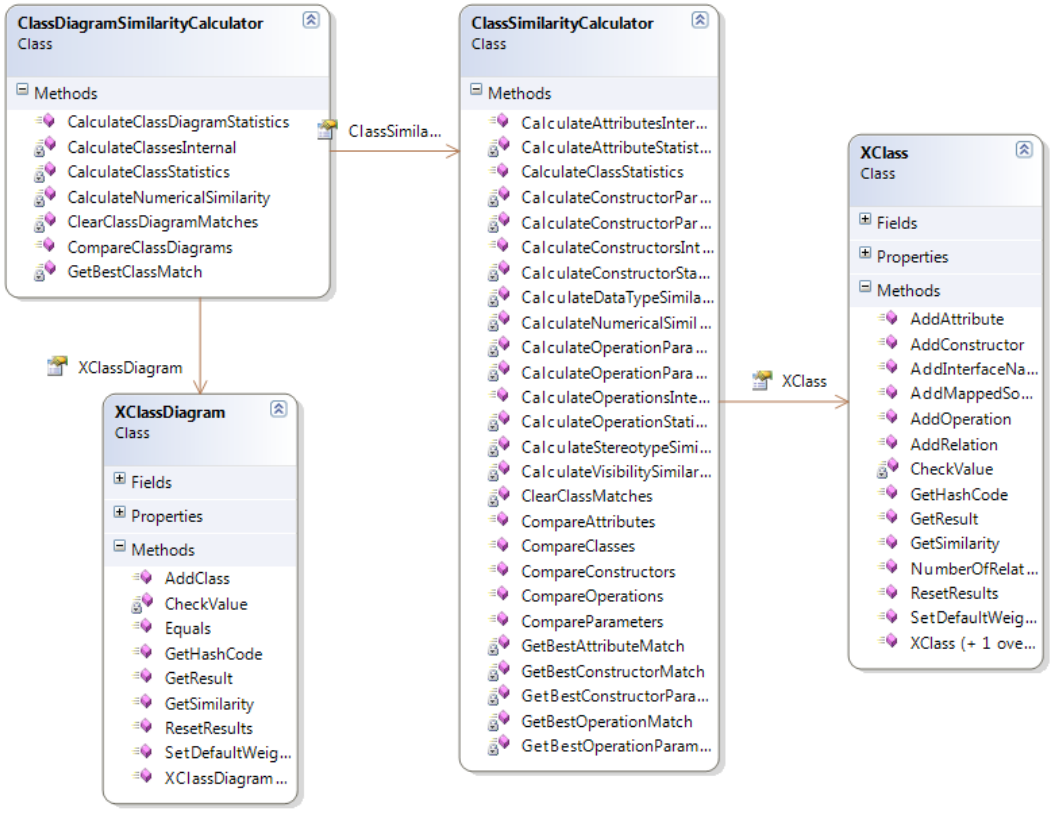

Figure 53 - Class Diagram for Similarity Calculator

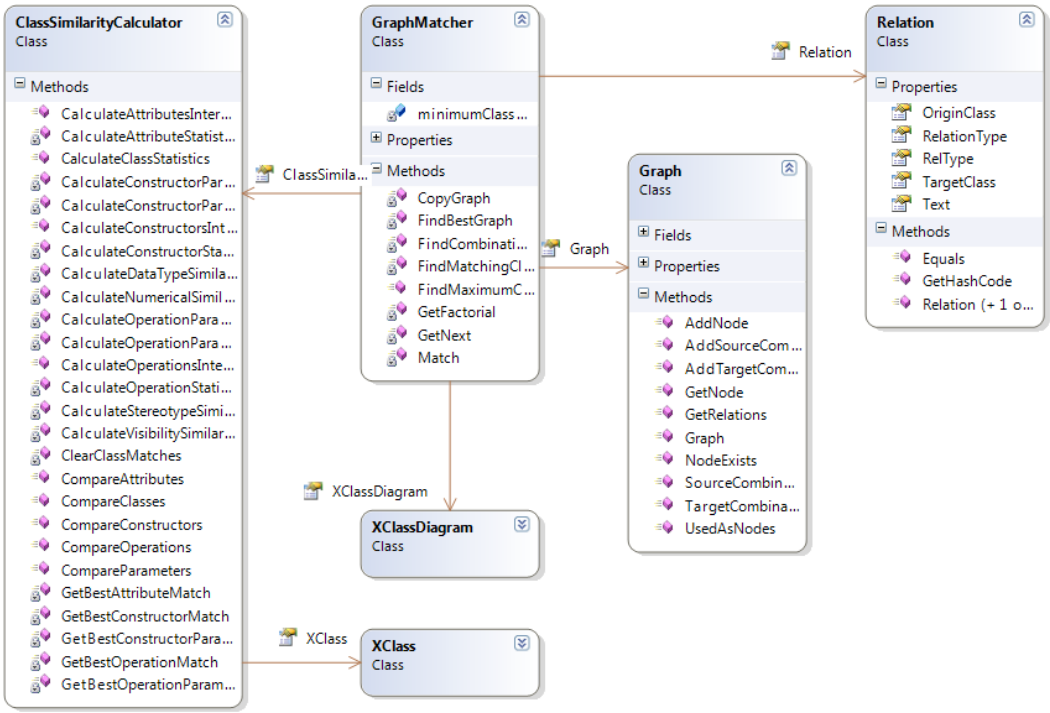

Figure 54 - Class Diagram for Graph Matcher

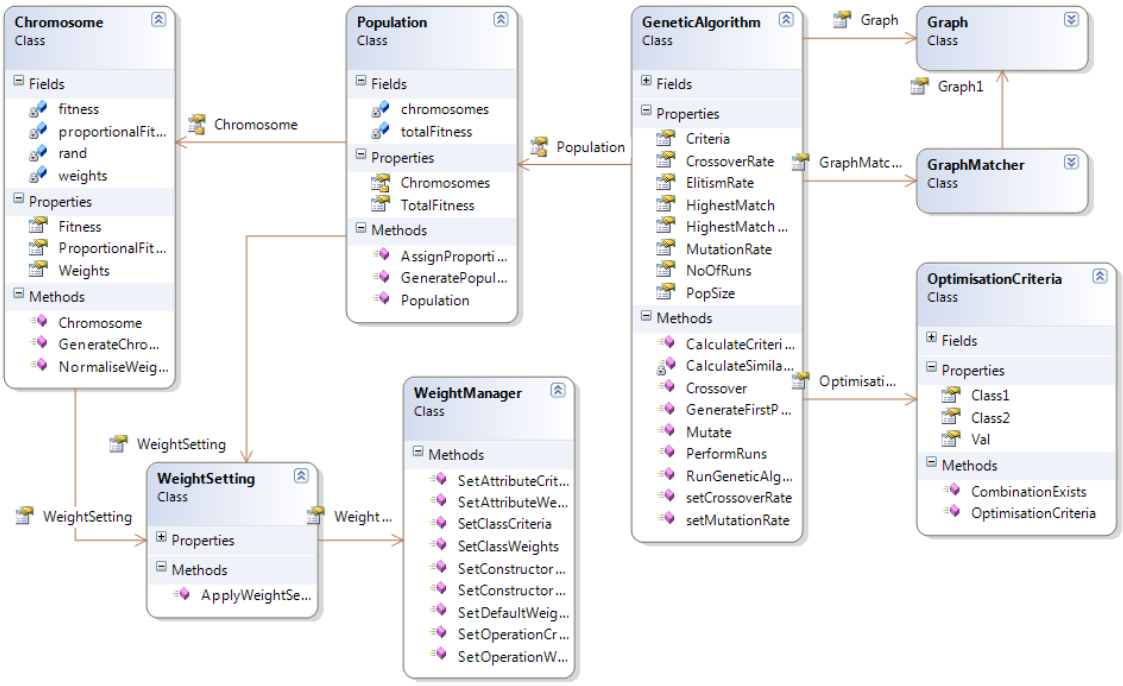

Figure 55 - Class Diagram for Weight Optimiser

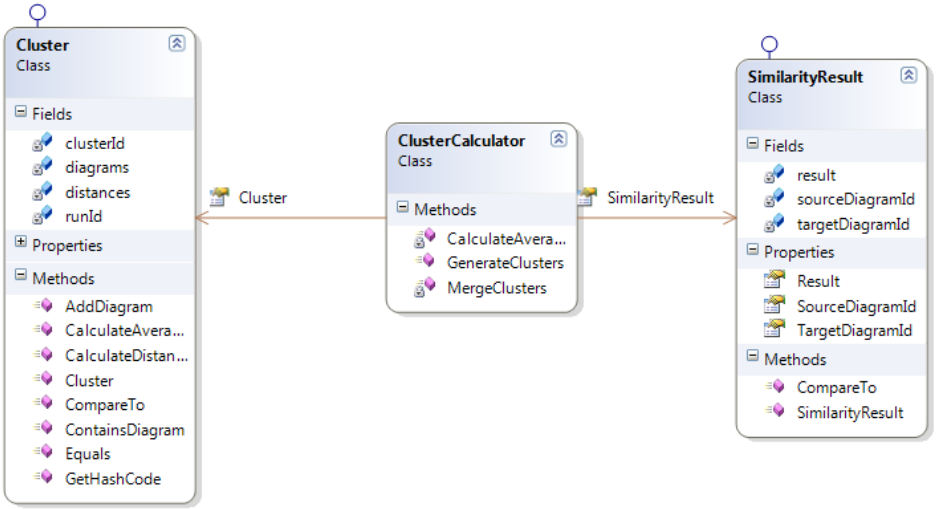

Figure 56 - Class Diagram for Clustering

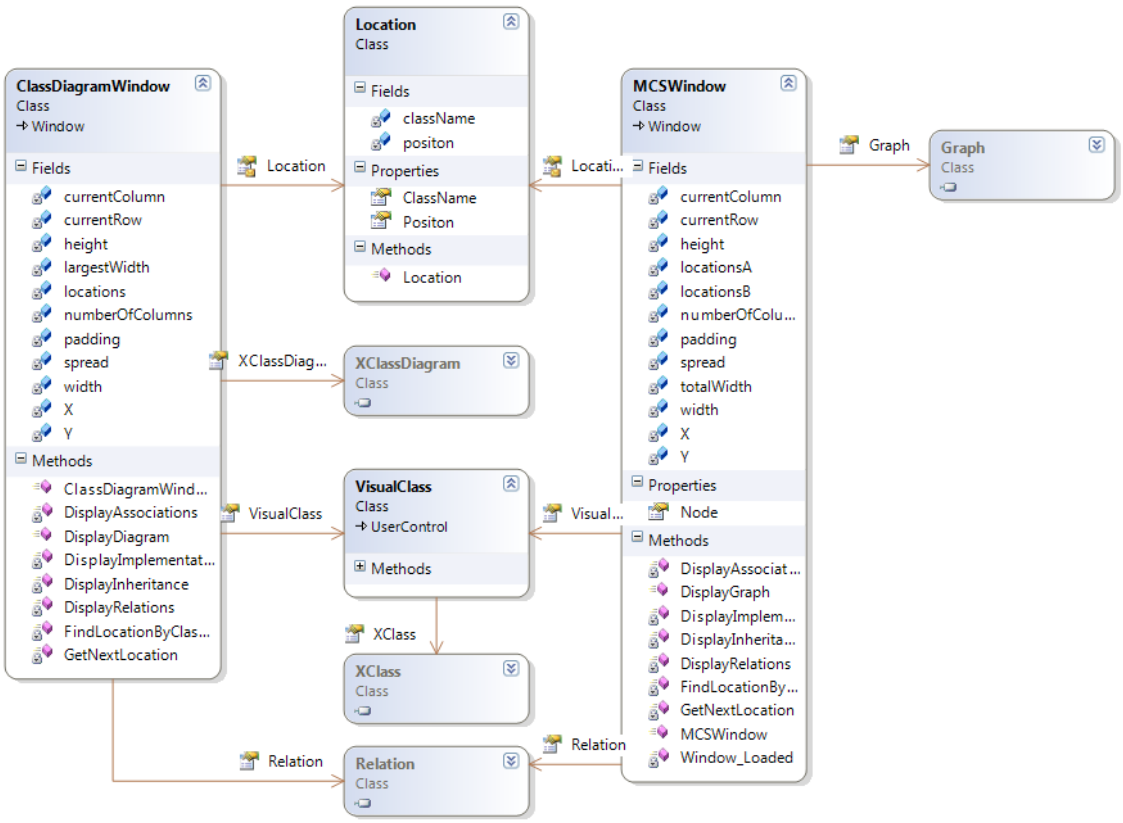

Figure 57 - Class Diagram for Visualiser

The class diagrams presented on the previous pages provide an overview of how the different modules of the UMLSimilator tool were implemented. To aid understanding of how the modules would interact with the classes from the core domain, these have been included in the module class diagrams where necessary.

8.2 Appendix 2 – Comparison of Nearest Neighbours

The following charts show the performance for one, three and five nearest neighbours when calculating structural similarity.

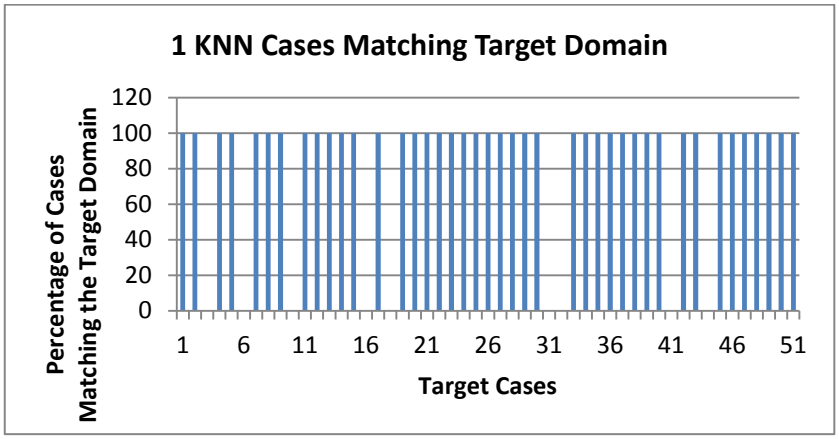

Figure 58 - Percentage of Cases Matching the Target Domain using One Nearest Neighbour

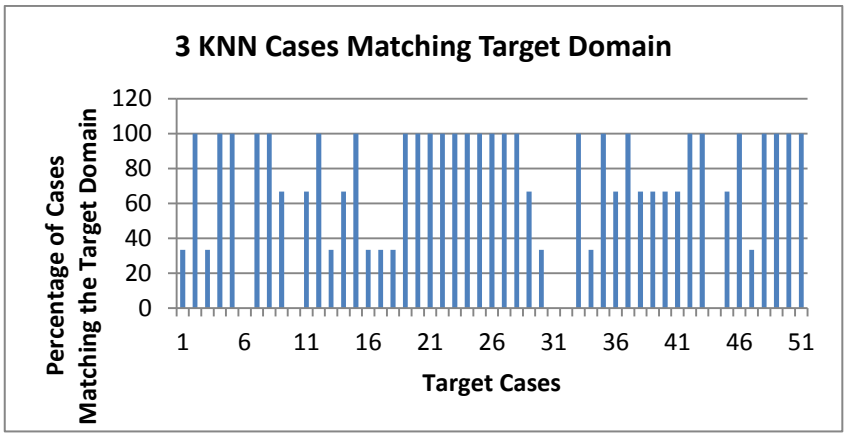

Figure 59 - Percentage of Cases Matching the Target Domain using Three Nearest Neighbours

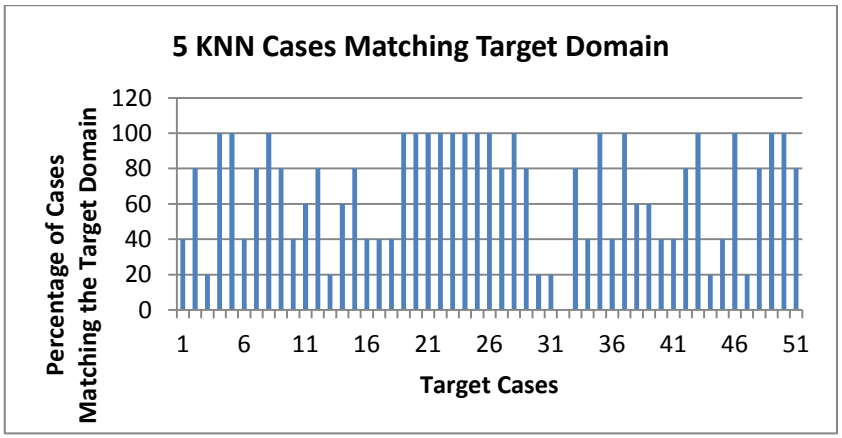

Figure 60 - Percentage of Cases Matching the Target Domain using Five Nearest Neighbours

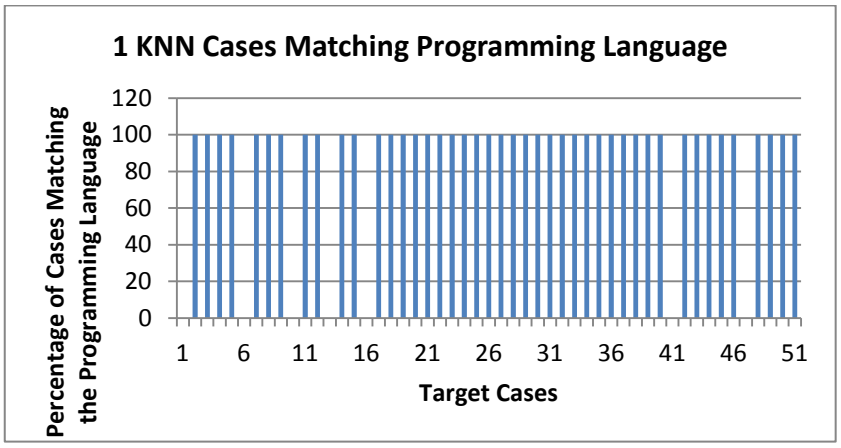

Figure 61 - Percentage of Cases Matching the Target Programming Language using One Nearest Neighbour

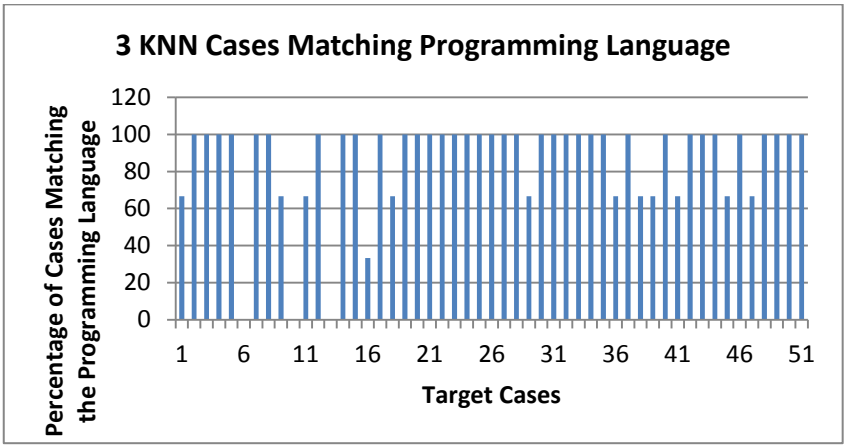

Figure 62 - Percentage of Cases Matching the Target Programming Language using Three Nearest Neighbours

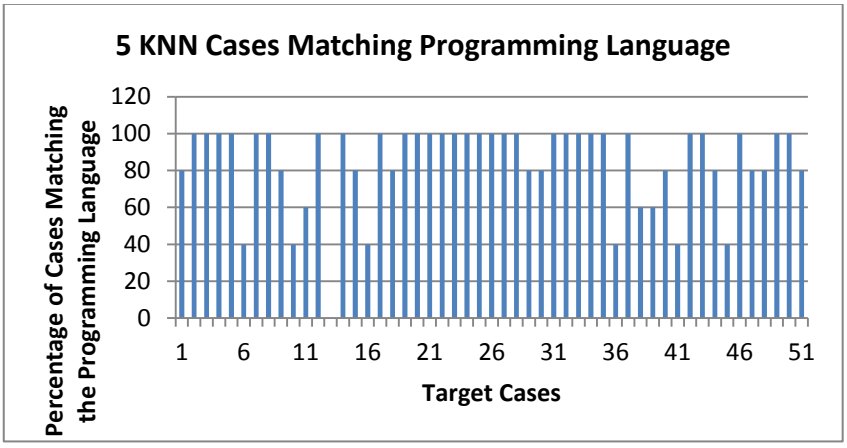

Figure 63 - Percentage of Cases Matching the Target Programming Language using Five Nearest Neighbours

For matching the domain and programming language one nearest neighbour generally performs best, but the matches are either 0% or 100%.

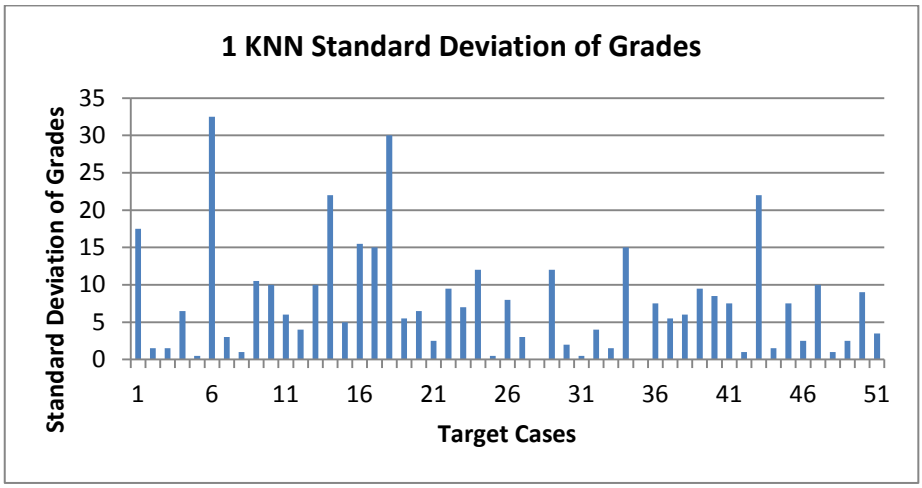

Figure 64 - Standard Deviation of Grades using One Nearest Neighbour

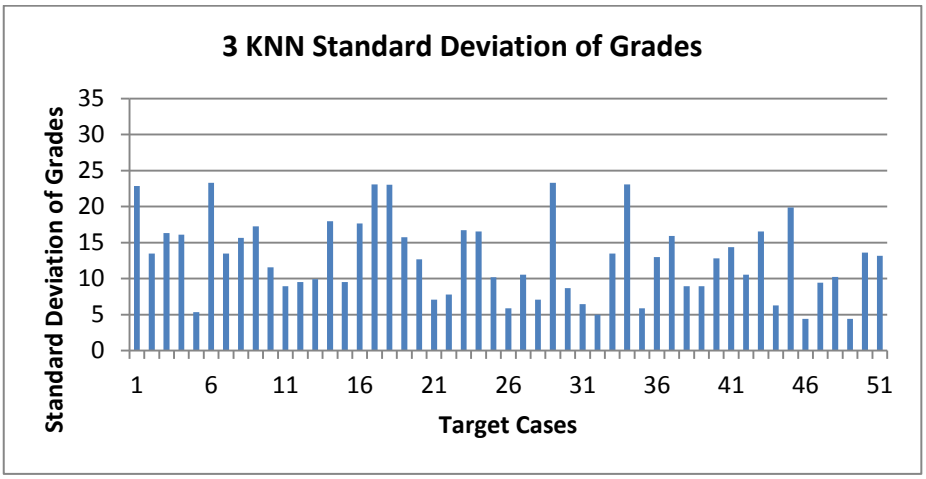

Figure 65 - Standard Deviation of Grades using Three Nearest Neighbours

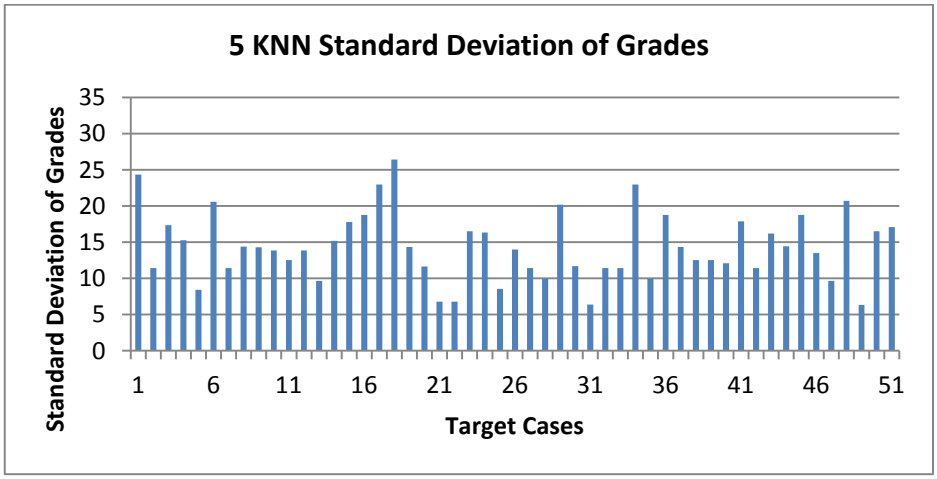

Figure 66 - Standard Deviation of Grades using Five Nearest Neighbour

Again, using one nearest neighbour creates the best results, although there are spikes for some cases, which in the case of three or five nearest neighbours were reduced.

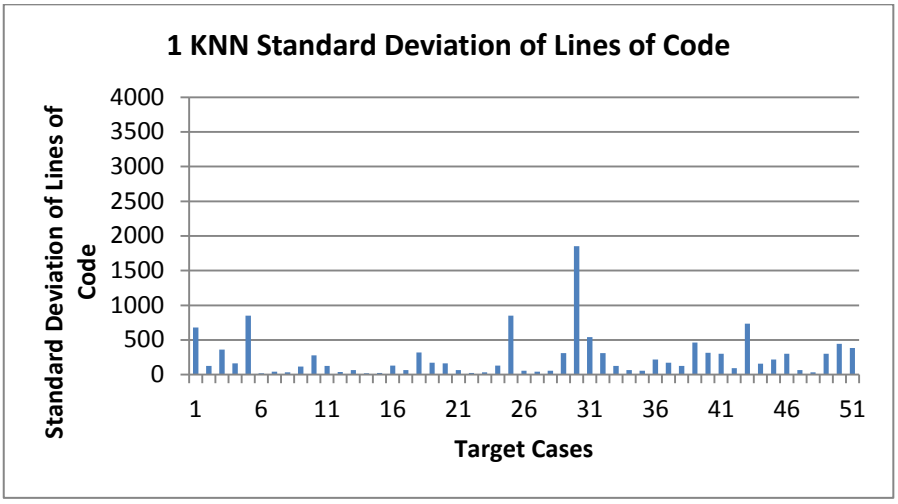

Figure 67 - Standard Deviation of Lines of Code using One Nearest Neighbour

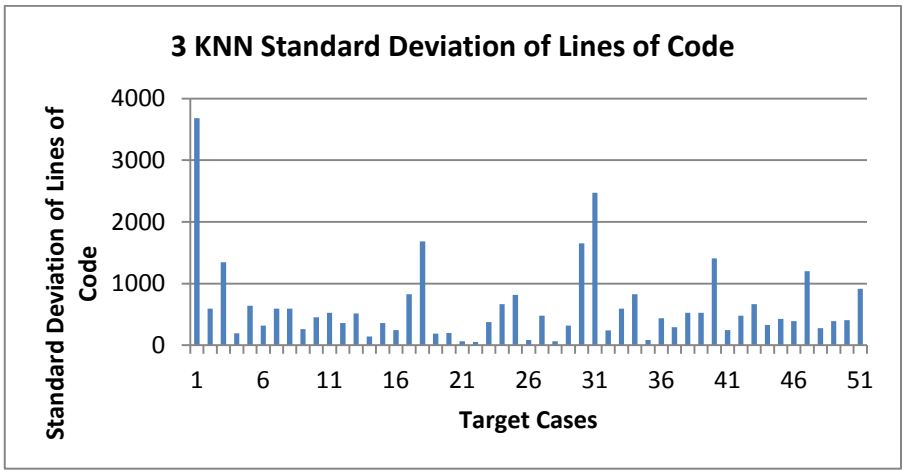

Figure 68 - Standard Deviation of Lines of Code using Three Nearest Neighbours

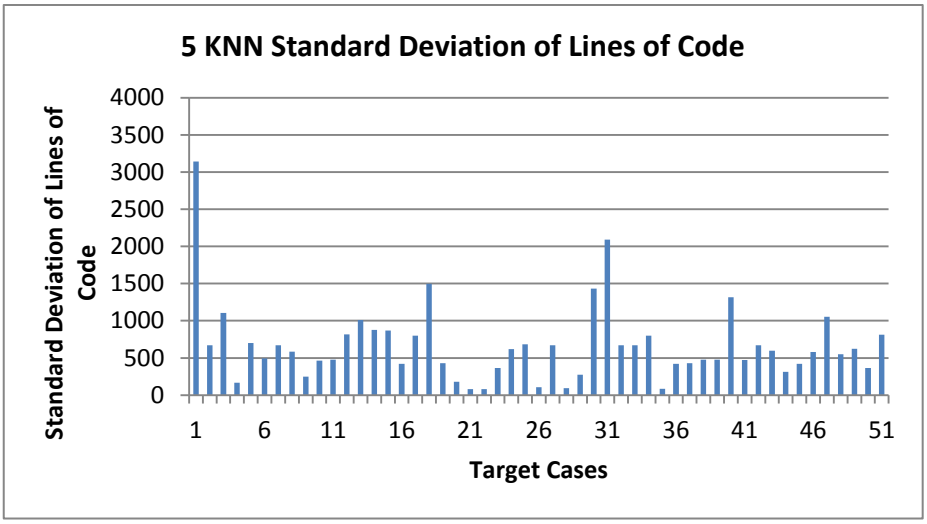

Figure 69 - Standard Deviation of Lines of Code using Five Nearest Neighbour

8.3 Appendix 3 – Minimum Class Match Threshold Settings

The minimum class match threshold is used to determine how similar two classes have to be in order for the algorithm to consider them as a potential pair in the maximum common subgraph.

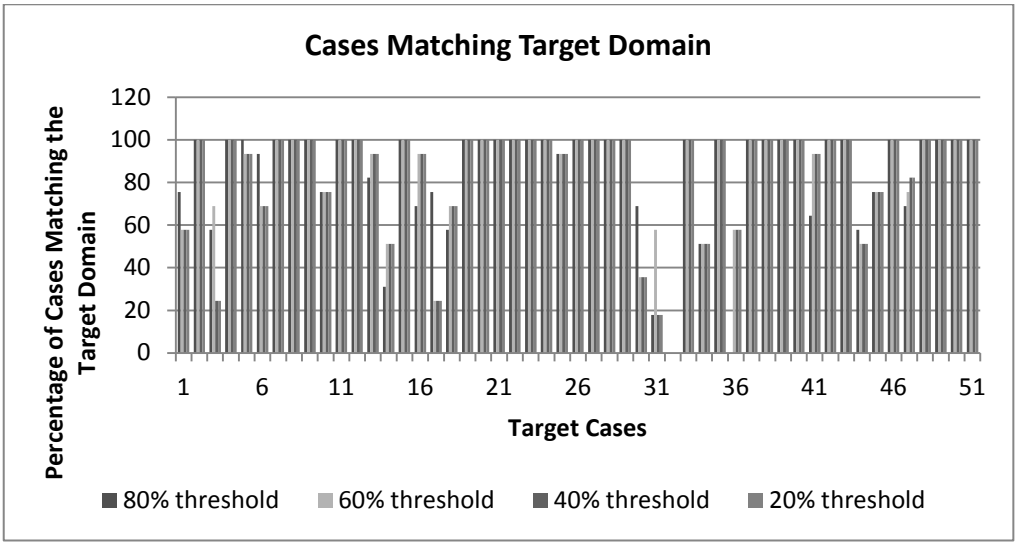

Figure 70 - Comparison of Different Threshold Settings for Matching Target Domain

For matching cases to the same domain as the target case, the 60% threshold performed best. It matched the cases correctly in 85.51% of cases.

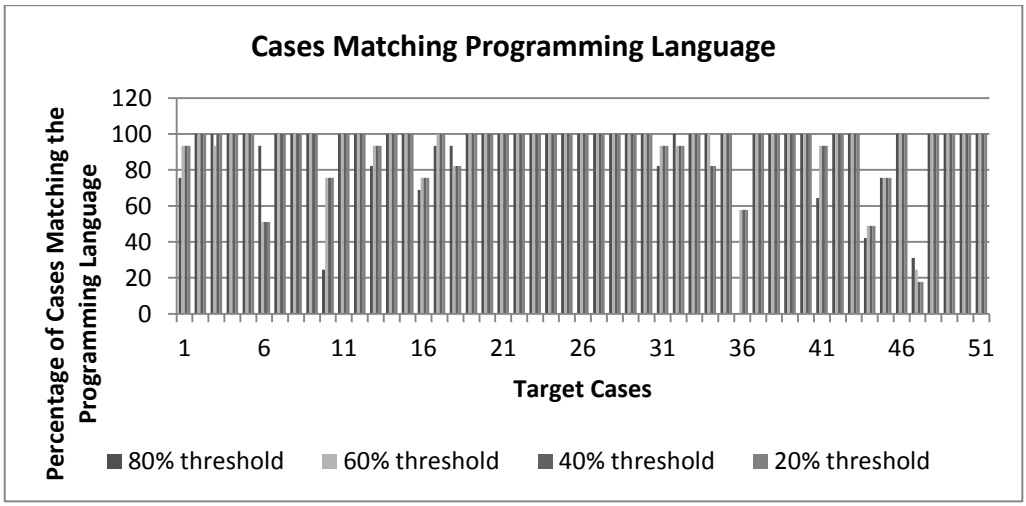

Figure 71 - Comparison of Different Threshold Settings for Matching Target Programming Language

For matching cases to the same programming language as the target case, again the 60% threshold performed best. It matched the cases correctly in 93.16% of cases.

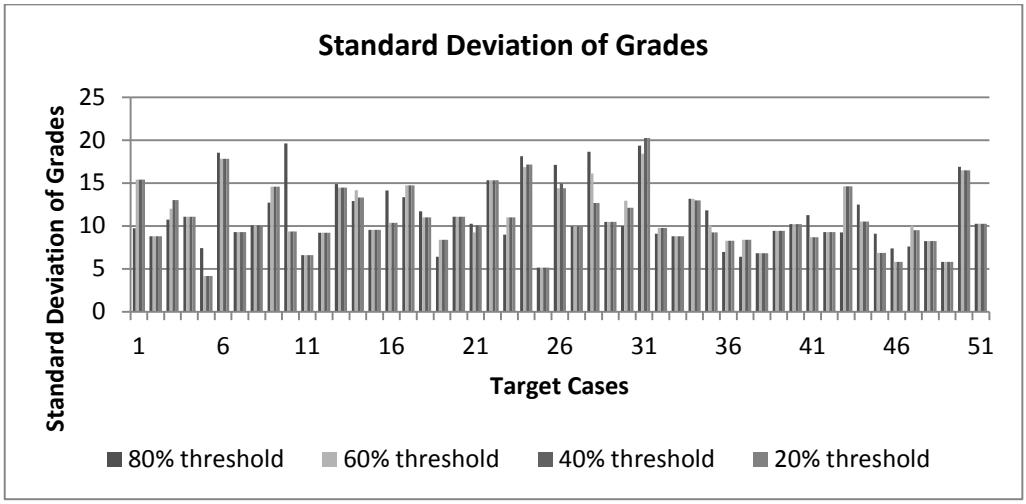

Figure 72 - Comparison of Different Threshold Settings for Measuring Standard Deviation of Grades

The best threshold setting for measuring standard deviation of grades was 20%, with an average standard deviation of grades of 10.79.

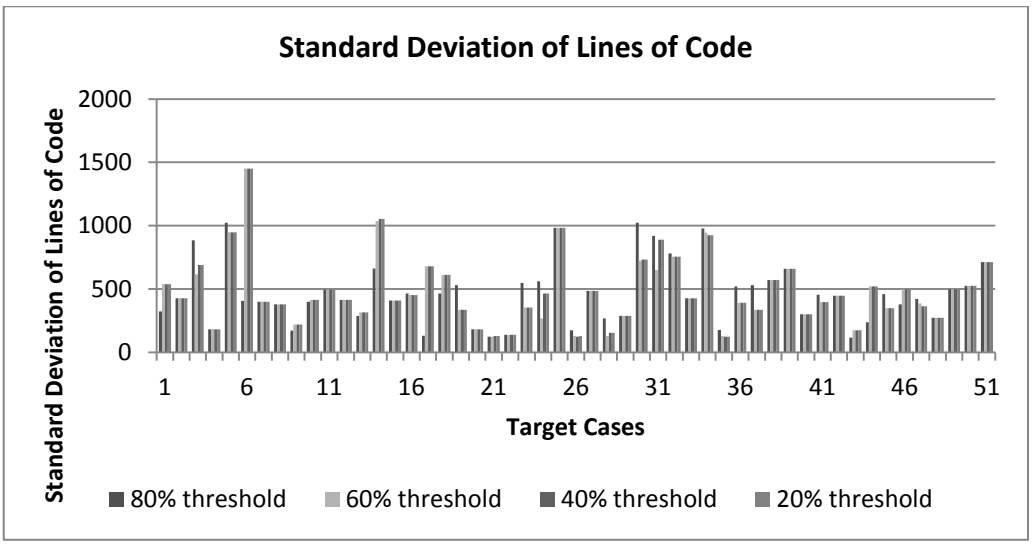

Figure 73 - Comparison of Different Threshold Settings for Measuring Standard Deviation of Lines of Code

The best threshold setting for measuring standard deviation of lines of code was 80%, with an average standard deviation of grades of 462.10.

Overall, the 60% threshold performed best. It performed best in two of the categories and was the only setting with all categories above the average. The variation in execution times between the categories varied greatly. With a lower threshold, the execution time would increase.

8.4 Appendix 4 – Expert Class Similarity Matches

The tables in this appendix show the desired class matches set by the expert for the provided class diagrams. The same class could be used in several matches. The range of values used was 2.5, 5, 7.5 and 10. The highest score obtained is also shown for each pair of class diagrams.

Classes From 3427	Classes From 3434	Expert Value	Highest Scoring Combination
DataServer	SetProject	2.5	
IPMSComponent	IPMSComponent	10	10
IProject	IProject	10	10
ITask	ITask	10	10
PMSServer	ProjectProgress	5	
Project	Project	5	
SQLServer	SetProject	2.5	
Task	Task	5	
Task	AddTask	2.5	
Team	ManageTeam	2.5	
Total			**30**

Table 24 – Expert Value Settings for Project Management Class Diagrams 3427 and 3434

Classes From 3401	Classes From 3407	Expert Value	Highest Scoring Combination
AdminForm	MainForm	7.5	
ClassResultWrapper	XClass	2.5	
FieldResultWrapper	XField	2.5	
IClassPersistence	IClassPersistence	10	10
MainForm	MainForm	5	
MethodResultWrapper	XMethod	2.5	
PersistenceService	DBPersistence	5	
RecordForm	AddForm	7.5	
StoreToSQL	DBPersistence	5	
WebService	WebService	5	
XMLPersistenceComponent	XMLComponent	5	
XMLPersistenceComponent	XMLPersistenceComponent	10	10
ZClass	ZClass	10	10
ZClass	XClass	5	
ZField	ZField	10	10
ZField	XField	5	
ZMethod	ZMethod	10	10
ZMethod	XMethod	5	
Total			**50**

Table 25 – Expert Value Settings for Software Cataloguing Class Diagrams 3401 and 3407

Classes From 3450	Classes From 3453	Expert Value	Highest Scoring Combination
AddCarForm	AddCar	7.5	
AddModelForm	AddModel	7.5	7.5
AddUpgradeForm	AddUpgrade	7.5	
Calculators	Discount	2.5	
Converter	Converter	7.5	
DelModelForm	DeleteCar	2.5	2.5
MainForm	FastLtdCon	5	
Operators	FastLtdCon	7.5	7.5
Search	ViewAllCars	5	5
Search	Search	7.5	
SellCarForm	CarsForSale	5	
Total			**22.5**

Table 26 – Expert Value Settings for Car Repair Shop Class Diagrams 3450 and 3453

Classes From 3488	Classes From 3489	Expert Value	Highest Scoring Combination
ArgsMainApplication	ArgsStores	5	5
Catalogue	EnquiryDesk	2.5	
ConnectArgsMySQL	DBHandler	5	
Products	NewEntries	2.5	2.5
StockList	StockUpdate	2.5	
Transactions	DBHandler	2.5	
Total			**7.5**

Table 27 – Expert Value Settings for Stock Management Class Diagrams 3488 and 3489

Classes From 3500	Classes From 3504	Expert Value	Highest Scoring Combination
Bid	Bid	10	10
Bidder	Bidder	10	
BidderPanel	BidderForm	7.5	
MainForm	AdminPanelForm	7.5	
PojectPanel	ViewProjectDetailForm	5	
Predecessor	Predecessor	7.5	7.5
Project	Project	7.5	7.5
ProjectDetails	ViewProjectDetailForm	7.5	
ProjectDetails	PredecessorForm	2.5	
ProjectPanel	ProjectForm	7.5	
Task	Task	10	10
Technology	MainTechnology	10	10
Total			**45**

Table 28 – Expert Value Settings for Project Bidding Class Diagrams 3500 and 3504

8.5 Appendix 5 – Weight Settings Obtained from Genetic Algorithm

Given that the class scoring didn't work, the genetic algorithm was used to generate weight settings that improved similarity matches between class diagrams selected by the expert. The table below shows the complete sets of weights values for all features.

Feature	Default Weights	Software Cataloguing Weights	Project Management Weights	Car Repair Shop Weights	Stock Management Weights	Project Bidding Weights
Final modifier	0.077	0.081	0.104	0.057	0.036	0.119
Visibility modifier	0.077	0.069	0.128	0.037	0.183	0.119
Abstract modifier	0.077	0.101	0.004	0.163	0.056	0.128
Stereotype (enumeration/interface)	0.077	0.060	0.164	0.106	0.129	0.092
Number of attributes	0.077	0.021	0.001	0.021	0.002	0.061
Number of constructors	0.077	0.110	0.039	0.088	0.083	0.038
Number of operations	0.077	0.113	0.049	0.014	0.020	0.003
Superclass	0.077	0.075	0.047	0.061	0.177	0.133
Number of implementations	0.077	0.044	0.100	0.046	0.094	0.095
Number of associations	0.077	0.035	0.001	0.072	0.026	0.036
Attributes (internal structure)	0.077	0.097	0.081	0.157	0.045	0.056
Constructors (internal structure)	0.077	0.112	0.156	0.043	0.038	0.020
Operations (internal structure)	0.077	0.081	0.125	0.135	0.113	0.099
Class Total	***1***	***1***	***1***	***1***	***1***	***1***
Data type	0.25	0.037	0.467	0.143	0.312	0.091
Final modifier	0.25	0.025	0.404	0.401	0.071	0.182
Static modifier	0.25	0.346	0.009	0.072	0.276	0.307
Visibility modifier	0.25	0.592	0.119	0.384	0.341	0.421
Attribute Total	***1***	***1***	***1***	***1***	***1***	***1***
Visibility modifier	0.333	0.096	0.263	0.760	0.414	0.171

Number of parameters	0.333	0.177	0.410	0.222	0.340	0.784
Parameters (internal structure)	0.333	0.727	0.326	0.018	0.246	0.045
Constructor Total	***1***	***1***	***1***	***1***	***1***	***1***
Visibility modifier	0.125	0.034	0.035	0.090	0.207	0.215
Return type	0.125	0.064	0.190	0.093	0.089	0.001
Final modifier	0.125	0.263	0.242	0.269	0.064	0.051
Static modifier	0.125	0.084	0.114	0.101	0.079	0.233
Abstract modifier	0.125	0.280	0.128	0.203	0.194	0.021
Synchronised modifier	0.125	0.022	0.214	0.003	0.155	0.195
Number of parameters	0.125	0.111	0.005	0.146	0.025	0.148
Parameters (internal structure)	0.125	0.142	0.073	0.095	0.187	0.136
Operations Total	***1***	***1***	***1***	***1***	***1***	***1***

Table 29 – Comparison of Default Weights and Weights Obtained using the Genetic Algorithm

The default weights are obtained by assigning equal weights to every feature in the same level of the class structure hierarchy.

For every domain a set of weights was generated by asking an expert to match the most similar class diagrams within a domain. The genetic algorithm was then used to maximise the overall similarity match by testing various weights.

The tables on the following pages show the class diagram pairs that the expert identified as being the best match within each of the domains.

This approach worked better than setting class scores. However, the automatically generated weight settings didn't improve results by as much as was initially expected.

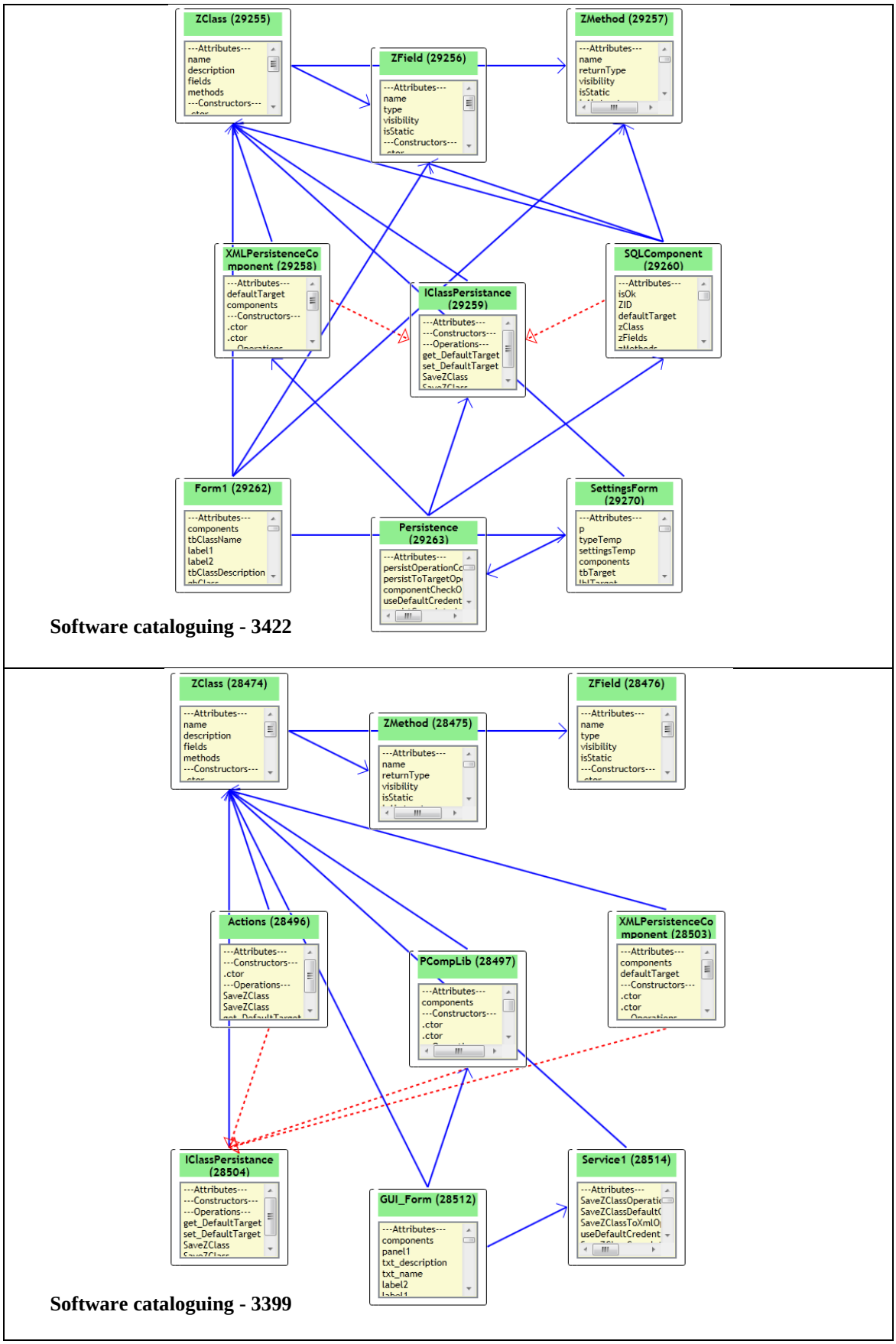

Figure 74 – Software Cataloguing Class Diagram and Expert's Best Choice

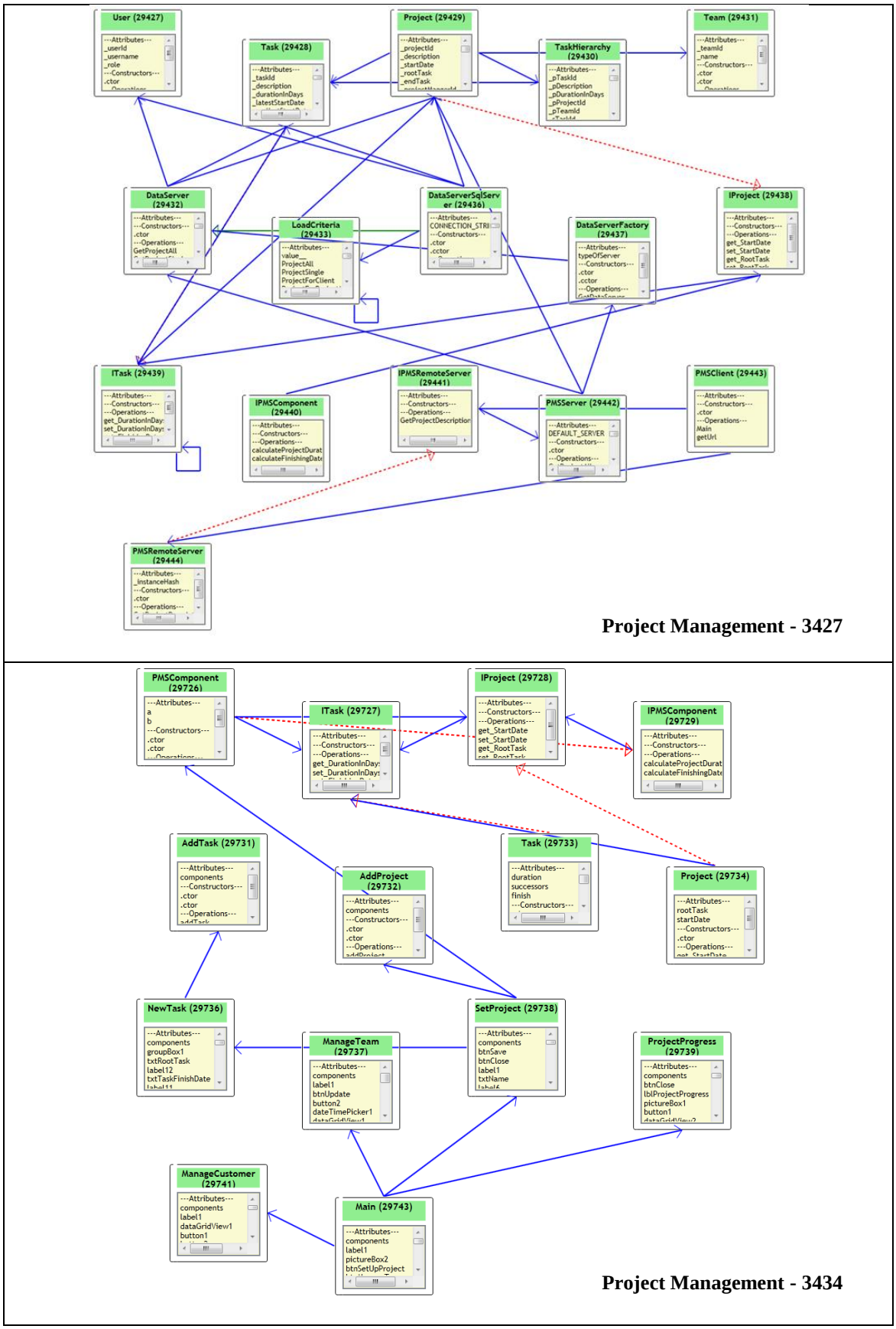

Figure 75 – Project Management Class Diagram and Expert's Best Choice

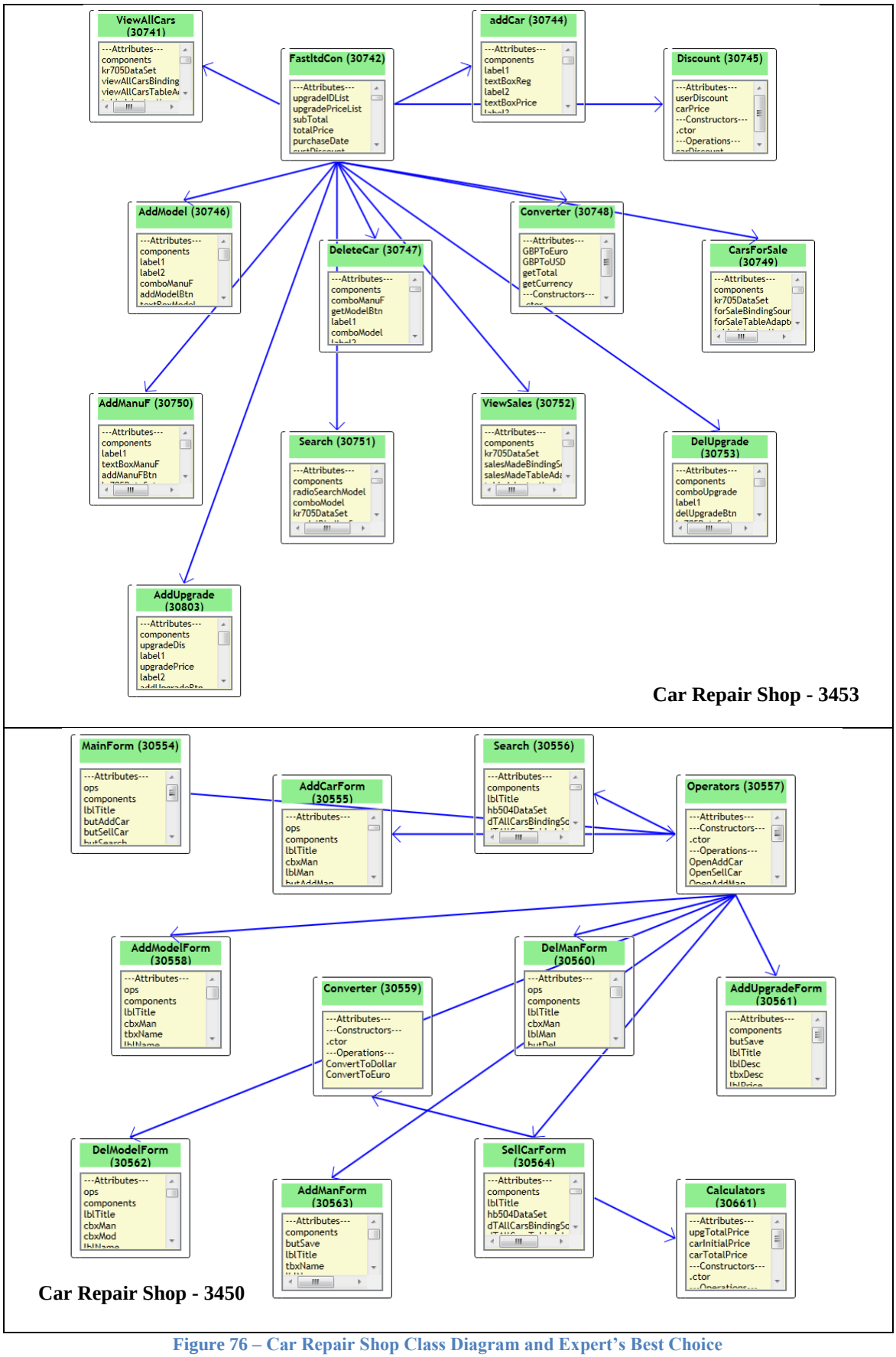

Figure 76 – Car Repair Shop Class Diagram and Expert's Best Choice

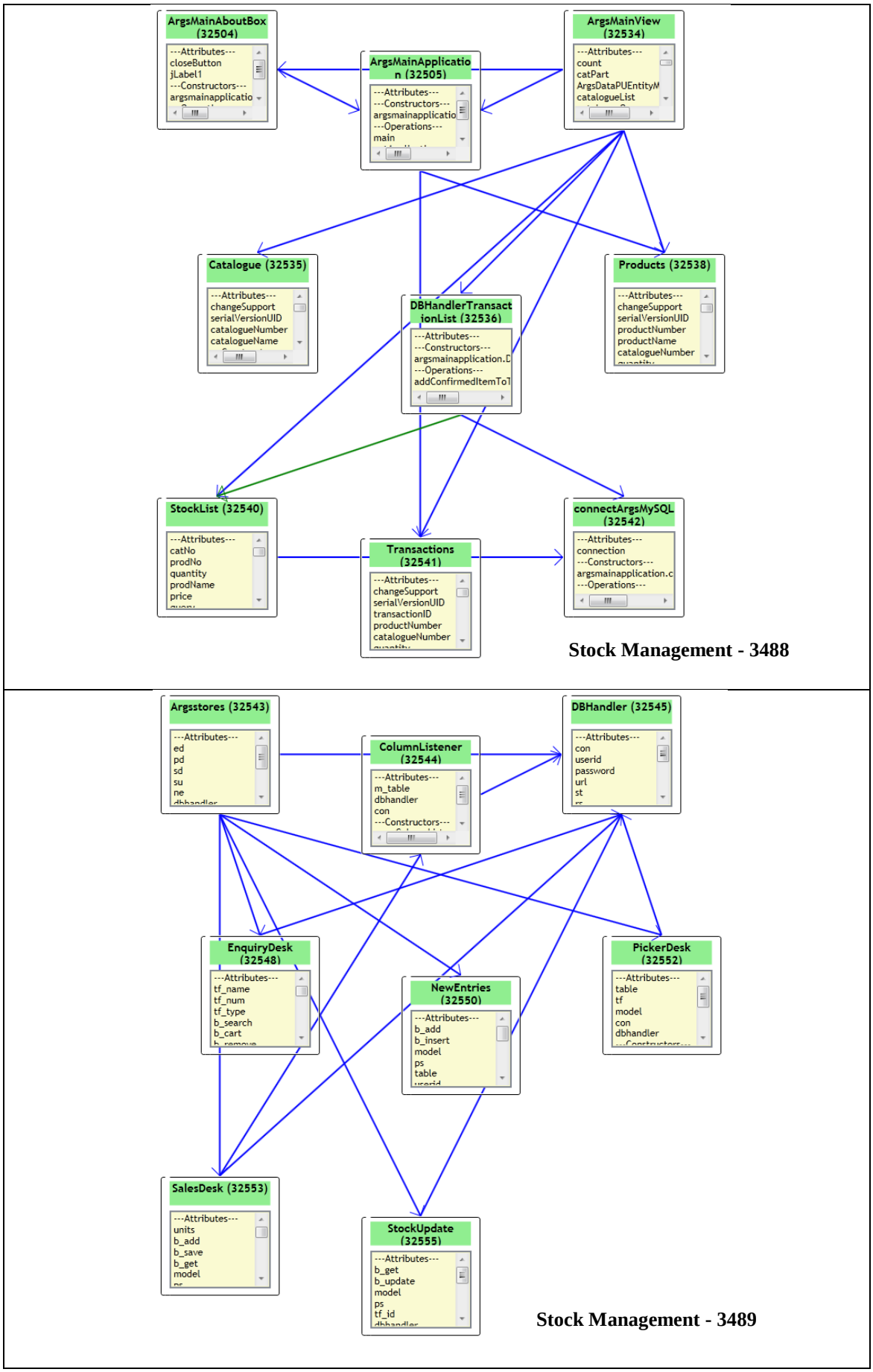

Figure 77 – Stock Management Class Diagram and Expert's Best Choice

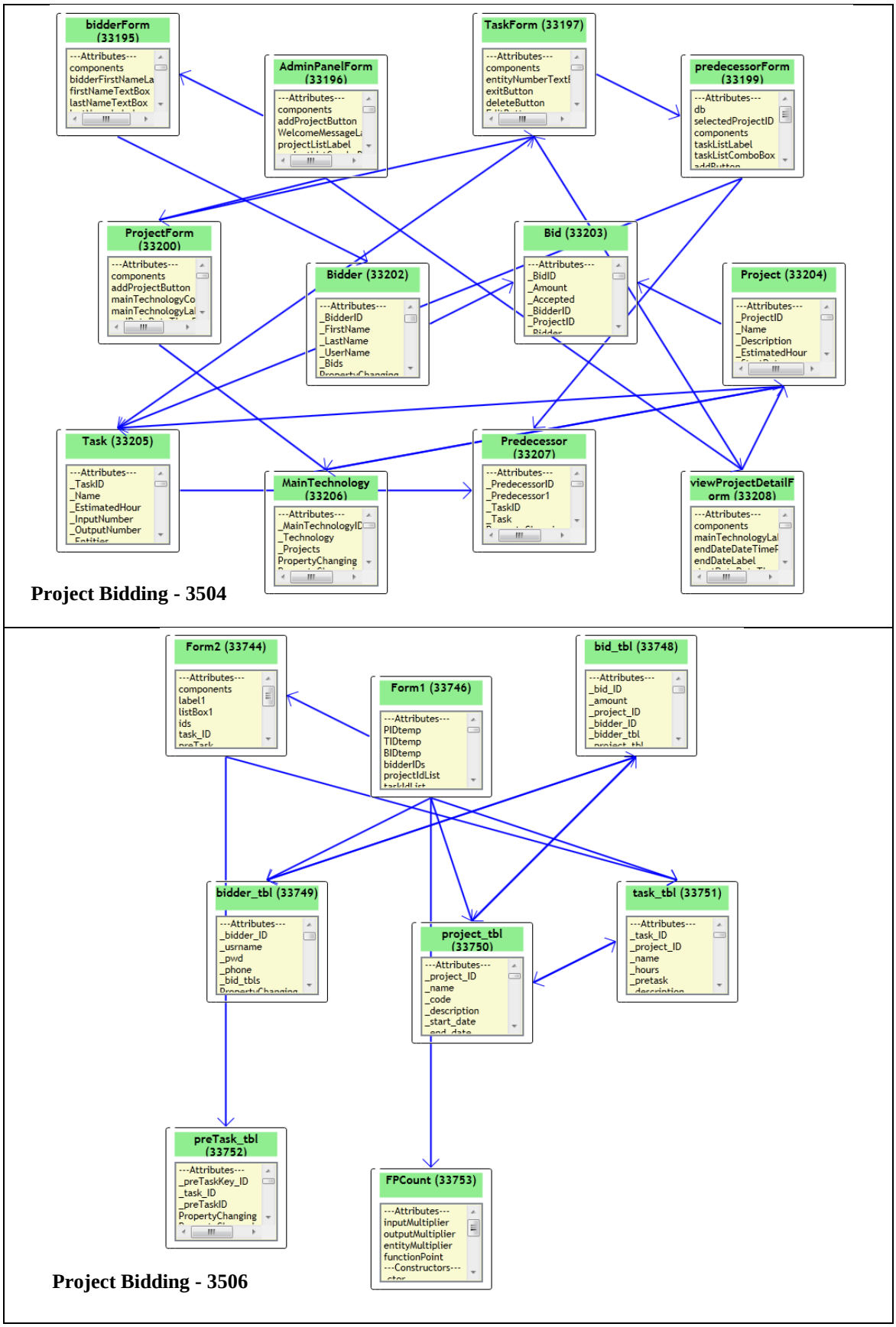

Figure 78 – Project Bidding Class Diagram and Expert's Best Choice

8.6 Appendix 6 – Results From Domain Weight Settings

The following charts show the results obtained by using the five weight settings generated using the genetic algorithm.

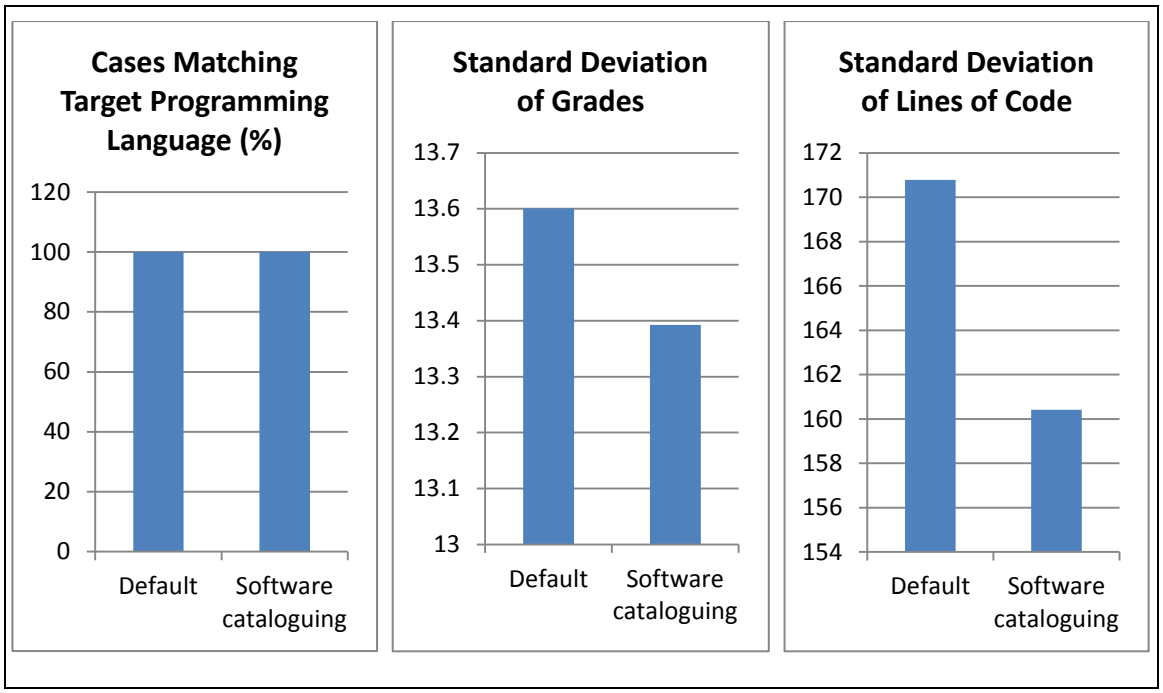

Figure 79 – Graph Similarity Results Using Default and Software Cataloguing Weights and Provenance

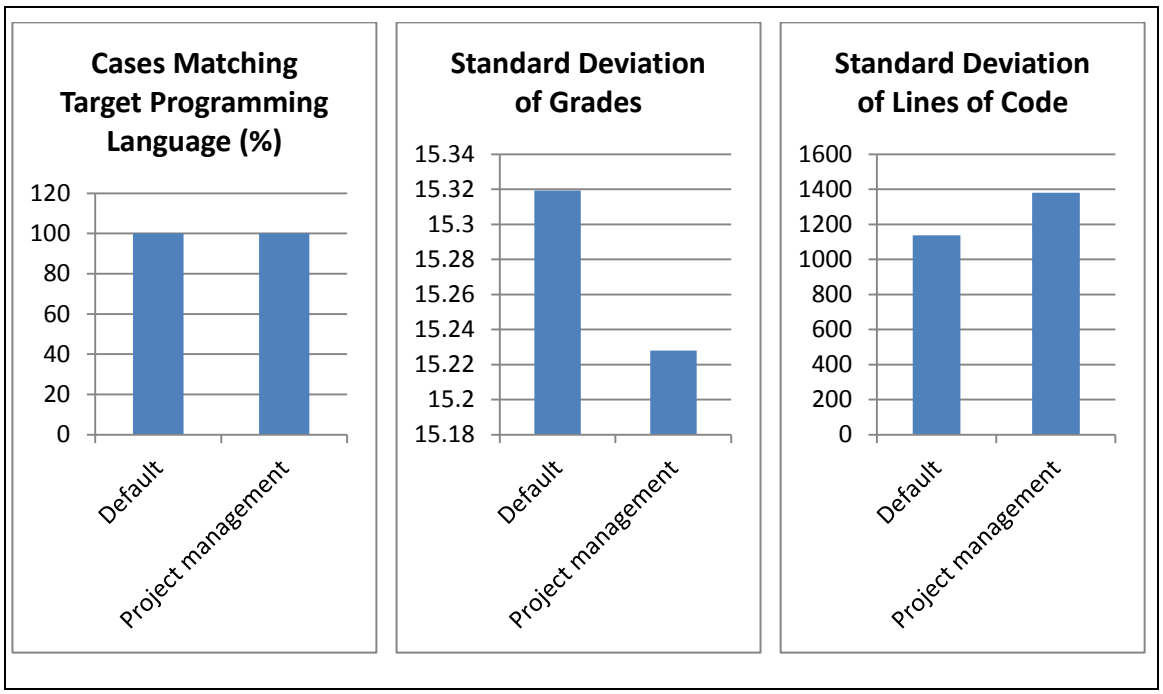

Figure 80 – Graph Similarity Results Using Default and Project Management Weights and Provenance

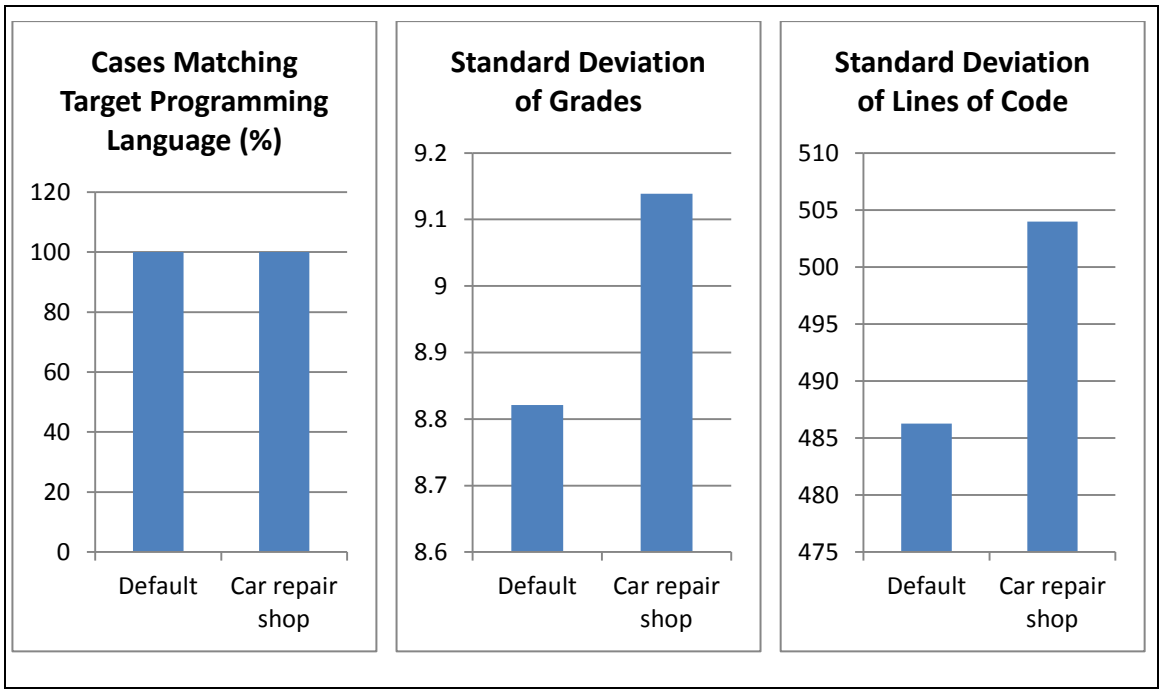

Figure 81 – Graph Similarity Results Using Default and Car Repair Shop Weights and Provenance

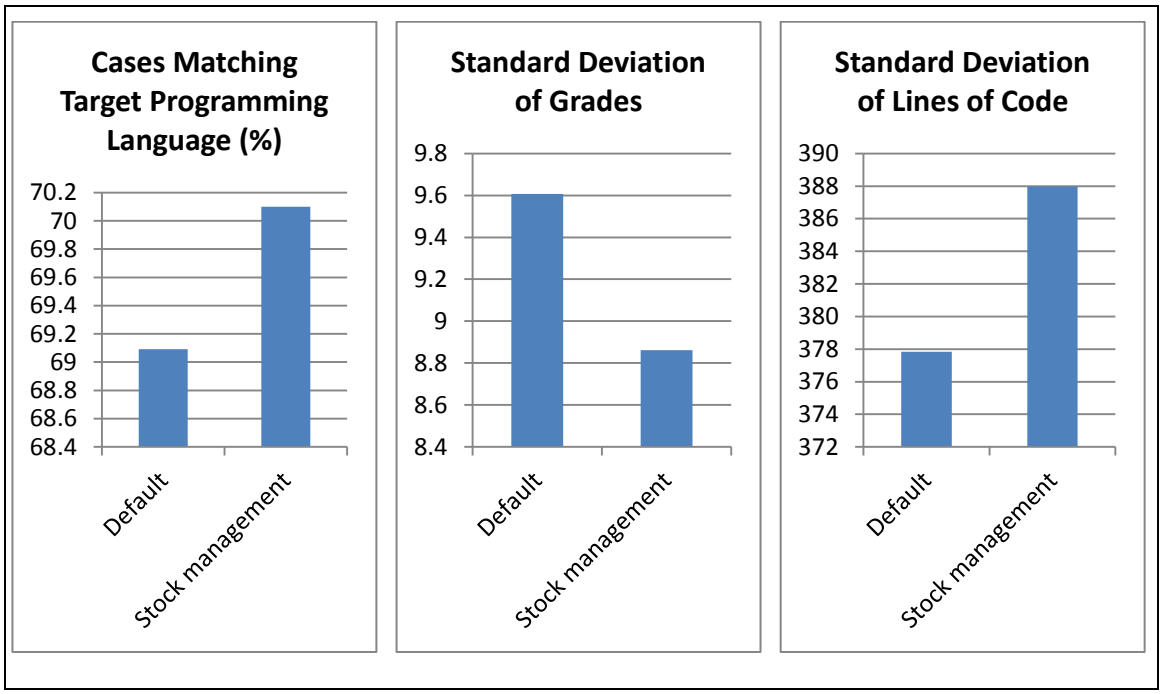

Figure 82 – Graph Similarity Results Using Default and Stock Management Weights and Provenance

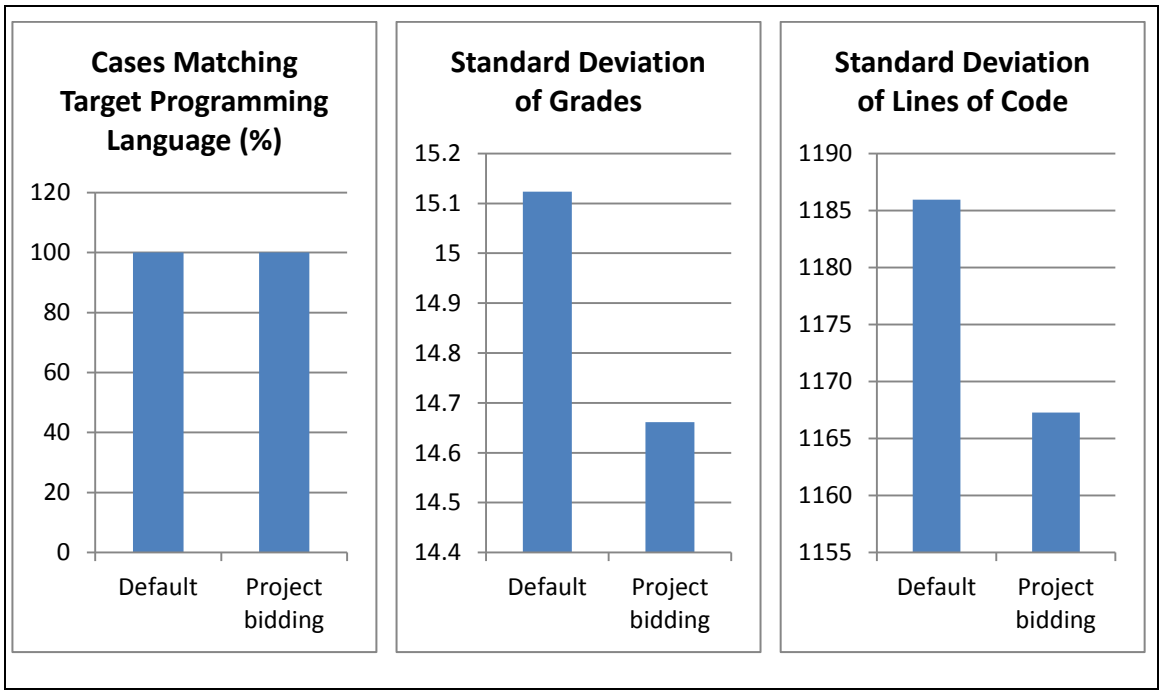

Figure 83 – Graph Similarity Results Using Default and Project Bidding Weights and Provenance

The percentage of cases matching the same domain has been omitted as this would always be 100%, given that cases are only selected from the same domain.

8.7 Appendix 7 – Publication from ECAI

The following paper has been published in the proceedings of the Artificial Intelligence Techniques in Software Engineering Workshop, 18th European Conference on Artificial Intelligence.

Measuring Similarity of Software Designs using Graph Matching for CBR

Markus Wolf and Miltos Petridis[7]

Abstract. This paper examines different ways for measuring similarity between software design models for Case Based Reasoning (CBR) to facilitate reuse of software design and code. The paper considers structural and behavioural aspects of similarity between software design models. Similarity metrics for comparing static class structures are defined and discussed. A Graph representation of UML class diagrams and corresponding similarity measures for UML class diagrams are defined. A full search graph matching algorithm for measuring structural similarity diagrams based on the identification of the Maximum Common Sub-graph (MCS) is presented. Finally, a simple evaluation of the approach is presented and discussed.

1 Introduction

The entire object-oriented paradigm revolves around reuse of software design and code. We create classes, which define blueprints for creating any number of objects. Reuse happens at an intra-class level, where we can factor code into methods which can be reused within the class. Inheritance enables us to reuse existing classes through extension and specialisation. Classes can be packaged as components, which can be reused. However, these are all examples of code reuse. Reuse in object-oriented development is not limited to the reuse of code, although this is the kind of reuse which has been applied at a practical level for a long time. Traditionally, less importance has been given to the reuse of software design, but this has changed, especially with the emergence of design patterns. In order to reuse software design it is essential to be able to compare designs and measure similarity of designs. Case-based reasoning [1] has been successfully applied to support reuse in an object-oriented environment [2]. Software designs are complex structures and there is no preset benchmark for comparing or defining similarity between software designs or elements of software designs. This makes calculating the similarity between software design models a somewhat difficult and complex process to automate. The key lies in classifying the characteristics that would make a human expert identify the similarity between given designs.
One important factor for a human expert is certainly semantic similarity. This will allow an expert to recognise what type of system a particular design depicts. Research has been undertaken by Gomes et al [3] into software design similarities using lexical similarity. This research makes use of WordNet, an electronic lexical database, which allows conceptual, as well as, lexical searches to measure semantic similarities.

Another interesting approach is that of Bjornestad [4], which uses analogical reasoning to measure similarity between object-oriented specifications. Emphasis is placed on the role of an object within the context of the software design.

The approach introduced in [5] and also adopted in this paper steps away from both, semantic and role similarities, and starts from the assumption that all information on a given software design model may be obfuscated. Thus, there may be no information about the design other than structural. It is important to distinguish between the structure and functionality of a software design, but the question is whether the structure contains within itself enough information to identify its functionality. At least to the extent that software designs with similar functionality can be found to have similar structures and behaviour.

Zaremski and Wing [11] use a different notion semantic similarity in their work on specification matching of software components. Their approach requires metadata in form of the specification of a component's behaviour. While not measuring semantic similarity of names used, their use of specifications in form of pre- and post-conditions provides semantic description of a component.

[8] Department of Computing Science, University of Greenwich, Park Row, London SE10 9LS email:{M.A.Wolf, M.Petridis}@gre.ac.uk

As with any complex structure, partitioning into substructures or composing elements is required in order to be able to compare software models. In order to define the overall similarity of the entire structure, these elements are compared against each other.

By dividing a software design into its composing elements, a hierarchy of elements is created. These can then be compared to the corresponding elements from another design at different levels of the hierarchy. As each element contributes to the overall similarity in uneven shares, a weighting scheme is used to assign the weights proportionally. This way the weights can be adapted individually to reflect the reasoning process of the human expert.

A way of creating substructures in a software design is to treat a software design as a graph composed of various design elements and their relationships. As discussed by Bergmann and Stahl [6], the similarity between two objects is a combination of the intra-class similarity and the inter-class similarity. By representing software designs as graphs, it is possible to compare a design with regards to its composing elements, but also their relationships.

Section 2 of this paper discusses similarity measures for the structural similarity between individual classes. Section 3 extends the approach to a similarity measure for UML class diagrams based on a graph representation of the structural information of the diagrams and presents a graph matching algorithm based on the identification of the Maximum Common Sub-graph (MCS). A simple evaluation of the approach is presented. Section 4 discusses the extension of the approach to other UML design artefacts representing behavioural aspects of software design.

2 Structural Similarity of Classes

Class diagrams are used in object-oriented software design to depict the classes of a software design and the way these relate to one another. Apart from the classes and their relationships, class diagrams often show additional information, such as certain class-specific properties, attributes and operations. If a class is regarded as a hierarchy, then they are the main elements of a class diagram and will therefore be the top element in the hierarchy of components of a software design.
Current object-oriented programming languages, such as Java and C#.NET, follow a standard model for determining the elements of the class structure. The class structure thus may contain modifiers, which state whether a class is final or abstract. It could also contain attributes or methods (operations) and constructors. The attributes, methods and constructors will in turn have defining properties themselves. This way the hierarchy of elements is created, as can be seen in figure 1.

Class modifiers state something about a class overall. Another relevant aspect of a class' similarity to other classes is its complexity. An indication of this is the number of attributes, constructors and methods. Furthermore, at a lower level, the number of constructor and method parameters. Any names of classes, attributes, etc. are disregarded. This is done due to the fact that this information is semantic rather than structural.

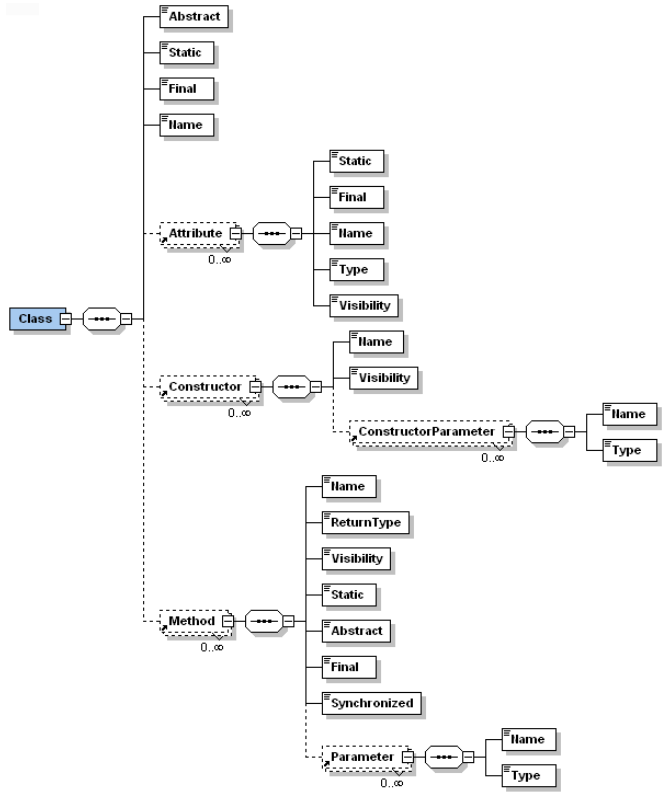

Figure 1. **Hierarchical structure of a class' composing elements**

The following rule applies when comparing elements between classes: the lack of elements in both classes being compared would denote a 100% similarity for that element, while the absence in merely one class would result in a 0% similarity. For example, if neither class had any attributes, the overall attribute similarity was 100%, while if one class contained attributes, but the other does not, 0% similarity would be returned.

In order to calculate the similarity between classes a software tool called UMLSimilator was developed.

XML was used to provide maximum flexibility in dealing with the case base. XML is well-structured and as a text-based format it is widely supported and easily transferable, yet capable of representing complex data structures through containment.

Existing implementations were used to populate the case base. The cases were obtained by reverse engineering existing implementations.

The UMLSimilator contains a module to convert classes from their compiled byte code into cases in XML format. This is done using the reflective ability of the Java programming language, by which means compiled classes are dynamically loaded at runtime and the features automatically extracted. The original design is thus reverse engineered from the byte code and stored in XML format for later retrieval. Existing tools for reverse engineering code have been discussed by Gorton and Zhu [13].

Research into this area has been undertaken by Tessem, Whitehurst and Powell [7], who use Java's reflective ability combined with Case-Based Reasoning to "retrieve case-based components in a prototyping tool for the Java programming language." This tool is used to aid class retrieval and reuse, but on a code rather than design level and even though our research is based on existing implemented designs, it is currently only concerned with the design level. Their approach is also oriented towards semantic similarity similarly to Gomes et al [3]. The research presented here investigates additional class elements, which might contain important information for measuring software design similarity based on structure. The following equation shows how the similarity between two classes is calculated.

$$S(C,C') = \omega_{hl} S_{hl}(C,C') + \omega_{struct} S_{struct}(C,C') \quad (1)$$

The overall similarity combines the structural and the high-level similarity between the two given classes, where ω_{hl} is the weight applied to the high-level similarity and ω_{struct} is the weight applied to the structural similarity

The high-level similarity between two classes is based on the defining characteristics of a class (its modifiers) and the number of attributes, methods and constructors. These are properties of a class which can be obtained without having to look at the detailed structure of the class.

The high-level similarity between two classes is obtained as follows:

$$S_{hl}(C,C') = \sum_{\substack{all \\ \text{mod}ifiers}} \omega_{\text{mod}} S_{\text{mod}}(C,C') +$$

$$+ \omega_{no.attr} S_{no.attr}(C,C') + \omega_{no.methd} S_{no.methd}(C,C') +$$

$$+ \omega_{no.constr} S_{no.constr}(C,C') \qquad (2)$$

where "all modifiers" are the modifiers of a class, ω_{mod} is the weight applied to the similarity of each of these modifiers, $\omega_{no.attr}$ is the weight assigned to the number of attributes and similarly $\omega_{no.methd}$ and $\omega_{no.constr}$ refer to the number of methods and number of constructors.

The structural similarity is concerned with the internal structure of a class (attributes, methods and constructors), as stated in equation 3, where ω_{attr} is the weight applied to the sum of all matched attributes' similarity, $n_{attr.matches}$ refers to the total number matches and A_x, A'_{bmx} refers to an attribute and its best match (bmx). Similarly, ω_{methd} refers to methods and ω_{constr} to constructors.

$$S_{struct}(C,C') = \omega_{attr} \frac{1}{n_{attr.matches}} \sum_{\substack{all \\ matched \\ Attributes}} S_{attr}(A_x, A'_{bmx}) +$$

$$\omega_{methd} \frac{1}{n_{methd.matches}} \sum_{\substack{all \\ matched \\ Methd}} S_{methd}(M_x, M'_{bmx}) + \qquad (3)$$

$$+ \omega_{constr} \frac{1}{n_{constr.matches}} \sum_{\substack{all \\ matched \\ constructors}} S_{constr}(Cs_x, Cs'_{bmx})$$

The similarity between two attributes is defined as follows:

$$S_{attr}(A,A') = \sum_{\substack{all \\ \text{mod}ifiers}} \omega_{\text{mod}} S_{\text{mod}}(A,A') + \omega_{type} S_{type}(A,A') \qquad (4)$$

The modifiers in this case are static, final and visibility. ω_{type} refers to the weight assigned to the type of an attribute.

When calculating the similarity between two methods, the modifiers specific to each method, the number of arguments and the similarity between these arguments are all taken into consideration:

$$S_{methd}(M,M') = \sum_{\substack{all \\ \text{modifiers}}} \omega_{\text{mod}} S_{\text{mod}}(M,M') +$$

$$+ \omega_{ret.type} S_{ret.type}(M,M') +$$

$$+ \omega_{\text{arg}} \frac{1}{n_{\text{arg}.matches}} \sum_{\substack{all \\ Matched \\ \text{arguments}}} S_{\text{arg}}(Arg_x, Arg_{bmx}) + \quad (5)$$

$$+ \omega_{no.\text{arg}} S_{no.\text{arg}}(M,M')$$

where the modifiers are visibility, static, synchronized, abstract and final. "retype" is the return type of a method, $n_{\text{arg}.matches}$ is the number of argument matches and "noarg" is the number of arguments.

Similarity metric 5 is also applied to constructors, but only the relevant terms apply.

In order to calculate the similarity between two given classes, the UMLSimilator takes every element within the hierarchical structure of the first class and compares it to the equivalent element(s) from the second class. It proceeds until every element is matched and the overall similarity is obtained, not allowing any element to be matched more than once. A full search algorithm is applied which ensures a good overall match. The algorithms in place will recursively rematch all elements until satisfactory matching is achieved.

This algorithm is applied on all levels of the element hierarchy. This algorithm does not guarantee optimum matching. This is due to the fact that by finding the best match for a particular element and then removing it from the pool of available elements, the overall similarity could actually be decreased. A slightly lower initial match could actually result in higher successive ones. However, the algorithm required to guarantee optimum matching would be too computationally demanding to be feasible.

3 Graph Representation and Matching of Software Designs

Most of the research done so far in the area of software reuse using CBR focuses on the definition of similarity between various design elements that are present in UML diagrams. However, a great deal of the knowledge associated with UML models is encoded in the links between these elements. Typical of this are the associations between classes in a UML class diagram and the message passing in interaction diagrams. In fact it can be argued that most of the practical reuse of design and code by software engineering practitioners is associated with design patterns that are related to patterns of interaction between objects. It is thus important to establish reliable similarity metrics to allow for case retrieval based on the structural similarity between design models.
Previous research [8],[9] has shown that competent case retrieval based on the structural similarity between cases can be achieved using graph matching techniques. The approach followed in this work views design models, such as class diagrams as graphs composed of nodes and arcs that link the elements. A full search graph matching algorithm has been adapted to be applied to UML class diagrams. Given the graph representations of two UML class diagrams, C and C´, the algorithm returns the Maximum Common Subgraph MCS(C,C´) present in both graphs.

The algorithm attempts to find the best matching elements in the two graphs based on the metrics presented in the previous sections of this paper. The similarity metric between arcs in the diagram is based on a simple classification of association types in terms of their multiplicity as follows:

a. one-to-one
b. one-to-many
c. many-to-one
d. many-to-many

3.1 Graph based similarity measures

In previous research, a simple similarity measure between classes was proposed [5]. However, this was not taking into account the structure of a class diagram in terms of associations between classes. However, a great deal of the information that can assist decisions for software design reuse is associated with the structure of a class diagram and particularly with the associations between classes. The measure proposed in this paper takes into account the structure of class diagrams.

The similarity between associations of the same type is defined as:

$$S(A_i, A_j) = \begin{cases} 1 & type(A_i) = type(A_j) \\ 0 & type(A_i) \neq type(A_j) \end{cases} \qquad (6)$$

The graph matching algorithm maps the maximum common connected sub-graph based on matches between individual elements of each graph, where a minimum threshold similarity value is satisfied.

The overall similarity between the two case graphs (G,G´) is then defined as:

$$S(G,G') = \frac{(\sum_{\substack{matches \\ C,C' \\ in \\ MCG}} S(C,C'))^2}{count(G).count(G')} \qquad (7)$$

where count(G) represents the number of nodes (classes) in graph G.

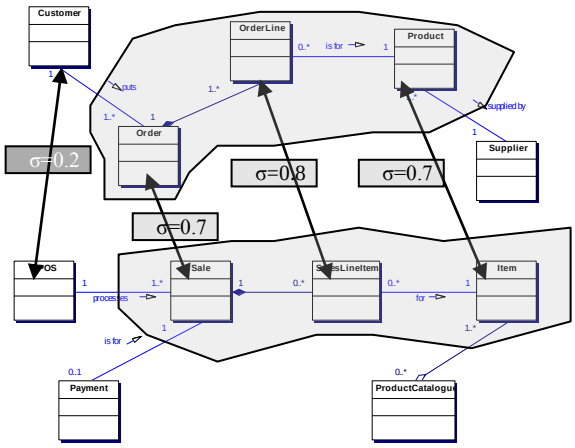

Figure 2. Graph similarity between UML class diagrams

Figure 2 shows an example application of the algorithm and measure. At a similarity threshold set to 0.6, three connected nodes in each graph have been picked up by the algorithm. A further match between the classes "Customer" and "EPOS" is rejected for having similarity value less than the threshold. Potential matches between "Customer" and "Payment" and between "Supplier" and "Product Catalogue" are not considered as the connecting associations are of different types.

In this example, the overall graph similarity between the two class diagrams is:

$(0.7+0.8+0.77)^2 /(5x6)=0.172$

3.2 Graph matching algorithm

The algorithm used in this research is based on the full recursive search of all elements in the graph representations of both target and source diagrams compared.
The algorithm attempts to match individual classes from the source and target diagram, where the similarity is greater than a predefined threshold. For each successful match, the algorithm attempts to match all respective matches of respective neighbours whose similarity measure is greater than the threshold. The algorithm finishes when no additional neighbour matches can be found. This algorithm has been extended from previous research [10] to follow links representing only similar types of associations as defined by equation 6 above.

Figure 3 outlines the algorithm used to identify the Maximum Common Sub-graph (MCS) between two graph representations of corresponding class diagrams.

```
for all possible matches (x,y)
   MCS:match(x,y)
endfor
```

```
match(x,y):
   calculate struct. similarity sim(x,y)
      if sim(x,y) > threshold
         find all matching neighbour(x_n,y_n)
         for each (x_n,y_n)
               MCS:Append(MCS,match(x_n,y_n))
               if MCS best so far -> retain
         endfor
      endif
      return MSC
```

Figure 3. The MCS graph matching algorithm

3.3 Experiments to evaluate the approach

In order to evaluate the similarity measures and algorithm proposed in the proposed approach, a set of limited experiments were conducted. A number of simple software design case studies (4) were each given to three students at a software engineering design course. The students submitted class diagrams for each of the case studies. Model solutions prepared by the tutor were also added to the case base. Out of the four case studies, two referred to very similar problems, one referring to an order system and one referring to a sales system. The other two case studies referred to a software simulation and a library book borrowing system.

The algorithm described above was used to identify the MCS between the graph representations of the class diagrams. The similarity measures described above were used to calculate the similarity between any two cases (class diagrams). The calculations were used to cluster the cases.

The exercise showed that this approach clustered correctly all diagrams referring to the same case study. It also showed that the approach can discriminate between class diagrams belonging to different case studies, with the exception of the two case studies that referred to the two systems that were isomorphic (order and sales systems).

Although this evaluation is by no means exhaustive, it indicates the suitability of the measures and algorithms when used within a CBR process.

4 The Importance of Behavioural Information in Software Designs

Within object-oriented development, software design is most commonly used to represent static structures. However, a design may not consist solely of a static model, but combine a number of different aspects of a software application or system using a range of appropriate diagrams, such as use case, state or interaction diagrams.
The strength of software lies in the fact that it is dynamic and has behaviour. Although it may consist of a static structure, it is the runtime behaviour and interaction between elements of the static structure that make it functional. Thus, the way in which a number of classes and objects are composed, in order to form larger structures, states a lot about a software design, yet much of its meaning can only be determined by the pattern of communication between these elements, such as the order in which messages are sent between them or the reactions triggered by certain messages, the overall complexity of the software and the assignment of responsibility.

Observing the interaction of elements within a software design may also be important, due to difference in object granularity. Different software designers have different styles and practices, which could result in applications or systems designed for the same purpose having a considerably different number of objects, due to the designer's personal perception of object granularity. However, it is possible that the messages sent between objects or possible states of an object show similarities of designs, which are not structurally similar.

A key issue is that a set of UML diagrams depicting a piece of software are related and complement each other. For instance, an interaction diagram should represent a particular use-case, but using elements and relationships from the class diagram. This makes is possible to treat a set of related diagrams as a unit representing a piece of software which can be compared to others.

With any UML diagram, measuring the similarity is a rather abstract undertaking as there are a number of elements that can be compared, especially if semantic similarity is not being taken into consideration. This means that the criteria that can be used in order to measure similarity between diagrams are limited to the composing elements and how they relate to one another. Nevertheless, it is possible to regard any of these diagrams as a graph of nodes and arcs, even if there are some additional constraints, such as containment or sequence.

5 Conclusion

Calculating the similarity between software designs for CBR is a complex undertaking. Software designs are elaborate structures, composed of a large number of elements arranged in many possible ways to achieve certain functionality. When comparing two designs, it is exactly that functionality which should determine the similarity between them. The problem lies in correctly identifying this functionality. Different approaches have been followed

in the past in order to reveal this functionality, such as semantic analysis, analogical reasoning, structural analysis or hybrid approaches.
In this paper we analysed a number of ways which can contribute to calculating the similarity of software designs, using class diagrams, focusing entirely on the structural analysis of software designs. This way, we are trying to determine the significance of the structure in determining the functionality of a design.

This research is work in progress. A tool has been developed in C#.NET, called UMLSimilator. This tool works with software designs stored in a determined XML format. It is used to calculate the similarity between classes, by applying different similarity metrics and a combination of weights. It allows element-specific similarity measuring to take place, as it allows the user to determine the element criteria for class comparisons.

Additionally, a graph matching algorithm and associated graph similarity metrics have been devised to capture structural information encoded in design models. This enables the analysis of class diagrams, not as a loose collection of classes, but as a connected arrangement of related classes.

Experiments with the UMLSimilator tool have been very promising and are encouraging the analysis of aspects, which we have not taken into consideration in our work up-to-date, such as data types, for example as defined in [12], and categorisations.

In our future research we intend to enhance the MCS similarity measures to discriminate better between different types of associations, including generalisation and composition. We are also looking into expanding the similarity measure on the use of behavioural information captured in other types of UML diagrams. Finally, a more sophisticated set of evaluation experiments are planned to establish the effectiveness and robustness of the approach.

References

[1] Kolodner J: Case-based Reasoning, Morgan Kaufmann, 1993
[2] Fernández-Chamizo C, González-Calero P, Gómez-Albarrán M, Hernández-Yánes L: Supporting Object Reuse Through Case-Based Reasoning, in Advances in Case-Based Reasoning pages 135-149. Springer-Verlag, 1996
[3] Gomes P, Gandola P, Cordeiro J: Helping Software Engineers Reusing UML Class Diagrams, in Proceedings of the 7th International Conference on Base-Based Reasoning (ICCBR'07) pp. 449-462, Springer, 2007
[4] Bjornestad S: Analogical Reasoning for Reuse of Object-Oriented Specifications, in Proceedings of the 5th International Conference on Case-Based Reasoning (ICCBR'03)
[5] Wolf M, Petridis M: Similarity Metrics for Reuse of Software Design using CBR, in Proceedings of the 8th UK Workshop on Case Based Reasoning, Cambridge
[6] Bergmann R, Stahl A: Similarity Measures for Object-Oriented Case Representations, in 4th European Conference on Case-Based Reasoning, Springer, 1998
[7] Tessem B, Whitehurst A, Powell C: Retrieval of Java Classes for Case-Based Reuse, in B.Smyth & P.Cunningham (eds.), Procs. of the Fourth European Workshop on Case-Based Reasoning, LNAI 1488, pp.148-159, Springer, 1998
[8] Knight B, Petridis M, Mejasson P, Norman P A: Intelligent design assistant (IDA): a Case Based Reasoning System for Materials Design, in Journal of Materials and Design 22 pp 163-170, 2001
[9] Woon F, Knight B, Petridis M: A Case Based System to assist in the design process in the manufacture of furniture products, in Proceedings 6th UK Workshop on Case Based Reasoning, Cambridge 2001
[10] Petridis, M., Saeed, S., Knight, B.: Refining Similarity Measures for Effective Reuse of Metal Casting Design Knowledge, Workshop Proceedings of 7th International Conference on Case-Based Reasoning, ICCBR-07, Belfast, Northern Ireland, 13-16 August 2007, pp. 16-32
[11] Zaremski A, Wing J: Specification Matching of Software Components, in ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 4, pp 333 – 369, 1997
[12] Meditskos G, Bassiliades N: Object-Oriented Similarity Measures for Semantic Web Service Matchmaking, in Proceedings 5th IEEE European Conference on Web Services, 2007
[13] Gorton I, Zhu L: Tool support for just-in-time architecture reconstruction and evaluation: an experience report, in Proceedings 27th International Conference on Software Engineering, pp 514 - 523, 2005