Independent degree project - second cycle

Computer Engineering

Automatic Source Code Classification
Classifying Source Code for a Case-Based Reasoning System

Markus Nordström
Abstract

This work has investigated the possibility of classifying Java source code into cases for a case-based reasoning system. A Case-Based Reasoning system is a problem solving method in Artificial Intelligence that uses knowledge of previously solved problems to solve new problems. A case in case-based reasoning consists of two parts: the problem part and solution part. The problem part describes a problem that needs to be solved and the solution part describes how this problem was solved. In this work, the problem is described as a Java source file using words that describes the content in the source file and the solution is a classification of the source file along with the source code. To classify Java source code, a classification system was developed. It consists of four analyzers: type filter, documentation analyzer, syntactic analyzer and semantic analyzer. The type filter determines if a Java source file contains a class or interface. The documentation analyzer determines the level of documentation in a source file to see the usefulness of a file. The syntactic analyzer extracts statistics from the source code to be used for similarity, and the semantic analyzer extracts semantics from the source code. The finished classification system is formed as a kd-tree, where the leaf nodes contain the classified source files i.e. the cases. Furthermore, a vocabulary was developed to contain the domain knowledge about the Java language. The resulting kd-tree was found to be imbalanced when tested, as the majority of source files analyzed were placed in the left-most leaf nodes. The conclusion from this was that using documentation as a part of the classification made the tree imbalanced and thus another way has to be found. This is due to the fact that source code is not documented to such an extent that it would be useful for this purpose.

Keywords: Artificial Intelligence, Case-Based Reasoning, CBR, Vocabulary, Classification, Similarity measure, Distance measure, Java, C++.
Table of Contents

Abstract ................................................................. 3
Terminology ............................................................ 6

1 Introduction .............................................................................................................. 7
1.1 Background and problem motivation ................................................................. 7
1.2 Overall aim ........................................................................................................... 8
1.3 Scope .................................................................................................................... 8
1.4 Concrete and verifiable goals ............................................................................. 9
1.5 Outline ............................................................................................................... 10
1.6 Contributions .................................................................................................... 10

2 Case-Based Reasoning ................................................................................... 11
2.1 Overview ............................................................................................................ 11
2.1.1 Similarity function ....................................................................................... 12
2.1.2 The Vocabulary ......................................................................................... 16
2.1.3 Ontology ..................................................................................................... 17
2.2 The Reasoning Process .................................................................................... 18
2.2.1 Retrieve ....................................................................................................... 18
2.2.2 Reuse .......................................................................................................... 21
2.2.3 Revise ......................................................................................................... 25
2.2.4 Retain .......................................................................................................... 28
2.3 Case Representation ....................................................................................... 29
2.3.1 Feature vector representation ..................................................................... 30
2.3.2 Frame-based representation ....................................................................... 30
2.3.3 Object-oriented representation .................................................................. 30
2.3.4 Textual representation ............................................................................... 30
2.3.5 Hierarchical case representation ................................................................ 31
2.3.6 Generalized case representation ................................................................ 31

3 Methodology ...................................................................................................... 32
3.1 Theoretical model ............................................................................................ 32
3.2 Practical model ................................................................................................ 32
3.3 Evaluation ......................................................................................................... 33

4 Design .................................................................................................................. 34
4.1 The case ............................................................................................................. 34
4.1.1 Problem ....................................................................................................... 34
4.1.2 Solution ....................................................................................................... 34
4.2 File classification system ................................................................................ 39
4.2.1 Type filter .................................................................................................. 42
4.2.2 Documentation analyzer ............................................................................. 42
4.2.3 Syntactic analyzer ...................................................................................... 43
4.2.4 Semantic analyzer ...................................................................................... 43
4.3 Vocabulary ....................................................................................................... 44
4.4 Distance Measure ............................................................................................ 45

5 Implementation ................................................................................................ 46
Automatic source code classification – Classifying source code for a Case-Based Reasoning system
Markus Nordström 2015-06-11

5.1 The case ........................................................................................................ 46
5.1.1 Problem object ....................................................................................... 46
5.1.2 Solution object ....................................................................................... 47
5.2 File classification system ........................................................................ 48
5.2.1 Type filter .............................................................................................. 49
5.2.2 Documentation analyzer ....................................................................... 50
5.2.3 Syntactic analyzer ................................................................................ 51
5.2.4 Semantic analyzer ................................................................................ 51
5.3 Vocabulary ................................................................................................. 52
5.4 Semantic analyzer ...................................................................................... 54

6 Results ............................................................................................................. 56
6.1 Test results type filter ............................................................................... 56
6.2 Test results documentation analyzer .......................................................... 56
6.3 Test results syntactic analyzer ................................................................... 57
6.4 Test results semantic analyzer ................................................................... 58
6.5 Test results semantic tree ......................................................................... 59
6.6 Example of a CBR system using the classification system ..................... 60

7 Evaluation ....................................................................................................... 64
7.1 Comparison with ACAI system ................................................................. 64
7.2 Evaluation of the complexity ..................................................................... 64
7.3 Evaluation of the result ............................................................................... 65
7.4 Future work ................................................................................................ 66
References ........................................................................................................ 68
Terminology

Acronyms/Abbreviations

A.I  Artificial Intelligence
CBR  Case-Based Reasoning
ML   Machine Learning
GP   Genetic Programming
CC   Comments versus code ratio
HF   Header versus function ratio
DOC  Documentation
1 Introduction

In machine learning (ML) research, it is becoming more popular to develop algorithms and systems that generate source code based on an abstract user input. Most of the current research is on genetic programming (GP) which, in short, evolves the source code according to a fitness function defined by the creator [1]. However, there is little research being done on the subject regarding case-based reasoning (CBR) which, rather than evolving, uses solutions from previous similar problems to solve new problems. This method of Artificial Intelligence (A.I) thus requires a knowledge base where problems and solutions can be stored, together called cases, so that they can be accessed at a later time [1]. In the context of source code generation, the knowledge base will need to classify source code in such a way that cases can be constructed for use by a CBR system. The construction of the knowledge base and its classification system will be the main challenge of this thesis.

1.1 Background and problem motivation

The concept of transforming code using A.I has been around since the 1950s when the first evolutionary algorithms appeared in evolutionary simulation systems. Since then, evolutionary algorithms have been implemented in many fields of research, such as biology, computation optimization and mathematics. Evolutionary algorithms are a part of the genetic programming paradigm which tries to implement evolution theory on program structures to improve or optimize them. [1]

The concept of CBR was not introduced until the 1980s when Roger Schank developed his model of dynamic memory which paved the way for the earliest CBR systems [2]. Among the first systems to appear are Janet Kolodner’s CYRUS [3] and Michael Lebowitz’s IPP [4]. CBR has been used in several fields of study, most notably in health science and computer science. One example of a successful CBR system is the Compaq SMART system used in help desk [5].

There is little research in CBR on source code generation and manipulation. One such example is the ACAI system developed by Danilchenko and Fox, which uses a combination of CBR, routine design and template-based programming to generate Java code based on abstract user input. However, this solution requires a library of generic code and a library of abstract programs to be able to generate code that satisfies the user’s goal. [6]

The CBR concept aims to mimic human problem solving by taking solutions to similar problems from the past and applying it to solve a new problem. To store this kind of information, the CBR system has a knowledge base which stores cases where each case has a problem and its solution. When a new problem is
encountered, the CBR system searches among the cases in the knowledge base to find a similar case to match the new problem and uses the solution associated to try and solve it. Thus the structure of this knowledge base is of utmost importance so that information may be retrieved and analyzed in a useful way to the CBR system. Although the concept is the same in all CBR system, the knowledge base is custom-made for every system because of the nature of the problem space it handles. This thesis will consider source code generation and manipulation as its main problem space. [7]

This thesis focuses on the structure of the knowledge base and its classification system of the source code. The reason for this is the complexity of the task of building a full CBR system and the time frame given for this thesis.

1.2 Overall aim

The overall aim of this thesis is to create the structure of a CBR knowledge base and classification system. This system will have source code generation and manipulation as problem space. A successful implementation would result in a knowledge base structure which would lay the foundation for future research into a CBR agent that would use this structure to approximate source code based on abstract user input. This in turn would save time for IT companies and developers since much source code could be approximated instead of being manually written. A successful implementation will hopefully prove a valuable contribution to the research field.

1.3 Scope

The theoretical part of this thesis regarding CBR and its concept is limited to giving the reader an introduction to the subject and is does not intend to cover all aspects of the CBR research field. Readers who would like to know more about the subject may be interested in reading [7]. The purpose of the theory part is to present an overview of the concept that is vital to understand in order to follow the reasoning in this thesis.

The result of this thesis is limited to developing a knowledge base for a CBR system and does not include the reasoning logic behind CBR. The reason for this is the time frame of this thesis is too narrow to implement the entire CBR system. The result will be a theoretical model and an implementation of the knowledge base and the classification system required to classify source code files for use in a CBR system.

The programming language that the knowledge base will handle is limited to Java EE source code only. This is motivated by the fact it is a high level language. The nature of high level languages makes them easier to analyze which will reduce the time spent on semantic and syntactic analysis of the source code. The concept of high level programming languages will not be explained in this thesis. Readers who do not posses the required knowledge is recommended to read [8] for an introduction to programming language theory.
The source files to be analyzed and categorized in the knowledge base will be limited to a certain degree of documentation and semantic. The reason for this is that the documentation provides information that will be a vital part of the process of analysis when categorizing source files. Without a reasonable degree of documentation the files will not provide enough metadata to make the categorization process as fundamental as possible. Thus, files with little or no documentation will not be considered in this thesis.

1.4 Concrete and verifiable goals

The challenge of the problem domain is to categorize and make cases of the source code files for the knowledge base. The categorization will need to be structured in such a way that it can provide usable information and make cases that are comprehensible for the CBR reasoner. The source code categories will need to be dependent on the type of content the source files provided contain. Since there is a great variety of content in source files it will be a challenge to find common factors among them for the categorization process.

Another challenge is to build the case representation since it is a description of how to store cases in the knowledge base. If implemented in the wrong way, the computational complexity will be too heavy for a basic computer to solve and thus the knowledge base will not produce any result.

The goal of the theoretical work of this thesis will be a theoretical study of the following research areas:

- Fundamental theory of case-based reasoning to provide basic background information and present concepts on the subject regarding the reasoning cycle.

- Methodology of case-based reasoning to give a brief overview of different approaches to the reasoning process.

- Case representation methods to show different ways to represent cases in a knowledge base.

The goal of the practical work of this thesis will be the following:

- Constructing a theoretical model of a CBR knowledge base and classification system from the background research conducted in the theory part of this work.

- Implementing a CBR knowledge base from the produced theoretical model.

- Analyze and evaluate the result in terms of goals and methodology.
1.5 Outline

This report includes a theoretical part which is presented in Chapter 2. It gives a brief overview of the case-base reasoning methodology and process. Here, an overview of case-based reasoning is presented along with the reasoning process and a few examples of how cases are represented.

Chapter 3 presents the methodology used in this work, where the methodology of the theoretical part of this work is presented along with the practical part. The evaluation methodology is also presented.

Chapter 4 presents the system design created in this work. It explains the design of the case including problem and solution parts. It also explain the design of all the analyzers in the classification system. The vocabulary is also presented where each part of the knowledge containers are explained. Lastly, the distance measure is presented.

Chapter 5 outlines the implementation of the classification system. Each part presented in Chapter 4 is presented here but with implementation as focus rather than design.

Chapter 6 presents the test results of the classification system along with an example of a case-based reasoning system that could use this classification system.

Chapter 7 presents the evaluation carried out after the test results have been obtained. It also proposes future work.

1.6 Contributions

The Java source files that was used to test the classification system in this work was provided by Dewire.
2 Case-Based Reasoning

In this chapter the background theories required to follow the reasoning in this work will be presented.

2.1 Overview

CBR uses experience from a past problem to solve a new problem. This can be illustrated by an example: Imagine a car mechanic who just received a new car to fix. The only thing his customer told him is that the car will not start. At this point the mechanic does not know exactly what is wrong with the car. However, he knows the car brand and that this brand often has a battery problem. So he replaces the battery with a new one and the car works again. Here the mechanic used past knowledge of the car brand to solve a new problem. This is the principle behind CBR. It tries to mimic the human problem solving process to try to solve new problems. There is no limitation to what type of problem CBR can solve; it is used for anything from exact sciences to mundane tasks. However, there are problem types which are more suited for CBR than others. [7]

As the name suggests, case-based reasoning is centered around cases. A case is a recorded previous experience about a problem solving episode, but the exact content of a case varies between implementations. Usually it is the problem domain that dictates the contents of a case and the reason for this is that each type of problem can be described in different ways. A case can be represented in different ways - as section 2.3 Case representation will show - but the basic representation is on the form (problem, solution) pairs. The cases are stored in a case base also called the representation space, which consists of a problem space and solution space. This means, traditionally, a case is a bijective mapping between two points in the problem space and the solution space. However, since one problem can have many solutions it is not sufficient to have a bijective mapping. In CBR it is also of interest to save non-working solutions since the system needs to learn what solutions to avoid. It is thus necessary to have one problem mapped to several solutions for learning purposes. [7]

Problems are central to CBR since it is a problem solving methodology. There are two kinds of problems in CBR: the problem stored in the case base, this is referred to as “problem”, and the query problem, which is referred to as “new problem” posed by the user to be solved by the CBR system. It is sometimes difficult to formulate a problem because it often refers to its context. This means that each problem needs a different solution. Consider the following question: How much does the house cost? There are two possible answers to this question:

- Too expensive for me
Each of these answers depend on the problem context which means that somehow the context has to be known in order to find the correct solution. Defining the context of a problem can be beneficial for finding a suitable solution but it also introduces metadata for the case, which means each case will require more space in the case base. Thus finding the right problem formulation is important for more than one reason. [7]

Solutions can also be represented in different ways, depending on the implementation and intended use of the CBR system. In a simple case the solution contains only a name or simple data such as objects with a value attached to them. It can be expected temperature or simple sensor values or predefined values for projects. There are also complex solutions which contains more complex structures such as an object-oriented description of a technical object. The most complex of all solutions are those made for planning and those in textual and image form. The reason for this complexity is that unexpected events could be involved, which means the solution fails. For example: flying from one destination to another, there could be any number of interruptions caused by several unforeseen factors, such as extreme weather. Thus this kind of solution requires an extension that describes what happened so that future adaption of the solution is possible. [7]

In CBR, cases are stored in case bases, but there are different ways to store them, this is called case organization. There are three different types of case organization: flat, structured and unstructured. Flat structure is the simplest way to organize cases and it is used when few cases are needed in the system. The cases have no relation to each other and each case contains only basic data. Structured organization most suitable when there is a very large number of cases. It is not necessarily a hierarchical structure even if it is the most common way. Object-oriented structure is a good example of structured organization. Unstructured organization is when cases are represented as texts or images. [7]

### 2.1.1 Similarity function

To obtain a solution from a CBR system, a retrieval function is required. The purpose of the retrieval function is to obtain the case most similar to the query problem as possible. This is done by comparing the query problem to the problems in the case base and finding the most similar one. The ideal scenario is to find an equivalent problem and use that solution for the query problem. This is rarely the case, so instead the retrieval function tries to find the most similar problem. This is called similarity assessment and is defined by a similarity function. [9]

The similarity function is unique for each implementation since it is dependent on how the cases are represented but one example is the attribute-value pair case. The similarity function in this type of case would be to compare similarity
between each attribute and then compare the relevance of each attribute. Each attribute would require its own similarity function which returns 1 if two attributes are equal and 0 if they have nothing in common. Instead of a real number between [0, 1] the similarity function can also return a boolean value of true or false, this is all dependent on the nature of the attribute. The relevance of an attribute is also denoted by a number where a large number is of more importance than a lower number, because some attributes are of more importance than others. Similarities between attributes are called local similarities and relevance between attributes are called weights. In [7] they define similarity as follows: [9]

**Definition 1:** Let CB be a set of objects and p be an object; then some s of CB is a nearest neighbour to p if there is no object in the CB that has a higher similarity to p than s.

The similarity between two cases is called global similarity and is defined as a sum of all local similarities between the cases. Consider an arbitrary domain of\
\[ x = \{ x_1, x_2, x_3, ..., x_n \} \] and \\
\[ y = \{ y_1, y_2, y_3, ..., y_n \} \] attribute-value pairs of Attributes: Attribute\textsubscript{1}, Attribute\textsubscript{2}, Attribute\textsubscript{3},..., Attribute\textsubscript{n}. Then the local similarity is defined as follows: [7]

\[ \text{sim}\textsubscript{i}(x\textsubscript{i}, y\textsubscript{i}), \text{sim}\textsubscript{2}(x\textsubscript{2}, y\textsubscript{2}), \text{sim}\textsubscript{3}(x\textsubscript{3}, y\textsubscript{3}),..., \text{sim}\textsubscript{n}(x\textsubscript{n}, y\textsubscript{n}); x\textsubscript{i}, y\textsubscript{i} \in \text{dom}(\text{Attribute}) \]

And global similarity is defined as: [7]

\[
\sum_{i=1}^{n} (\omega_i \ast \text{similarity}(x_i, y_i)) | 1 \leq i \leq n \] Where \( \omega_i \) is a calculated weight , \( n \in \mathbb{N} \)

The weight \( \omega_i \) is of global measure since the relevance of attributes are different between cases and this makes it more difficult to calculate. [7]

Similarity can be viewed as a relation because it is possible to say that two specific objects are more similar than two other objects. There are three relational models that describes this. [7]

- Binary similarity predicate
  - \( \text{SIM}(x, y) \Leftrightarrow \text{“} \text{x and y are similar} \text{“} \)

- Binary dissimilarity predicate
  - \( \text{DISSIM}(x, y) \Leftrightarrow \text{“} \text{x and y are dissimilar} \text{“} \)

- Similarity as a partial order relation
  - \( \text{R}(x, y, z) \Leftrightarrow \text{“} \text{x is at least as similar to y as x to z} \text{“} \)
Automatic source code classification – Classifying source code for a Case-Based Reasoning system
Markus Nordström

The first two predicates are binary so they can only return one of the two answers true or false and does not indicate the degree of similarity between the two objects. Moreover, they are the negation of each other. However, the third predicate is much more useful since it implies different degrees of similarity. One generalization of it could be: [7]

- \( S(x, y, u, v) \iff \text{"x is at least as similar to y as u is to v"} \)

Thus it is possible to define the partial order \( R \) such that:

- \( R(x, y, z) \iff S(x, y, x, z) \)

This means that it creates a partial ordering with regards to an arbitrary object \( x \). This kind of ordering shows objects are more or less close to object \( x \). In addition, this kind of relation needs to have a few properties associated with it. It is thus necessary to define a few axioms that must hold at all times. Consider the set of objects \( x, y, z, u, v, s \) and \( t \). [7]

- Reflexivity
  - S(\( x, x, u, v \))
  - All objects are at least as similar to themselves as two other objects could be to each other.

- Symmetry
  - S(\( x, y, u, v \)) \iff S(\( y, x, u, v \)) \iff S(\( x, y, v, u \))
  - \( x \) is as similar to \( y \) as \( y \) is to \( x \). \( u \) is as similar to \( v \) as \( v \) is to \( u \).

- Transitivity
  - S(\( x, y, u, v \)) \land S(\( u, v, s, t \)) \Rightarrow S(\( x, y, s, t \))
  - \( x \) is at least as similar to \( y \) as \( u \) is to \( v \) AND \( u \) is at least as similar to \( v \) as \( s \) is to \( t \) then \( x \) is at least as similar to \( y \) as \( s \) is to \( t \).

For the remainder of the report another notation for similarity relations will be used. They are defined below. [7]

- \( y \geq z \iff R(x, y, z) \)
  - \( x \) is at least as similar to \( x \) as to \( z \)

- \( y > z \iff(y > z) \land \neg(z \geq y) \)
Similarity can also be viewed as mathematical functions. This is a numerical way to express similarity between two objects and it is more detailed than the relational approach but it is also more difficult to formulate. The similarity function is defined as: [7]

**Definition 4:** A similarity measure for a problem space $P$ is a function

$sim: P \times P \rightarrow [0, 1]$. 

This means that for a similarity of 1, two objects are similar and are of highest usefulness. In the same way, a similarity of 0 means that two objects are not similar and of lowest usefulness. To find the nearest neighbour to an object, the following definition is used: [7]

**Definition 5:** For a given problem $P$, a nearest neighbour is a problem $P'$ that has maximal similarity among the problems in $P$.

Just like the relational similarity, the numerical similarity method also requires basic properties to hold. These axioms are defined as follows: [7]

- Reflexivity
  - $sim(x, x) = 1$

- Constancy of self-similarity
  - $sim(x, x) = sim(y, y)$

- Symmetry
  - $sim(x, y) = sim(y, x)$

- Transitivity
  - $sim(x, y) \geq \sup(z \mid \{inf(sim(x, z), sim(z, x))\})$

Another type of numerical similarity measure is distance. The difference between numerical similarity and distance similarity is that distance does not have to be bounded. Distances are defined as: [7]

**Definition 6:** A distance measure on a set $U$ is a real-valued function

$d: U \times U \rightarrow [0, \infty)$
In contrast to numerical similarity, no upper bound in the distance implies no lower bound in the similarity measure. If an object is far away from another object, they are not similar. [7]

\[ d: U \times U \rightarrow [0, \infty) \iff \text{sim: } U \times U \rightarrow (-\infty, 0] \]

The distance measure also require certain axioms to hold, these are defined below:

- Reflexivity
  - For all \( x \): \( d(x, x) = 0 \)
- Strong reflexivity
  - For all \( x, y \): \( d(x, y) = 0 \iff x = y \)
- Symmetry
  - For all \( x, y \): \( d(x, y) = d(y, x) \)
- Triangle Inequality
  - For all \( x, y, z \): \( d(x, y) + d(y, z) \geq d(x, z) \)

It has some axioms in common with the other types of similarity except Triangle Inequality and strong reflexivity. As such, a distance function is pseudo-metric if reflexivity, symmetry and triangle inequality holds. For the function to be metric, the strong reflexivity axiom must hold as well. [7]

One method of distance is the cosine distance measure, also called cosine similarity. It is not a metric measure because the triangle inequality does not hold. Consider two vectors of attributes \( A \) and \( B \) then the cosine distance is defined as follows: [10]

\[
\text{Similarity} = \cos(\theta) = \frac{(A \cdot B)}{\|A\| \|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}
\]

This means that the result of this method ranges from -1 to 1 where -1 depicts dissimilarity, 1 depicts similarity and 0 depicts orthogonality. [10]

2.1.2 The Vocabulary

For a CBR system to be able to perform reasoning tasks it requires a so-called vocabulary. The vocabulary defines all terms and concepts needed to under-
stand the problem domain. The reason for this is simple, no one can discuss a subject they have never heard of before. Therefore the knowledge in the vocabulary is extensive and may include many aspects of the same object as there are nearly an infinite number of terms defining that object. Thus it is important to choose the information contained in the vocabulary carefully as storage space is not unlimited. [7]

There may also be several sub-containers in the vocabulary to describe certain types of objects. Consider a company; the vocabulary would contain information on how to calculate economic factors but it would also contain knowledge about employees such as names, salary, position in the company etc. There could also be knowledge about products in the supermarket. The important thing in the vocabulary is that all knowledge required for reasoning is present and well-defined. [7]

### 2.1.3 Ontology

An ontology is an entity represented in a declarative and structural way which means it describes what an entity is and how it relates to other entities in a specified scope. Relations between entities are hierarchically organized. Ontologies include large amounts of information about entities such as types, constraints, cardinalities, semantic entailments, functions and axioms. Relationships can be formally defined in various ways such as inclusion, spatial, cause-effect, rationale, means-end, sequence and attribution. Knowledge contained in an ontology usually comes from different sources with different semantics for describing objects. Thus, the idea behind an ontology is to collect, combine and restructure knowledge for the CBR system. The major types of ontology are: [7]

- **Formal**
  - It is machine understandable.

- **Explicit specification**
  - It contains explicitly defined concepts, properties, relations, functions, constraints and axioms.

- **Shared**
  - It contains consensual knowledge of a community

- **Conceptualization**
  - It is an abstract model of a phenomenon in the world.
Ontologies were invented to provide commonsense reasoning to AI and this means that they require their own types and purposes. The most general ontologies, the top-level ontologies, are called base or abstract ontologies. They conceptualize the most basic of entities such as set, mass, relation, gravity etc. Using this as a base, several other types of ontologies could be derived, such as domain, application and linguistic ontologies, each with its own extended specialization of the base ontology. An ontology is structured as a directed graph which is often just a tree. [7]

In CBR, ontologies can be used to provide its commonsense reasoning to support processes such similarity assessments. This is the role of the ontology implemented in the vocabulary since that is where all definitions and domain knowledge can be found. The ontology can in this case be implemented as an object-oriented structure which will coincide well with the case base since there is a way to represent cases by using object-orientation. This means that the ontology will provide the taxonomic structure for the similarity measure. [7]

2.2 The Reasoning Process

The reasoning process in CBR is what produces a new solution to a query problem based on stored knowledge in the case base. The process is divided into four steps: retrieval, reuse, revise and retain. Each of these steps retrieves, manipulates and learns while solving a query problem. The steps will be explained in this chapter.

2.2.1 Retrieve

In CBR, retrieval is not a simple database retrieval. The retrieval function finds the most similar case matching the query problem. This is done by using the similarity function, which makes the retrieval process unique to every implementation as it is highly connected to the case representation. There are two major types of retrieval function: [11]

- Retrieval algorithms: Algorithms that require a fixed database to perform search on.
- Agent-based retrieval algorithms: Algorithms that can perform searches in multiple databases that do not have to be fixed, such as a distributed system or a central system.

The retrieval function is not a linear search for a desired object instead it is a query object to which it tries to find the nearest neighbour in the case base. There are two problems associated with this. Firstly, the intended object may be in the database but not formulated in a satisfying way, which may leave it overlooked. Secondly, the query object may not describe the intended object well enough, which also means it will be overlooked.
Traditionally in searching, a keyword is used to locate a desired object, but the problem with this method could be that either there will be no answer, called silence, or too many answers, called noise. This means all cases need to be indexed because in CBR there must always be an answer, and noise can be reduced as the number of retrieved nearest neighbours can be controlled. [11]

To describe the retrieval process consider the following: Let CB be an arbitrary case base defined as $\text{CB} = \{c_1, c_2, c_3, ..., c_n\}$. Let sim be a similarity function defined on the CB. Now let $q$ be the query problem. Then the retrieval function will retrieve the following from the case base: [7]

1. the object $c_i$ that is most similar to $q$
2. or the $k$ most similar objects $\{c_{i_1}, c_{i_2}, c_{i_3}, ..., c_{i_k}\}$ to $q$
3. or all objects $c_i$ that have at least $\text{sim}_{\text{min}}$ with regards to $q$

These three rules imply that there is no empty answer from the case base. There are also three important properties associated with the retrieval function. [12]

- Correctness
- Completeness
- Efficiency

Given the similarity function $\text{sim}$ and a query problem $q$, the retrieval function tries to retrieve the $k$-nearest neighbours. Suppose it finds two answers, $a$ and $b$, then it has to decide which of them are closest to $q$. This means it needs to find that either ($a \geq_b b$) or ($b \geq_a a$) is satisfied. However, this implies its own ordering of the object based on how cases are retrieved. This ordering can differ from the ordering defined in the similarity function which means that the retrieval function imposes its own similarity measure $\text{sim}_{\text{ret}}$ and this means there could be a conflict between $\text{sim}$ and $\text{sim}_{\text{ret}}$. Of course, there can be no difference between the two, so definition 7 below will see to that there is no conflict between them. [7]

**Definition 7:** Let $\text{ret}$ be an arbitrary retrieval function.

1. The function $\text{ret}$ is correct with respect to $\text{sim}$ if for all $a, b, q$: ($a \geq_{\text{ret}_q} b$) $\Rightarrow (a \geq_{\text{sim}_q} b)$.
2. The function $\text{ret}$ is complete with respect to $\text{sim}$ if for all $a, b, q$: ($a \geq_{\text{sim}_q} b$) $\Rightarrow (a \geq_{\text{ret}_q} b)$.
3. The function $\text{ret}$ is complete if it retrieves the nearest neighbours of the similarity measure.
The only problem that remains is that completeness and efficiency is competing with each other, meaning that the more completeness gained the less efficiency and vice versa. Efficiency is closely related to case representation, the structure of the case base and the similarity measure. It is also related to the way objects are described and how accurate this description is. [13]

There are two kinds of errors associated with retrieval, α- and β-errors and (α+β)-errors. They are represented in Illustration 1. Rectangles represent retrieved set from the case base and circles represents the true nearest neighbours. [7]

Consider a case \( c \) and a query \( q \), then an α-error is when the retrieved set is smaller than the true nearest neighbours. Thus there is a probability that the most similar \( c \) in regards to \( q \) in not selected. β-error is when the retrieve function is not accurate enough and selects a too large subset, which means irrelevant objects are retrieved but never used. Performance-wise this is undesirable. [7]

Consider a case \( c \) and a query \( q \), then an α-error is when the retrieved set is smaller than the true nearest neighbours. Thus there is a probability that the most similar \( c \) in regards to \( q \) in not selected. β-error is when the retrieve function is not accurate enough and selects a too large subset which means irrelevant objects are retrieved but never used. Performance-wise this is undesirable. [7]

There are many retrieval methods but the one that is most used is kd-trees because of its indexing nature and the way it limits the search space. The name kd-tree stands for k-dimensional tree and they are used as an index structure in CBR. Kd-trees are commonly used in database retrieval, the only difference is that in CBR it is required to backtrack a tree if the desired object is not found. The purpose of representing cases in this way is to keep them homogenous but this requires that the attributes of the objects are ordered and are compatible with the ordering in the similarity measure. Consider the attributes \( x_i < i x_i' \) then the following statement must hold: [7]

\[
\text{sim}( (x_1, x_2, x_3, ..., x_d), (x_1, x_2, x_3, ..., x_i, ..., x_d) ) \geq \text{sim}( (x_1, x_2, x_3, ..., x_d), (x_1, x_2, x_3, ..., x_i, x_d) )
\]

The actual tree is constructed as part of the preprocessing step. Suppose there
are \( k \) ordered domains \( T_1, T_2, T_3, \ldots, T_k \) for the attributes \( A_1, A_2, A_3, \ldots, A_k \). Furthermore let \( CB \) be a case base such that \( CB \subseteq T_1 \times T_2 \times T_3 \times \cdots \times T_k \). There is also a user-defined parameter \( b \) for the size of the leaf nodes, called buckets. The reason for this is that the buckets will be sets which will be sequentially searched and the parameter \( b \) will dictate the computational complexity for this. [7]

**Definition 8:** A kd-tree \( \text{Tree}(CB) \) for the case base \( CB \) is a recursively defined binary tree:

- if \( |CB| \leq b \) then \( \text{Tree}(CB) \) is a leaf node
- if \( |CB| > b \) then \( \text{Tree}(CB) \) is a tree where the root is annotated with some attribute \( A_i \) and some value \( v_i \in T_i \).

Beneath the root there are two successor trees \( \text{Tree}(CB \leq) \) and \( \text{Tree}(CB >) \) defined as: [7]

- Let \( CB \leq := \{ x \in CB \mid x \leq v_i \} \)
- Let \( CB > := \{ x \in CB \mid x > v_i \} \)

This means that a kd-tree partitions a case base iteratively based on attribute values. The buckets are not partitioned as they represent the linear search space. [14]

The advantage of using kd-trees is that the effort to create it varies depending on how many object that have to be found. It is also possible to extend the model, should new objects be added. This way of representing objects also makes it possible to store them in a regular database. The disadvantages of this kind of model is that it requires great effort to build the tree, and the tree itself is inefficient when there is a need for backtracking. It is also applicable on ordered domains only and cannot handle unknown or missing values. [14]

### 2.2.2 Reuse

When encountering a new problem it is rare that there is an exactly corresponding solved problem in the case base. Usually there is a similar problem and solution, and this solution will work for the new problem if modified. This aspect is the major focus in the reuse part of the CBR cycle. To reuse a previous solution a change to the solution is required, and in CBR this is called adaptation. The actual changes in the solution require actions and these actions cause changes to the solution. The actions are described by rules and they dictate under what conditions and circumstances the adaptations are possible. When adapting solutions there are two approaches that could be useful in different situations. The first approach is to take a previous solution and adapt it to fit the new problem as mentioned before. The second approach is to extract the strategy used to derive the solution and create a new adapted solution to fit the new problem. Both these approaches are performed by using actions and these ac-
Actions can be formally defined as a set of preconditions that have to be met in order to perform the action on a solution. Here $\phi_i$ is a precondition. [7]

$$\phi_1 \land \phi_2 \land \phi_3 \land \ldots \land \phi_n \Rightarrow \text{Action}$$

Each precondition consists of a collection of conditions that have to be met in order for the precondition to be true. This means that the conditions are hard constraints, and they are variables that are local to the rule they create. These variables may very well be shared across cases since they might have them in common. When they are applied to a case they are instantiated with the values they have in the current context of the case considered. [7]

All cases in a case base are described using the same form of representation. This means that the actions used to perform adaptation is structured based on the representation form. Actions have operators that modifies the state of a case. [7]

Operator: $\text{states} \rightarrow \text{states}$

A state describes the current context of a case at a given time. It does, however, not describe the whole case but rather a part of it that is deemed to be relevant. There are three major types of actions: Add/create, delete and modify. These actions affect variables in the case or whole parts of a case. The operators can be arranged in a sequence to create a chain of operators. [7]

$$\text{state}_1 \rightarrow \text{state}_2 \rightarrow \text{state}_3 \rightarrow \ldots \rightarrow \text{state}_n$$

This kind of event chain can create a composite action consisting of several different actions which will alter a case in a specific way.

There are two kinds of rules: completion rules and adaptation rules. Completion rules adapt the case description in the query because the query can be insufficiently described while adaptation rules adapt the retrieved solution. [15]

Completion rules arise because the query may not be describing the problem correctly or sufficiently. Such a query is called incomplete and the completion rules intend to complete them, see Illustration 2. In incomplete queries some variables may not be instantiated or have incomprehensible values and this is highly undesirable since the variables may be important when searching for solutions. The problem with missing values is that there is no way to compute what the value should have been. However, there are ways to approach this problem: use a default value, compute an average or use optimistic/pessimistic approximation. In addition, there are also other tasks the completion rules must manage. Since the calculation of weights is part of the preprocessing step it is
Automatic source code classification – Classifying source code for a Case-Based Reasoning system

Markus Nordström

2015-06-11

up to the completion rules to calculate and assign weights according to the current context. Lastly, completion rules are in charge of calculating data that is not a part of the reasoning but rather of interest to the user, namely the metadata. It can be average values for prices or tendencies of sensor values, for example. [16]

Illustration 2: Completion rules.

Adaptation rules perform actions in the form:

Adapt: Solutions → Solutions

Case adaptation refers to three case descriptions: the query case, the retrieved case and the goal case, see Illustration 3. The adaptation rules consider all three cases in order to make a goal case as good as possible. The adaptation can be performed on single objects as well as complex solutions with multiple objects. When considering single objects for adaptation, the rule only changes a single attribute’s value. To adapt a complex solution, a sequence of single object rules are required to change the solution’s objects since each object needs its own adaptation rule. There are preconditions that dictate when a rule is valid and that the action will result in a valid solution. [17]
There are several types of adaptation. The reason for this is that all objects under consideration may not have the same properties. Some objects may be static, meaning that they cannot be changed because of their nature. Then there are objects that must be transformed using rules defined specifically for these objects. Lastly, there are objects requiring so-called generative adaptations because the current context demands that the adaptation is generated from a derivation of a problem. This approach generates a solution more or less from scratch. These approaches are formally called: [18]

- Null adaptation
- Transformation-based adaptation
- Generative adaptation

There are two kinds of transformation-based adaptation: substitution and structural transformation. Substitution replaces parts of the retrieved solution with valid parts that transforms the retrieved solution towards the goal solution. Structural transformation, however, changes the structure of the retrieved solution and reorganises it by adding or deleting structural parts of it. [18]
Traditionally in computer science, rules are logical and always true. In CBR this means that rules are treated as logical inferences. The inferences are concatenations of operators and are formally defined below. [7]

**Definition 9:** An operator sequence $\text{op} = \text{op}_1 \circ \text{op}_2 \circ \text{op}_3 \circ \ldots \circ \text{op}_n$ is a sequence of operators where the preconditions of each operator are not violated by the results of the preceding operators.

This means that a sequence of operators $\text{op}$ transforms a product $p$ into an adapted product $p_{\text{op}}$.

$$ p_{\text{op}} = \text{op}_1(p) \circ \text{op}_2(p) \circ \text{op}_3(p) \circ \ldots \circ \text{op}_n(p) $$

Since the retrieved solution is not enough to solve the problem the operators will transform it towards the goal solution. This means, however, that the first sequence of operators may not be enough. This is why it is necessary to have a way to see if the adapted solution is better than the retrieved solution. [7]

$$ \text{sim}(\text{wanted, retrieved}) < \text{sim}(\text{wanted, adapted}) $$

By using this relation it can be established if the adaptation was successful and to what degree it was successful. [7]

### 2.2.3 Revise

There is a revision step because no matter how thoroughly a recommended solution is created it may still be wrong. Like all other computer systems there may be weaknesses and defects that needs to be discovered and corrected. The revise step in CBR consists of three substeps: [19]

- Evaluation
- Revision
- Learning

Evaluation checks whether the proposed solution is good enough and is meant to provide some kind of feedback on the retrieve step. Revision and learning may sound like similar operations and in reality there is no sharp line between them. The essential difference is that revision is performed locally in the sense that only a single case is considered, whereas learning is global in the sense that it treats several cases and processes from global aspects. [19]

There are three types of evaluation in CBR, each with its own methodology and requirements. The three different types are presented below: [19]

- Evaluation by teacher
Evaluation by teacher means that the system requires a human expert to evaluate a proposed solution. This is both expensive as a human expert requires payment and it also introduces the human error factor into the CBR system. It is also possible that no human expert is available. [19]

Evaluation in the real world means that there is some kind of auxiliary system that reports changes in the environment related to solutions to prior problems. An example of an auxiliary system is an integrated real-time sensor system that measures the consequences of a solution in the real world. There could also be a function that judges if the solution was appropriate for the problem, enabling the system to correct the issue. [19]

Evaluation in the model is a self-analysis of the system and can be done in several ways. One way is to search for inconsistencies in the symbolic model. If an inconsistency is found then the retrieved case in question has to be removed. Another way is to make a numerical simulation based on evidence categories created from evaluation functions to determine whether a solution works or not. There is also a way to statistically analyze the model, but it requires several examples to make it possible to determine whether a solution is valid. In addition it requires evaluation criteria and testing methods. [19]

The revision substep focuses on observations from the real world, making it possible to record consequences of solutions and learn from them. This extends the contents of a case to include the consequences which will be defined as “event occurred”. [20]

Case = (Problem, (Solution, event occurred, other))

The consequences will explain what happened when the solution was applied. The consequence may be positive or negative in nature and will give an indication of whether or not there is need for revision. The difference from the adaptation step is that adaptation is part of the system and is priori defined. The revise step, however, originates from outside the system and is used if no adaptation is found. It is not represented by rules like the adaptation. When revision is applied it produces a new case: [20]

(Problem, Revise(Solution))

There are three methods that could be used to create the revise step. Because revise is mainly done as an external resource some of the methods are more appropriate in certain situations, all depending on the CBR system’s nature. The three methods are: [21]

• Self-repair
Automatic source code classification – Classifying source code for a Case-Based Reasoning system
Markus Nordström 2015-06-11

This is automatically done by the system. It generates a repaired solution by itself through domain knowledge or simulation.

User repair

A user has to find the repaired solution. There may be automated auxiliary systems involved to help the user, such as search engines.

Validity measure

Measure the validity of the knowledge structure by finding contradictions in the case base.

The learning substep involves learning from the acquired experience in the previous substep. Revision only takes place on demand for specific reasons. Learning is motivated by several reasons. [21]

Expensive mistakes.

Missed opportunities and results below expectations.

New technologies, aging technologies and productivity changes.

New businesses and improved competitiveness.

The purpose of the learning substep is to improve some part of the system. When a system is new a lot of learning is required, as many factors may be incorrect or missing. There are two types of learning when it comes to CBR, eager learning and lazy learning. Eager learning takes place at compile time for later use. This is useful when learning weights and similarity measures. Lazy learning takes place at runtime and its result is produced when used. This is the most common way in CBR since cases and solutions are considered at runtime. [22]

Learning can be divided into two methods: learning without feedback and with feedback. When learning there has to be some way to know what is good and what is better. In machine learning this cannot be precisely formulated because the algorithms would be too complex to be executed in reasonable time. Instead the system has to accept some degree of fault in the learning. This is represented by a user-defined threshold $\varepsilon$ and errors below this value are accepted. To extend this method there could be a probability $1 - \delta$ to only accept errors above this value. [22]

$$\text{Prob}(\text{error}(R) \leq \varepsilon) \geq 1 - \delta$$

This kind of learning is called Probably Almost Correct (PAC) learning and reduces the complexity of the learning algorithms. Since the parameters of a CBR
system cannot be determined exactly, the PAC learning method is very suitable. CBR is mostly concerned with approximations close to the truth but not exactly the truth. This method can also be used to determine how close a subset $X$ of cases to the whole case base $CB$ is. [22]

**Definition 10:** Let $CB$ be a case base contained in a real space. Then $X$ is an $(\varepsilon, \delta)$-approximation of $CB$ if for each case $c$ except for some set $Y$ with $|Y| < \varepsilon$ there is some $b$ in $X$ such that $d(c, b) < 1 - \delta$.

### 2.2.4 Retain

The retain step in the CBR process is responsible for storing cases in the case base. This seems to be a mundane task but in fact there is some learning involved here as well. Since the space where cases can be stored is not unlimited, there has to be some way to determine what cases should be retained and what cases should be omitted. This means that there are two factors that require consideration for the case base: the case base needs to be well informed to provide good solutions, but also be small and efficient for retrieval. These two factors conflict with each other. There are three basic algorithms that try to solve this: IB1, IB2 and IB3. IBL in this context means Instance-Based Learning. [23]

The simplest of the algorithms is IB1. It stores each new case that is created. The algorithm is primitive in this sense and has the disadvantage of making the case base grow very fast. This is not desirable since storage space is limited and it is important to keep the case base minimal for retrieval purposes. In theory, if there is an unlimited amount of cases in the case base then the probability of finding a satisfactory solution within is high. [23]

The second algorithm, IB2, tries to improve on IB1 by only storing cases based on nearest neighbour in the case base. The weakness of this method is that it depends on the presentation sequence. This means that even though a case is correctly classified in the training phase, and therefore not added to the case base, that very case may be misclassified later in the final case base. However, IB2 stores much fewer cases than IB1, but still has a classification performance similar to IB1. [23]

IB3 deals with the weakness of IB2 by removing bad cases from the case base. There are two predicates to determine the quality of a case. [23]

- **Acceptable**($c$) meaning that case $c$ should be added to the case base.
- **Bad**($c$) meaning that case $c$ is significantly bad and should never be added to the case base.

**Definition 11:** Let $c$ be an arbitrary case then:
The call probability is \( \text{Call}(c) = \text{Prob}(c \text{ is a nearest neighbour to an arbitrary problem } p) \)

The precision of \( c \) is \( \text{Prec}(c) = \text{Prob}(c \text{ gives a correct answer } | \ c \text{ is a nearest neighbour}) \)

The usefulness of \( c \) is \( \text{Usef}(c) = \text{Prec}(c) \cdot \text{Call}(c) \)

This definition only refers to the actual case base \( CB \) and not to a set of cases. This means these predicates are always updated in the loop of the algorithm. The goal of this is to teach a case base \( CB_{\text{acc}} \) that consist only of acceptable cases. [23]

In IB3 there is an acceptance scale defined on \([0, 1]\) where there are two user defined thresholds \( \alpha \) and \( \beta \). [23]

- Acceptable case \( \geq \beta \)
- Bad case \( \leq \alpha \)

This means that the parameters \( \alpha \) and \( \beta \) control the IB3 algorithm. IB2 and IB3 are also involved in forgetting cases. The important part in forgetting cases is that even if a case is removed from the case base it retains its current competence. [23]

### 2.3 Case Representation

In CBR a case could be represented in many ways. The representation is highly dependent on the purpose of the reasoner and how the system reuses cases. The more formal definition of a case is: “A case is a contextualized piece of knowledge representing an experience that teaches a lesson fundamental to achieving the goal of the reasoner” [24]. For a case to be comprehensive, in theory, it should contain the following five parts [25].

1. A situation and its goal.

2. The solution and the means with which it is derived.

3. The result of the solution. Was the result what was expected?

4. An explanation of the result. Why did it work? Why not?

5. The lessons that can be learned from this solution.

However, essentially, a case is composed of a problem description and a solu-
2.3.1 Feature vector representation

Cases that are represented as feature vectors are numerically transformed into attribute-value pairs stored in a vector. This means it should be possible to describe the object to be represented in a feature vector numerically. An attribute-value pair is just an attribute name associated with an attribute value. The advantages of this representation is that it is easy to perform processing and statistical analysis since the object is numerically encoded. [25]

2.3.2 Frame-based representation

Frame-based representations are partially formalized using description logics. Frame-based representation is a form of structured case representation that stores case information in a kind of frame structure. This has been further improved by treating such cases as terms in feature logics, a kind of description logic. Domain knowledge is integrated using a sort hierarchy. To treat composite cases that consist of several objects or sub-cases the representation method assumes that a sub-term is also a term. The similarity measure is utilizing the methods subsumption and anti-unification from feature logics. [25]

2.3.3 Object-oriented representation

The object-oriented case representation uses the concept of the object-oriented paradigm, namely the is-a and part-of relation. This means the case will be a hierarchical structure that uses inheritance. Cases are thus represented as a collection of objects where each object consists of a collection of attribute-value pairs much like the feature vectors. An object class describes the structure of an object. This approach is effective when the domain is complex with many different factors. [25]

2.3.4 Textual representation

Textual case representations are needed when a case is contained in textual form. This representation method decomposes the text into so called Information Entities (IEs). An IE can be either a word or phrase in the text that is deemed important to determine the reusability of a case. Using an extensive vocabulary consisting of relevant words and phrases, texts can be mined for IEs. This means that case acquisition can be automated. In this method, the case base is organized by a Case Retrieval Net (CRN) which is a directed graph with nodes representing cases and their IEs. Similarity determines the links between
2.3.5 Hierarchical case representation

All basic approaches to case representations consider cases at one level of abstraction. However, in this method cases are represented at multiple levels of abstraction. The basic idea is to represent cases at multiple levels of detail, there may even be different vocabularies. This means that when solving a new problem, cases are retrieved at appropriate levels. These cases are then combined and refined. [25]

2.3.6 Generalized case representation

All basic approaches to case representations consider only one point in the case space. However, a generalized case covers a subspace of the case space instead, which means that it provides solutions to a set of similar problems rather than a single problem. [25]
3 Methodology

In this chapter the methodology to create a knowledge base structure and classification system for a CBR system will be presented.

3.1 Theoretical model

In order to create a classification system for source files, a theoretical model was constructed to give an overview of the intended system and its functionality. The system was designed to be a tree structure because of the computational complexity that a tree provides. The tree had five levels, each carrying out its own task within the system, where each level analyses a source file for different attributes that are important when making a new case.

The theoretical model of a case was constructed by manually looking through a great amount of source files for common factors and characteristics that would prove useful attributes for a case. Since a case in CBR is split into two parts, a problem part and a solution part, each part was individually developed since they have different functions in a CBR system. Since a problem needs to describe the situation where the problem was found, its attributes are different from a solution. The solution describes how a problem was solved and thus its attributes are different from a problem’s attributes.

In CBR, having a way to determine distance between two cases is necessary, therefore a distance measure was developed. The distance measure was designed to calculate distance with regards to the case attributes.

In order for a CBR system to understand the terms used in the source files it requires a vocabulary. The vocabulary was developed to include terms used in Java EE programming language. In addition, the vocabulary was developed to include Javadoc as well, in order to make the system understand Javadoc tags.

3.2 Practical model

The practical work is an implementation of the theoretical model developed previously. The language that was used to implement the model was C++ version 11 [26], and Microsoft Visual Studio [27] was used as integrated development environment (IDE). In order to test the system and gather statistics the company Dewire supplied a large number of source files. However, the source files are confidential and as such the details of them are undisclosed.

Implementation of the different layers in the tree structure used the Standard Template Library (STL) provided by C++ version 11 [26].

3.3 Evaluation

To evaluate the results of this thesis, a comparison was made to a similar system called the ACAI system. The comparison included how ACAI stores cases and what information it holds about a case, as well as what kind of methodology ACAI uses when constructing a case.

Since CBR is known to have computational complexity problems, the evaluation also contained a calculated approximation of the system’s overall computational complexity.

An evaluation of the result of the classification process was also made. The factors that were investigated is the distribution of the resulting cases from the classification in the tree. The resulting distribution in the tree was analyzed and an attempt to explain the reason for that distribution was made.
4 Design

In this chapter, the system design will be presented. Each part of the system will be explained in its respective chapter.

4.1 The case

As explained in section 2.1, a case includes both a problem and a solution where both have different purposes and attributes. In this chapter the attributes chosen for both problem and solution will be presented.

4.1.1 Problem

Traditionally, a problem in CBR describes a situation, as mentioned in section 2.1, but in the context of source code that is not suitable. In this work, a problem describes a source file using keywords found in the documentation of the file. Since a Java file either contains a class or interface, the name of it is also used to describe the file. The keywords will describe the purpose and content of the source file and be used for similarity later.

To describe a source file, a vector of terms is used. This vector contains all the keywords found in the documentation of the file plus the name of the class/interface and will be called the file abstraction vector in this work. The keywords that are searched for are all defined in the vocabulary and thus part of the domain knowledge, see section 4.3 and section 5.3.

When a file is being classified, the system searches the file using a top-down linear methodology. However, only rows that contain comments will be searched.

4.1.2 Solution

The solution part of a case in this work will encompass several attributes, all extracted using the different analyzer modules described in section 4.2. The purpose of these attributes is to give a thorough representation of a source file, but not contain the entire source file. In this way, the contents of the source file will be abstracted into data used for similarity. However, there is a record in the solution that points out the original source file in case it is needed again.

Some of the attributes in a solution will need a well defined non-value assigned to them at the time as a new case is instantiated in the system. The reason for this is that when the new case is instantiated in the system it is in a default state meaning that not all attributes have been assigned a value from the system. In this default state the case requires well defined non-values so that no default case can be mistaken for a complete case.
Below is a list of all attributes included in a solution and their explanation.

- **Pointer to the original source file:** This is necessary because when a good similarity is found it would be useful to access the original file with all its content.

- **Comment versus Code ratio:** This ratio was developed to help determine the usefulness of a source file. The reason for this is that the more comments there are in a source file the more reliable the file abstraction vector will be since it has a greater amount of descriptive comments to access. This means that the more reliable the file abstraction vector is the more accurate the similarity measure will be, which is important for the reliability of the CBR system to provide a good solution. The documentation analyzer (see section 4.2) will calculate the ratio by counting the number of comments versus the amount of executable source code and use the ratio to determine which documentation class the source file belongs to. The ratio is defined on an interval $[0, 1)$ which means that a source file with over 1 in comments versus code ratio will likely have a good documentation class and thus be very useful for the CBR system. There are also three thresholds defined on this interval namely; $\alpha$, $\beta$ and 1. See Illustration 4.

![](Illustration_4.png)

*Illustration 4: The comment versus code ratio illustrated on the real axis*

These depict thresholds for sub-intervals on $[0, 1)$ each of which represents a comment versus code ratio class. These classes will be an abstraction of the ratio value and will be used in the similarity measure. The abbreviation CC stands for Comment versus Code. The classes are defined as follows:

- **CC_CLASS1:** This is the lowest class of comments versus code ratio. It represents very little to no documentation and thus source files that fall under this category is of little usefulness to the system. This class is defined between 0 and $\alpha$.

- **CC_CLASS2:** This class represents a good degree of comments versus code ratio and a good usefulness to the system. This class is defined between $\alpha$ and $\beta$. 
Automatic source code classification – Classifying source code for a Case-Based Reasoning system
Markus Nordström 2015-06-11

- **CC_CLASS3**: This class represents a very good degree of comments versus code ratio. Source files in this class are very useful for the system since there is a good number of comments that describe the file. This means that the probability that this source file's file abstraction vector is describing the file accurately, and is of high usefulness to the system. This class is defined between $\beta$ and 1.

- **CC_CLASS4**: This is the most useful class a source file can be in. Source files that are classified under this category are the most useful files for the CBR system since there is a great number of comments. This means that it is probable that the file abstraction vector will be accurate. This class is defined above 1.

- **CC_NOT_SET**: This is the non-value for the comment versus code ratio.

The thresholds $\alpha$ and $\beta$ are user defined and can be changed during run-time or at compile-time.

- **Header versus Function ratio**: This ratio was developed to help determine the usefulness of a source file, just like the comments versus code ratio, however, this ratio has a much bigger impact on the documentation class. The reason for this is that a header describes a function and thus has a bigger impact on the file abstraction vector. This means that it should have a bigger impact on the determination of documentation class. The documentation analyzer (see section 4.2) will calculate the ratio by counting the number of headers versus the number of functions. The ratio is defined on an interval $[0, 1]$ and the reason for this is that there cannot be more headers than functions. There are three threshold variables defined on this interval namely; $\alpha$, $\beta$ and $\gamma$. See Illustration 5.

Illustration 5: The Header versus Function ratio illustrated on the real axis.

Similar to the comment versus code ratio, these threshold variables depict sub-intervals on $[0, 1]$ where each of the sub-intervals represents a class in header versus function ratio. These classes will be an abstraction of the actual value of the header versus function ratio and will be used in the similarity measure. The abbreviation $HF$ stands for Header versus Function. The classes are defined as follows:
Automatic source code classification – Classifying source code for a Case-Based Reasoning system
Markus Nordström
2015-06-11

- **HF_CLASS1**: This is the lowest class of header versus function ratio. Files that are classified here have very few headers that belong to a function. This means that these files are of very little usefulness to the system since there is almost no documentation to indicate the purpose of the functions. This class is defined between 0 and $\alpha$.

- **HF_CLASS2**: This class represents an acceptable number of headers. Files classified here are somewhat useful to the system but provide unreliable information to the file abstraction vector since there is not much documentation. This class is defined between $\alpha$ and $\beta$.

- **HF_CLASS3**: Source files that are classified in this interval are very useful to the system since there is a good number of headers that describe the functions within the class or interface. This class is defined between $\beta$ and $\gamma$.

- **HF_CLASS4**: This class is the most useful of the header versus function classes. Source files classified here are well documented documents that provide excellent information to the file abstraction vector since there is a header to almost every function. This class is defined between $\gamma$ and 1.

- **HF_NOT_SET**: This is the non-value for the header versus function ratio.

The thresholds $\alpha$, $\beta$ and $\gamma$ are user defined and can be changed during run-time or during compile-time.

- **Documentation class**: The purpose of the documentation class is to provide an abstraction of the level of documentation within a source file and give the overall usefulness of a source file. The documentation class is calculated using the comments versus code ratio and header versus function ratio with the following formula:

$$\text{Documentation class} = \omega_{cc} \cdot \text{CCRatio} + \omega_{hf} \cdot \text{HFRatio}$$

Where $\omega_{cc}$ and $\omega_{hf}$ are weights used to indicate the significance of the different ratios as both of them may not be equally significant to the level of documentation. The weights can be changed during run-time as well as compile-time. Similar to the ratios, the documentation class is defined on a an interval $[0, 1)$ with two thresholds depicting sub-intervals $\alpha$ and $\beta$. See Illustration 6.
Automatic source code classification – Classifying source code for a Case-Based Reasoning system
Markus Nordström
2015-06-11

DOC_CLASS1: This class indicates a source file that will have very little usefulness for the system since the combined ratios are both low. This class is defined between 0 and $\alpha$.

DOC_CLASS2: This class indicates an average usefulness of a source file since the level of documentation is not extensive. This class is defined between $\alpha$ and $\beta$.

DOC_CLASS3: Source files classified in this class are considered to be very useful since the ratios that define the documentation class would be high. This class is defined between $\beta$ and 1.

DOC_CLASS4: The source files that are classified here are the most useful files since there is a high probability that they have an accurate file abstraction vector, making them the most useful for similarity checks. This class is defined above 1.

DOC_CLASS_NOT_SET: This is the well defined non-value for the documentation class.

The thresholds $\alpha$ and $\beta$ can be changed during run-time or compile-time.

- **Code size class:** This class is an abstraction of the size of a source file. Size in this context means the amount of code that the file contains. This is also used in similarity since a very large file cannot perform the same functionality as a very small file. This means it acts as a kind of filter for similarity checks. There are four thresholds in this class which defines an interval $[0, \delta]$. The thresholds are $\alpha$, $\beta$, $\gamma$ and $\delta$. These thresholds represent sub-intervals. See figure.

Illustration 6: The documentation classes represented on the real axis.

Illustration 7: The code size classes represented on the real axis.

Each of these classes are just an abstraction of the number of code rows in the source file and they have no other purpose. The thresholds $\alpha$, $\beta$, $\gamma$
and $\delta$ are user defined and should represent an increased number of code rows. They can be changed during run-time and compile-time.

- **Indicator whether the source file contains a class or interface**: This indicator will tell if the file contains a class or an interface. The reason for this is that there is a significant difference between classes and interfaces and thus there is a need to distinguish between them.

- **The common name of the class/interface, if any**: This attribute contains a common name if there is one. This means that if the class name or interface name contains a name, or part of a name, that is common among class/interface names then it will be stored here. Examples of common names would be Factory, Repository or Service.

- **The parent name of the class/interface, if any**: This attribute contains the name of the parent of this class or interface, if there is a parent.

- **The number of member functions**: Contains an integer that indicates the number of member functions contained in the source file.

- **The number of member variables**: This attribute contains an integer that indicates the number of member variables of a source file.

- **A composition of the member variables**: This attribute is a composition of member variable types. This means that it contains a table of each member variable type and the number of them in the source file. This is useful when calculating similarity.

- **A composition of the member functions**: This attribute contains a table similar to the member variable composition but in addition to storing return types of the functions it also stores the types of arguments of the corresponding function. This kind of table is useful when calculating similarity.

- **A semantic lookup table**: The semantic lookup table contains all the names of the member variables, the names of the functions and their arguments. This table is used when checking similarity.

All of these attributes are stored in each new case that is constructed.

### 4.2 File classification system

The file classification system was developed to analyze and extract data from a source file. This data is then put into a new case, to each of the respective attributes explained in section 4.1.2. In this chapter all the different analyzers are presented and explained on a conceptual level, and explained in-depth in sec-
This chapter presents the type filter, documentation analyzer, syntactic analyzer and semantic analyzer.

The classification system is designed to work as a tree structure. This means that when a new case have been constructed it will end up in a leaf node of the tree. See Illustration 8.

This figure illustrates the four levels developed in this work. When the type filter has determined that the source file is either a class or an interface and assigned it to the case, the documentation analyzer determines the level of documentation in the source file and assigns a class to the case. The classes are called $C_1$, $C_2$, $C_3$ and $C_4$ in the figure, and these are the documentation classes which will be explained further in section 4.1.2. These classes have their own node in the tree, hence the multiple connection 4 in Illustration 8.

When the documentation classes have been created, the syntactic analyzer assigns further classes to the case, here called $C_1$, $C_2$, $C_3$, $C_4$ and $C_5$. These classes are the code size classes which are explained further in section 4.1.2. These classes have their own node in the tree, hence the multiple connection 16 in Illustration 8.

When the code size class has been determined, the semantic analyzer extracts semantic data from the source file such as variable names and function names.
and enters it into the semantic table, see section 4.2.4. When this is done, the case is put in the respective leaf node of the tree, namely one of the rectangles at the bottom of the tree.

Each of the leaf nodes contains a so-called semantic tree. The semantic tree is a kd-tree the purpose of which is to improve the search time in the node. See Illustration 9.

![Illustration 9: A graphical illustration of the semantic tree.](image)

To determine the reference value in the root all cases in the node are sorted alphabetically by their class/interface name and a median is selected from among them. This median is then used as reference and all other cases are measured against it. The measure uses the cosine distance function defined in section 2.1.1 to determine whether the case in question is going left or right in the tree. The attributes that are compared are return types of the functions as well as the types of the arguments to the functions. When it comes to variables, the variable types are compared. The same method is used in the level two nodes, but the median is selected from among cases that ended up in that node.

Each level is presented and explained in its respective chapter. The rectangles at the bottom represents the leaf nodes in the tree and this is where the completed case will end up after the classification is completed.
4.2.1 Type filter

The purpose of the type filter is to determine whether the source file being analyzed is a class or an interface. The file classifier is supposed to be a part of a CBR system, which means that it has access to the domain knowledge, i.e. that the type filter uses the knowledge contained in the vocabulary to determine the source file’s content.

4.2.2 Documentation analyzer

The documentation analyzer has three tasks to carry out when it analyzes a source file. Firstly to determine the comment versus code ratio. Secondly, it determines the header versus function ratio. Lastly, it combines the two ratios into the documentation level formula defined in section 4.1.2.

When the comment versus code ratio is calculated, the documentation analyzer counts the number of comments in the source file and the amount of executable code in the source file to use in the formula below.

\[
\text{Comment versus code ratio} = \frac{|\text{Comments}|}{|\text{executable code}|}
\]

The result is stored internally within the analyzer to be used later when the documentation level is calculated.

In the same manner, the header versus function ratio is calculated by identifying the number of functions and the number of functions with a header above and use these values in the below formula.

\[
\text{Header versus function ratio} = \frac{|\text{Headers}|}{|\text{functions}|}
\]

The result is stored internally within the analyzer to be used later when the documentation level is calculated.

When both the ratios have been calculated, the documentation level is calculated using the below formula.

\[
\text{Documentation class} = \omega_{cc} \text{CCRatio} + \omega_{hf} \text{HFRatio}
\]

Where \(\omega_{cc}\) and \(\omega_{hf}\) are weights used to indicate the importance of the corresponding ratio. The reason for this is that the ratios may not have the same importance when the system have been running for a time. These weights can be changed at any time during run-time or at compile-time.
4.2.3 Syntactic analyzer

The syntactic analyzer analyzes the source file for statistics about the content. These statistics will be used for the similarity measure since it provides a numerical way of performing a similarity check. It provides data about the following attributes:

- number of member variables
- number of member functions
- composition of the member variables
- composition of the member functions
- common name of the class/interface
- parent name of the class/interface

The number of variables and functions are calculated by counting the corresponding variables/functions. The compositions are tables which contain all variable types and the number of variables. When it comes to functions, the table contains the the return type of the function, and in addition it also contains the types of arguments that belong to the function’s argument list. Lastly, the analyzer determines whether the class/interface has a common name as explained in section 4.1.2. It also determines whether the class/interface has a parent. However, neither the common name nor the parent name is stored in this analyzer, that is the job of the semantic analyzer. The rest of the variables are stored internally within the analyzer and later assigned to the new case.

In order to analyze the source file for data to be used in the attributes, the analyzer utilizes knowledge from the vocabulary, i.e. knowledge about standard Java keywords, how a variable is defined and how a function is defined. See section 4.3 where the vocabulary is described.

4.2.4 Semantic analyzer

The semantic analyzer extracts semantics from the source file being analyzed. This data will be used in the similarity measure for comparison between cases. The semantic analyzer provides data for the following attributes:

- common name of the class/interface
- parent name of the class/interface
- semantic lookup table
Firstly, the analyzer checks whether the syntactic analyzer sets the flags for the presence of a common name or a parent. If any of these flags are set then the semantic analyzer extracts the data and stores it. The primary purpose of the semantic analyzer is to construct the semantic lookup table. This table is similar to the compositions used in the syntactic analyzer but instead of storing types the semantic table stores the names of variables, functions and their arguments. This data will be used in the similarity measure.

4.3 Vocabulary

For a CBR system to comprehend the problem domain there is a need for a vocabulary, see section 2.1.2. Everything the system needs to know about a Java file is stored in the vocabulary. If it is not in the vocabulary, the system will not know about it. The vocabulary in this system contains knowledge about the Java language keywords [28] and knowledge about how comments are represented in Java. It also contains information about the different classes used to abstract the ratio values, see section 4.1.2. Knowledge in the vocabulary is sorted by type in different so called knowledge containers. The different knowledge containers are listed below:

- **Comments versus code ratio container**: This knowledge container provides the knowledge about comments versus code ratio classes. It contains data on CC_CLASS1, CC_CLASS2, CC_CLASS3, CC_CLASS4 and CC_NOT_SET. See section 4.1.2 for the definition of these classes.

- **Header versus function ratio container**: The data contained in this knowledge container is the header versus function ratio classes. It contains HF_CLASS1, HF_CLASS2, HF_CLASS3, HF_CLASS4 and HF_NOT_SET. See section 4.1.2 for the definition of these classes.

- **Documentation level container**: This knowledge container contains data on the different classes used to describe the level of documentation in a source file. It contains DOC_CLASS1, DOC_CLASS2, DOC_CLASS3, DOC_CLASS4 and DOC_CLASS_NOT_SET. See describe 4.1.2 for the definition of these classes.

- **Code size class container**: This knowledge container includes the code size classes. It contains CODE_SIZE_CLASS1, CODE_SIZE_CLASS2, CODE_SIZE_CLASS3, CODE_SIZE_CLASS4, CODE_SIZE_CLASS5, CODE_SIZE_NOT_SET. See section 4.1.2 for the definition of these classes.

- **Java keywords container**: This knowledge container holds all the keywords used in the Java language. This container is used by the syntactic analyzer to find member variables and functions. See section 5.3.
Automatic source code classification – Classifying source code for a Case-Based Reasoning system
Markus Nordström

• Comments container: This knowledge container provides knowledge of how comments are represented in the Java language. It contains knowledge about row comments, block comments and Javadoc tags [29].

There is also knowledge about how to measure distance between two cases. In this work, the cosine distance is used because it calculates the difference using two vectors of attributes, see section 2.1.1. This means that the vocabulary contains the formula used to calculate the cosine distance. The way the two attribute vectors are represented is defined in the knowledge containers listed above, and since the attributes in a case are mostly numerical, there is no need to convert values into numerical form. The attributes that are non-numerical in a case are only used to check similarity against non-numerical values.

4.4 Distance Measure

In this work, the distance measure is simple in nature. It only takes two different cases and calculates the distance between them by using the formula defined in section 2.1.1. It calculates the cosine distance between two cases. This is also used in the semantic tree as governing function to build the tree.
5 Implementation

This chapter outlines the implementation part of this work. The implementation of each part of the system is explained in its respective chapter.

5.1 The case

The case object is a reflection of the definition of a case, see section 2.1. It contains a problem object and a solution object, see Illustration 10. Each of them are presented in this chapter.

Illustration 10: Class diagram of the caseObject class.

When constructing a new case, this is the object used. It initializes both the problem and solution object in their guaranteed default state. The reason for this is that the case should contain only non-values before the analysis begins. The case object has all the methods to pass data to and from the problem object and solution object respectively. They have been omitted in this report as their implementation is self-explanatory.

5.1.1 Problem object

The problem object contains the file abstraction vector that is used as problem within this system, see Illustration 11. The vector contains the descriptive terms that represent the file that the problem is associated with. The object is initialized with a default state where the file abstraction vector is empty. There are also methods for adding and removing terms from the vector as well as returning the size of the vector, meaning the number of contained terms.
5.1.2 Solution object

The solution object contains all attributes defined for a case, see section 4.1.2. Most of the field names are similar to their respective attribute, see Illustration 12, but there are a few fields that are not, and these will be explained in this chapter.
In this class there are three flags; `commonNameFlag`, `interfaceFlag` and `parentFlag`. These flags indicate the presence of specific content in the case. The `commonNameFlag` indicates that the case contains a common name and the `commonName` field contains a such name. The `interfaceFlag` indicates that the case is an interface rather than a class. The `parentFlag` indicates that the case has a parent and that the `parentName` field contains a such name. These flags are not of boolean type, they are a custom type called `CBRType`, which is a composite type that can be true, false or null. The reason for this is that there is a need to have a well defined non-value also for the flags.

The solution object also contains a pointer to the original file for reference, indicated by the `solutionFile` field.

### 5.2 File classification system

The file classifier is the implementation of the file classification system, see chapter 4.2. The class itself contains a `newCase` field which is the case that the source file’s attributes will be added to, see Illustration 13. It also contains default instances of all the analyzers at the time of initialization.

*Illustration 13: The file classifier class diagram.*
When the file classifier is instantiated, a source file must be provided. The source file is then read using the `readFile` method which reads the content of the file and places it in the `fileContents` field which is later used to pass the contents of the file to the different analyzers.

Each analyzer has its own method `analyzeFile`, which carries out different things. In this class the `analyzeFile` function calls each of the analyzer’s respective `analyzeFile` with the source file as argument and receives the attributes for the new case in return.

Each analyzer has its own instance of the vocabulary in order to be provided with the required domain knowledge for their task.

The tree structure mentioned in section 4.2 is implemented in this class through the `typeFilter`, `documentationAnalyzer`, `syntacticAnalyzer` and `semanticAnalyzer` fields. The `typeFilter` represents level 1: Type filter in the figure and thus determines the class or interface indicator. The `documentationAnalyzer` field represents level 2: Documentation analyzer in the figure and performs the documentation analysis. The `syntacticAnalyzer` field represents level 3: Syntactic analyzer in the figure and analyzes the syntax. Lastly, the `semanticAnalyzer` field represents the level 4: semantic analyzer in the figure and extracts the semantics from the source file.

All of the analyzers are explained in this chapter.

### 5.2.1 Type filter

The type filter implementation analyzes the file to determine whether it contains a class or interface, see section 4.2.1. The type filter searches for the class or interface keyword using a linear search, giving it a $O(n)$ in computational complexity. It contains the `interfaceFlag`, which, when the analyzer is done, is passed back to the file classifier to be added to the new case. It also contains the `voc` field which is this analyzer’s instance of the vocabulary. See Illustration 14.

![Illustration 14: The type filter class diagram](Image)
5.2.2 Documentation analyzer

This is the implementation of the documentation analyzer. This class contains all the fields to analyze the documentation of the source file and give it a classification, see section 4.2.2.

The difference between documentation class and documentation level in this implementation is that the documentation class is the abstraction of the documentation level absolute value, see Illustration 15. The \texttt{ccRangeUpperBound} field is the comments versus code ratio upper bound which is equal to 1. In addition, all the thresholds are defined here and there are methods defined to change these as well.

The search for documentation is of linear character throughout the whole file which gives it a $O(n)$ in computational complexity.

This analyzer also has an instance of the the vocabulary in the form of the \texttt{voc} field.
5.2.3 **Syntactic analyzer**

This chapter outlines the implementation of the syntactic analyzer. This class contains all the fields necessary to analyze the syntax of a source file, see section 4.2.3. The search for variables and functions in a source file is linear and thus it is O(n) in computational complexity. Here, \( \alpha_1 \) up to \( \alpha_4 \) is used to depict the thresholds for the \( \text{codeSizeClass} \) field. It is in this analyzer that the \( \text{parentFlag} \) is set and later used by the semantic analyzer. The \( \text{voc} \) field represents the vocabulary instance in this analyzer, see Illustration 16.

![Illustration 16: The syntactic analyzer class diagram.](image)

When the analyzer processes a file it will enter data into \( \text{variableComposition} \) and \( \text{functionComposition} \) as well as both \( \text{Count} \) fields. When the analysis is done, the attributes are passed back in the file classifier to be added to the new case.

5.2.4 **Semantic analyzer**

This chapter outlines the implementation of the semantic analyzer. This class contains all the fields necessary to perform a semantic analysis of a source file. When searching for variable and function names a linear search is used, giving it O(n) in computational complexity. Here the parent flag is checked to see if it is necessary to extract the name of the parent. If it is necessary then the name of the parent is saved to the \( \text{parentName} \) field, see Illustration 17.
Data is also added to the semantic table field as the analysis progresses. If there is a common name contained in the source file then the \textit{commonNameFlag} is set and the name is saved to the \textit{commonName} field. This class also contains a vocabulary instance in the form of the \textit{voc} field.

\section*{5.3 Vocabulary}

As explained in section 4.3, the vocabulary contains all the domain knowledge the system needs. Some of the methods in the vocabulary are omitted as their purpose is not important for the understanding of the whole system and their names are self-explanatory. Below is the class diagram of the vocabulary. See Illustration 18.
The vocabulary object contains three fields: `comments`, `javaDocTags` and `keywords`. These fields contain knowledge about the Java language such as keywords and how to represent comments in the language. All of these fields are initialized in the `init` method. When a default object of the vocabulary is instantiated, the `init` method is executed. This guarantees that two separate vocabulary object contains the same knowledge no matter where in the system they are created.

The `comments` field contains knowledge about the representation of comments. The comments field is a C++ map [30] which is a hash map. This hash map is initialized in the `init` function as mentioned and uses an enumeration as key, see Illustration 19, and a string representation of a comment character as value.

Illustration 19: The COMMENT enumeration.

The map is initialized using the comment enumeration and its associated value in the `init` function as follows:

```cpp
this->comments.insert(pair<COMMENT, string>(ROW_COMMENT, "//"));
this->comments.insert(pair<COMMENT, string>(BLOCKCOMMENT_START, "/*"));
this->comments.insert(pair<COMMENT, string>(BLOCKCOMMENT_MID, 
"*"));
this->comments.insert(pair<COMMENT, string>(BLOCKCOMMENT_END, 
"*/"));
```

In the same manner, the `keywords` field is a map which takes an enumeration and its associated value and stores it. The enumeration contains all the java keywords defined in the language and is stored in the `keywords` map along with its associated string representation value. The `keywords` map is also initialized in the `init` function.

The `javaDocTags` field is also a map defined much in the same way as `keywords` and `comments`. It takes an enumeration and its associated string represen-
In the vocabulary there are also four other enumerations used to represent the different classes a case can be assigned. The enumerations are: DOCUMENTATION_CLASS, CC_CLASS, HF_CLASS and CODE_SIZE_CLASS.

The DOCUMENTATION_CLASS enumeration contains the different documentation classes a case can have as well as the well defined non-value for the documentation analyzer’s default state. This enumeration is used in the documentation analyzer (see section 4.2.2) when the documentation level is calculated.

The CC_CLASS enumeration contains all the comments versus code classes a case can be assigned, see Illustration 20. It also contains the well defined non-value for the documentation analyzer’s default state. This enumeration is used in the documentation analyzer.

Illustration 20: The CC_CLASS enumeration.

The HF_CLASS enumeration contains the different header versus function classes a case can be assigned in the documentation analyzer as well as the well defined non-value for the default state.

Lastly, the vocabulary contains the CODE_SIZE_CLASS. This enumeration contains all the classes a case can be assigned in the syntactic analyzer, see section 4.2.3. It also contains the well defined non-value for the syntactic analyzer’s default state.

5.4 Semantic analyzer

The distance measure is an object that defines the distance operator used in the system, see Illustration 21. This object implements the formula presented in section 2.1.1. The method cosineDistance takes two arguments, one left hand side argument and one right hand side argument. The arguments are of solutionObject type, see section 5.1.3. The returned value from cosineDistance is of the double type.
Illustration 21: The class diagram of the distance measure.
6 Results

In this chapter, all the test results from the classification system is presented level by level. The results are distributions among the nodes in the tree. The distribution is in percent because the testing set that was used to carry out the testing was under NDA and cannot be revealed. The test set comprised of 1701 Java source files.

6.1 Test results type filter

In this chapter, the result of the type filter is presented. This is the first step in the classification system and after processing all of the 1701 source files the resulting distribution is presented in the graph. The majority of the source files provided were Java classes.

![Graph 1: Graph of the distribution in the type filter.](image)

6.2 Test results documentation analyzer

In this chapter, the resulting distribution of the documentation classes are presented. The result is calculated in the class node since that is where most files are located. The result is presented in Graph 2. Most files were classified in DOC_CLASS2, and the remaining files were classified in DOC_CLASS4.
6.3 Test results syntactic analyzer

This chapter presents the results in the syntactic analyzer. Since most source files were classified in DOC_CLASS2, the distribution is presented from that class. Most source files were classified in the CODE_SIZE_CLASS2, and the remaining were classified in CODE_SIZE_CLASS4. No files were classified in the CODE_SIZE_CLASS5, see Graph 3.
6.4 Test results semantic analyzer

This chapter presents the resulting distribution in the leaf nodes as well as the final distribution of the interface files. Notable in the class distribution is that the result is weighted towards the left of the graph, see Graph 4.

Graph 3: Graph displaying the distribution of DOC_CLASS2.

Graph 4: Class distribution in the leaf nodes.
Graph 5 shows the distribution of the interface files. Most of these files were classified in Node 1.

6.5 Test results semantic tree

The result of the semantic tree in Node 6 from the class distribution is presented in this chapter. The reason for this is that most files were classified in Node 6. Notable in Graph 6 is that the balance in the tree is shifted to the left-most leaf node.
6.6 Example of a CBR system using the classification system

In this chapter an example of a CBR system is presented. This CBR system utilizes the classification system developed in this work. Some assumptions are made about the current context prior to the example; First, it is assumed that the case base is trained and contains an acceptable amount of cases to carry out the example. Secondly, it is assumed that the classification tree and semantic trees are balanced in the current state.

For this example, a RESTful service will be generated using CBR. The requirements of this service are listed below.

- RESTful service
  - Write welcome message
  - Write the message in uppercase
  - Write the message in lowercase

This list is thus the problem formulation for this example.

Now, that the problem has been defined it is time for the next step in the CBR process. The next step is the retrieve step which will retrieve the most similar case from the case base, see section 5.2.1 for more information on the retrieve step. To retrieve the most similar case, the classification tree is traversed to find
Automatic source code classification – Classifying source code for a Case-Based Reasoning system
Markus Nordström
2015-06-11

the appropriate leaf node. In this leaf node, the semantic tree is searched to find the most similar case.

To find the most similar case in the semantic tree, the cosine similarity was used. The most similar case that was found had the following properties:

- RESTful service
  - Write welcome message
  - Write message in lowercase
  - Print system time

The code in this case is presented below. Some of the code is omitted for space reasons.

```java
@Path("/greeting")
public class GreetingService {
  @GET
  public String message() {
    return "Welcome!";
  }

  @POST
  public String lowerCase(final String message) {
    return "Welcome!".toLowerCase();
  }

  @POST
  public String systemTime() {
    /*Code omitted*/
  }
}
```

This case is quite similar to the problem that is defined but it is missing the uppercase function. In addition, the case also has a print system time function which is not required. With this information gathered it is time for the reuse step in the CBR process.

In the reuse step, the found case is transformed to fit the goal problem. This is accomplished by using the structural transformation-based adaptation method presented in section 5.2.2. This method along with the domain knowledge of the Java language will adapt the case towards the goal. This means that the only thing missing is the uppercase function so it is added to the case, see below.
Automatic source code classification – Classifying source code for a Case-Based Reasoning system
Markus Nordström 2015-06-11

@Path("/greeting")
public class GreetingService {
    @GET
    public String message() {
        return "Welcome!";
    }

    @POST
    public String lowerCase(final String message) {
        return "Welcome!".toLowerCase();
    }

    @POST
    public String upperCase() {
        return "Welcome!".toUpperCase();
    }

    @POST
    public String systemTime() {
        /*Code omitted*/
    }
}

After this modification the case now solves its goal problem definition. However, the system time function is still there. This is ignored for now. It is now time for the next step in the CBR process, revise.

In the revise step it is time to check for inconsistencies in the proposed solution. It is found that the system time function produces unwanted results for the solution. To solve this, the system time function is simply removed, see the code snippet below.

@Path("/greeting")
public class GreetingService {
    @GET
    public String message() {
        return "Welcome!";
    }

    @POST
    public String lowerCase(final String message) {
        return "Welcome!".toLowerCase();
    }

    @POST
    public String upperCase() {
        return "Welcome!".toUpperCase();
    }
When the revision is done, the proposed solution behaves as expected. This is then learned by the CBR system. With this step completed, the last step is performed.

In the retain step the updated solution is stored in the case base using the IB3 algorithm, see section 5.2.4.
7 Evaluation

In this chapter, an evaluation is made of the classification system by comparing it to an existing CBR system and evaluating the complexity of the system as well as the results obtained in the system test.

7.1 Comparison with ACAI system

The Automated Coder using Artificial Intelligence (ACAI) system is a system similar to this work in the sense that it utilizes the CBR methodology to generate Java source code. Its purpose is to solve database-type problems in the medical domain by generating programs that perform queries on a database of records.

As opposed to this work, the ACAI system stores case problems as plans rather than file abstraction vectors. These plans are XML documents that contain instructions for generating an executable program. Each plan consists of a name and description along with an input, operation and output steps. The input step defines where the program gets its input. Operation defines the executable parts required in the program to solve the problem. Finally, the output step defines where the program outputs its data. This approach to describing problems is more structured than using a file abstraction vector since the vector may be inaccurate or describe the wrong purpose of the source file. However, creating a plan in the ACAI system requires knowledge on how to write all the different steps used in the XML document, which means that there is a risk of human error whereas the file abstraction vector contains only real data from existing source files.

The ACAI system uses a form of template based programming to build its programs. This means that, as opposed to this work, it defines a domain of code templates called components which acts as building blocks when constructing its programs. This way of defining the domain knowledge of the programming language means that the system will need a human expert to expand the domain of code templates while the classification system in this work guarantees that new code is always added to the domain knowledge, expanding it continuously.

7.2 Evaluation of the complexity

This chapter will evaluate the overall computational complexity of the classification system. The classification system is structured as a kd-tree but when analyzing source files each analyzer uses a linear search algorithm to find items in the source file giving the analyzing part \( O(n) \) for one file. This means that the overall analyzing computational complexity is \( O(s*n) \), where \( s \) is the depth of the tree and \( n \) is the number of files to be analyzed. This is of course not acceptable for the long term of the system. However, since source files are only analyzed once and not all source files are added at any one given time, instead files
are added over time. This means that since the number of files is spread over a period of time the overall computational complexity would be $O(n)$, rather than $O(s*n)$, which is more acceptable.

7.3 Evaluation of the result

In this chapter, the result of the testing done with the classification system is evaluated. For more information on the test results, see Chapter 6. This chapter will discuss the results given in the classification tree and try to give an explanation to the results. In addition, possible solutions will be suggested in order to solve the issues discovered.

Firstly, the majority of the files classified in the type filter are Java classes, this was expected since Java is an object-oriented programming language. Java defines a class as an object. This means, however, that the right part of the tree is much smaller than the left which contributes to imbalance. This can be solved by separating the tree at the root, meaning that there are two different trees: one for classes and one for interfaces.

In the documentation analyzer, most source files were classified in the lower classes; DOC_CLASS1 and DOC_CLASS2. This means that most source files have low usefulness for the end system since the file abstraction vectors from these classes are unreliable and inaccurate in their description of a source file. There is not much that can be done about this fact other than picking a more well documented test set and blacklisting low documented source files from the system. Another solution would be to find another way to describe a source file that does not require documentation. This result also shows that the tree is in imbalance in this analyzer which means that the tree needs to be rebalanced at some point.

The results of the syntactic analyzer in DOC_CLASS2 shows that most files are classified as CODE_SIZE_CLASS2. This implies that the tree is again imbalanced but this can be solved by redefining the thresholds that makes up the classification in the analyzer so that it fits the set being analyzed.

In the semantic analyzer, the results in all the leaf nodes are presented since the analyzer does not partition the file set any further. However, each leaf node contains a semantic tree to optimize search time. The results that are presented in graph peaks around Node 6 and disappears at Node 15. This shows that the distribution is shifted to the left of the graph. This implies that the overall usefulness of the source file set is low. It would have been optimal if the graph was shifted to the right instead, that would indicate a high usefulness of the source file set. The main reason for this is found in the documentation analyzer where most source files are classified with low documentation. This means that, in order to solve this issue, it might be necessary to find another way to classify source files independent of documentation in the file. The reason why there is a documentation analyzer is because it helps to construct the file abstraction vector. This implies that if a new method is to be used independent of the docu
The result in the interface leaf nodes were expected since there are so few interfaces and the fact that interfaces are just an abstract structure that other classes implements.

The result in the semantic tree in Node 6 indicates that many functions had little or no arguments. This is only statistics and it makes the tree imbalanced but it can be solved by rebalancing the tree. Since the purpose of the tree is to decrease search time, so there is no valuable classification data here.

### 7.4 Future work

This chapter will propose future work based on this thesis. It will explain what parts need to be developed in order to create a fully functional CBR system.

In this work, a classification system used for classifying source code into cases has been developed. The case has also been developed including the problem and solution parts as well as a way to store cases in the case base using a kd-tree. In addition, a similarity measure has been developed to enable a method to determine similarity between cases, see Illustration 22.

![Illustration 22: The CBR process model.](image)
This means that the work that remains is to develop each of the four steps of CBR: retrieve, reuse, revise and retain. When developing the retrieve process, the similarity measure and the classification tree should be integrated. These parts are developed to provide good compatibility with the retrieve process.

The reuse process also needs to be developed. The vocabulary developed in this work can be of use in the development process since the adaptation part is heavily dependent on the domain knowledge. However, the vocabulary needs to be extended to include the Java programming grammar as well.

The revise process will have to be developed from scratch since there is not much time devoted to it in this work. A learning algorithm needs to be developed as well, as a method of performing self-evaluation of a proposed solution. Here, the extended vocabulary would be useful since it contains the Java grammar.

The retain step, in this proposal, would need to implement the IB3 algorithm presented in section 5.2.4.
References


Automatic source code classification – Classifying source code for a Case-Based Reasoning system
Markus Nordström


Automatic source code classification – Classifying source code for a Case-Based Reasoning system
Markus Nordström
2015-06-11

[27] Microsoft Visual Studio
https://www.visualstudio.com/

[28] Java language keywords
https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html

[29] How to Write Doc Comments for the Javadoc Tool
http://www.oracle.com/technetwork/articles/java/index-137868.html

[30] map
C++ Reference
http://www.cplusplus.com/reference/map/map/?kw=map