UNIVERSITATEA POLITEHNICA BUCUREŞTI
FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

LUCRARE DE LICENŢĂ

Generatoare Automate de Cod folosind Mecanisme Dedicate Detectarii de Pattern-uri

Coordonatori ştiinţifici:
Prof. dr. ing. Valentin Cristea
Dr. ing. Paul-Alexandru Chiriţă

Absolvent:
Mihai-Alexandru Ciorobea

BUCUREŞTI
2013

UNIVERSITY POLITECHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE DEPARTMENT

DIPLOMA PROJECT

Intelligent Code Generation based on Pattern Detection Mechanisms

Thesis supervisors:
Prof. dr. eng. Valentin Cristea
Dr. eng. Paul-Alexandru Chiriţă

Student:
Mihai-Alexandru Ciorobea

BUCHAREST
2013

Table of Contents

List of Abbreviations

AI .....Artificial Intelligence
AST …..... Abstract Syntax Tree
DTO.....Data Transfer Object
IDE……....Integrated Development Environments

Today, when writing a software program, we have to achieve at least three major requirements: Availability, Reliability and Scalability. In order to have all of those we usually use Stability Patterns. This is a vital component in the software evolution. Structural and Design Patterns are the secret for a good and scalable application. When we write a program that will achieve all of these three conditions, we usually need to write pretty much code that is similar but not the same.

Writing so much code, just to meet the specified requirements, we may insert many bugs. If we could generate these structural patterns, we would have less manual written code and less possible bugs. As Bill Gates said: "Measuring programming progress by lines of code is like measuring aircraft building progress by weight", we try to make the software programs line code measurement irrelevant.

Keywords: Code Generation, Compilers, AST Tree, Rewrite AST Tree, Pattern Detection, Template, Algorithm and File Generation

1. Introduction

What is software? In theory software is “a program that enables a computer to perform a specific task”. It also can be described as “a collection of instructions that describe a task, or set of tasks, to be carried out by a computer”.

Over the time, society increasingly relies on software, at all levels. Nevertheless, software quality generally continues to fall short of expectations, and software systems continue to suffer from symptoms of aging, as they are adapted to changing requirements and environments. The only way to overcome or avoid the negative effects of software aging is by placing change and evolution in the center of the software development process.

A software program is the result of the programmers work, but software is regarded like a classical engineering product, and it is more complex than any other human artifact. Its complexity may come from at least two different ways. In theory the maintenance is treated as a lowly activity but in practice, the cost of the maintenance may rise up to 75%-95%.

Software is actually evolving due to business and technology drivers. The systems that don’t change continuously are actually dead. In software development, evolution is crucial, and this is because we think at new and better solutions while we create one.

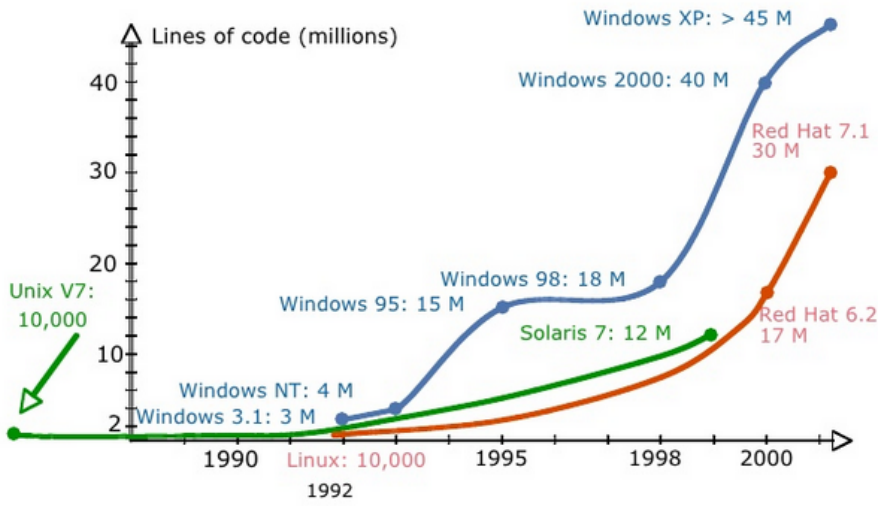

Fig. 1.1 How Large is Software

As we can see in the Figure 1.1 the big software applications or services, have grown over the years. When we are developing in a project at these dimensions is very hard for a programmer to write high quality code. This is happening because in big and complex software architecture is very hard to write code in a part of the application without affecting something else. A solution for this problem is usually resolved using a big and complex architectural design.

For implementing this architectural design, usually there are some very strict rules that need to be followed by all the programmers that are contributing to the software product. Among this rules, there are many implementations details that need to be respected to obtain the desired effect.

As we mentioned before, for this complex architectural design to be implemented we need to write some code that respects the required specification. The code written for a feature may vary from a couple of lines to thousands of lines. Even if it may not be so easy to write so much code, this is the best approach there is.

The drawback of this approach is the time spent on trying to write code without any bugs and trying to resolve any problems that may appear. The biggest problem is that even if we are very careful not to make any mistake, it's human to not be perfect, and because of this, much time can be lost on trying to resolve problems that will appear.

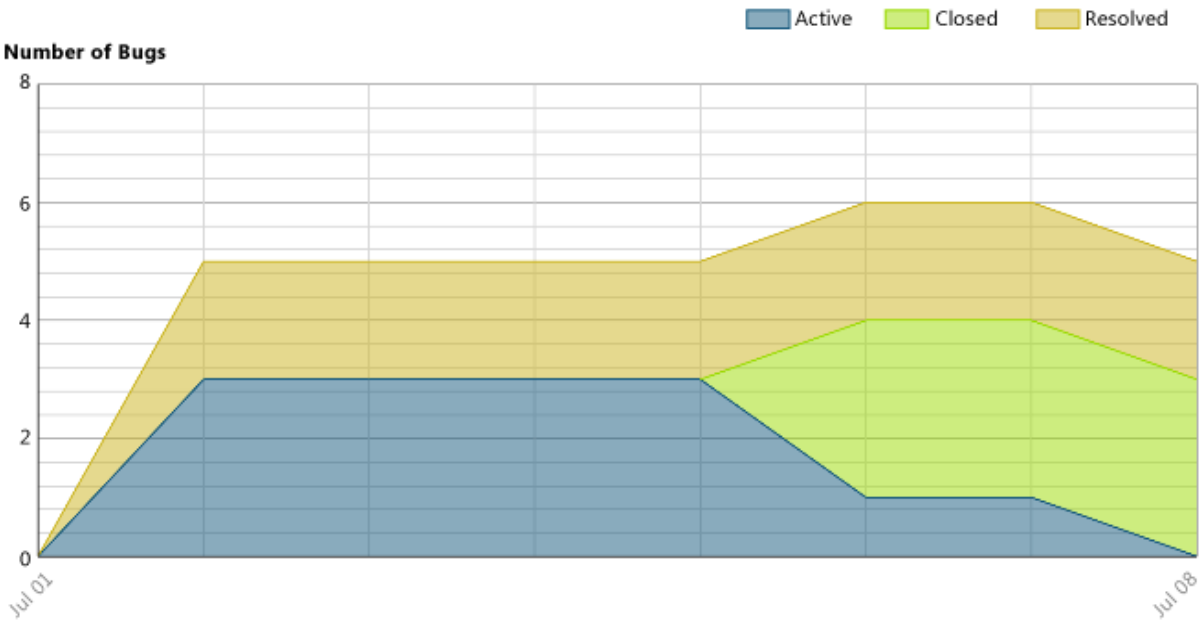

Fig. 1.2 Bug Status Report

All the bugs we are talking about are for the part of the code that is written only to respect the architectural design. In conformity with a Microsoft Bug Status Report, a programmer in inserting in average of 6 bugs per week. In figure 1.2 we can see the evolution of the bug status in a week.

Software systems are becoming more and more sophisticated, complex, and ubiquitous, while standards for software quality and productivity are becoming higher and higher. Developers constantly seek various techniques, tools, and practices to speed up software development without introducing additional software defects. One commonly used tactic for this purpose is to reuse prior knowledge existing in previous software projects.

Source code is one of the most directly reusable forms of prior knowledge, along with specifications, design documents, and test suites used for previous projects. Developers often copy code to quickly implement functionalities that have been implemented before. This is the reason why for many people, is something normal. This is usually done not only inside a project but even cross projects. According to [4,5,6,7], nowadays a large program may have more than 20% of its code taken from other small projects.

The main problem here is not the code reusing itself, but the cost and the challenges for management and maintenance. Lets take an example, the well-known

problem of multiple bug existence. The problem starts when we place the same piece of code in different portions on the application. After a period of time, is discovered a bug in that specific code. In this moment is more then complicated to find all the files where the code was placed then the actual fix.

Reducing maintenance cost is thus a strong motivation for detecting similar code in large projects, refactoring them to reduce redundancy and improve software quality, and tracking and managing their migration and evolution.

In addition, detecting, tracking, and managing similar code may also help improve software productivity, enabling faster code development by providing effective ways for developers to utilize prior knowledge embedded in existing code bases. Based on the frequency at which a piece of code is used, a large, searchable code library containing commonly used code can be constructed; the pieces of code in the library can have various granularities and modalities, from sequences of primary code statements to sets of functions and modules, from co-existent function calls to programming rules, design patterns and even software architectures.

When developers can reuse the code library for development of new software in an easy, error-proof way, instead of copying, pasting, or inlining code, new software can then be developed faster with fewer defects since the reused parts are built on the top of previous proven code bases and developers can be freed from many coding details and focus more on overall design issues and new features needed during the development.

Exactly this is the problem we are trying to reduce. We say reduce because is not possible to eliminate the chance of inserting duplicated code into an application, but we can reduce its number, by creating a tool that will help programmers to generate the code that usually it repeats itself but can't be reused. This kind of code is usually time consuming and it reduces the programmers productivity up to 10-50%, depending on the experiences and agility of the programmer.

After reading the book The Pragmatic Programmer, one of the arguments I found most interesting was "write code that writes code". This is very inspiring for exactly what we are trying to do here: to create a tool that will help programmers to generate code, using templates.

Templates are used to generate all kinds of text, including computer code. The last decade, the use of templates gained a lot of popularity due to the increase of dynamic web applications. Templates are tools for programmers, and implementations of template are most times based on practical experience rather than based on a theoretical background. This is one of the reasons that we are allowing each programmer to create his own.

In this paper we present a tool that will be useful to almost all the software solutions with big coding needs for scalability and availability. This solution will be used by engineers, to generate code, for the product they are working for. This is needed for reducing the workload and minimizing the risk that comes with big applications that require much code written per feature. This constraint comes with the benefit that the application will continue to be scalable and to have a considerable uptime.

In the next chapters we will present multiple approaches concerning the problem of code generation. This is a topic that has not been thoroughly before, or not in this form. We are now trying to generate code with business logic that can be used in later feature, referring the same subject, or even in maintenance. The problem of code generation was and is still challenged by compilers and IDEs.

Today, those programs are facing with the problem of stubs generation or runtime optimizations. The biggest difference between those tools and what we are going to do is that we don't need to generate code that can be used for any approach, but only for some. There also is a big advantage that we are exploring, the one of actually having code to base on. This is a big benefit we are counting on, the power of getting the essential code fraction for a code class.

Using these mechanisms we will actually expect to generate up to 12% of the Business Catalyst code. This will have a big impact on reducing the workload and minimizing the defect risk that occurs when we are improving a big and complex architecture that needs to stay scalable and maintainable.

1.1. Overview

Based on the code duplication and architecture design problem we started working at this code generator that is detecting patterns on a solution and is able to detect and extract similar code from it.

In this paper we first handled the problem with detecting the patterns, even if they are architectural patterns or just code that is written just for implementing the architectural design. For detecting the similarities between files there are multiple methods that can be applied. At the beginning we analyze all of them and after that we will chose one for the tool implementation.

After we detected all the similarity between those files, we can now extract a template what will help us to generate code in a later step. Those templates can also be written by hand, not just extracted. This use case is usually helpful when we know that we are going to create a design what will be used very often in other features.

The generation code part is the last part of our study, and this is the moment when all our efforts will be rewarded. Even when we already have the templates, the code generation problem is still not simple; we have to deal with renaming, replacing and more other problems that can appear in this kind of problems. As a last part, we are going to generate code for any database table; thing that is also time consuming.

When we write code inside a project we may write fractions of code that may already exist in the project. In this paper we will analyze how can we detect the duplicated code and how to extract it into a method.

Benefits. This generation tool can be used to reduce the number or hours spent on writing features that may already have some code written in the existing solution. This is a main common problem that occurs to the big architectural solutions. At this kind of problems we sometimes need to write fractions of template code, only to keep the scalability up and maintain the reliability at some standards.

A common use case for this application is a software solution that requires multiple features with similar logic but not reusable reason. This means that the code may be very similar but not the same and also may contain some similar logic but it cannot be written as a generic code.

2. Previous Work

Automatic programming identifies the type of computer programming in which the human programmers use some mechanisms to generate code at a higher abstraction level.

According to the Mitsubishi Electric Research Laboratories the “Automatic programming has been a goal of computer science and artificial intelligence since the first programmer came face to face with the difficulties of programming. As bets such a long-term goal, it has been a moving target -- constantly shifting to reflect increasing expectations.” Even if the first implemented program of this kind was written in early 50’s, there are a number of important development groups that are still researching in this domain. For example there is a project at MIT that focuses on automatic programming, named “Programmer’s Apprentice”.

A common approach to automatic programming is based on reuse of generic algorithms through views. A generic algorithm performs some task, such sorting a data structure, based on abstract descriptions of the data on which the program operates. A view describes how actual application data corresponds to the abstract data as used in the generic algorithm. Given a view, a generic algorithm can be customized for that view to produce a version of the algorithm that is optimal on the application data.

Another perspective on automatic programming can be that the computer is generating programs, usually based on specification that are higher-level and easier for humans to specify than ordinary programming languages.

One of the most used expressions for automatic programming is “Artificial Intelligence meets Compilers” and this is because the AI can write programs based on knowledge of algorithms, data structures and design patterns. AI techniques are needed to represent, find and instantiate the design patterns. In the analyzed expression we meet the word compilers; this is because compilers can generate and manipulate programs, using compiler techniques. Also the central representation of the program in the compiler’s view is the Abstract Syntax Tree (AST), on which all the compiler optimizations are made.

2.1. Code Generation based on Pattern Detection

Intelligent code generation is a delicate subject, which is and will be researched for a long time from now. Over the time it had more losses than winnings, but researchers still have a great interest in this subject.

The most approached problem was the one of generating code for the most common situations or from patterns. The first case, is usually used by Integrated Development Environments (IDE) for automatically generating a skeleton source file. Each programmer then customizes this file in order to resolve the problem he is facing.

Since the solutions for resolving problems are becoming much bigger and complex, the engineers in computer science have created a general reusable solution for commonly occurring problem within a given context. This solution is called software design pattern and it can be described as a template. Patterns are formalized best practices that programmers must implement themselves in the application. Because this approach has gained much popularity after the book "Design Patterns: Elements of Reusable Object-Oriented Software", a lot research was made over the years regarding this issue.

Nowadays we are trying to get to the next level, and that would be to generate software design patterns or even architecture patterns from the code we already have. This problem, according to Wikipedia and some other related documents, is a NP-complete problem.

Because of the problem complexity and generalization, we are going to divide it in three smaller problems.

One of the key objectives towards the automatic code generation on a large complex software system is the identification of common patterns in the code. A key issue in design recognition is to localize patterns of code that may implement a particular plan or algorithm.

2.1.1. Pattern Detection

A number of research teams have developed tools and techniques for localizing specific code patterns.

As described in [1] there are many tools that are very efficient in localizing patterns but do not provide any way for partial and hierarchical matching. Moreover, they do not provide any similarity measure between the pattern and the input string.

Other tools have been developed to browse source code and query software repositories based on structure, permanent relations between code fragments, keywords, and control or data flow relationships. Such tools include CIA, Microscope, Rigi, SCAN, and REFINE. These tools are efficient on representing and storing in local repositories relationships between program components. Moreover, they provide effective mechanisms for querying and updating their local repositories. However, they do not provide any other mechanism to localize code fragments except the stored relations. Moreover no partial matching and no similarity measures between a query and a source code entity can be calculated.

One of the key objectives towards the design recognition of a large complex software system is the identification of common patterns in the code. These common patterns may reveal important operations on data types, identify system clusters, help reorganize the system in an object-oriented way, and suggest points where potential errors can be found.

There is an approach where the source code is parsed and represented as an annotated Abstract Syntax Tree (AST) (chapter 2.2.1.). Nodes in the AST represent language components (e.g. statements, expressions) and arcs represent relationships between these language components. For example an IF-statement is represented as an AST node with three arcs pointing to the condition, the THEN-part, and the ELSE-part.

Pattern detection systems use a variety of methods to localize a code fragment given a model or a pattern. One category of such tools uses structure graphs to identify the "fingerprint" (chapter 2.2.2) of a program [2].

There is a third method for detecting pattern (chapter 2.2.3.) and that is usually to convert the program into a stream of tokens (thus ignoring easily changeable information such as indentation, line breaks, comments etc.) and then comparing these token streams to find common segments.

Detecting the design patterns in a big software system helps all the engineers to understand the product design and architecture. Each pattern solves design problems that occur in everyday software development. The detection of design patterns during the process of software reverse engineering can provide a better understanding of the software system. In our approach (chapter 2.2.4.) we will use static and dynamic code analysis to detect design pattern.

2.1.2. Template extraction

According to Wikipedia, a template may mean a pre-developed page layout in electronic or paper media used to make new pages with a similar design, pattern, or style. In Computer Science, template may mean standardized non-executable file type used by computer software as a pre-formatted example (template file), or a programming technique used by a compiler to generate temporary source code (template meta-programming) or even an object-oriented design pattern (template method).

To achieve our goal, to generate code, we will need to extract a template from multiple input files. Meaning, we will need to use the result of the pattern detection, and transform it into a template that will help us to generate code at the next step.

Even if it sounds pretty easy, this step is almost as hard as the previous step. At the beginning we will need to define a language for templateing. There are many languages for this kind of task, but all of them are used for automatically generating a skeleton source file. In our approach we already have the files containing the source code, and all we need to do is remove, rename or parameterize the existing code.

In our first approach (chapter 2.3.1.) we extract a template file, by applying the pattern detection on the input files, then the result is compared with every input files. By this, we can analyze the result and we can realize which part of the code is matching and what can be templated or not.

This method for extracting templates is very good and efficient when we already have the code and we only want to make it easier for future developing. Consider this, we now think that we know that some pattern we are going to write is a very used template. To avoid writing it multiple times for extracting a template, we've documented the templateing language (chapter 2.3.2.) for building the template from scratch.

In order to maintain the architecture design of the big and complex software solutions, and because is basically too hard for any heuristic method to detect this type of design, we will provide some predefined patterns used for writing code using the required architecture design (chapter 2.3.3.)

2.1.3. Code Generation

A code generator [3] is a program being able to generate code based on an input specification, see Figure 2.1. Such programs are written to automate repetitive work, but also when a system needs to instantiate textual representations of a model, like HTML pages in web applications. Code generators are a subclass of meta-programs; the set of programs analyzing or manipulating other programs. A code generator can be written in an ad-hoc fashion or using sophisticated methodologies from compiler research.

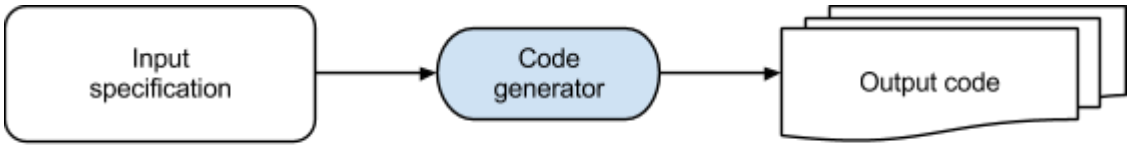

Fig. 2.1. A code generator

Writing code generators is not a trivial task. First, the input language has to be designed (template language). This language should provide a complete vocabulary at the right level of abstraction for the problem domain. Defining this vocabulary requires an extensive knowledge of the problem domain. This requirement is not limited to writing code generators, but also for writing understandable and reusable domain specific libraries.

Second, writing and understanding a code generator is hard, as it contains code artifacts executed at different stages. It contains code executed at generation time, *meta-code*, and code used as building block for the output code, the *object code*. . A programmer has to be aware of the different execution stages of the different code artifacts. More confusing, artifacts belonging to different execution stages are mixed

We concern improving the reliability of template based code generators. Text-templates are known from their use in instantiating HTML in web applications. As a result of the popularity of templates in web applications, numerous template evaluators are designed for instantiating HTML. Besides generating web content, text-templates can be used for generating all kinds of unstructured text, like e-mails or code. Figure 2.2 shows the four artifacts involved in a text-template based generator (chapter 2.4.1.).

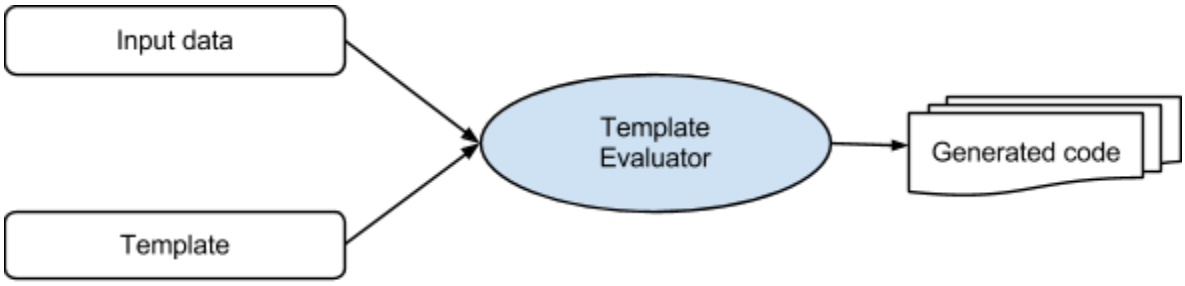

Fig. 2.2 Template based generator

2.2. Pattern Detection

One of the key objectives towards the design recovery of a large complex software system is the identification of common patterns in the code. These common patterns may reveal important operations on data types, identify system clusters, help reorganize the system in an object-oriented way, and suggest points where potential errors can be found.

According to [8] there have been many studies that define similar code differently. In software engineering, similar code is also known as cloned code (or code clones), reflecting the fact that much similar code occurs due to the practice of copying and pasting during software development process. Code clones introduced by copying and pasting are mostly of similar appearance since they are derived from the same origin.

As we mentioned before, there have been quite a number of detection techniques that target different kinds of code clones classified in the spectrum in Figure 2.1.

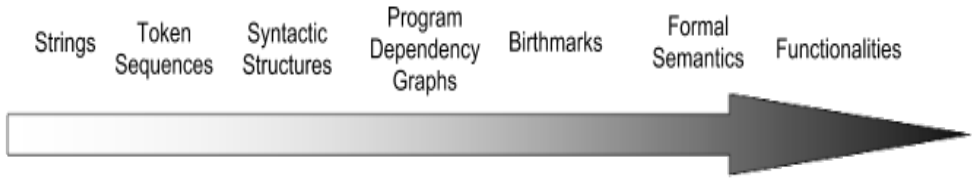

Fig. 2.1 Semantic Awareness of Code Clones

When analyzing for pattern detection we need to take in consideration:

- Verbatim copying,
- Changing comments,
- Changing whitespace and formatting,
- Renaming identifiers,
- Reordering code blocks,
- Reordering statements within code blocks,
- Changing the order of operands/operators in expressions,
- Changing data types,
- Adding redundant statements or variables, and
- Replacing control structures with equivalent structures.

All the descriptions we have mentioned so far appear to be short and rather incomplete. Similarity between two files is not based on the entire files, but rather on the source code fragments they contain.

2.2.1. Scalable and Accurate Tree-based approach

Tree method of detecting similar code, is one of the most accurate methods there is. Many previous studies have developed various techniques and tools for clone detection. Over time there were found multiple techniques to approach this problem from the tree view: there is a method which is token-based and for example CCFinder [5] and CP-Miner[10] are using it to find similar code blocks. Also there is second method, used by CloneDR [11,12], which is a syntax tree-based method.

According to [8] “they either do not scale to large code bases due to expensive algorithms for tree-comparison or are not robust against minor code modifications due to their token-based nature.”

Parse Tree Comparison Algorithms. Parse tree comparison algorithms implement comparison techniques that compare the structure of files [13]. One way is to represent each file as a parse tree, and then to use string representations of parse trees it compares file pairs for similarity. The comparison process begins by converting each source-code file into a string of tokens where each token is replaced by a fixed set of tokens. After tokenization, the token streams of the files are divided into sections and their sections are aligned. This technique allows shuffled code fragments to be detected.

We can use string-matching algorithms to compare file pairs represented as parse trees. As with most string algorithms, we can search for maximal common sub sequence of tokens. Using this method we can detect similar files that contain common modifications such as name changes, reordering of statements, and adding or removing comments and white spaces [14].

Also there is a new method for tree comparison, method used by Brass system [15]. In this new algorithm each program is represented as a structure chart, which is a graphical design consisting of a tree representation of each file. The root of the tree specifies the file header, and the child nodes specify the statements and control structures found in the file. The tree is then converted into a binary tree and a symbol table (data dictionary), which holds information about the variables and data structures used in the files.

Combining the algorithms we can obtain a more accurate result. In the beginning we use two algorithms for comparing the structure trees of files and the third algorithm involves comparing the data dictionary representations of the files. Initially, similar file pairs are detected using the first comparison algorithm. Thereafter, the second and third comparison algorithms are applied to the detected file pairs.

Tree comparison involves using algorithms more complex than string matching algorithms. From this it can be assumed that systems based in tree comparison algorithms are slower than systems based on string matching algorithms [13].

Numerical vectors in the Euclidean space. In this chapter we will present an efficient algorithm used in identifying similar subtrees that are obtained from source code files. The algorithm is based on a new and unique characterization of subtrees with numerical vectors in the Euclidean space $\mathbb{R}^n$ and an efficient hashing algorithm to cluster these vectors into the Euclidean distance metric. In the end all the subtrees in one cluster are considered similar. The algorithm is also language independent, because all he uses is the syntax trees generated from a language.

According to [8], tools that implement this algorithm for code similarity finding are more accurate than other tree-based algorithms and as scalable as a token-based algorithm.

The main idea is to create particular characteristic vectors to approximate structural information within trees and then adopt Locality Sensitive Hashing (LSH) [16] to efficiently cluster similar vectors.

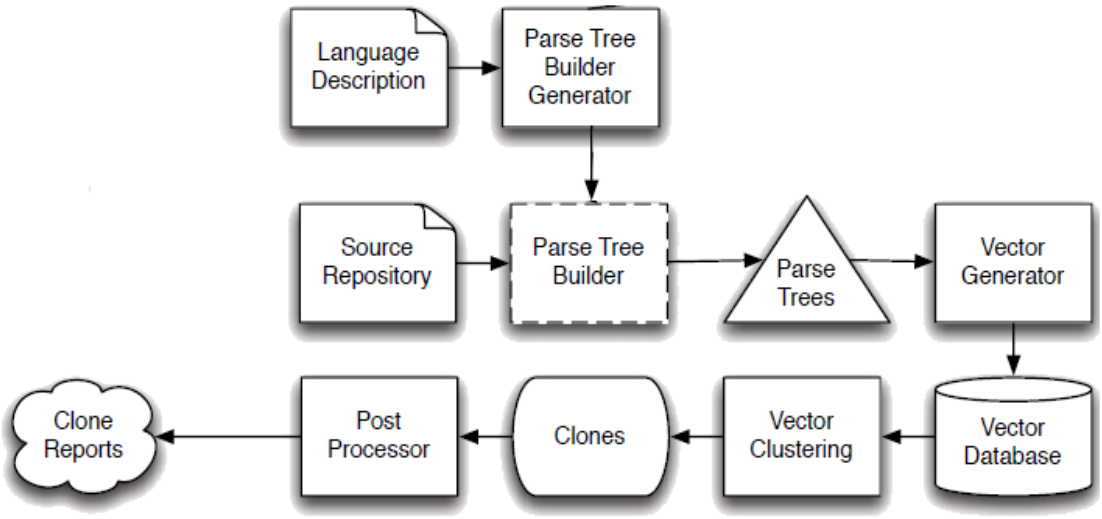

Fig. 2.2 High Level Architecture

Figure 2.2 shows the steps that need to be taken to detect the similarities between source code files:

- A parser is generated from a formal grammar of a language.
- The parser translates sources files in a program or a set of programs into parse trees.
- The parse trees are processed to produce a set of characteristic vectors whose dimensions are fixed for a particular programming language and capture the syntactic information of parse trees.
- The vectors are clustered by special hashing algorithms; close ones are put into the same cluster.
- Additional post-processing heuristics, such as filtering out clusters with tiny or overlapping code fragments, are used to generate code clone reports.

Characteristic vectors are introduced to capture the structural information of trees. This is the first key step in our algorithm. Each vector represents a subtree and is equivalent to a point (c_1, c_2 …, c_n) in the Euclidean space, where each c_i represents the count of occurrences of a specific tree pattern in the subtree.

Because we are looking for similar code portions, we will not consider all the nodes in parse trees to be essential. This is the reason we are distinguishing between **relevant and irrelevant** nodes. For example irrelevant nodes include C tokens '[' or ']' used in array expressions. Irrelevant nodes do not have an associated pattern or dimension in our vectors.

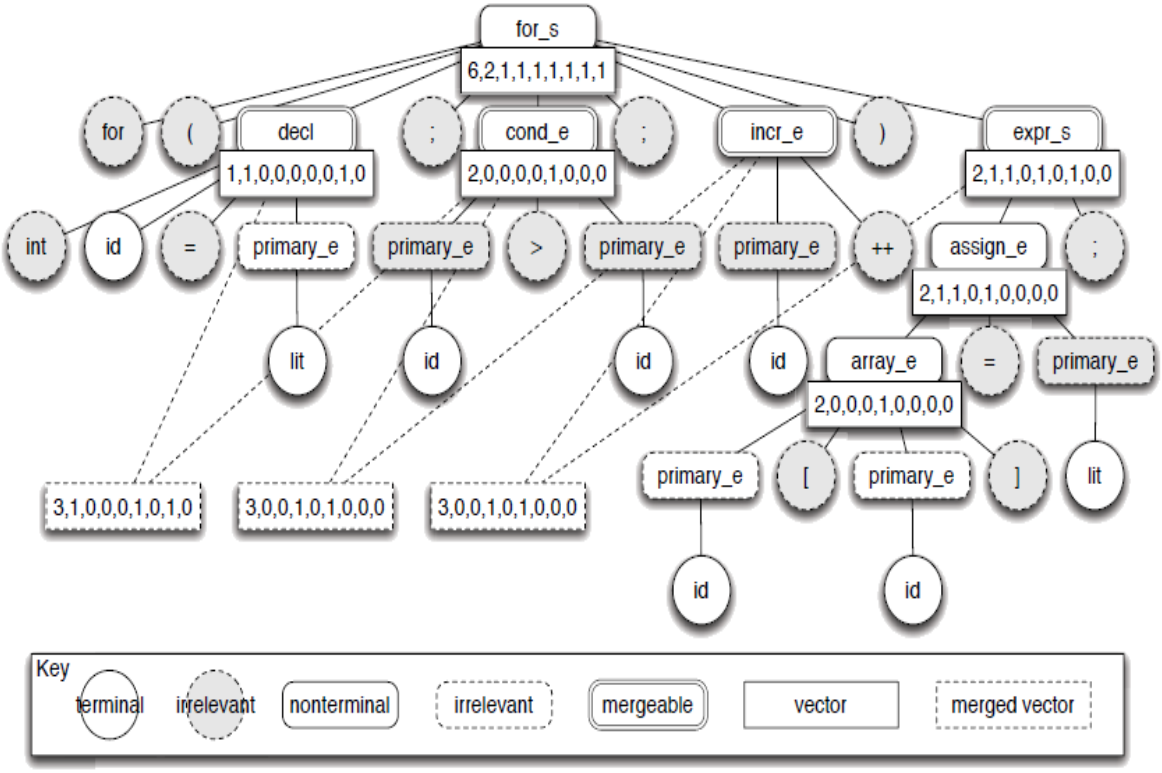

Figure 2.3 A sample parse tree with generated characteristic vectors.

Each element in the characteristic vector represents the occurrence counts of the relevant nodes usage like: identifiers, literals, assignments, increments, array usages, conditions, expressions, declarations or loop usages. This is the reason why the subtree under "decl" node has (1, 1, 0, 0, 0, 0, 0, 1, 0) as a characteristic vector. We create this vector with a post-order traversal of the parse tree by summing up the vectors for children with the vector for the parent's node.

For avoiding small reports we can set a minimum value for the vectors that will be analyzed.

One of the biggest problems is when we need to analyze subtrees, and we don't know how big the optimal subtree needs to be. Because developers often insert code fragments within some larger context. Differences in the surrounding nodes may prevent the parents from being detected as clones. For this problem we use a second phase of characteristic vector generation, called **vector merging**, to sum up the vectors of certain node sequences.

In this phase, a sliding window moves along a serialized form of the parse tree. The windows are chosen so that a merged vector contains a large enough code fragment. In our example we merged vectors of declaration with condition and we obtained the vector (3, 1, 0, 0, 0, 1, 0, 1, 0) for the combined code fragment.

This is very important because, depending on which nodes we chose to merge, is affecting which fragments from source code will be considered in the code analysis. Roots of expression trees, likely to be units for copy pasting, are often good choices for merging vectors. We call such chosen nodes mergeable nodes. The four children of the for statement are used as **mergeable nodes**. It is not necessary for mergeable nodes to be on the same level.

```
for( int i= 0; i < n; i ++ )
    x[i] = 0;
```

```
for( int i = 0; i < n; i ++ )
    y[i] = " ";
```

After we have selected the characteristic vectors, our algorithm clusters similar characteristic vectors using Euclidean distances to detect similar code. If applying the described algorithm on those two code fragments we will obtain the same characteristic vector (6, 2, 1, 1, 1, 1, 1, 1, 1), and we will consider them clones. As we can see in our first example the subtree rooted at expr_s illustrates the need for post-processing. When a particular subtree has a low branching factor, the vectors for a child and its parent are usually very similar and thus likely to be detected as similar.

Because the number of generated vectors can be large, an efficient **clustering algorithm** is needed. This step is made right after we employ the post-processing phase to filter such spurious overlapping clones.

2.2.2. N-Gram-Based approach

A program is processed to produce a token sequence, which is scanned for duplicated token subsequences that indicate potential code clones. Compared to string-based approaches, a token-based approach is usually more robust against code changes (e.g., comments and spacing). CCFinder [5] and CP-Miner [7] are perhaps the most well-known among token-based techniques, where CCFinder employes a suffix-tree algorithm [9] to find similar token subsequences and CP-Miner uses a frequent subsequence mining algorithm in data mining, CloSpan [17,18], to find repeatedly occurring subsequences even with gaps. Some software plagiarism detection tools (e.g., Moss [2] and JPlag [19]) also use token-based techniques to search for similar code fragments.

A k-gram is a contiguous substring of length k. Divide a document into k-grams, where k is a parameter chosen by the user. We will then hash each substring, giving it a hash number. This will be then used for a document fingerprints.

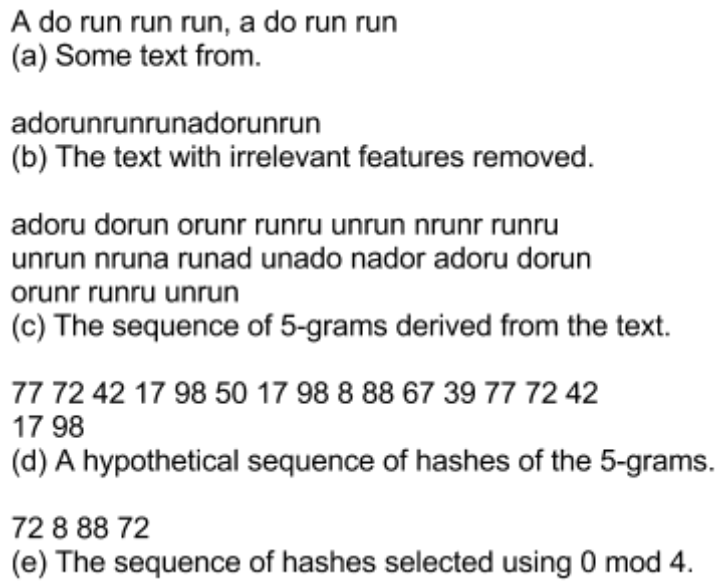

Fig. 2.4 Fingerprinting some text

For the example in the figure 2.4 we can see all the steps for fingerprinting some sample text. Here we can see that the white spaces are not taken in consideration and they are removed from the beginning. After this we obtained a long string that will be transformed into 5-grams. The number 5 is for our example the optional number, because if we use a smaller number, we would analyze inside words similarity. If we try to use bigger k-gram strings, then we will analyze the similarity of the expressions. After we've split the text into k-grams we will now hash each string, obtaining multiple hash numbers specific for the given text, named document's fingerprints. In all practical approaches, the set of fingerprints is a small subset of the set of all k-gram hashes.

For efficiency, only a subset of the hashes should retained as the document's fingerprints. In all practical the most used approach is to choose all hashes that are 0 mod p, for some fixed p. This is not CPU intensive and it reduces the set of the fingerprints to 1/p from the total hashes. Meaningful measures of document similarity can also be derived from the number of fingerprints shared between documents [2].

The drawback of the approach is that it doesn't guarantee that all of the similarity are detected: a k-gram shared between documents is detected only if its hash is 0 mod p. An efficient algorithm that guarantees that at least part of any sufficiently long match is detected uses a window of size w to be w consecutive hashes of k-grams in a document. By selecting at least one fingerprint from every window our algorithm limits the maximum gap (the distance between consecutive selected fingerprints) between fingerprints. In fact, our algorithm is guaranteed to detect at least one k-gram in any shared substring of length at least $w + k - 1$.

We define an algorithm as local if, for every window of w consecutive hashes $h_i,\ldots, h_{i+w-1}$, the algorithm decides to select one of these hashes as a fingerprint and this choice depends only on the window's contents $h_i,\ldots, h_{i+w-1}$.

Document fingerprinting has worked extremely well in this setting. Because the basic idea of hashing k-grams makes minimal assumptions about the form of the input, it is easy to incorporate fingerprinting for a new data format without disturbing the underlying hashing engine.

Because the winnowing algorithm is one of the best approaches in this situation, we will talk more about it. We give an upper bound on the performance of winnowing, expressed as a trade-off between the number of fingerprints that must be selected and the shortest match that we are guaranteed to detect. In this case, we expect to detect any similarity that is at least as long as the guarantee threshold (t), and not to detect any matches shorter than the noise threshold (k).

We guaranty that the match we detect has at least the length k, by using the k-grams. The larger k is, the more confident we can be that matches between documents are not coincidental. On the other hand, larger values of k also limit the sensitivity to reordering of document contents, as we cannot detect the relocation of any substring of length less than k. This is the reason why we need to choose k to be the minimum value that eliminates coincidental matches.

If we have *n* hashes and $n > t - k$, then at least one of the element must be chosen as fingerprint in order to guarantee detection of all matches of length at least t. This suggests that the window size be $w = t - k + 1$.

According to [2] the optimal hash to represent as a fingerprint is the minimum hash value and if there is more than one hash with the minimum value, select the rightmost occurrence. In the end all of those selected hashes represents the fingerprint of the document. It has been remarked that the algorithm works better if when we select a hash as a minimum value, we shouldn't extract the hash again if it is the minimum value in the window. See figure 2.5

A do run run run, a do run run
a) Some text.

adorunrunrunadorunrun
b) The text with irrelevant features removed.

adoru dorun orunr runru unrun nrunr runru unrun nruna runad unado nador adoru dorun orunr runru unrun
c) The sequence of 5-grams

77 74 42 17 98 50 17 98 8 88 67 39 77 74 42 17 98
d) A sequence of hashes of the 5-grams.

(77, 74, 42, **17**), (74, 42, 17, 98),
(42, 17, 98, 50) (17, 98, 50, 17)
(98, 50, **17**, 98) (50, 17, 98, **8**)
(17, 98, 8, 88) (98, 8, 88, 67)
(8, 88, 67, 39) (88, 67, **39**, 77)
(67, 39, 77, 74) (39, 77, 74, 42)
(77, 74, 42, **17**) (74, 42, 17, 98)
e) Windows of hashes of length 4.

17 17 8 39 17
f) Fingerprints selected by winnowing.

[17,3] [17,6] [8,8] [39,11] [17,15]
g) Fingerprints paired with positional information

Fig. 2.5 Winnowing sample text

Since a local minimum value will appear in multiple windows and is very likely to remain the minimum hash in adjacent windows. Thus, many overlapping windows select the same hash, and the number of fingerprints selected is far smaller than the number of windows while still maintaining the guarantee. In our example we can see that from all that windows we only selected five hashes. This is a major step in reducing the fingerprint but still not modifying the guarantee or noise threshold.

Also in many applications it is useful to record not only the fingerprints of a document, but also the position of the fingerprints in the document. For example, this info is used to show the parts where the analyzed documents are similar, figure 2.5 g).

2.2.3. Rewrite code

In this approach we can easily distinguish several approaches to detect source code pattern detection. We can identify two main categories of detection: feature comparison and structure comparison.

Feature comparison is an early approach to detect source code similarity; its method is based on the notion of a feature vector. The software metrics for each program is mapped to a point in an n-dimensional Cartesian space. The similarity score for two compared programs is estimated using various metrics like: number of unique operators, number of unique operands, total number of operators, total number of operands, average number of characters per line, the number of comment lines, the number of indented lines, the number of blank lines. Using all of those feature comparison categories we can create three profiles: Physical profile (takes in comparison the physical attributes, such as the number of lines, words, characters), Halstead profile (Characterizes a program based on its token types and frequencies) and Composite profile (this profile combines the other two profiles). The final score is made calculating those profiles for each program, and then the numbers obtained are normalized. The final step is to calculate the Euclidean distance between their profiles. Unfortunately, the accuracy of this method is moderate and this is due to the feature vectors that can hardly have good performance, when talking about big programs with too much structural information for the vectors to handle.

A more recent method for detecting source code similarity is name structure comparison. The better approach is due to a structure comparison rather than just summary indicators. There are also hybrids systems that compare programs using structure and metric comparison [20, 21], others rely on structure comparison alone.

In the latter case, the systems, are converting the input program into a stream of tokens (ignoring the irrelevant structure items, like white spaces, line breaks or comments). Also in this approach we can identify two phases:

1. All programs are parsed and converted into token strings.
2. Those token strings are compared in pairs for determining the similarity of each pair.

Converting the programs into token strings. This is an important step in the similarity process because here is the moment when we can add more semantic information to be represented in the token string, for example we can distinguish from BEGIN_METHOD token instead of OPEN_BRACE token.

In this step of the similarity detection we show select only the essence of a program, the part that characterize what the program is doing with its algorithm. For example we should never choose to reduce to a token the white spaces or comments. For the selected tokens we will store the line where it originates from, this because we will need to represent the similar parts of the application. See the example from figure 2.6.

	Source code	Generated tokens
1	public class MyClass {	BEGINCLASS
2	public void myMethod(Object arg) {	VARDEF, BEGINMETHOD
3	int count = 0;	VARDEF, ASSIGN
4	while (count < 100)	APPLY, BEGINWHILE
5	count ++;	ASSIGN, ENDWHILE
6	Sytem.out.println("Finish .");	APPLY
7	}	ENDMETHOD
8	}	ENDCLASS

Fig. 2.6 Example for source text and generated tokens

Comparing two token strings. One of the methods for comparing two token strings is "Greedy String Tiling". This algorithm was introduced by Michael Wise[22] and involves that when comparing two strings A and B, we need to find a set of substrings that are the same and they satisfy the following rules:

1. Any token of A may only be matched with exactly one token from B. This rule implies that it is impossible to completely match parts of the source text that have been duplicated in a plagiarized program.
2. Substrings are to be found independent of their position in the string. This rules implies that reordering parts of the source code is no effective attack.
3. Long substring matches are preferred over short ones, because they are more reliable. Short matches are more likely to be spurious.

Because of the last rule, we can say this algorithm is a greedy algorithm, and on a closer look we can identify two phases:

1. In this first phase, we are searching for the biggest contiguous match and this is done using 3 imbricated loops. First two loops are checking in all the strings A if they are identical with a string B. If there is a match a third loop will try to expand this match as much as possible.
2. Phase 2 will mark all matches of maximal length found in the first phase. All the tokens used in this match will be marked so they will not be used in a further match of phase 1 for a subsequent iteration. This part is the part that guarantees that any token will only be used in one match. It is possible that some of the matches may overlap, but will respect that an token is used only once, and the first match is going to use the token, the others will be ignored.

We will repeat those two phases until there is no match found. We guarantee that the algorithm will terminate and no more matches will be found because the maximal length will decrease at each step. Also we will need to set a minimum token length match, for avoiding most of the spontaneous similarity. This will be named "Minimum Match Length". See the pseudocode of the algorithm in figure 2.7.

```
1   Greedy-String-Tiling(String A, String B) {
2     do{
3       maxMatch = MinimumMatchLenght;
4       matches = {};
5       foreach unmarked token A_a in A {
6         foreach unmarkerd token B_b in B {
7           j = 0;
8           while ( A_a+j == B_b+j )
9             j++;
10          if ( j == maxMatch )
11            matches = matches U match(a, b, j);
12          else if ( j > maxMatch ) {
13            matches = { match(a, b, j) };
14            maxMatch = j;
15          }
16        }
17      }
18      foreach match( a, b, maxMatch) ϵ matches {
19        for j = 0 to maxMatch -1 {
20          mark( A_a+j );
21          mark( B_b+j );
22        }
23        tiles = tiles U match(a; b; maxmatch);
24      }
25    } while ( maxMatch > MinimumMatchLength );
26  }
```

Fig. 2.7 Greedy String Tiling

The triple match (a, b, l) denotes an association between identical substrings of A and B, starting at positions A_a and B_b respectively, with a length of l. As we can see the first phase start at line 5, where we compare each unmarked token from A with each unmarked token from B. Here is where we save the maximum match length with the matches array. The phase 2 starts at line 18 where we mark all the used tokens so we will not use them again.

According to [19] the similarity measure should reflect the fraction of tokens from the original programs that are covered by matches. There are two sensible choices: If we want a similarity of 100% to mean that the two token strings are equivalent, we must consider both token strings in the computation. If, on the other hand, we prefer that each program that has been copied completely (and then perhaps extended) will result in a similarity of 100%, we must consider the shorter token string only. We choose the first variant here, which results in the following similarity measure sim:

$$\mathrm{sim}(A, B) = \frac{2 * \mathrm{coverage}(\mathrm{tiles})}{|A| + |B|}$$

$$\mathrm{coverage}(\mathrm{tiles}) = \sum_{\mathrm{match}(a,b,\mathrm{length}) \in \mathrm{tiles}} \mathrm{length}$$

Run time complexity. In the worst case, lets consider that |A| = |B| = n, because this will not affect the final theoretical complexity. Also in the worst case, the minimum match length (MML) to be 1 and the match is always located at the very end of the remaining string. In this case, it is easy to say that the time complexity is $O(n^3)$.

Run time optimizations. Even if the worst case complexity can not be reduced, it is possible to obtain an average complexity of O(n). This huge difference is obtained with an idea from the Karp-Rabin algorithm [23]. This algorithm tries to find all occurrences of a short string (named pattern P) in a longer text (T) by using hash function. For this to be done we need to calculate all the hashes for all the substrings with length |P|. This can be done in linear time by using a hash function H that is able to compute the value of $h(T_tT_{t+1}...T_{t+|P|-1})$ from the values of $h(T_{t-1}\ T_t\ ...\ T_{t+|P|-2})$, T_{t-1} and $T_{t+|P|-1}$. The complexity of this algorithm in practice is almost linear.

There is also a method that uses Dynamic Programming to determine the substring similarity from source code, but that is not the scope of the paper.

2.3. Template Extraction and Generation

According to Wikipedia, a template may mean a pre-developed page layout used to make new pages with a similar design, pattern or style (figure 2.8). A template can be used for assuring a custom standard layout or look and feel on multiple pages. The main advantage is that when we use a template, we need to update only one file, and all the pages based on it will automatically update as well.

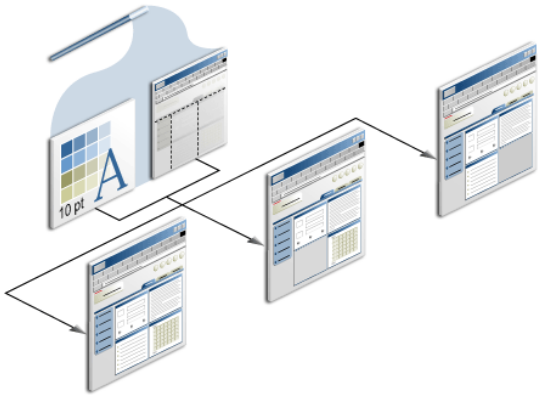

Fig. 2.8 Template file.

In computer science, a template may have many possible means. We are going to talk about a file that is a generalization of a source code and to explain we will take an example. In figure 2.9.a.we can see a source code template, which is going to be instantiated by assigning a name to the parameter variable. The code obtained is in figure 2.9.b. This example is a simple one, in real application we may need to set values or names to multiple tokens from the template file.

```
1   template <int N> struct Factorial {
2           return (N==0)? 1 : n * factorial(N-1);
3   };
```
a)

```
unsigned int factorial(unsigned int n) {
        return (n==0)? 1 : n * factorial(n-1);
}
```
b)

Fig. 2.9 Template example

Because is very easy to write code with bugs and because there is lots a code that may look the same in a common application, there are lots of softwares that are using code templates for code generation. For example in Visual Studio, a T4 text template is a mixture of text blocks and control logic that can generate a text file. The control logic is written as fragments of program code in Visual C# or Visual Basic. The generated file can be text of any kind, such as a Web page, or a resource file, or program source code in any language. There are two kinds of T4 text templates:

- **Run time T4 text templates**('preprocessed' templates) are executed in the application to produce text strings, typically as part of its output. See an example at figure 2.10.

```
<html><body>
 The date and time now is: <#= DateTime.Now #>
</body></html>
```

Figure 2.10 Run time text template

- **Design-time T4 text templates** are executed in Visual Studio to define part of the source code and other resources of your application. See an design template in figure 2.11 and the resulted code after instantiating the template in figure 2.12.

```
<#@ output extension=".txt" #>
<#@ assembly name="System.Xml" #>
<#
 System.Xml.XmlDocument configurationData = ...; // Read a data file here.
#>
namespace Fabrikam.<#= configurationData.SelectSingleNode("jobName").Value #>
{
  ... // More code here.
}
```

Figure 2.11 Design text template

```
namespace Fabrikam.FirstJob
{
  ... // More code here.
}
```

Figure 2.12 Source code obtained from design template

Also there are many similar applications with the same usage in the software market. One of the best there are is a tool named Code Smith. This tool is a template driven **Source Code Generator** used to automate the creation of common application source code for any text-based language.

All of those tools and applications can help us to produce higher-quality, more consistent code in less time and enables software developers to efficiently:

- Reduce repetitive coding.
- Generate your code in less time with fewer bugs.
- Produce consistent code that adheres to your standards.
- Create your own custom templates for any language.

2.3.1. Automatic extraction using Pattern Detection

To achieve our goal, to generate code, we will need to extract a template from multiple input files. Meaning, we will need to use the result of the pattern detection, and transform it into a template that will help us to generate code at the next step.

This is a big step for our goal, because where is where we decide what is the structure of the final code and after this we only can change small parts of it when we instantiate the template.

Because this operation is made after the pattern detection step, we already have the similar code as our input, also we have the original code. Using those inputs we can extract the exact original code and on it we can make it more general replacing in the template the variable names or the loops indicators and more others input tokens.

In this approach we already have the similar code parts, what we only need to do, is to compare the original code lines and extract from them the template for variables. The figure 2.13 shows the generalization made by the automatic extraction on the variables names for “product” and “site” method extraction.

```
1   List<#NAME1#> get#NAME1#sForUserId(int userId) {
2       List<#NAME1#> rez = new ArrayList<#NAME1#>();
3       List<#NAME1#> #NAME1#List = get#NAME1#s();
4       for(#NAME1# #NAME1#: #NAME1#List ) {
5           if ( #NAME1#.getUserId == userId) {
6               rez.add(#NAME1#);
7           }
8       }
9       return rez;
10  }
```

Fig 2.13 Generated template

We can see that the template variable “#NAME1#” was created from two other variables: “Site” and “Product”. As we observe the generated template has only one template variable, one that is used in assigns or method names.

2.3.2. Manual Writing

There is always a chance that we may want to modify the resulted template file, even if we only want to change we predefined variable name, or to add / remove code from the template. This is a good reason that the generated files need to be editable.

Also if you know that the code you are going to write will be used in a later feature, you may want to write a template file from the beginning so that your colleagues would use it. This use case is usually used for architectural features. All of the tools on the market have this feature, being the main usage of the applications. For us, this editable option is only to let the user to make small changes to the template.

2.4. Code Generation

The common approach for generating HTML pages in web applications is a text-template. But, besides generation HTML, text-templates can also be used in generating all kinds of unstructured text, like e-mails or code. In a common template based generator, we will have four components:

- Template
- Input data
- Template evaluator
- Generated code

A definition provided by Parr is telling us that : “A template is an output document with embedded actions which are evaluated when rendering the template.”. Using this definition we can identify two entity types in a code template. One is a plain text, that will not change when the template is instantiated and the second one is called placeholder and it is a syntactical entity that will be changed when the template is used. Usually the placeholder is the part of the template that differ in the files that were received as an input.

According to [3] the process delegated with the automatic processing of the template is called template evaluator. This is a process that parses the template and searches for placeholders. It also replaces all the placeholders with input text received from the user that is generating code. The combination of template and template evaluator constitutes a code generator.

2.4.1. Template usage

Ones we have the template, and we decide to instantiate a template, we need to provide an text instance to all the unique placeholders from the chosen template. Code generator is a software development tool to help you get your job done faster. Technically speaking it is a template driven Source Code Generator that automates the creation of common application source code for any language. A great motto for this is: “Your code. Your way. Faster”.

The figure 2.15.a shows us the original template code. After we instantiated the two variables #ID1# and #ID2# to the values “index” and “stop” we obtained the code from figure b. Also in this step we can change the template structure, by removing or adding other text parts. We can even add placeholders, but in order for our tool to identify the new placeholders we need to announce that we added new variables in our template.

```
1  for( int #ID1# = 0; #ID1# <= #ID2#; #ID1# ++){
2          //code here using  #ID1# and #ID2#
3  }
   a)
```

```
for( int index = 0; index  <= stop; index  ++){
        //code here using  index and stop
}
b)
```

Fig. 2.15 Code generation from template

2.4.2. Database Model

Another great feature for code generation is the possibility to create repositories or queries ones we have created a table in the database. This functionality is provided by many other tools or plugins. The main problem with those applications is that they are providing the solution for a common approach. This is usually appropriate for the many software solutions. The problem appears when we are not developing in the most common technique. One example is when we don’t use an ORM for retrieving the information from the database. We can use a repository layer that will interact with the database directly and it will provide the data model to the service layer.

3. Code Generation based on Code Rewriting Techniques

For a team with a big architectural design, a tool that will help improve the productivity is always wanted. The best part of the code generation program is that once we have a template extracted, all our colleagues can use it and the productivity is getting higher and higher.

3.1. Selecting the Algorithm

For the final algorithm we had to choose an algorithm from Parse Tree Comparison Algorithms, Numerical Vectors in the Euclidean Space, N-Gram Based Algorithm and Code Rewrite.

There are pros and contras al all of them. First approach is one of the best, more accurate than others and capable of detecting the tiniest change in the parsing tree. It has two big problems; the first one is about complexity and time consuming. But the biggest problem is to detect small changes that affect the entire tree. The Numerical Vectors approach is the most scalable and method there is. It combines the power of a compiler and Artificial Intelligence algorithms using clustering and other adaptive grouping methods. This method has its problems too, for example to be sure that we have the good and reliable result we need to run the algorithm with multiple configurations and multiple times.

The N-Gram based approach is a very good method, which is already used for example by Moss [2] or JPlag [19]. The algorithm is more than reliable when talking about False Positive but it has no guaranty for False Negative. It is usually very fast and has a hashing technique that allows saving all the data we use.

The last method, named Code Rewrite, is the lightest method there is. Is actually using a compiler and is serializing the parsing tree. From this point it can compare two programs as it would compare two arrays.

We decided to use an algorithm that will take as much as possible from each and every solution mentioned before. Our algorithm is used only for this purpose; it has to detect all the similarities at the structural level. For example we would like to identify as template when we analyze three programs, one iterating an array using a ‘for’ statement, one using a ‘foreach’ and the last one using a ‘while’. Also there are lots of structural similarities such as iterating with the stop clause set to ‘true’ but having a conditional instruction that will stop the iteration.

As in figure 3.1, we decided to use compiler level. Having this we could have language independent specification that would provide a common interface. This decision was made to easily identify components in the program structure.

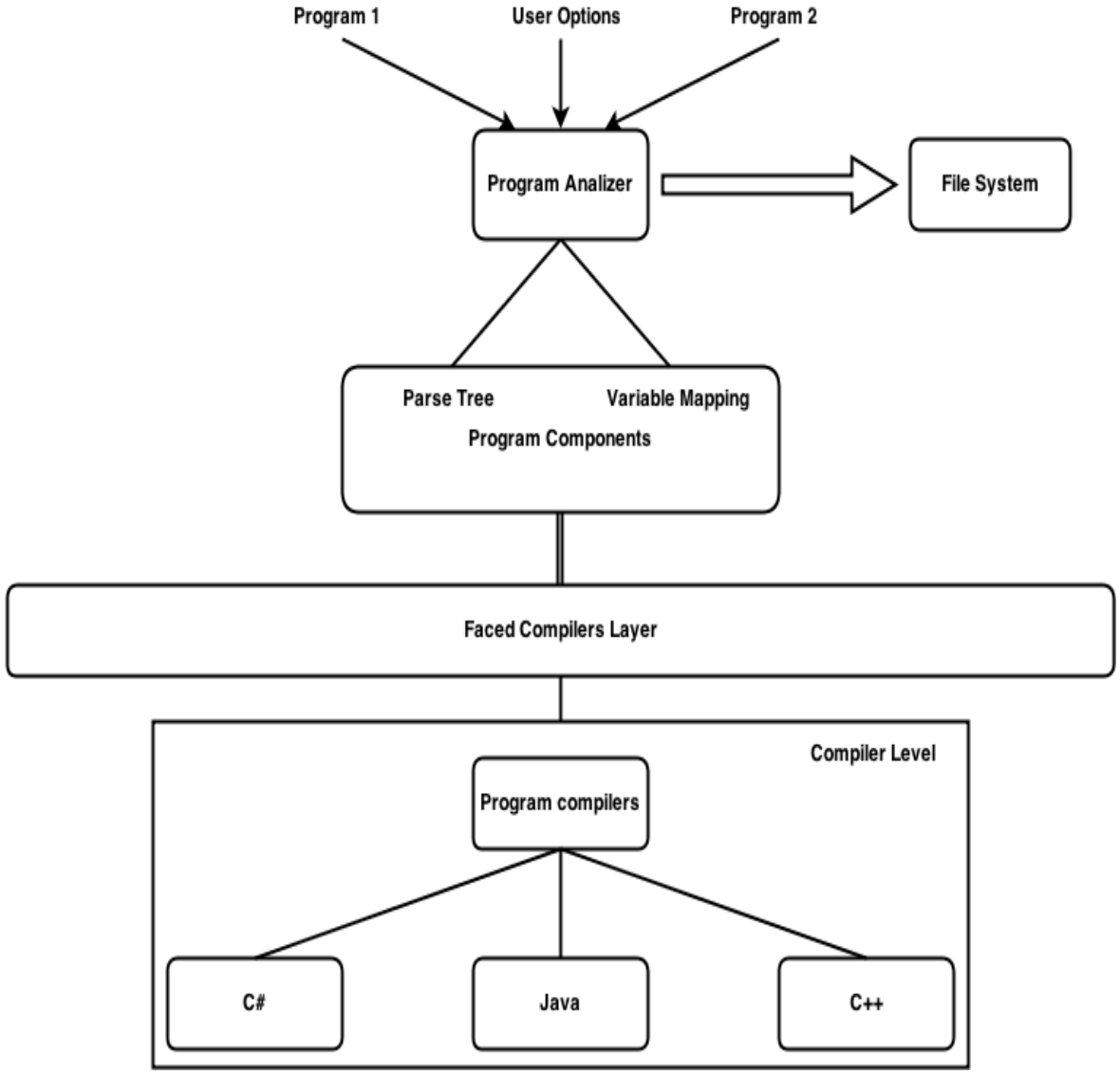

Fig. 3.1 Internal Design

Above the compiler, we decided that we need to add a faced layer in order to have the independent language support as mentioned before. The faced layer is exposing only the parsing tree with its labels. After we added the faced interface we serialized the given tree to an array.

In this moment we could analyze the given programs as string arrays. Here there are at least two approaches. First one is to use the Dynamic Programming algorithm for determine the longest common substring, this being helpful at similarity for source code. The second method, the one also implemented in this tool, the one named Greedy String Tiling. We selected the second one, to identify code rearrange. As we can see in figure 3.2 the greedy method Karp Rabin Greedy String Tiling has a better detection rate, when compared with dynamic programming longest common substring algorithm. The greedy method is actually much better in this case. And this is because it runs multiple times and every time is eliminating the used tokens from the searching context.

```
1   class P1{                       class P1{
2           void Method1(){                 void Method2(){
3                   [...]                           [...]
4           }                               }
5           void Method2(){                 void Method1(){
6                   [...]                           [...]
7           }                               }
8   }                               }
            a)                              b)

1   class P1{                       class P1{
2           void Method1(){                 void Method2(){
3                   [...]                           [...]
4           }                               }
5           void Method2(){                 void Method1(){
6                   [...]                           [...]
7           }                               }
8   }                               }
            c)                              d)
```

a) Given program 1
b) Given program 2
c) Template resulted using Greedy Algorithm
d) Template resulted using Dynamic Programming
Fig. 3.2 Greedy String Tiling vs. Dynamic Programming

3.2. Implementation Details

After analyzing all the given solutions, we had to implement that approach that was more relevant to our needs. The specific solution needed to have at least 75% accuracy. This constraint helped in selecting the approach that will reduce the false positives or the false negatives in an average input test. This specific requirement was made to assure that using the code generation tool we will have a big impact.

In this chapter we will describe the problems and the structural architecture witch is behind the code generation tool. We will have to be very careful with technical details, because this tool will have multiple usages, one of them been to detect real time if there are code blocks that are similar with the one the engineer is writing when developing to a feature.

3.2.1. Algorithm Details

We had to decide what compiler to use. We had some options as: Mono compiler or NRefactory library. We selected the last one, being an open source library that is actually a layer on the Mono Cecil and .NET 4.0.

We had to rewrite a part of the NRefactory library, in order to manage it to do what we needed. For the tree rewrite we used reflection making a mapping between the old labels and the new wanted one. Here we used for mapping an enum, with metadata. This was used in order to have all the old labels in our classes and if we want to change any of that to be easy to add a rewriting rule to do that. For example if we want to rewrite 'while' label into a 'loop' label this can be done with only one rewrite rule added to the reflection part of the application.

After this we had a parsing tree with our labels in it. From this moment in our application we started creating from scratch and not building on some library on compiler. We had the AST with what we needed.

From here we flattened the tree, for multiple reasons, we could compare much easier two string arrays with tokens. Graph isomorphism problem is neither known to be NP-complete nor to be tractable.

3.2.2. Similarity between files

Now that we have all the data needed, we will start comparing the sources between them. As mentioned before we will use Greedy String Tiling method for measuring the similarity between files.

The greedy algorithm is one of the best approaches there is. It can match on multiple portions of the code. This is the biggest advantage we can use.

As described in the chapter 2.2.3, when comparing two programs, we actually need to compare rewrite tokens. The problem with this approach is that we would like it to be easy but in order to have this easy way to compare we needed to add an extra layer to our similarity block. This was made by adding another metadata to the originally rewriting layer, specifying the method used for comparing the current node. For example, we would like to compare two invocations like "DoX()" and "DoY()", all we would like to do here is check if the method is doing the same thing. But for example when comparing two field declarations like "public int var1" and "private MyInt variableName" we would like to look at multiple other things like modifiers or class name.

Similarity between variables and similarity between methods. In our last example we wanted to compare two variable declarations and all we have done was to check if the modifiers are matching and also the type of the variables. But the problem is this is not enough and we will explain why. First of all, when writing two different classes, even if they represent the same thing or they represent something similar, the variable name will never be the same. Let's take an example. If we write two web services one for products and the other one for customers. When we will send the data on the network we will use two different names for the data transfer objects(DTO); probably they will be named 'ProductDTO' and 'CustomerDTO'. When

we are trying to match the declarations of those two variables, we will find that they are different. In our case, whey represent different things but they have the same structural speaking.

Name similarity. After this first review on similarity between variables or methods, we realized that we, as programmers, are naming the variable with the same structural meaning almost the same. This helped us to create our new algorithm for detecting the similar variables. In order for us to do that, we decided that there are at least two possible approaches. First one is when we compare the number or characters that match. To do that, we decided that the best approach is to get the longest common substring from the variables names. We considered this as a good metric because in the resulted name we will get the only the characters from the both strings that are in the same order. This was a big success, letting us to identify a very big class of similar variables. But we decided not to stop at this solution, and that was the moment when we focused our attention on words inside a variable name. We realized that variables with similar structural meaning have in their components similar words. Lets take the previous example: both of the variable names have the word 'DTO' meaning that both of them represent the same thing. As a result we gave the user the possibility to choose between **character matching** and **word matching**.

Stack similarity. After we done that, we still realized that is not enough. And the biggest problem we were facing was that we couldn't match variables that wouldn't have similar names and types. We decided that we need a new approach. Lets take a simple example, in figure 3.3 we would like to match the variable as a common element between the two programs.

```
1   class P1{                          class P1{
2           void Method1(){                    void Method1(){
3                   int x;                             int y;
4                   x++;                               y = y + 1;
5                   if ( 1 == 2 )                      if ( 1 == 2 )
6                           print(x);                          print(y);
7           }                                  }
8   }                                  }
                    a)                                 b)
```

Fig. 3.3. Variable Stack Matching

Looking at this code we can easily tell that the variable 'x' and 'y' have the same structural meaning. the only difference is the name. The easy solution here would be for our tool to not check the variables names, but this would give too many false positives and it would be in conflict with all we mentioned before at name matching.

In additional we start looking at what means that two variables are similar, structural speaking. We finally realized that the name or modifiers or type are not important when searching for structural similarity. The important fact is that the compared input must be used in the same places. This is the first step we took in getting closer to our goal. We decided that we attach a 'stack' to each variable usage.

This stack is representing where the variable is used. As you can see, in the figure 3.4, we have the previous program but we show what a stack means.

A variable has attached to it, a series of stacks representing where the variable is used. When two arrays containing stacks are similar in some percentage, we consider the variables having the same structural meaning. We decided to let the user define how big the similarity percentage needs to be.

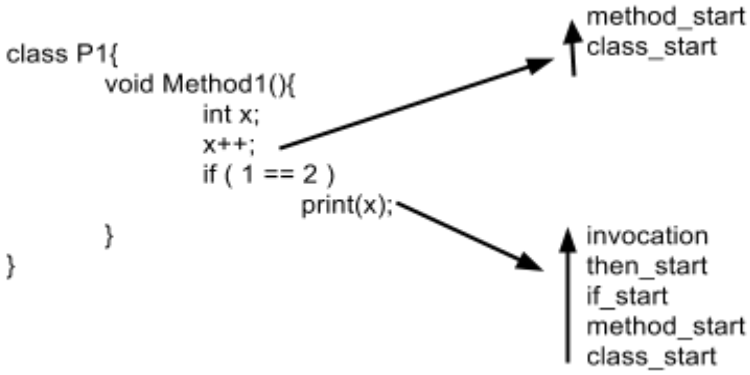

Fig. 3.4. Variable Stack Example

Loop Similarity. When talking about loops, we refer at statements like ‘for’, ‘while’, ‘foreach’. When we compare two programs, we probably find many statements like the ones enumerated here or similar with those. We could consider all of them as one, and match a ‘while’ loop with a ‘for’ loop. This is not necessarily bad, we would replace each label with a ‘loop’ label, and the matching would come from it. In this case, we would go to a lower level when speaking about structural meaning. But after some research we finally realized that each and every loop is usually used in a particular case and is not always good to consider all as one. For example the ‘while’ family statement is used when we don’t know from the beginning how much we will iterate, in contrast with ‘for’ statement when we know the size of the iterated array. Another example is a ‘foreach’ statement, which is used when we want to iterate through an array, but we don’t need the index for any of the elements, like finding the maximum value.

Manual editing. After we our tool is generating a template from the given programs, the user can add, remove or even rename variables in the final text. Also we can write some templates by hand, and this can be really helpful sometimes, but as a programmer you need to have some language to write this kind of templates, so before starting to write a new template a user needs to know what the template looks like. This is simple because all that can be changed in any template instantiation is the variable name. All others elements inside a template will stay unmodified. If we need to modify any other statements, we will modify them on the concrete instantiation. As a final specification, the names of the variables that represent the same thing are identified by having a ‘#’ sign before and after the variable name. This represents the text between the markers is a variable which that may need to change the name according to the needs of the instantiation purpose.

3.3. Predefined Architecture Patterns

According to Microsoft layered application guidelines, you can decompose the design into logical groupings of software components. These logical groupings are called layers. Layers help to differentiate between the different kinds of tasks performed by the components, making it easier to create a design that supports reusability of components. Each logical layer contains a number of discrete component types grouped into sub layers, with each sub layer performing a specific type of task. Using these layers, we can optimize the way that the application works when deployed in different ways, and provides a clean delineation between locations where certain technology or design decisions must be made.

As we can see in the figure 2.14, the Microsoft Corporation is giving us a high level for the logical architecture view of a layered system. It doesn't matter if the layers may be located on the same physical tier, or may be located on separate tiers, your design must accommodate those guidelines.

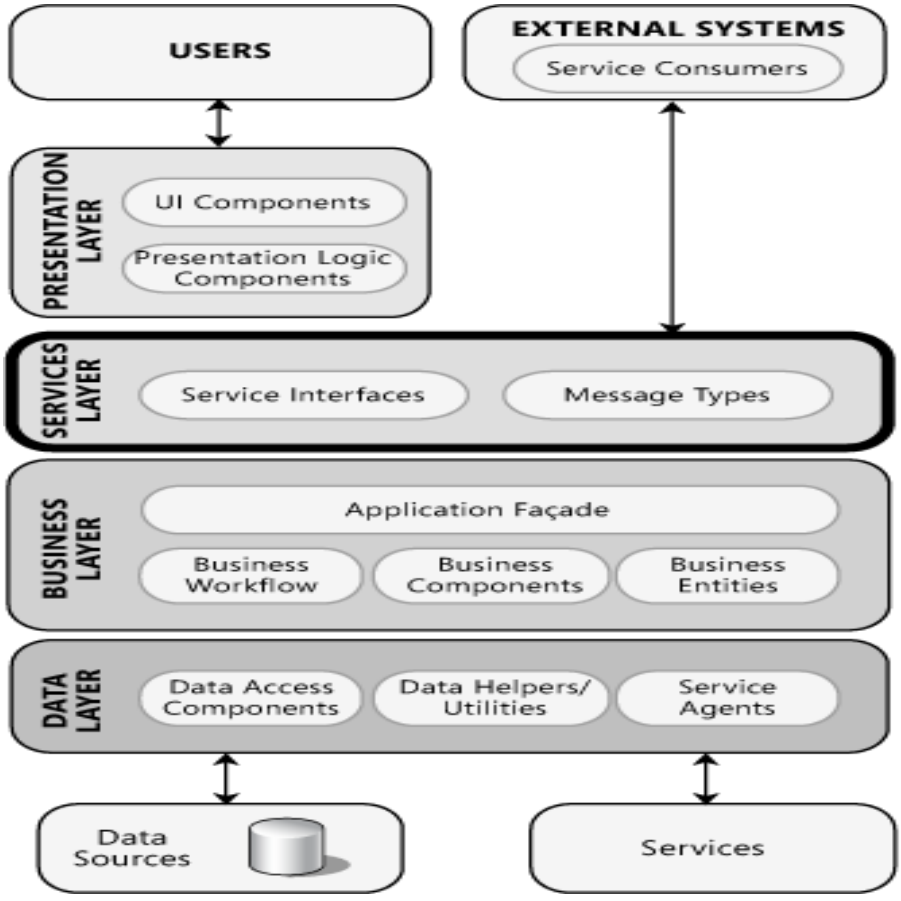

Fig. 2.14 Presentation, Business, and Data Layers

Presentation layer. This first layer is the user-oriented layer. It is responsible for managing user interaction with the system, and generally consists of components that provide a common bridge into the core business logic encapsulated in the business layer.

Business layer. This is the software logic and it implements the code functionality of the system. It generally consists of components, some of which may expose service interfaces that other callers can use.

Data layer. This is the part of the application, where we keep all our important information. It provides access to data hosted within the boundaries of the system. The data layer exposes generic interfaces that the components in the business layer can consume.

Because our goal is to generate code for a Microsoft application, we also have all of those layers. As in any big application, this guideline has small or big modifications in order to be as scalable as we want and flexible as we can change or replacing any layers without interfering with the system logic.

In order to do this, we divided the business layer into two layers. One is a Service Layer, and it has all the logic that the application is required to simulate. This layer is communication with the Data Layer with model entities; those are a description of the objects represented by a computer system together with their properties and relationships.

Also the Service Layer is exposing a unitary interface layer that is more light throw the Facade Layer. This last component is called from a Web Service Layer that supports all the requests from the client interaction. Also between those two layer, we have integrated a Proxy handle, used for redirecting requests to the right component, or just for small verifications.

Our goal is to have this predefined structural design as a template and when we need to add a vertical feature

3.4. Automatic Code Duplication Detection

A good usage for our final application is to use the powerful pattern detection implemented. It has combined multiple approaches, for detecting structural similarities. Due to this, we have a powerful mechanisms designed for detecting what may be alike in multiple file source.

The main idea for this feature is that inside a given project. At a first look, this would not provide us with much information. The surprising result of this main approach is that, when there are multiple engineer programmers which are involved in the same application, they may write methods, or fractions of them which structural speaking are the same. This can be easily explained using an example like: when there are two programmers, one is using products as objects and the other one is using users as objects. Lets say they both need to sort a list of their object. It is obvious that both of them may use the same technique for sorting a list. After both features are ready, we final application will contain the same sorting method but written for two different objects. When integrating the final code in the main breach, our programmers may see, or may not see that their colleague has written a similar block somewhere in the other part of the application.

As we proved in our example we may write similar code blocks (not necessary an entire function, maybe only three-four files of code) in the same application in the

same time or even later. Running the tool, we can be announced with a warning flag that will show you the file with the some code.

This is would be a great feature to the code generation application, no in the sense of generating code, but to detect the code that is structural identical and it may be extracted as a separate method.

The next step of extracting the common lines into a function can be made using ReSharper or even the Visual Studio IDE, which is providing an extract method feature. All of those functionalities are very helpful in order to have a scalable and maintainable code for our solution.

4. Results

The final application is an easy to use web interface. At the beginning we need to select the files we consider similar. As we can see in the figure 4.1, in the same page we need to set some options related to the algorithm options. Between all the configurations we need to set how to compare the variable or methods names, character count or word count.

This is very important for our generation file. Due to the configuration we are making here we may have a generation file that may be more appropriate with our needs.

For example, there are two options that may be selected: Before and After analyze. Those options will take a look at the variabiles and all what can be diferent due to user input. This is important because we can consider variables to be the same from the begging which will make the pattern detection act differently. And also there is a good chanse of detecting similarities after all the algorithms have runed and we only analyse the remaining code that has not matched on.

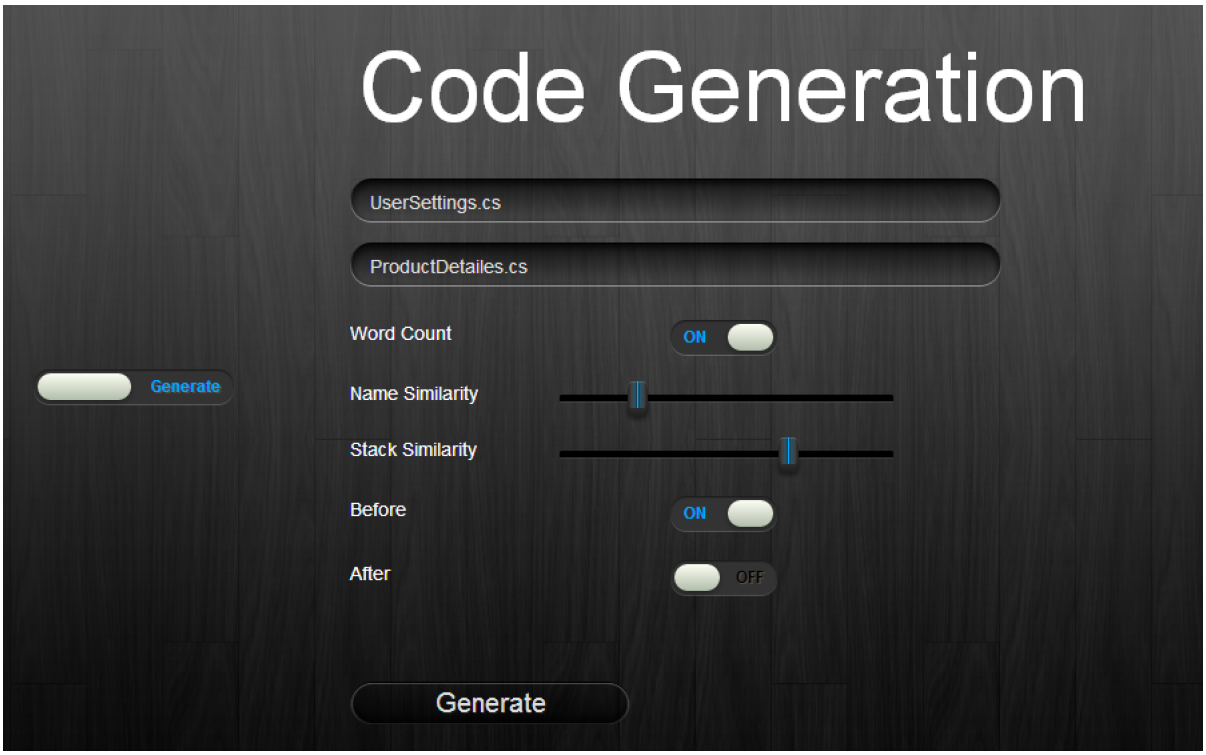

Fig. 4.1 Settings for Code Genetation Tool

After generating the initial template, the user needs to select a default program. This program will be the text that the initial template will look like.

In the figure 4.2 we now have the initial extracted code, from here the user can drag and drop lines from any original program, or even change the order in the resulted template. Also he can add, edit or remove any line from the generated template. And before we can move on, we need to set a name to this final template.

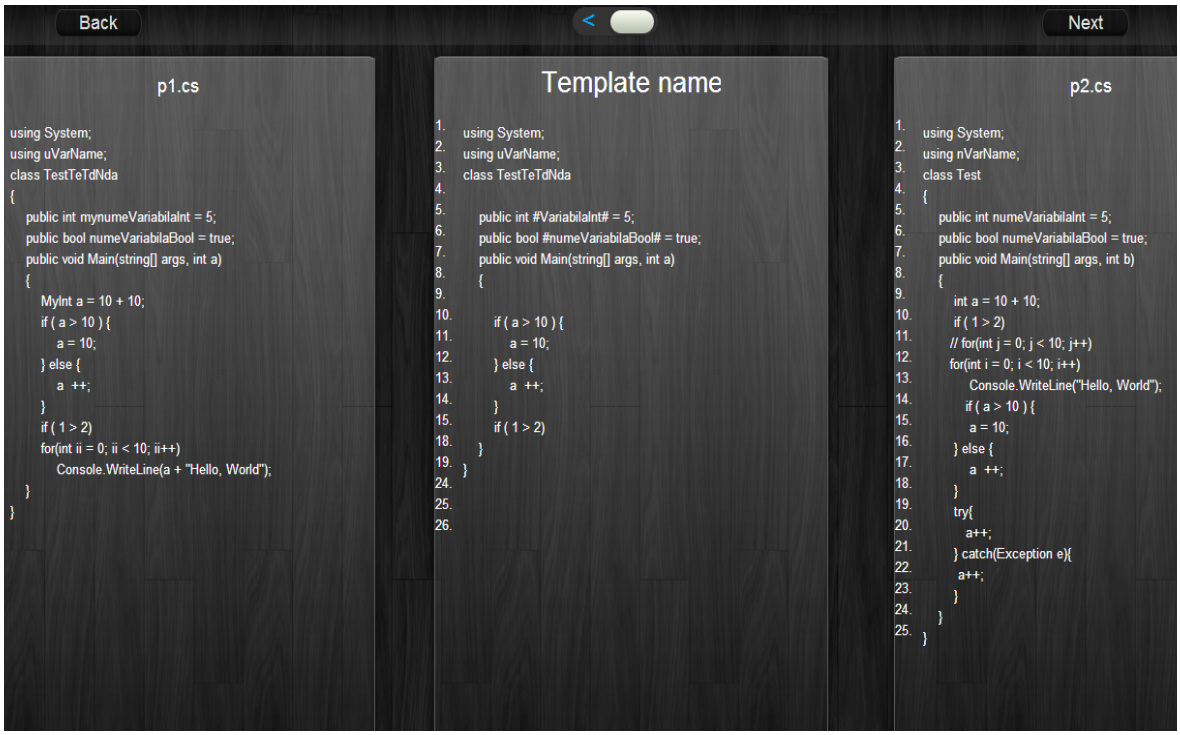

Figure 4.2 Editing the extracted code

In the last step of code generation, is using the template. We can see in figure 4.3 that we may need to rename any variable. We have a rename variable panel witch will rename all the occurrences of that variable in the text. We can also make changes to the template code. All our changes will be saved as original template.

Also after saving a final template, we can come back later, or even share this with the team and use this template as a ground for a feature. We can see in the figure 4.3 our final code is displayed, as it will be instantiated in the application.

After presenting the final tool to all our colleges, they were all happy to try it. The feedback from our developers was a good one; given them the possibility to concentrate on what is important in our application and not blocking on things that can be automated.

As a final estimation our developer productivity increased with 27% when writing features that could use this kind of automation templateing. Also their overall productivity increased by 5%. We know this is not very much when talking about a single developer, but when we talk about 20, the time estimation is begging to be important. When we say important we mean that relative to a 6 hour productivity day and 20 developers we save about 6 hours per day. This is almost the productivity of a full time job employ.

Figure 4.3 Final templates

As a result of using this application, the team talked about the time lost for writing some tasks before and the time it takes to make the same feature using the product of my thesis. Mentioning the number of bugs that can be avoided using code generators tools

5. Conclusions

When we are facing project with thousands of lines in code, we need to be very careful with all the structural and design patterns that assure that the application is scalable and keep the performance high enough. As in figure 5.1, we as programmers, write code that is placed on the top of an older code and later our code will be the ground of an others programmers code.

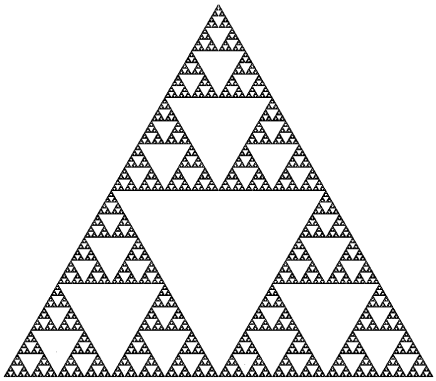

Fig. 5.1. Pyramid of Code

In order to write high quality code we need to concentrate very hard on what we write. In order to do that, we need a tool that will take away our time consuming tasks that can be automated.

Using a tool that can extract templates and let's you generate code using those templates, will save time, money and will make an engineer to focus on is important in his project.

Taking all of those in consideration, we realized we need a tool to help the engineers to focus on real problems, not on structural problems they have already established. Now we can generate templates from existing code and this is big step in the code generation evolution. Due to this, we realized we not only increased the productivity, but also we increased other people skills not only because we are now more focused on what matters but because we are also minimized the defect risk.

Despite the challenges posed by the task, initial result indicates that a satisfactory percentage can be obtained using the current application. Continuing the current work and improving the existent approaches can finally lead to a solution which can be used by more and more programmers when writing an application.

BIBLIOGRAPHY

[1]. K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, M. Bernstein, Pattern Matching for Clone and Concept Detection, 1996

[2]. Saul Schleimer, Daniel S. Wilkerson, Alex Aiken, Winnowing: Local Algorithms for Document Fingerprinting, 1994

[3]. Jeroen Arnoldus, Mark van den Brand, A. Serebrenik, J.J. Brunekreef, Code Generation with Templates, Volume 1, 2004

[4]. Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and St´ephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. 2007.

[5]. Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. 2002.

[6]. Miryung Kim, Vibha Sazawal, and David Notkin. An empirical study of code clone genealogies, 2005. ACM.

[7]. Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. 2004.

[8]. Lingxiao Jiang, Scalable Detection of Similar Code: Techniques and Applications, December 2009

[9]. Dan Gusfield. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press,1997.

[10]. Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. 2004. USENIX Association.

[11]. Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS R: Program transformations for practical scalable software evolution. 2004. IEEE Computer Society.

[12]. Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees.1998. IEEE Computer Society.

[13]. M. Mozgovoy. Desktop tools for offline plagiarism detection in computer programs. Informatics in Education,, 2006.

[14]. D. Gitchell and N. Tran. Sim: a utility for detecting similarity in computer programs. SIGCSE Bulletin, 1999.

[15]. B. Belkhouche, A. Nix, and J. Hassell. Plagiarism detection in software designs. In ACM-SE 42: Proceedings of the 42nd Annual Southeast Regional Conference, New York, NY, USA, 2004. ACM.

[16]. Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Localitysensitive hashing scheme based on p-stable distributions.2004. ACM.

[17]. Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. 1995. IEEE Computer Society.

[18]. Xifeng Yan, Jiawei Han, and Ramin Afshar. CloSpan: Mining closed sequential patterns in large databases. USA, 2003.

[19]. Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs. 2000.

[20]. John L. Donaldson, Ann-Marie Lancaster, and Paul H. Sposato. A plagiarism detection system. ACM, February 1981.

[21]. H. T. Jankowitz. Detecting plagiarism in student Pascal programs. The Computer Journal, 1988.
[22]. Michael J. Wise. String similarity via greedy string tiling and running Karp-Rabin matching. Dept. of CS, University of Sydney, 1993.
[23]. R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. IBM J. of Research and Development, 1987.