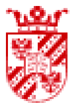
AUTOMATICALLY REDUCING CODE DUPLICATION

Finding code duplication and eliminating it by method extraction

Liewe Kwakman



university of groningen

faculty of mathematics and natural sciences

computing science

September 2010

Thesis supervisors:

prof. dr. A. C. Telea

prof. dr. P. Avgeriou

0 Index

0 Index 2
1 Abstract 6
2 Introduction 7
3 Related work 10
3.1 Code clone detection 10
3.1.1 Text based code clone detection 10
3.1.2 Token based code clone detection 10
3.1.3 Pattern matching based code clone detection 11
3.1.4 AST based code clone detection 11
3.1.5 Slicing based code clone detection 11
3.2 Code refactoring 11
3.2.1 Extract method 12
3.2.2 Metric based refactoring suggestions 12
3.2.3 Slicing based code clones refactoring 12
4 Code analysis 13
4.1 Project analysis 13
4.1.1 Project file structure 14
4.1.2 Source code files 15
4.1.3 Root namespace 15
4.1.4 Project Id 15
4.1.5 References 15
4.1.6 Imports 16
4.2 Lexical analysis 16
4.3 Syntactic analysis 17
4.4 Keeping track of scopes 19
4.4.1 Scope tree construction 20
4.4.2 Qualified location 22
4.4.3 Scopes of partial types 22
4.5 Type system 23
4.5.1 Types and members 23
4.5.2 Identifying types with fully qualified names and assembly identifiers 29
4.6 Populating the type system 30

4.6.1 Library reference resolving 31
4.6.2 Analyzing types in two stages 32
4.6.3 Type scanning 32
4.6.4 Type building 34
4.7 Summary 37
5 Classification 39
5.1 Classification Result 39
5.2 Type classification 43
5.2.1 Fetching types using a fully qualified name 43
5.2.2 Querying types which reside in modules 44
5.2.3 Type resolution 46
5.3 Classifying member and local variable references 49
5.3.1 Classifying a simple named identifier 49
5.3.2 Classifying a qualified named member 52
5.3.3 Classifying a module member 53
5.3.4 Finding accessible modules 55
5.3.5 Finding the correct member call or indexing on a type 56
5.3.6 Member resolution 57
5.4 Expression classification 65
5.4.1 Operator resolution 65
5.4.2 Expressions 66
5.5 Justification of the classification against the coding specification 73
5.5.1 Type resolution justification 73
5.5.2 Simple name expressions 78
5.5.3 Member Access Expressions 81
5.5.4 Overloaded Method Resolution 83
5.5.5 Applicable Methods 85
5.5.6 Type Argument Inference 86
5.5.7 Index Expressions 88
5.5.8 Operator resolution 89
5.6 Summary 91
6 Code refactoring 92
6.1 Matching 95
6.1.1 Finding matching code fragments 95

6.1.2 Matching statements ... 96
6.1.3 Checking dataflow consistency ... 115
6.2 Selection ... 117
6.2.1 Refactoring preconditions ... 117
6.2.2 Checking a fragment conditional returns ... 118
6.2.3 Return values of an extracted method ... 120
6.2.4 Side effects ... 121
6.2.5 Selecting a code clone to rewrite ... 122
6.3 Rewriting ... 122
6.3.1 Creating the method body ... 123
6.3.2 Determining variables ... 124
6.3.3 Determining which variables should be parameters and which should be returned ... 125
6.3.4 Determining variable dependencies ... 127
6.3.5 Creating the unifying extracted method ... 130
6.3.6 Replacing the code fragments with a call to the extracted method 131
6.4 Summary ... 134
7 Application ... 135
7.1 Show case ... 135
7.2 Case Study ... 138
8 Discussion ... 143
8.1 Increasing usability ... 143
8.2 Increasing the number of code clones found ... 144
8.3 Finding more than two instances of a clone at once ... 144
9 Conclusions ... 146
10 References ... 148
11 Appendix 1: Visual Basic 8.0 grammar ... 150
11.1 Lexical Grammar ... 150
11.1.1 Characters and Lines ... 150
11.1.2 Identifiers ... 150
11.1.3 Keywords ... 151
11.1.4 Literals ... 151
11.2 Preprocessing Directives ... 153
11.2.1 Conditional Compilation ... 153

11.2.2 External Source Directives 154
11.2.3 Region Directives 154
11.2.4 External Checksum Directives 155
11.3 Syntactic Grammar 155
11.3.1 Attributes 155
11.3.2 Source Files and Namespaces 156
11.3.3 Types 157
11.3.4 Type Members 159
11.3.5 Statements 165
11.3.6 Expressions 169

1 Abstract

When coding large projects, chances are that existing code fragments will be re-written over and again. This can be done unconsciously but also by copy-pasting; this is unfortunately a widespread practice. In some case studies of commercial software it was shown that 5% of the code was represented by clones. It is well-known that code duplication decreases the maintainability of the code rapidly. In this thesis we describe the design and implementation of a software tool which can detect code duplicates in programs written in Visual Basic. The tool offers the possibility to automatically remove these duplicates by using a method extraction refactoring. As a non-trivial side result in building this tool, we constructed a full static analysis pipeline involving parsing, disambiguation, and deep semantic analysis of general Visual Basic programs. One additional novel part of our tool is that semantical analysis is performed to ensure the duplicates presented can be refactored without changing the external behaviour of the program.

2 Introduction

When coding large projects, chances are that existing code fragments will be re-written over and again. This can be done unconsciously but also by copy-pasting, this is unfortunately a widespread practice. In some case studies of commercial software 5% of the code was reported to be clones (1) in an open source project, this amount was reported to be higher: 12% (2). Newer surveys indicate similar figures across a wide range of types of open source software (3). It is well-known that code duplication decreases the maintainability of the code rapidly. It is desirable to remove these clones by merging them into one instance of the code.

To be able to remove clones they first need to be found. When working on a large code base with hundreds of thousands of lines of code or more which has a lot of duplicate code, it is quite difficult to find all this duplicate code. Although code duplication detection tools exist, these are, for their greatest majority, based on textual (2) or token (1) (4) based analysis of the code. Textual analysis only will find code which is textually exactly the same, if for example someone changes a variable name slightly; the similarity of the code is no longer recognized. Also it is not guaranteed that two pieces of code which are textually the same are semantically the same. Token based analysis will find more code duplications, because it compares token types, for example an identifier will match an identifier. This also means more code clones will be found which are not semantically the same.

Even if we assume the availability of state-of-the-art code duplication detectors which can detect semantically identical, but potentially syntactically different, duplicates, the elimination of the dyplicates is typically done manually. This process involves a laborious activity of checking the instances of a duplicate one by one and editing the code in various places to replace all duplicates by a single instance. Henc, a second challenge for producing maintainable code, after detecting the duplicates, is to suggest (or perform) automatic refactoring, i.e. replace all instances of a duplicate (also called a clone) with one single instance. The advantages of refactored code are numerous: the code base to maintain becomes smaller; the overall structure can improve, testing and understanding activities decrease in cost, since one has now only to test or understand a single code instance rather than all its clones.

The goal of this thesis project is to build a tool which locates pieces of code which are semantically the same or similar, whether they are textually the same or not and, where suitable, suggest and perform code refactoring operations to eliminate this duplication. This code refactoring operations will be in the form of unifying method extractions in such a way that a call to the resulting method with the right arguments can replace both semantically similar pieces of code without changing anything to the external behaviour of the code. Given that this project is executed in the context of software development activities at an IT

company specialized in Visual Basic development, we shall limit our clone detection and refactoring scope to Visual Basic code.

Tools exists which suggest the removal of code clones in java source code using the so-called "Extract Method" or "Pull Up Method" refactoring (5) using only syntactical information. However, syntactical information is not enough though to actually perform the code refactorings. Hence, we need a mechanism to extract both syntactic and semantic information from Visual Basic source code to support our automatic clone removal requirement.

Because code clones need to be found which can be automatically removed by method extraction without changing the external behaviour of the code, the code clones cannot be found using just textual or token based duplication detection. It is necessary to exactly now which variables and members are used in the code and what is done with them. This, again, can only be done by doing a deep semantic analysis on the code.

To limit the scope and effort of this work, the envisaged solution should be able to parse and analyze code written in Visual Basic 8.0. Preconditions for this code are; it is provided with a Visual Studio 2008 project file and the code should be compilable with option 'strict' on. However, the main design principles involved in creating the solution should be generally applicable to languages beyond Visual Basic.

As a first step of the proposed solution, a static code analyzer is constructed. This code analyzer analyses a Visual Basic 8 project and loads all files necessary. In the first phase, all files containing source code are parsed to an abstract syntax tree (AST). In the second phase, a semantic analysis is performed over the AST. In this phase, the AST is used to populate the so-called type system, which keeps track of all types existent in the program. Next, an expression resolver is constructed to resolve any expression against the type system. This step is required to be able to determine how methods are called in the program, as a preliminary support step for the refactoring.

In the second stage fragments of similar code are detected. Because the envisioned way to handle the found code clones is to remove them by extracting one method from such fragments, next to the matching of the code, the restrictions of the method extraction operation must be kept in mind.

The method used to find matches is loosely based on the unifying algorithm to match expressions. To match two fragments the kinds of every pair of statements must be the same, but under the right circumstances expressions can be substituted with new variables introduced in the extracted method. Sometimes this is necessary because the expressions refer to local variables which are in another scope then they will be after the refactoring; sometimes it is necessary because two expressions differ; in that second case, the resulting

value of the evaluated expressions can be passed to the extracted method as an argument.

From the code clones found, candidates for method extraction must be chosen. Some code clones must be rejected because they are impossible to replace with an extracted method, others must be rejected because it is not desirable to replace them with an extracted method.

Ones a code clone is selected, it can be rewritten. A method must be created which with the correct arguments can be called in replacement of the two matching fragments of code in the selected code clone. First must be distinguished which substitute variables in the extracted method need to be parameters and which need to be returned. The method and must be added in the source code and both fragments need to be replaced by the correct call. If the extracted method returns a value it must be assigned to the correct variable or property.

The following diagram gives an overview on what has to be done to find and refactor code clones.

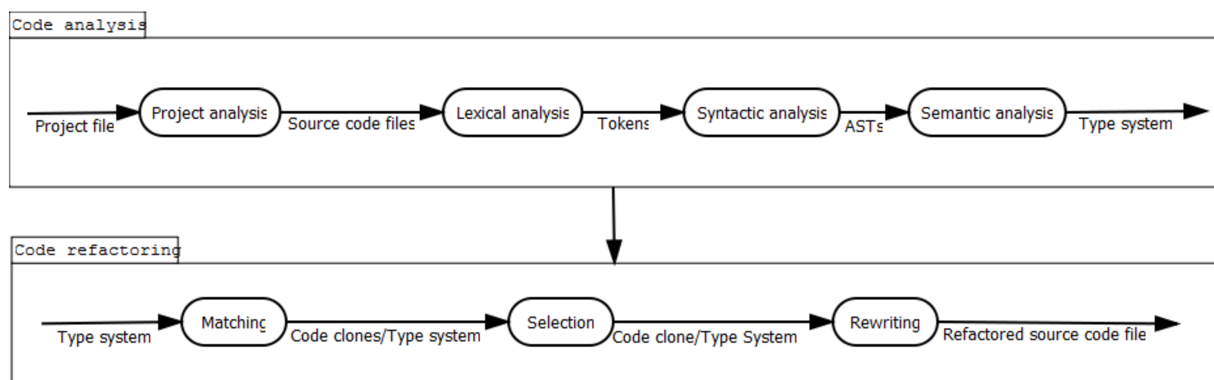


Figure 1 Overview of the clone detection and refactor process

The remainder of this thesis is structured as follows.

Section 3 discusses the work that has already been done in the area of code clone detection and refactoring. Section 4 discusses how the code analysis is performed and a type system is built. Section 5 discusses how expressions can be classified using the type system. Section 6 discusses the code clone detection itself, the selection of code clones and the rewriting of the code. In section **Error! Reference source not found.** some examples of the use of the software created are shown. Section 8 discusses our solution. Section 9 concludes the thesis and outlines directions of future work.

3 Related work

There is little work to be found about the combination of code clone detection and refactoring combined. However, a lot of research is done in the area of code clone detection. Also method extraction of single code fragments is researched. Most scientific work is intended for languages like C or Java, which do not have the 'Property' and 'Module' construction present in Visual Basic, so these results are not directly applicable to Visual Basic. In section 3.1 work in the area of clone detection is discussed, section 3.2 discusses work in the area of code refactoring.

3.1 Code clone detection

Multiple approaches are used to find code clones; none of them as far as we know find code clones which are guaranteed to be removable using method extraction. Generally variable names and literal values are abstracted away and ignored, resulting in simpler and faster algorithms, but to guarantee a code clone is really a clone, or is similar, and refactoring would be semantic invariant, semantic analysis is necessary as done by the tool presented in this thesis.

3.1.1 Text based code clone detection

In (2) uses a textual based approach to find code duplication or near-duplication. Two code fragments are considered a match if they are the same after some global substitutions of variables and literal values, whitespace within lines and comments are ignored. This approach is based on a parameterized suffix tree. It is very fragile, if a new line is added in a clone, it is not found. In (6) whole lines are compared to each other by hashing the strings. To further support clone analysis, visualization is used: In a dotplot, or matrix plot, matching lines can be visualized; diagonals in the dotplot are consecutive lines. However interesting, this visualization approach does not really scale well to very large code bases.

3.1.2 Token based code clone detection

A commonly used tool to find code clones is CCFinder (4). CCFinder uses lexical analysis to transform the source files into a token sequence. This token sequence is transformed using a set of transformation rules in such a way it generalizes similar tokens; among others, variable names and literals are abstracted away. After this sequences of equal tokens are sought to find code clones. Some metrics and heuristics are used to filter out the less interesting code clones. To cope with scale, CCFinder uses suffix trees to find sequences of equal tokens. CCFinder also uses the lexical knowledge of programming languages to find code clones, which is an improvement to the text based approach. A refinement of the CCFinder approach that also uses more sophisticated visualizations to show the code clones together with the overall hierarchical program structure is implemented by the SolidSDD tool (7). However, neither CCFinder nor SolidSDD are designed in such a way that they can be adapted to perform method-based clone removal.

3.1.3 Pattern matching based code clone detection

In (8)methods are presented to find code clones using pattern matching techniques. A similarity metric between all pairs of begin-end blocks in the program is computed. Code clones are not really found but only a similarity measure. To find code clones, the user has to look manually through high scoring pairs. The methods only compare whole blocks of code; code clones which are not whole blocks will thus not be detected.

3.1.4 AST based code clone detection

In (1) abstract syntax trees are used to find code clones. Near misses are found by comparing trees for similarity rather than exact equality; variable names and literal values are ignored. Because this method uses abstract syntax trees, the syntactical knowledge of the programming language is used; this is a great advance in respect to textual code clone detection. But because of the ignoring of the variable names matches could be found which are not meaningfully extractable. To cope with scale, in the method sub-trees are categorized using hash values which allows straightforward matching of exact sub-tree clones. The solution presented in this thesis also makes use of AST's to find clones, near misses and less near misses are found by using the type an expression classifies to instead of only the fact it is the same kind of expression. In our solution selection procedure guarantees the presented code duplicates are meaningful and extractable. Moreover, our method focuses on Visual Basic, which was not one of the languages covered by (1). In (9) code clones are sought to serialise ASTs and using suffix tree detection on the serialized ASTs.

3.1.5 Slicing based code clone detection

In (10) duplicates are found which don't have to be continuous and of which the statements don't have to be in the same order. A program slicing approach is used to find these duplicates. First groups of matching statements are made, after that groups of matching statements are combined to find bigger duplicates. To do this a program dependency graph is used. In short a statement is dependent on another statement if the other statement alters a variable used by the statement or if the other statement is a control statement influencing if and how a statement is evaluated. To expand a matching pair of statements, the predecessors in the dependency graph are matched, if they match, they can be added to the resulting duplicate. Just as in previous approaches described variable names and literal values are ignored.

3.2 Code refactoring

Not much is written about automatically refactoring code clones, and there are virtually no records of showing such techniques implemented by actual tools. All approaches found, such as the ones discussed above, abstract variable names and literal values away and very few approaches (if any) do deep semantic analysis. Hence, such approaches are less suitable (if suitable at all) to perform automatic code clone refactoring, for the reasons discussed earlier in this chapter.

3.2.1 Extract method

Fowler presented in his book (11) an elaborate list of refactorings. In this thesis the focus is on the "Extract Method" described on page 110. The refactoring described in the book is done manually and used to break big methods up in smaller methods. One of the motivations is that smaller methods increase the chances that other methods also can use it, which is exactly what is done in this project: pieces of code which are reused are extracted. However, we do this automatically, and not manually.

3.2.2 Metric based refactoring suggestions

A tool with common goals as this thesis is Aries (5) Code refactoring opportunities are suggested to the user in the form of the tool Aries in the form of the "Extract Method" and "Pull Up Method" refactoring patterns also described in (11). Aries uses the output of CCFinder (4) and performs code metrics on it to suggest which code clones found are suitable for refactoring. CCFinder only uses lexical analysis, in Aries syntactical information is added by parsing source files where code clones are found. Aries does not look at semantics and types though. It can thus only base it suggestions on syntax thus it cannot be guaranteed the suggested refactorings are possible, especially not if it would be applied to a language as Visual Basic with structures as properties and modules. Hance, Aries is suitable as a help for further manual refactoring, but does not solve the challenge of deep automatic refactoring.

3.2.3 Slicing based code clones refactoring

In (12) duplicated code is found using a program-slicing based approach as described in 3.1.5. Algorithms are provided to extract methods for duplicated code. These were only partial implemented, dependencies between statements still needed to be manually determined. Two nodes are regarded matching if their internal expressions are identical ignoring all variable names and literal values. Hence, determining the parameters of the extracted method would thus not be trivial, making this method less suitable for automatic method-based clone refactoring.

4 Code analysis

To be able to refactor code and eliminate duplicates, a first fundamental step is to find the duplicates. As already mentioned in the introduction, we are looking for duplicates from a semantic, rather than syntactic or lexical perspective. This guarantees, first of all, refactoring correctness. Secondly, because knowledge of types allows identifying expressions which are syntactically different but result in the same type, expectations are this will yield a higher duplication rate, meaning ultimately a better refactoring.

To extract the semantic information from the source code, the code is first transformed into abstract syntax trees (AST's) and a type system is built. An abstract syntax tree of a program text is a data structure which represents program text in terms of the programming language's grammar (13). The type system is a table of all types declared and referenced(used) in the source code.

To build the ASTs and the type system a code analysis is done in following four stages:

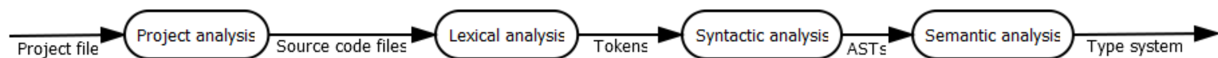


Figure 2 Overview of the code analysis

In the project analysis an input project file is analyzed and together with some general project information. All referenced source files, projects and libraries are loaded as discussed in section 4.1. During the lexical analysis the raw source code is transformed to a stream of language specific tokens, which is discussed in section 4.2. In the syntactical analysis, discussed in section 4.3, these tokens are used to build an abstract syntax tree. In the semantic analysis, a so-called scope tree is built, representing the scopes in the source code, and the type system is populated. Section 4.4 discusses how scope trees are constructed, in section 4.5 is discussed what the type system is and how it works and section 4.6 discusses how the type system is populated.

4.1 Project analysis

The goal of project analysis is to analyze the project at a high level and determine the information needed for subsequent per-file syntax and semantic analysis. This means we have to determine which code files, referenced projects and libraries to load along with other project information. There are two reasons why this stage is needed:

- Automation: ideally we want to apply our code refactoring solution on entire projects with minimal user intervention. For this, a means should be devised to analyze and process entire projects rather than individual files.

Correctness and completeness: the semantics of a given code fragment (whether it is a file, function, class, or something else) is, in most programming languages including Visual Basic, not independent of its larger context. For instance, the

code constructs in a file typically depend on included files and referenced projects and libraries. Also some build options used to compile that file are only available at the project level. Hence, the analysis of individual files is implicitly dependent on information available at project level.

All information about which files are included in a project, which projects and libraries are referenced and other general project information in Visual Basic is stored in a project file. This project file is analyzed. The result of this stage will be a collection of *Project* objects containing the appropriate *CodeFile* objects, references and other project information.

The Visual Basic 8 project files are XML files, therefore parsing is not a problem; many libraries are available taking care of parsing XML. The following information is extracted from the project file:

- Which source code files are a part of the project
- The root-namespace of the project
- The id of the project
- Referenced projects and libraries
- Global imports

Section 4.1.1 discussed how a Visual Basic project file generally is formatted. Section 4.1.2 discusses how the source code files referenced by the project file are loaded. Every Visual Basic project has a so-called root-namespace helping to identify types declared in a specific project, section 4.1.3 discusses the root-namespace of a project. Although the root-namespace of a project can be used to distinguish types from different projects, the root-namespaces of different projects can be the same. Every project has however a unique id which is discussed in section 4.1.4. Section 4.1.5 discusses how referenced projects and libraries are loaded. Section 4.1.6 discusses how imports are extracted from the project file.

4.1.1 Project file structure

As said before, a project file is an XML file. The information in the file is divided into properties and items. Properties contain general information about the project such as the root-namespace and project id. Properties are grouped in tags with the name *PropertyGroup.* Items represent source code file names and references; they are grouped in tags with the name *ItemGroup*. Both the *PropertyGroup* and *ItemGroup* tags are direct children of the root tag *Project.*

A project file looks as follows, where the items and properties are replaced by dots:

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="3.5" DefaultTargets="Build"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    …
```

```
  </PropertyGroup>
  <ItemGroup>
    …
  </ItemGroup>
</Project>
```

4.1.2 Source code files

The source code in a Visual Basic project is distributed over different source code files. To be able to analyze the code correctly, all source code files associated with the project need to be loaded, i.e. read into the analysis tool.

Files are included in the project file using the *Compile* tag which is a child of *ItemGroup*:

```
<Compile Include="Class1.vb" />
```

The code files are distinguished by their extension, files referenced in the project file ending with the extension 'vb' are considered code files; other files, for example images or other non-compilable resources, are ignored. For every code file a *CodeFile* object is added to the project; this object further loads the content of the source file.

4.1.3 Root namespace

The root-namespace is the namespace in which every type and subsequent namespace in the project is automatically placed in. It can be used to distinguish types declared in different projects. Hence, the root-namespace will be the first part of every qualified name of types declared in the project. The root-namespace is stored in the *Project* object. In the project file, the root-namespace is found in the *RootNamespace* element which is a child element of a *PropertyGroup* element.

```
<RootNamespace>rootnamespace</RootNamespace>
```

4.1.4 Project Id

Every project has a unique id which is used to identify it with. In theory multiple types with the same qualified path can exist in different projects, this id is used to distinguish between them. It can be found in the *ProjectGuid* element which is a child of *PropertyGroup* element.

```
<ProjectGuid>{19C0B3FD-3C43-46D2-8DAA-DE50C4B5DB4D}</ProjectGuid>
```

The project id is stored in the *Project o*bject.

4.1.5 References

A project can reference other projects to be able to use types declared in those projects. There are two types of references: library references and project references. A library reference references a so-called assembly contained in a dynamic link library (DLL) file; these references are saved in the project file for

later use. The references can be found in the *Reference* element, which is a child of an *ItemGroup* element.

```
    <Reference Include="ReferencedLibrary.Windows,
Version=2.0.1207.0, Culture=neutral,
PublicKeyToken=21d5517571b185bf">
      <HintPath>..\..\Externals\
ReferencedLibrary.Windows.dll</HintPath>
    </Reference>
```

A project can also reference another project. Projects which are referenced need to be analyzed as well. So if a project is referenced, its project file is loaded and analyzed as well. The id of the referenced project is saved in the *Project* object. Reference projects can be found in the *ProjectReference* element which is a child of an *ItemGroup* element.

```
  <ProjectReference Include="..\
ReferencedLibrary\ReferencedLibrary.vbproj">
    <Project>{B2BBB9DD-7561-4D2F-8A45-77755E9CC104}</Project>
    <Name> ReferencedLibrary </Name>
  </ProjectReference>
```

4.1.6 Imports

Global imports are imports which apply for every file in the project. They are saved in a list in the *Project* object. They can be found in the *Import* tag which is a child of *Itemgroup*.

```
<Import Include="Microsoft.VisualBasic" />
```

4.2 Lexical analysis

Once all the files are loaded the next step of the code analysis is the lexical analysis. A lexical analyzer or scanner takes the raw source code as its input and turns it into a stream of tokens. A token represents a couple of characters, separated by whitespace, which together mean something in the language analyzed.

Regard the following code which represents a get accessor of a property.

```
1    Get
2       Return _TabSpaces
3    End Get
```

If this code is scanned, we get five tokens; a GET token, a RETURN token, an Identifier token '_TabSpaces', an END token and a GET token. Whether a group of characters is a token or not is fairly context free, in the example 'Get' is a token because it consists of a 'g' followed by an 'e' followed by a 't' and it is surrounded by whitespace. A token doesn't have any further meaning (semantics) at this stage. The first 'Get' in the example has a different

syntactical meaning than the second one, but at this stage it has the same token type. The first one is the start of a get accessor of a property, the second one together with the 'End', is the end of a get accessor. This syntactical difference is determined in the next stage of the code analysis, the syntax analysis.

For the project an external third-party lexical analyzer was acquired and used out of the box (14), so further discussion about this subject falls out of the scope of this thesis.

4.3 Syntactic analysis

Syntax analysis or parsing takes tokens from the lexical analyzer and uses them to build an abstract syntax tree (AST) or parse tree. In this stage syntax, i.e. the order of the tokens is taken in to account to give more meaning to the tokens. Normally the grammar of a language is used to model a parser and the AST. The next piece of Visual Basic code, for example, represents a property.

```
1   Public Property MyProperty() As MyType
2       Get
3           Return _ MyPropertiesBackingField
4       End Get
5       Set(ByVal value As MyType)
6           _ MyPropertiesBackingField = value
7       End Set
8   End Property
```

The grammar for a property in Visual Basic 8 is as follows [3].

PropertyMemberDeclaration ::=
    RegularPropertyMemberDeclaration |
    MustOverridePropertyMemberDeclaration

RegularPropertyMemberDeclaration ::=
    [*Attributes*] [*PropertyModifier+*] `Property` *FunctionSignature* [*ImplementsClause*]
        LineTerminator
    PropertyAccessorDeclaration+
    `End Property` *StatementTerminator*

MustOverridePropertyMemberDeclaration ::=
    [*Attributes*] [*MustOverridePropertyModifier+*] `Property` *FunctionSignature* [
ImplementsClause]
        StatementTerminator

PropertyModifier ::= *ProcedureModifier* | `Default` | `ReadOnly` | `WriteOnly`

MustOverridePropertyModifier ::= *PropertyModifier* | `MustOverride`

PropertyAccessorDeclaration ::= *PropertyGetDeclaration* | *PropertySetDeclaration*

PropertyGetDeclaration ::=
    [*Attributes*] [*AccessModifier*] `Get` *LineTerminator*
    [*Block*]
    `End Get` *StatementTerminator*

PropertySetDeclaration ::=
    [*Attributes*] [*AccessModifier*] `Set` [`(` *ParameterList* `)`] *LineTerminator*
    [*Block*]
    `End Set` *StatementTerminator*

The parser, which is based on the Visual Basic 8.0 grammar, tries to match the tokens in the order specified by the grammar and builds an AST of the tokens.

After parsing the example above, the AST representing the property will look like this:

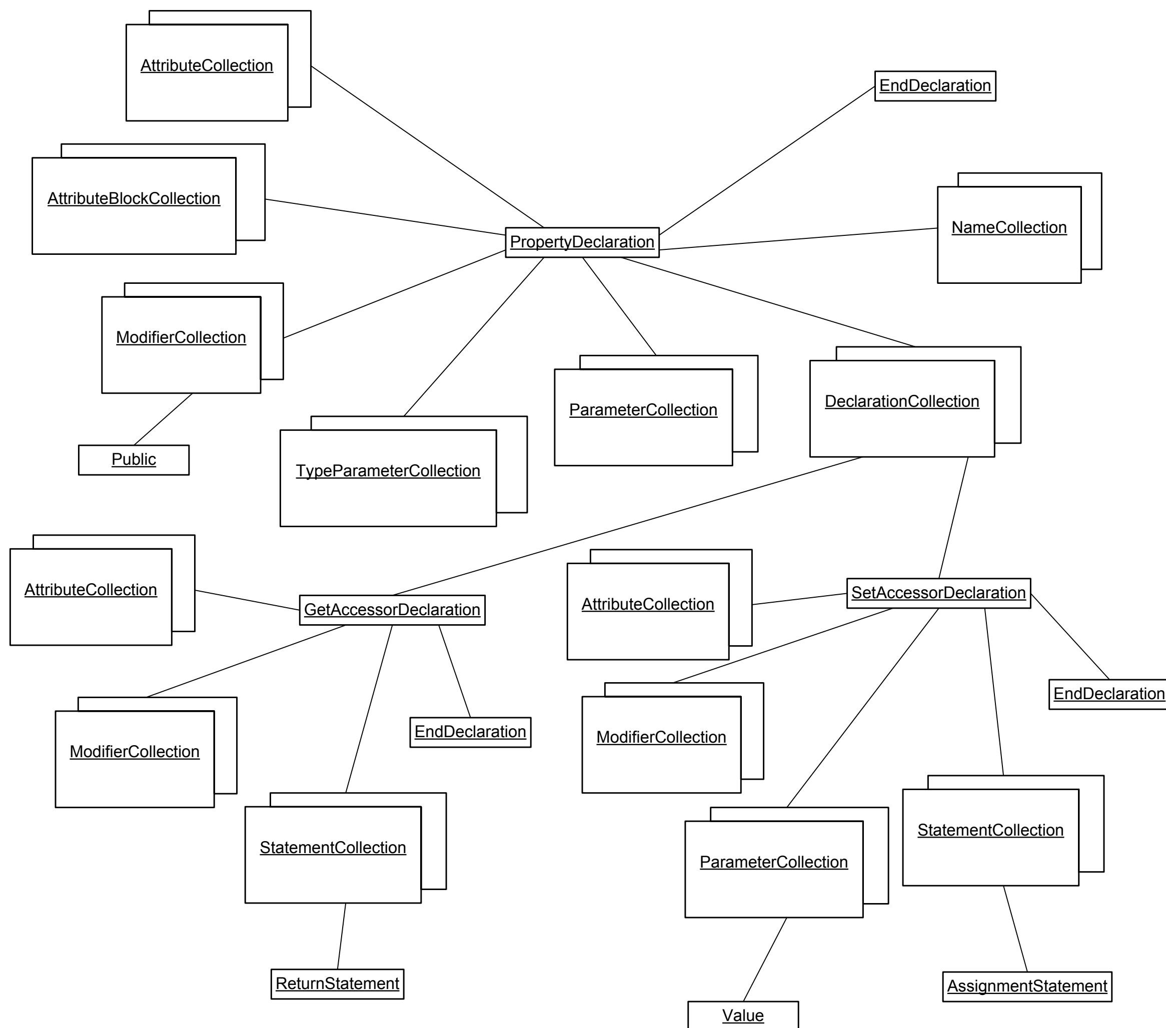


Figure 3 Example of an AST

For every file in the projects loaded an AST is constructed. The complete grammar from the code specification (15) is included in Appendix 1. The

syntactic analyzer that we used in our work was acquired together with the lexical analyzer (14), so the subject is not discussed any further in this thesis.

4.4 Keeping track of scopes

The scope of an entity's name is the set of all declaration spaces within which it is possible to refer to that name without qualification (15). To be able to classify expressions correctly at a later stage it is necessary to know in which scopes to search for declarations. To facilitate the search for scopes, a so-called scope tree is constructed. The scope tree represents the scope structures in the source code as illustrated by the next example:

```
1    Public Class Scope
2       Private var1 As Integer
3       Public Sub ScopeSub1(ByVal var1 As Integer)
4       End Sub
5
6       Public Sub ScopeSub2(ByVal var2 As Integer)
7       End Sub
8    End Class
```

The code in the example has five scopes. The first scope, which is not really obvious is the global scope; the global scope is an implicit (i.e. not declared by explicit syntax) scope which contains everything from all projects loaded in the analysis at a given moment. Every project in Visual Basic has a root namespace; everything within a project is nested in this root namespace. The Class 'Scope' is declared in this root namespace scope and is itself also a scope boundary; the two methods 'ScopeSub1' and 'ScopeSub2' are nested within the class's scope and also represent scopes themselves.

An important step in the syntactic and semantic analysis of source code is qualification of identifiers. By this, we mean finding the complete scope path, from the root scope (global scope) up to the innermost scope which contains the location of a given identifier. When an identifier needs to be classified, scopes are sought through from the scope the identifier is directly contained in, to the outermost scope until the identifier is classified. The scope path produced as described above generates a so-called fully qualified identifier name. Fully qualified names are useful as they uniquely describe identifiers with potentially identical non-qualified names throughout a project.

So when 'var1' is referenced within method 'ScopeSub1', this means the parameter in the method signature on line 3 is referenced. If an identifier with the same name would be used within method 'ScopeSub2', it would mean the field declared on line 2 would be referenced.

This shows that to classify identifiers correctly, it is necessary to know the scope structure in the source code. We should be able to determine which the direct containing scope of an entity is and which entities reside in a scope. It is also

necessary to be able to find the direct containing scope of any scope. To facilitate all these requirements we capture the scope structure in a tree, a scope tree.

How such a scope tree is constructed is discussed next in section 4.4.1. The fully qualified name of a type is tightly coupled to the scope structure; every structure that adds a part to a qualified name of a type it contains, also starts a new sub-scope. The scope tree is thus an ideal structure to keep track of this "qualified location"; this is discussed in section 4.4.2. A further complication to scopes appears: In Visual Basic, the notion of partial types exists. A partial type is a type whose declaration can be distributed in multiple files; still their inner scope is the same one. Section 4.4.3 discusses the scopes of partial types.

4.4.1 Scope tree construction

The scope tree is a tree in which every scope in the code is represented by a node. Every directly nested scope in a scope is represented by a child node of the node representing that scope. Every scope node contains references to scope members (i.e. symbols, or identifiers) directly contained by the corresponding scope in the source code. To be able to determine the directly containing scope for an identifier, for every node in the AST a reference to the corresponding scope node in the scope tree is stored. In this way, we can easily access the scope and, if needed, fully qualified name of any symbol in the AST.

The code shown above results in a scope tree as shown in the diagram below, the scope tree of the example above is shown on the left, a simplified version of the AST is shown on the right. A reference is stored from the AST nodes to the scope node which is positioned between the same dashed lines.

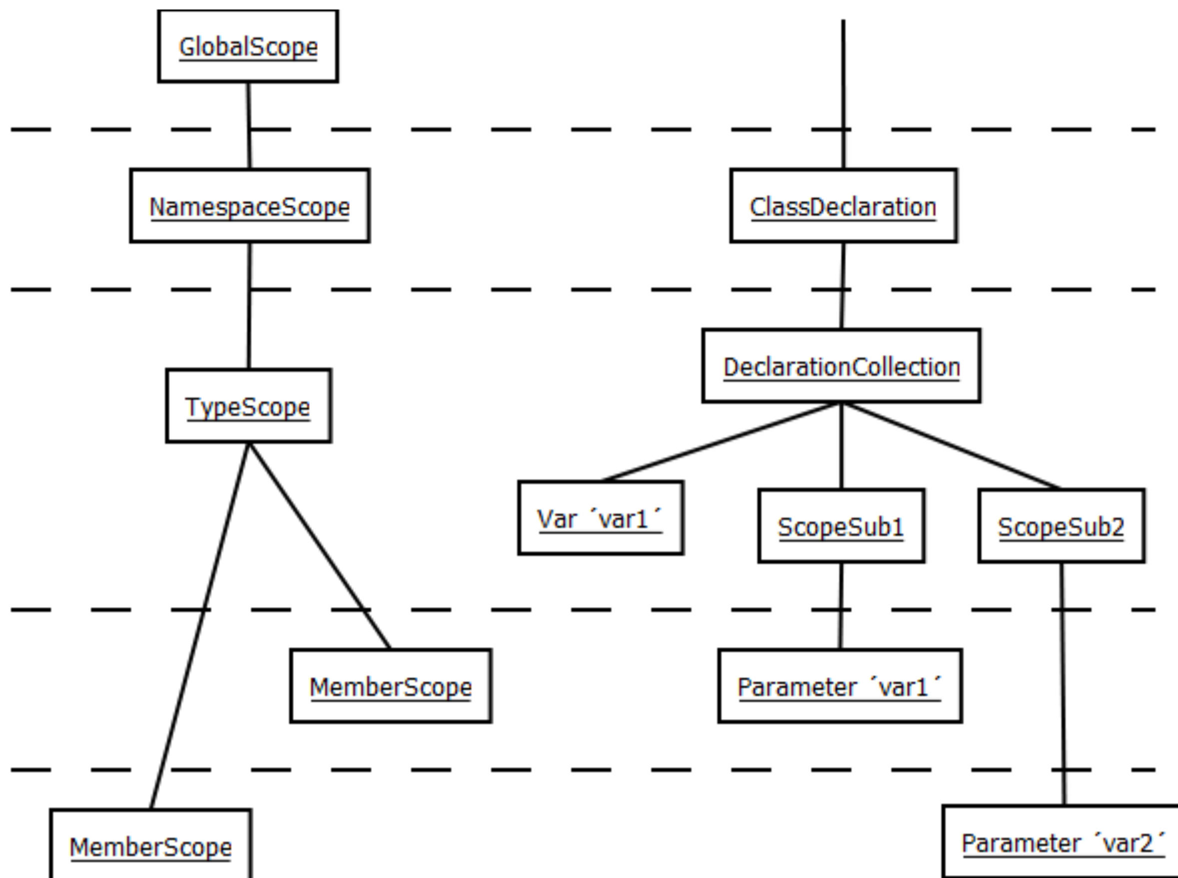


Figure 4 Scope tree (left) with the corresponding AST (right)

To distinguish between different types of scopes, five types of scope-nodes are defined:

- **Global:** A singleton root-node
- **Namespace:** Represents namespace scopes
- **Type:** Represent type scopes; modules, classes, structures, enums and interfaces
- **Member:** Represent member scopes; properties, functions, subs, and so on
- **Block:** Represent block scopes; where, if, and so on

The scope tree is built by recursively traversing the AST of each file, starting at the root, passing a reference to the current scope node while doing so. The root scope node is the same scope for all ASTs, it represents the global scope. Every time a node is encountered in the AST which starts a new nested scope, such as a class declaration, a new scope node is added as a new child of the current scope node. The new scope node is passed on as the current scope when the children of the AST node are recursively traversed.

If the node represents a scope defined by a type, a reference to this type can be stored in the scope node for later use in the type building phase; more about this is explained in section 4.5.

4.4.2 Qualified location

The qualified location of each new scope is the qualified name of the entity which starts the new scope; this can for example be a class name or a method name. The qualified location can be used to determine in which namespaces and locations can be sought for a type. The qualified name is constructed by taking the unqualified name of the entity which start the new scope and qualify it with the qualified location of the containing scope.

Consider the example in 4.4 again , if in the root-namespace of the project is 'Root' then the namespace scope node would have the qualified location 'Root', the type scope node would have qualified location 'Root.Scope' and the member scope nodes the qualified locations 'Root.Scope.ScopeSub1' and 'Root.Scope.ScopeSub1'.

4.4.3 Scopes of partial types

Partial type declarations are class or structure declarations that may be spread across multiple partial declarations within the program (15). This means the declaration space and thus the inner scope of this type is distributed as well among several files.

Consider the following code:

```
1    Partial Class PartialClass
2       Sub DoSomething()
3
4       End Sub
5    End Class
6
7    Partial Class PartialClass
8       Sub DoSomethingMore()
9
10      End Sub
11   End Class
```

This example will result in the class 'PartialClass' with two methods 'DoSomething' and 'DoSomethingMore'. The inner scope of both parts of 'PartialClass' is the same.

Because the inner scope of these partial declarations is the same, they should be represented in the scope tree by the same single scope node. Therefore when a class or structure is encountered, first a scope node is sought within the same project with the same fully qualified location, if such a scope is found it is used instead of creating a new scope. In this way all members which are added to the scopes type are added to the same type and for all AST nodes within the partial type a reference to the same scope is stored.

4.5 Type system

The type system is the heart of the type semantic analyzing mechanism. To be able to determine if two pieces of code are semantically the same, it is necessary to know the types of the identifiers in it. The type system is the central system which keeps account of all types found and referenced in the source code. The type system mainly consists of a type table containing every declared type exactly once. The word 'type' can be a bit hard to define; is used with numerous potentially different, senses in the static analysis literature. Section 4.5.1 discusses how the term is used in this thesis, what types and members exist in Visual Basic and how they are represented in the type system. Because only one representation of every type may exist in the type system, there must be a way to identify them; section 4.5.2 discusses how types are identified in the type system.

4.5.1 Types and members

It is very hard to find a single definition of the word 'type' in the static analysis literature, so first we should clarify the meaning of the word type in this thesis. We don't use the word type as it is often used when talking about compilers, i.e. the semantic information associated with a program construct like a variable, function or class definition. We use here the word 'type' in the same narrow sense it is used in programming languages when referring to a 'data type'. More precisely, in this thesis, types are entities which can be instantiated. For example a class is a type, but also intrinsic types and arrays are types. The only exception to the above rule are modules. A module is not a data type in the sense that it cannot be instantiated, but in our discussions it is regarded also as a type.

Functions are not regarded as types in this thesis (in contrast to some works in compiler theory where functions are types in a type system); they are referred to as members, just as fields, properties or any other structure which is a part of a type. All types can have members; members are declarations directly contained by the types declaration like functions, fields, properties or inner types.

Every kind of type or member in Visual Basic has its own properties, some of them are shared, and some of them are unique to a specific kind. For the complete description of Visual Basic 8 the reader is referred to [3]. Only properties which influence the semantics are of use to us. To represent the types and members we defined a class hierarchy as shown below. The central element is TypeDeclaration, which models a type. Its subclasses model specializations of the type notion (as shown in the lower part of the diagram). As explained above, a type can contain members, modelled by MemberDeclaration. Different kinds of members are modelled by specializations of MemberDeclaration (as shown in the top part of the diagram).

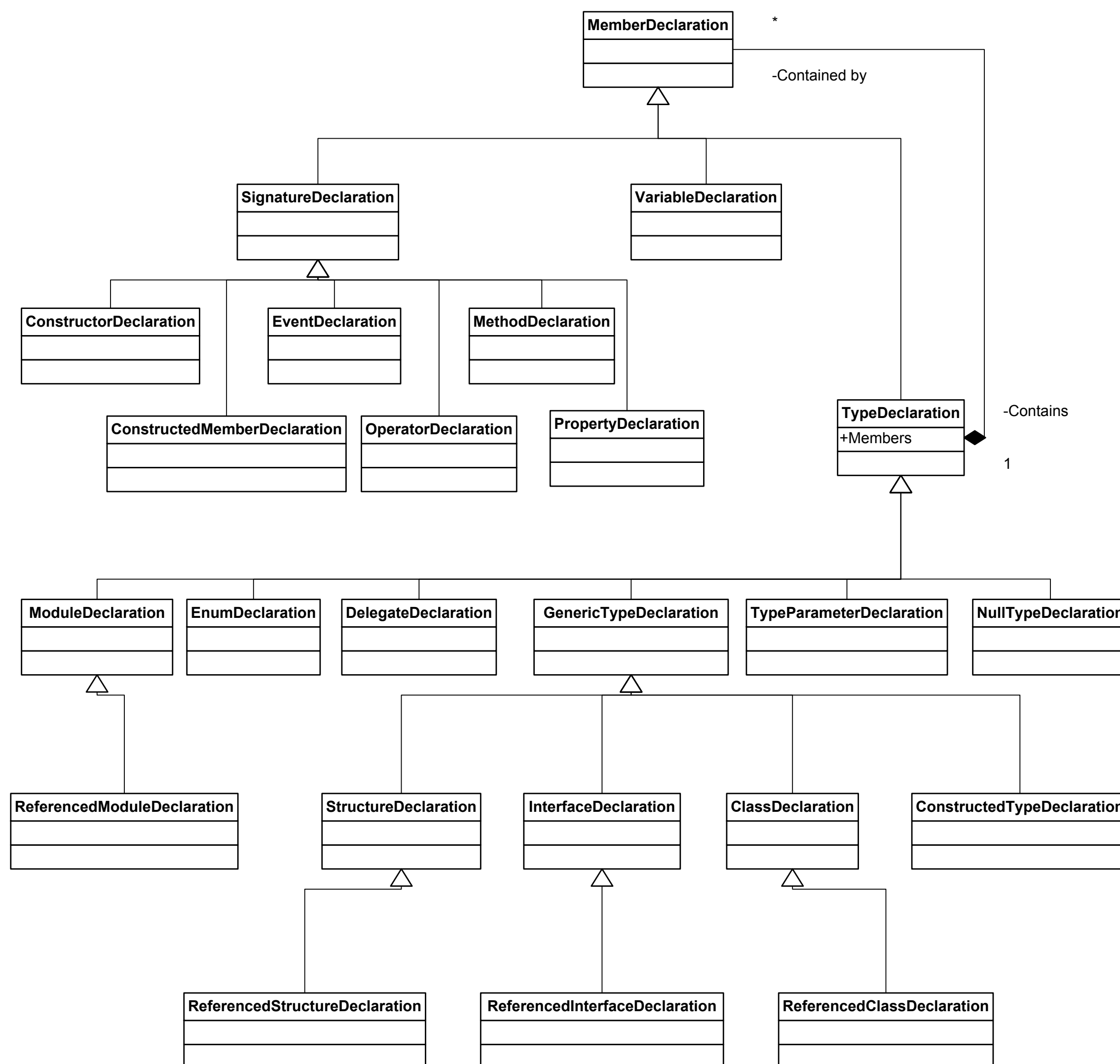


Figure 5 Overview of the type representation hierarchy

In the next sections each of the different kinds of types and members are discussed.

4.5.1.1 MemberDeclaration

The *MemberDeclaration* class is the base of the representation for all types and members. It might seem strange that representations for types are a subclass of the *MemberDeclaration* class because a type often isn't a member at all. Types can be nested though; if this is the case they should be regarded a member of the containing type; to model this, it is convenient to have both types and other members have the same super class. If a type is not a nested type it could with some leeway be considered a member of its containing namespace and therefore this construction is justified.

The *MemberDeclaration* class contains the following properties[1]:

[1] The type system is implemented itself, as Visual Basic, in the .Net framework, hence the use of similair terminology here and in the following discussions.

- **Name:** the name of the entity.
- **IsShared:** a flag, which when set means the member is static; this means it can be accessed directly on the uninstantiated type instead off only on an initialized one.
- **Shadows:** a flag, which when set means all members with the same name contained by super types of the containing type, are hidden.
- **AccessModifier:** the access modifier defines how accessible the member is, there are five values the access modifier can have:
  - **Public:** accessible to any type.
  - **Private:** accessible only within the containing type.
  - **Protected:** accessible only from within containing type and subtypes of it.
  - **Friend:** accessible only from within the same project.
  - **ProtectedFriend:** accessible from types within the same project or from subtypes of which this is a member.

4.5.1.2 SignatureDeclaration

The *SignatureDeclaration* class is the base class of all members that have a signature, i.e. have a return type, a name and parameters. The class contains the following properties:

- **ParameterTypes:** the parameters in the signature. This property consists of a list of objects of the type *ParameterType*. The *ParameterType* class has the following properties:
  - **Type:** a reference to the *TypeDefinition* which represents the type of the parameter
  - **Name:** the name of the parameter
  - **IsByRef:** a flag which when set means a reference to the value is passed instead of the value itself.
  - **IsOptional:** a flag which when set means the parameter is optional.
  - **IsParamArray:** a flag which when set means the parameter is an array which can be passed as a list of comma separated parameters.
- **TypeParameters:** Represent the type parameters of the member when the member is a generic member. Within the scope of this member they behave like types themselves. More about type parameters in section 4.5.1.15.

4.5.1.3 ConstructorDeclaration

The *ConstructorDeclaration* class represents a constructor of a type. It is a subtype of *SignatureDeclaration* and it has no further properties on its own. It is necessary to be able to distinguish between constructors and other methods because constructors can only be called at the moment a type is instantiated.

4.5.1.4 EventDeclaration

The *EventDeclaration* class represents an event declaration in a type.

4.5.1.5 MethodDeclaration

The *MethodDeclaration* represents a Sub or Function definition (in Visual Basic terminology). It is a subclass of *SignatureDeclaration* and does not have extra properties. A sub and a function are both methods with the difference that sub returns void and a function always returns a type.

4.5.1.6 GenericTypeDeclaration

The *GenericTypeDeclaration* class is the base class for all so-called generic types. A generic type is a type which has generic type parameters. These parameters are placeholders for types that are only specified at the place an instance is made of the type. The class contains the following properties:

- **ImplementedTypes:** A list of *TypeDeclarations* which represent the interfaces the type implements.
- **TypeParameters:** A list of *TypeParameters* which represent the type parameters of this generic type.

4.5.1.7 ConstructedMemberDeclaration

The *ConstructedDeclaration* class represents a specific case of a generic member. The class provides functionalities to represent its containing generic type but with the type parameters replaced by actual types. It has the following properties:

- **GenericMember:** A reference to the generic signature declaration of which this is a instantiation.
- **TypeArguments:** Represent the arguments which are matching type parameters of the signature. It consists out of a list of references to type definitions, for every type parameter in the generic signature declaration a type argument exists at the same index in the list.

4.5.1.8 OperatorDeclaration

The *OperatorDeclaration* represents a user defined operator overloading declaration. It contains the following properties:

- **OperatorToken:** The token initially produced by the lexical analyzer, it's used to determine which operator is overloaded.
- **IsWidening:** Applicable when the operator is the Visual Basic 'CType' operator. The 'CType' operator is used to cast an object of one type to another. For specific cases this cast can be overloaded, this overload should explicitly state if the cast is a widening or a narrowing cast. Widening casts can be done with no loss of data and therefore are implicit. No extra effort has to be done to write one type into a variable of another type if there is a widening cast available. A narrowing cast means data or precision could be lost. A narrowing cast is explicit, the 'CType' operator should be added, else the compiler won't accept the cast. This property is set when the operator is a widening 'CType'.
- **IsNarrowing:** See above. This property is set when the operator is a narrowing 'CType'.

4.5.1.9 PropertyDeclaration

The *PropertyDeclaration* class represents a property of a type as present in the Visual Basic language. It contains the following properties:

- **IsDefault:** When set states the property is default. A default property is a property that is called when an instance of its containing type is indexed.

The next type in the next example has a default property.

```
1 Public Class DefaultPropertyClass
2    Private _list As List(Of Integer) = new List(Of Integer)()
3
4    Public Default Property ListItem(ByVal index As Integer) As Integer
5       Get
6          Return _list.Item(index)
7       End Get
8       Set(ByVal value As T)
9          _list.Item(index) = value
10      End Set
11   End Property
12 End Class
```

When a variable is declared with the type:

```
Dim defaultPropertyValue As DefaultPropertyClass =
     New DefaultPropertyClass()
```

Then the next expression will call the default property:

```
Dim defaultPropertyResult As Integer = defaultPropertyValue(1)
```

4.5.1.10VariableDeclaration

The *VariableDeclaration* represents a variable declaration in a type.

4.5.1.11 TypeDeclaration

The *TypeDeclaration* class is the base class for all types. It contains the following properties:

- **Members:** A list of *MemberDeclarations* which represent the members of the type.
- **Bases:** A list of *TypeDeclarations* which represent the base types of the type, usually this is only one, but in case of an interface it can be more.
- **QualifiedPath:** The full namespace path of the type. For example a type is declared within 'Namespace1' in a project with a root namespace 'Namespace0' the QualifiedPath will be 'Namespace0.Namespace1'.
- **ProjectGuid:** The id of the project this type is declared in.

4.5.1.12 ModuleDeclaration
The *ModuleDeclaration* class represents a module. A module is a static type of which instances cannot be made. When a module is imported, its members will become available everywhere within the file, or project it's imported in.

4.5.1.13 EnumDeclaration
The *EnumDeclaration* class represents an enum. It contains the following property:

- **UnderlyingType**: An enum is actually just a number represented by a name. Its underlying type can be changed, it can be one of Byte, SByte, UShort, Short, UInteger, Integer, ULong or Long. This property holds a reference to the *TypeDeclaration* representing the underlying type.

4.5.1.14 DelegateDeclaration
The *DelegateDeclaration* class represents a delegate declaration. A delegate has a signature and therefore one could think it should be a subclass of the *SignatureDeclaration* class, but because instances can be made of it and it has extra default members, it behaves more as a type. The signature is added as a nameless method declaration.

4.5.1.15 TypeParameterDeclaration
The *TypeParameterDeclaration* class represents a type parameter of a generic type. It contains the following property:

- **Constraints:** This is a list of *TypeDeclarations* which represent the constraints of this type parameter. It means the type should be of the (sub-)types in this list. This information is useful because these constrains imply that members that exist in instances of one of these types, also exist in the type parameter that has these constrains. These constrains thus are necessary to resolve the members on this type parameter. There is also a special case of constraint, the 'New' constraint means the type should have a parameterless constructor. This constraint however is uninteresting for the semantic analysis and thus ignored.

4.5.1.16 NullTypeDeclaration
The *NullTypeDeclaration* class represents the 'Nothing' Visual Basic keyword. It represents the value of a reference to an instance of a certain type, not pointing at anything, also known as NULL or null in other programming languages like C or C++.

4.5.1.17 StructureDeclaration
The *StructureDeclaration* class represents a structure or compound type.

4.5.1.18 InterfaceDeclaration
The *InterfaceDeclaration* class represents an interface.

4.5.1.19 ClassDeclaration
The *ClassDeclaration* class is represents a class.

4.5.1.20 *ConstructedTypeDeclaration*

The *ConstructedTypeDeclaration* represents a specification of a generic type. The class contains functionality to represent itself as if it was a normal type. This is done to substitute all the instances of the type parameters with its type arguments. It contains the following properties:

- **GenericType:** A reference to the *GenericTypeDeclaration* this type specifies.
- **TypeArguments:** Represent the arguments which are matching type parameters of the generic type. It consists out of a list of references to type definitions, for every type parameter in the generic type declaration a type argument exists at the same index in the list.

4.5.2 Identifying types with fully qualified names and assembly identifiers

The core of the type system is a collection of every type known in a given program. Every type has a unique key (among all possible types in all projects) which is constructed of a project identifier and a fully qualified name. A fully qualified name contains the complete type path, the types name and possibly its generic rank, array rank or type arguments.

For example the next code with root namespace 'Root':

```
Namespace Namespace1
  Class Class1
    Class Class2

    End Class
  End Class
End Namespace
```

In this example there are two types, class Class1 and class Class2, the fully qualified names of these two will be 'Root.Namespace1.Class1' and 'Root.Namespace1.Class1.Class2'.

It is not allowed for a type reference to be ambiguous; it always should be possible to resolve the type or the source code cannot compile. With the precondition that the code must be compilable, i.e. is complete and correct, this guarantees a reference is never ambiguous. This means a distinction between referenced types can always be made, which implies fully qualified names of referenced types are always unique in the containing type of the type reference. Thus the fully qualified name together with the reference containing type is always enough to identify the correct type referenced somewhere in a program.

It is possible to have a reference to a type for which another type exists with the same (fully qualified) name within the same environment, which can only be distinguishable by the context of the type reference. If for example a project *MainProject* references two other projects *ReferencedProject1* and *ReferencedProject2*, *ReferencedProject1* contains a type with the fully qualified name *Namespace1.Class1*, *ReferencedProject2* containing a reference to project

ReferencedProject3 which on his turn also has a type with the fully qualified name *Namespace1.Class1*. If within *MainProject* a type is referenced with the fully qualified name *Namespace1.Class1* or part of this name, from the context can be deducted the type from project *ReferencedProject1* is the one referenced. This because project *ReferencedProject3* is not referenced by *MainProject*, thus types in *ReferencedProject3* are not visible in the context of the type reference.

Within a project or a library, two types with the same fully qualified name are not allowed. This, together with precondition the code has to be compilable, makes the fully qualified name within a project or library guaranteed unique.

To distinguish between two types with the same fully qualified name, a unique key is constructed from the identifier of the containing project or library, together with the fully qualified name. How this key is used to find a referenced type is discussed in chapter 5.

4.6 Populating the type system

In the previous section was shown what the type system is and what kind of types and members can be stored in it. Before the type system can be used for classification it needs to be populated with all the types existent in the program analyzed. Constructed and array types can be added when needed, but all basic types need to be present for correct resolution.

The next diagram shows an overview of the type system construction process.

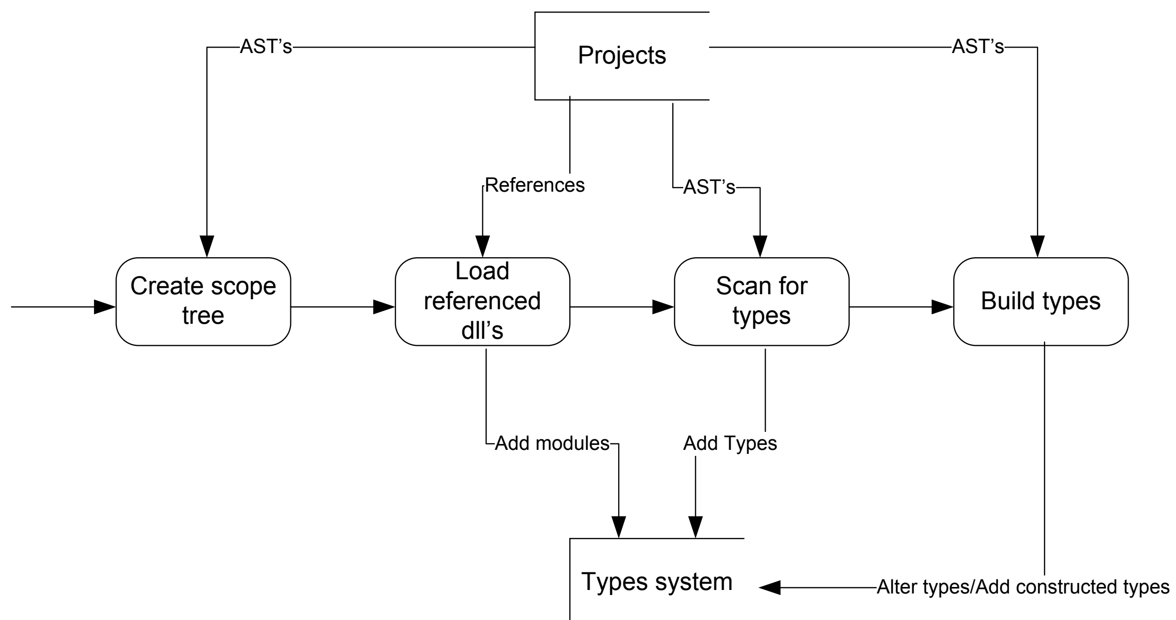


Figure 6 Overview of the type system construction

Populating the type system is done three stages. The first step is library reference resolving; in this stage referenced libraries are loaded and types within the libraries are extracted; this step is discussed in section 4.6.1. After this, all types declared within the analyzed code are added. At first the types are scanned

superficially; only their name, type and location are extracted. In the stage following the types are further constructed, and their inheritance and all of their members are extracted. Section 4.6.2 discusses why the types are extracted in two stages. Section 4.6.3 describes the type scanning and type building is described in section 4.6.4.

4.6.1 Library reference resolving

Not all types referenced within the code files can be found in the code itself; some of the types reside in external libraries. For proper type resolution it is necessary to load the referenced types into the type system. In this stage of the type system population, the referenced libraries are loaded and analyzed.

First the referenced libraries need to be found. System libraries are often referenced by name only. These system libraries can be found at different standard locations. To find a referenced library these standard locations are probed to find it.

When a referenced library is found, reflection is used to analyze the types present in the library. Reflection is a technique that for dynamic type loading from libraries and examine their properties. When needed types are loaded from the referenced libraries, analyzed and stored into the type system. Reflection is natively supported in the .NET framework.

Not all referenced types are loaded straight away; for performance reasons referenced types are loaded lazily. The libraries are loaded in memory, but not yet analyzed; only modules, which are a special case are extracted from the libraries at this point.

The rest of the types are only loaded only on request by the type system. When a type reference is being resolved by the type system, it sends a request to the reference resolver which tries to find a type with that name in the appropriate library and just in time adds it to the type system when found. This way, type's only get loaded when needed and no resources are wasted in loading types that are never used.

Not only are types from referenced libraries loaded lazily, a type's members, bases and implemented types are loaded lazily as well. Members of a type often have a return type or parameter types. Constructing referenced types together with their members would mean all types used in these members have to be loaded as well. These types have also members of which the types also should be loaded. So still a lot of types would be loaded which never are used in the code that is to be analyzed.

To load members lazy the type classes are sub-classed with special reference versions which load members, bases and implemented types only when needed, using a design pattern similar to the well-known Proxy pattern. These sub classes are:

- **ReferencedModuleDeclaration**, a subclass of *ModuleDeclaration.*
- **ReferencedClassDeclaration**, a subclass of *ClassDeclaration.*
- **ReferencedStructureDeclaration**, a subclass of *StructureDeclaration.*
- **ReferencedInterfaceDeclaration**, a subclass of *InterfaceDeclaration.*

These classes hold a reference to the library type. When a member's property is accessed for the first time, the library type is analyzed and the type's members are added together with the types used in these members. The same happens with the bases and implemented types properties, the bases and implemented types are analyzed and added the first time the property is accessed.

The performance gained with lazy reference loading is considerable; the number of types loaded because of standard imports went down a factor 10..15. Because of this 10ths of seconds where shaved of the type system building process.

As stated above modules are loaded without delay. As will be explained later, it should be possible to find all modules in given namespace, without knowing their name in advance. The standard functionality of reflection doesn't provide functionality to find a type by the namespace it is contained in. To be able to find modules, all modules are loaded from referenced libraries without delay.

4.6.2 Analyzing types in two stages

Members often have return types or parameter types. To construct a representation of these members, those types have to be resolved as well. If types would be loaded and constructed completely in one pass, this would mean that at some point a type needs to be resolved while the type system is not completely populated yet. It is obvious this would mean there is a chance the type sought would be not yet present in the type system and no type is found, but worse it could also happen a type is found but it is not the correct one, so type resolution isn't guaranteed to be correct before all types are loaded. Because of this type resolution can only be done after all types exist in the type system, this implies type loading has to be done in two passes.

There are different approaches possible to split the loading in two stages. For example the type reference could be stored in some sort of 'pseudo type' placeholder to be resolved in the second pass. For convenience is chosen to only load the skeleton of the types, only the information needed to resolve a type, in the first pass, i.e. the name, the access modifiers and the generic rank of the type. All the members and inheritance are loaded in the second pass.

4.6.3 Type scanning

In this step of populating the type system we find out which types are declared in the source code files. As described in the previous section, in this stage only a skeleton of every type is added to the type system.

To scan for types, the process walks through the AST of all source files in the program. Every time a type is encountered, the type is added to the type system. The qualified location is extracted from the corresponding scope node

and the qualified location of the type is set to this value. The name of the type, generic rank and access modifier is extracted from the AST node and saved in the type representation.

Next to the types themselves, type parameters are treated as types as well and added to the type system.

Together with the type scanning two other types of statements are scanned, the import statement and the with statement. These are piggybacked onto this stage because separately walking through the AST just to find these statements would be a relatively big impact on the performance.

4.6.3.1 Imports

There two types of imports namespace imports and alias imports.

```
ImportsStatement ::= Imports ImportsClauses StatementTerminator

ImportsClauses ::=
        ImportsClause |
        ImportsClauses , ImportsClause

ImportsClause ::= ImportsAliasClause | ImportsNamespaceClause

ImportsAliasClause ::=
        Identifier = QualifiedIdentifier |
        Identifier = ConstructedTypeName

ImportsNamespaceClause ::=
        QualifiedIdentifier |
        ConstructedTypeName
```

In the AST these are represented by *NameImport* and *AliasImport* objects. When one of these representations is encountered, the import is added to the list of imports in the current *CodeFile* object for later use in type resolution.

4.6.3.2 With block

A with block is a block of code where an expression is first specified, and then implicitly used.

```
WithStatement ::=
        With Expression StatementTerminator
        [ Block ]
        End With StatementTerminator
```

For example the following code:

```
1    With Var
2        .FunctionOnVar()
3        .AnotherFunctionOnVar()
4    End With
```

Here 'Var' is the With-expression, the variable produced by the expression is implicit within the with block. The two function calls in the block are called on the variable produced by the With-statement, thus on the variable 'Var'.

In the AST it is represented with a *WithBlockStatement* object. The expression is saved in the current scope node for later use in type resolution.

4.6.4 Type building

Once all types are known, the skeleton types scanned in the previous phase can be filled in. Its members need to be extracted from the code, implemented types, inherited types, type parameter constraints and underlying types. Next to members and other dependencies, it is also necessary to be able to qualify local variables, so these are added to the system as well.

For this the ASTs are traversed one more time, as follows.

In the next sections, we discuss members and variables are discussed, implemented types, inherited types, type constraints, and underlying types.

4.6.4.1 Members and variables

When a member or variable is encountered, the scope node corresponding with the AST node representing the member or variable is fetched. The member or variable is added to the list of members in the scope so it can be found during classification.

After this the scope is searched for the innermost type containing the member or variable, if the qualified name of this type equals the qualified location of the scope, it means the member is in fact a member of the type and not a local variable. If a variable is a member, it is often referred to as a field. If it is a member, it is added to the member list of the type.

The following kinds of members are added to types:

- Sub
- Function
- Property
- External Sub
- External Function
- Event
- Custom Event
- Constructor
- Field

Note that nested types are not treated as the rest of the member. They are in fact members but they are treated as separate types; there is no way of listing the nested types of a type.

As already explained earlier, a signature member is a member that has a name, parameters a return type and possibly type parameters. Except for variables, all

of these members have such a signature and have the same basic form. A variable has just a name and a type. All members, including fields can have access and other modifiers.

If a member is encountered while traversing an AST, the name and modifiers are extracted and saved in a member representation. After this all dependent types, which are the result type, parameter types and type parameter types, are resolved and saved in the member representation.

The scope node corresponding with the AST node representing the member is fetched. The member is added to this scope. The scope is searched for the innermost type containing the member, if the qualified name of this type equals the qualified location of the scope, it means the member is a direct member of the type and it is added to its member list. A member is for example not a direct member if it is a variable in a method.

4.6.4.1.1 Extracting member access modifiers

For all members their access modifiers are extracted and saved in the member's representation.

An access modifier is a modifier which defines the access rights to the member from outside the type it which is declared. The access modifiers can be one of the following:

- Public
- Protected
- Friend
- Private

The protected and friend can be combined. The use of an access modifier is not mandatory; if no access modifier is provided, the default access modifier is public. For completeness:

Public means there is no access restriction to this member. Protected means the member is only available to sub-types. Friend means the member is only available within the same assembly. Private means no access is allowed from outside the type. The combination of friend and protected means the member is available from within the same assembly as well as to all sub-types.

4.6.4.1.2 Extracting other member modifiers

Next to access modifiers there are some other modifiers which can influence the semantics of a member. Some of these are extracted and saved in the member representation, others can be safely ignored because of the precondition the input code should be compilable. Of all non access modifiers only the shadows and shared modifier are saved.

The shadows modifier hides all members with the same name in super-types, this modifier influence type resolution so it is saved.

When a member has the shared modifier it is accessible on the uninitialized type. In many other languages the word static is used for this. To which member a member call resolves on an uninitialized type is dependent on this modifier, thus it is saved.

4.6.4.1.3 Extracting type parameters

If a member has type parameters and thus is a generic member, the type parameters are extracted and added to the member representation in the same order as they have been put in the code*.* The type parameters are also added to the type system as they can be seen as types themselves within the member's scope.

4.6.4.1.4 Extracting a member return types

After extracting the type parameters, the return type is resolved using the type system. If the member does not have a return type specified, the return type implicitly is the "System.Void" type.

4.6.4.1.5 Extraction of member parameters

When the member has parameters, their types are resolved using the type system. Parameters are added to the member in a list of objects of the type *ParameterType*. Parameters can be optional, when the optional keyword is added, the IsOptional property in the *ParameterType* object is set to true. The last parameter in the list can be a parameter array, if the 'ParamArray' keyword is provided, the IsParamArray property in the *ParameterType* object is set to true. Within the scope of the member it is a variable, so a variable with the parameter name and type is added to the current scope node.

4.6.4.1.6 Member type specific processing

Next to the standard member properties, some member types need some extra processing done, as follows.

In the scope of a Function the function name can be used as a variable, the function result will be the content of this variable if not a specific value is returned. A variable with the name of the function and the functions return type as type is added to the scope of the function, to be able to resolve this variable.

When the member is an operator, the operator token is saved. An operator can have two special modifiers, the Narrowing and Widening modifier. One of these modifiers should be provided when the operator is of the type CType which is the cast operator. The Widening modifier states a cast can be done without data loss. The Narrowing modifier states the cast can lead to data or precision loss. The *OperatorDeclaration* has the properties IsNarrowing and IsWidening which are set to true when the modifiers are provided.

4.6.4.2 Variables

Variables can be members or local. Their type is resolved using the type system. All variables encountered are added to the scope, the variables which are a member are also added to the members of its containing type.

4.6.4.3 Generic types

Generic types can inherit and implement other types. Generic types are classes, structures and interfaces. When the definition of a generic type is encountered, the types which are inherited and implemented are resolved using the type system and saved in the *GenericTypeDeclaration* in the properties Bases and ImplementedTypes.

4.6.4.4 Underlying types

An enum has an underlying type. If this underlying type is provided, the type is resolved using the type system. If the type is not provided, the default type is "System.Int32".

4.6.4.5 Type Parameters

If a type parameter is encountered. Its type constraints are extracted. There are two types of type constraints, one is the new keyword and the other is a type name. The new keyword means a type should have a default constructor. This doesn't add any information which useful to resolving, so this constraints are ignored. The type name constraint means, the type should be an instance of this type or implement it. All type constraints are resolved using the type system and added to the *TypeParameter* object.

4.6.4.6 Delegate methods

Delegate subs and functions are a particular feature of Visual Basic. They are a hybrid between a type and a member. They can be called as a normal sub or function, but they also inherit the *Delegate* class which implement some default specific methods. These default methods are:

- **Invoke**, which has the same parameters as the delegate method.
- **BeginInvoke**, which has the same parameters as the delegate method plus an extra parameter of type "System.AsyncCallback" with the name DelegateCallback and a parameter of type "System.Object" with the name DelegateAsyncState.
- **EndInvoke**, which has the parameters as the delegate method which are passed by reference together with a parameter of type "System.AsyncCallback" with the name DelegateCallback.

These methods are added to the members of the *DelegateDeclaration* object.

4.7 Summary

In this chapter has been described how source code is analyzed, resulting in an abstract syntax tree for every code file associated with a project and a type system containing a representation of every type used in the analyzed program. This is done in four stages; project analysis, lexical analysis, syntactic analysis and semantic analysis.

In the project analysis a project file is analyzed and all properties, associated project, source files and libraries are loaded. In the lexical analysis every source file is converted into a stream of Visual Basic specific tokens. These tokens are

used in the syntactic analysis to construct an abstract syntax tree for every file. After this first the referenced libraries are analyzed, modules in the libraries are loaded completely, all other types are loaded lazily when needed. In the semantic analysis the types declared in the source code itself are extracted from the ASTs in two phases; first only the name and rank of the type is extracted. After all types extracted, the types are built further. Finally, a complete type system is created which enables us to carry out several types of semantic analyses required for the clone detection and refactoring steps described next.

5 Classification

In the previous chapter was explained how abstract syntax trees and scope trees are built and the type system gets populated. The last step in the semantic analysis is to *classify* every expression in the source code using the type system.

By classification, we mean analyzing expressions to deduce refined information which will further enable us to detect clones and ultimately refactor them. The term 'classification' used here is of our own choice. In the compiler literature, similar refined analyses on syntax and semantic information have different names, e.g. disambiguation, elaboration or advanced semantic analysis. This step is typically related to the separation of syntax and semantic analysis: typical (simpler) parsers operate context-free. However, in context-dependent languages, such as Visual Basic or C/C++, some constructs cannot be fully classified (i.e. determined what they are) at parsing stage, so this step is left for the subsequent semantic analysis. Since there is no established name for the specific type of analysis we do, and since our analysis is quite specific to our goals of clone detection and refactoring, we chose to introduce our own term: classification.

This step is quit a elaborate one; many resolution steps and rules apply to finding the correct type, member or operator. Instead of classifying each node in the abstract syntax tree, complete expressions are classified. The result of the classifying is formed in such a way the complete origin of the data type and value to which an expression classifies is deductable from the result, the result format is discussed in section 5.1. The classification of identifiers can be divided in three processes; type resolution, member reference classification and expression classification. If a node of the AST needs to be classified, it is deductable from the type of the node which of the processes is needed to classify the node. If the AST node refers to a data type, the type resolution process is needed; type resolution is discussed in section 5.2. If the node is an identifier, but doesn't refer a data type, it must refer a member or local variable, the classification of such nodes are discussed in section 5.3. The last classification case is the classification of expressions which is discussed in section 5.4.

5.1 Classification Result

To be able to match for example two expressions, it is necessary to know how they are related to each other. It's not only necessary to know if the types they classify to are equal or interchangeable, but it is also necessary to know which variables and members are accessed resulting into this type. If for example a certain member of a certain type is called on two instances of this type, it can only be said the two are the same if the instance of the type is the same; the origin of the instance of the type is thus important to know.

Often the result of a classified expression influences the classification of another expression. For example to which overloaded method is applicable depends on

the types its arguments classify to or sometimes even to which constant value the argument results in.

To provide the information needed for further classification and matching, all classification algorithms in the project return an object of the type *ResolveResult*. This object describes the origin of the resulting type of an expression. The information the object contains is dependent on the type of node which got classified. The *ResolveResult* always contains the classified data type the expression results in.

Some type members are accessible on types which aren't instantiated, others only on instances of types. Therefore the result always has a flag which states the resulting type of an expression is instantiated or not. If the instantiated flag is set, it means the resulting type should be interpreted as an instance of the type, if not it should be interpreted as a reference to the type itself.

Constant expressions of some intrinsic types can implicitly be converted to a narrower type when the value is within the range of the destination type. If the result of an expression is the argument in a method call or an operand of an operator, it the fact the expression is constant or not can influence the resolution process. Therefore the *ResolveResult* has a flag which states the resulting value is constant when set.

There are different implementations of the *ResolveResult* each reflecting a different classification case. The hierarchy of the *ResolveResult* types is shown below and explained in the following.

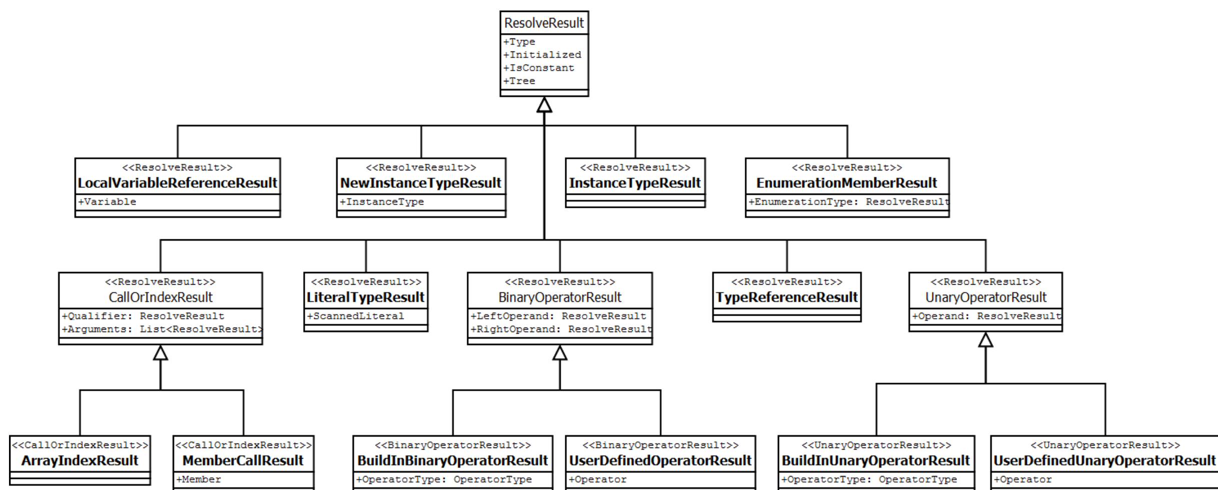


Figure 7 Overview of the result types of the classification process

LocalVariableReferenceResult

When an expression is classified as a local variable, which is a variable which is declared within the scope of a member, a *LocalVariableReferenceResult* is returned. It contains a reference to the variable referenced.

NewInstanceTypeResult

The *NewInstanceTypeResult* is returned when the classified expression instantiates a type *NewInstanceTypeResult* holds a reference to the *ResolveResult* of the type itself.

InstanceTypeResult

The *InstanceTypeResult* is returned when the resolved expression is an instance expression or if the type was implicit. An instance expression is the Visual Basic keyword Me, MyClass or MyBase, which refer to the type itself and the base type. When a member call is done on a member of the containing type, the member does not have to be qualified, in this case an *InstanceTypeResult* for the containing type is added.

EnumerationMemberResult

The *EnumerationMemberResult* is returned when the expression resolves to a member of an enumeration. *EnumerationMemberResult* holds a reference to the *ResolveResult* of the qualifying enumeration .

ArrayIndexResult

The *ArrayIndexResult* is returned when the expression classifies to the indexing of an array. It holds a reference to the *ResolveResult* of the array itself and a list of *ResolveResults* of the arguments used to index the array.

MemberCallResult

The *MemberCallResult* is returned when the expression classifies to a member call. It holds a reference to the member, a reference to the *ResolveResult* of the qualifying type and a list of *ResolveResults* of the arguments passed in the member call.

LiteralTypeResult

The *LiteralTypeResult* is returned when the expression classified is as a literal, the *LiteralTypeResult* contains the scanned literal. This variant of the *TypeResult* has the constant flag always set.

BuiltInBinaryOperatorResult

The *BuiltInBinaryOperatorResult* is returned when the expression classified is a binary expression of which the operator resolution results in a built in operator. *BuiltInBinaryOperatorResult* contains the *ResolveResult* of both operands and the operation type. The constant flag is set when both operands are constant and the operator type is ´plus´, ´minus´, ´multiply´, ´power´, ´modules´, ´divide´, ´integral divide´, ´and´, ´or´ or ´xor´.

UserDefinedOperatorResult

The *UserDefinedOperatorResult* is returned when the expression classified is a binary expression of which the operator resolution results in a user defined operator. *UserDefinedOperatorResult* contains the *ResolveResult* of both operands and a reference to the operator. The fact that the operator is user defined implies that at least one of the operands is a non intrinsic type, therefore the constant flag is always unset.

TypeReferenceResult

The *TypeReferenceResult* is returned when an AST node classifies to a type. This is the only variant of *ResolveResult* which has instantiated flag unset.

BuiltInUnaryOperatorResult

The *BuiltInUnaryOperatorResult* is returned when the expression classified is a unary expression of which the operator resolution results in a built in operator. *BuiltInUnaryOperatorResult* contains the *ResolveResult* of the operand and the operation type. The constant flag is set when the operand is constant.

UserDefinedUnaryOperatorResult

The *UserDefinedUnaryOperatorResult* is returned when the expression classified is a unary expression of which the operator resolution results in a user defined operator. *UserDefinedUnaryOperatorResult* contains the *ResolveResult* of the operand and a reference to the operator. The fact that the operator was user defined implies the operand cannot be an intrinsic type, therefore the constant flag is always unset.

Consider for example the following code:

```
1   Class A
2      Dim instanceOfB As B
3      Sub DoSomething()
4         Dim var1 As Integer
5         Dim var2 As Integer
6         instanceOfB.Something(var1, var2)
7      End Sub
8   End Class
9
10  Class B
11     Function Something(ByVal param1 As Integer, ByVal param2 As
Integer) As Integer
12        Return 0
13     End Function
14  End Class
```

If the expression ´InstanceOfB.Something(var1, var2)´ in line 6 is classified, the result will be a *MemberCallResult* where the type will be ´System.Int32´, and the member will be a reference to the method ´Something´ of class ´B´. In

the arguments are the two argument *ResolveResults*, both of type *LocalVariableReferenceResult* with type ´System.Int32´ and references to variables ´var1´ and ´var2´. In the qualifier of the *MemberCallResult* there is a also a *MemberCallResult* with type ´B´, its member will be de ´instanceOfB´, its qualifier is an *InstanceTypeResult* of type ´A´.

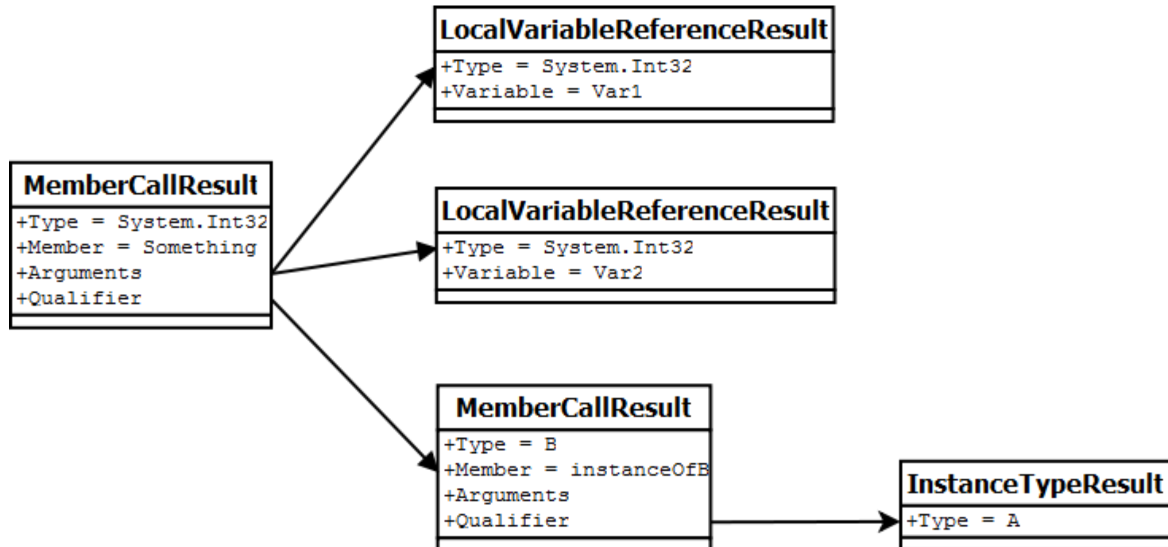


Figure 8 Example of a classification result

5.2 Type classification

In the type classification process, for a type reference in the code the correct type needs to be found in the type system. It can be that a type reference is just the simple name of a type, but it can also be the type is qualified. As discussed in section 4.5.2, sometimes different types with the same simple name can exist. To which of those types the reference classifies depends on multiple factors; which namespaces are imported in the file or in the project, which assemblies and projects are referenced by the project, which is the containing namespace of the type reference and how is the type reference qualified.

Section 5.2.1 discusses how, given a fully qualified name, can a type be fetched from the type system which is reachable and visible from a type reference. Modules are constructions in Visual Basic of which the members are implicit reachable if the module itself is reachable. Special actions need to be taken to facilitate classifying types declared within modules; these are discussed in section 5.2.2. To find the correct type resolution steps are defined in the language specification, section 5.2.3 discusses how these are implemented.

5.2.1 Fetching types using a fully qualified name

As explained in the previous section a fully qualified name itself is not enough to identify a type. Which type is found and if the type is found depends on the scope of the type reference. The process of finding a type given its fully qualified name and its containing type is referred to in the remainder of the thesis as a type *lookup*.

From the containing type the containing project can be derived. The containing project has references to other projects and libraries, these are the projects and libraries which are reachable from the type reference and thus together with the containing project itself, are the projects and libraries the type should be sought in. The containing type determines if a type is visible to a type reference. A type is visible to a type reference if one of the following statements holds for the containing type:

- The containing type is nested within the referenced type.
- The containing type is a direct parent of the referenced type.
- The referenced type has a public access modifier and if the type is a nested type, the parent is visible.
- The referenced type has a protected access modifier and the containing type inherits from the referenced type.
- The referenced type has a friend access modifier, the containing type resides in the same project modifier and if the type is a nested type, the parent is visible.

To check if a fully qualified name candidate C matches a type the following algorithm is used:

```
for all identifiers G of the projects and libraries referenced by the containing project
        construct a key using G and C
        if there is a type matching this key
                if the type found visible to the containing type
                        return the type
return null //no match found
```

As explained in the section 4.5.2; in the context of a type reference, for every referenced type with fully qualified name N, there cannot be another type within this same context, which is reachable and visible, also having the fully qualified name N. This means given a fully qualified name, the first type found by the algorithm shown above must be the correct type.

5.2.2 Querying types which reside in modules

There is a relatively odd construction in the Visual Basic language which complicates type lookup a bit: the standard module. For now only the implications of types declared within modules are discussed. Nested types in modules are reachable just as nested types in other types, but, the name of a standard module is implicit and can be left out. If a module is directly visible within the context of a type reference, a directly nested type in the module can directly be referenced without referring the module itself. If the module resides within a namespace, nested type can be accessed directly on a reference to that namespace. If for example a module *Module1* with nested type *Type1* exists in namespace *Namespace1* and *Type1* is accessible with *Namespace1.Module1.Type1*, then because the module name is implicit, it is also

accessible with *Namespace1.Type1*, the fully qualified name still is *Namespace1.Module1.Type1* though.

This difficulty is handled in the type lookup mechanism. If a fully qualified name candidate is tried out, more candidates are synthesized from it by adding potentially left out module names. Two parts of the fully qualified name are passed to the type lookup process; an implicit part, and an explicit part.

The implicit part is a qualified path which might be added to the name to search for the type in that location, it is not explicitly provided in the type reference (why this is done is explained in section 5.2). The explicit part is the name as stated in the code, or constructed from it using an import. The explicit part can contain a "hidden" standard module name part. If the beginning of a type reference, which is the explicit part of the fully qualified name, refers to a namespace, this namespace may contain a standard module, which could contain the rest of the fully qualified name.

When a fully qualified name is passed to the type lookup algorithm all the candidates with potentially hidden module names inserted are synthesized and queried as follows:

Try to lookup the fully qualified name as is

```
if a type is found
        return the type
split the explicit name part into a list of its name parts
for all n = 0 to the number of name parts
        construct the qualified name beginning S by combining the first n name parts combined with
         the implicit part of the qualified name
        construct the qualified name ending E by combining the rest of the name parts
        for every reachable module in the namespace with name S
                try to lookup a type with the fully qualified name S.M.E where M is the name of the
                module
                if a type is found
                        return the type
return null //no match found
```

In above algorithm a lookup is done for every reachable module in a namespace, how this list of reachable modules is constructed is explained in section 5.3.4.

It would be incorrect to apply the above algorithm on the whole fully qualified name candidate instead of only on the explicit part. Consider the following example:

```
1   Namespace Namespace1
2      Module Module1
3         Class ClassOrNamespace1
4            Class NestedClass
5
```

```
6            End Class
7         End Class
8      End Module
9
10     Namespace ClassOrNamespace1
11        Class AnotherClass
12           Dim var As NestedClass
13        End Class
14     End Namespace
15  End Namespace
```

'NestedClass' is a nested class in class 'ClassOrNamespace1' which on its turn is a nested class in module 'Module1'. In 'AnotherClass' a variable is declared of type 'NestedClass'; this is not the 'NestedClass' declared in the example because this is a nested class and nested classes are not implicitly reachable without a qualifier from outside the containing class. If the algorithm described above would be applied to the complete fully qualified name instead of on only the explicit part, the 'NestedClass' declared in the example would be found though. In section 5.2.3.2 is shown how in the type resolution steps the qualified name 'Namespace1.ClassOrNamespace1.NestedClass' is tried in an attempt to classify the variables type. Would above algorithm also be applied to the implicit part, the algorithm would find 'Module1' in namespace 'Namespace1' and try candidate 'Namespace1.Module1.ClassOrNamespace1.NestedClass' which indeed results into a match, the nested class 'NestedClass' which as stated earlier is not correct.

5.2.3 Type resolution

Type resolution is the process of finding the correct type referred to by a type reference in the code. As discussed in the previous section, the context of the type reference determines which types are reachable from the type reference. If type references always would be fully qualified names the type reference could be passed to the type lookup mechanism and it would be done. Type references are not fully qualified names most of the time though, they are often just unqualified names. To find the correct type a given set of resolution steps is described in the code specification (15). To find the correct type fully qualified name candidates are constructed and looked up using the type lookup mechanism discussed in sections 5.2.1 and 5.2.2. This is done in such order the resolution steps in (15) are met.

The candidates are constructed in the following order:

1. Construct a candidate from a reference which begins with a the Global keyword
2. Construct candidates for the type in same or in a super namespace.
3. Construct candidates using aliased imports located in the source file.
4. Construct candidates using non aliased imports located in the source file.
5. Construct candidates using aliased imports located in the project.
6. Construct candidates using non aliased imports located in the project.

Section 5.2.3.1 discusses the case the reference starts with the 'Global' keyword, section 5.2.3.2 discusses the candidates in the super namespaces, section 5.2.3.3 discusses the construction of candidates given an aliased import and section 5.2.3.4 discusses the construction of candidates given a non aliased import.

5.2.3.1 Finding the type qualified with Global keyword

The Global keyword is used to identify the absolute root of the program; this means everything following the Global keyword in a type reference should be interpreted as a fully qualified name[2]. The first step of the type resolution is to look if the type reference starts with the global keyword, if it does, the keyword is striped of and the rest of the type reference is the first fully qualified name candidate. If the code is compilable, this candidate always will be a hit.

5.2.3.2 Finding the type in the same or in a super namespace

The second step is try is to find the type in the same or super qualified location. A qualified location is the full path from the root to the scope in which the type reference occurs.

For example:

```
1   Namespace MyNamespace
2      Namespace MySubNamespace
3         Class A
4            Dim InstanceOfA As MySubNamespace.A
5         End Class
6      End Namespace
7   End Namespace
```

As explained in section 4.4.2 the qualified location of 'InstanceOfA' is 'MyNamespace.MySubNamespace.A'.

To find the type, fully qualified name candidates are constructed by combining the type reference with all containing nested locations from the innermost to the outermost.

First a list of all super-locations is created from the innermost to the outermost location, in the case of the example this list would be:

- "MyNamespace.MySubNamespace.A"
- "MyNamespace.MySubNamespace"
- "MyNamespace"
- ""

[2] This construct is similar to the :: (double colon) scope operator at the beginning of identifiers in C++

A fully qualified name candidate is constructed by combining the type reference with these locations and looked up until the type is found or all super namespaces are tried. In the case of the example:

- "MyNamespace.MySubNamespace.A.MySubNamespace.A" Miss
- "MyNamespace.MySubNamespace.MySubNamespace.A" Miss
- "MyNamespace.MySubNamespace.A" Hit

5.2.3.3 *Finding the type using aliased imports*

Aliased imports are imports in which an import reference is aliased by an alias name. Whenever this alias name is used in the code, the name can be substituted by the imports reference.

Take for example:

```
Imports Alias = AliasedNamespace.AliasedSubNamespace
```

Now 'Alias' can be used in the following way:

```
Dim aliasClass As Alias.AliasedClass
```

Which is equivalent to:

```
Dim aliasClass As AliasedNamespace.AliasedSubNamespace.AliasedClass
```

In the third and fifth step of the type resolution fully qualified name candidates are made using aliased imports. These are the aliased imports imported in the containing file in the third step and imports imported in the project in the fifth step.

To try to resolve a type using aliased imports, the following steps are taken.

```
for every aliased import
        if the alias is not equal to the beginning of the type reference
                continue with the next import
         construct a fully qualified name by replacing the alias in the type reference by the import
        reference
        try to match the type using the type lookup mechanism.
        If a match is found
                return the match
return null //no match found
```

In the example the start of 'Alias.AliasedClass' equals 'Alias', this beginning is now replaced by the namespace of the import 'AliasedNamespace.AliasedSubNamespace' resulting in fully qualified name candidate 'AliasedNamespace.AliasedSubNamespace.AliasedClass'.

The imports reference is a fully qualified reference by definition, so substituting an alias with the import reference results in the only correct fully qualified name which can be made with an aliased import.

5.2.3.4 *Finding the type using non aliased imports*

Regular non aliased imports import namespaces or types. If a namespace is imported, this means reachable types and namespaces directly in this namespace can be referenced without any qualification, if a type is imported this means nested types in it can be referenced without qualification.

In the fourth and sixth step of the type resolution fully qualified name candidates are constructed using non aliased imports. These are the non aliased imports imported in the containing file in the third step and imports imported in the project in the fifth step.

```
for every non aliased import
        combine the import with the type reference and use the type lookup mechanism to match a
        type.
        if a type is found
                return the type
return null // no match found
```

5.3 Classifying member and local variable references

In the previous section was discussed how type references where classified. All other identifiers which don't classify to a data type, classify to a member or a local variable. This section discusses how these identifiers are classified.

Section 5.3.1 describes the resolution steps that have to be taken if the reference is a simple name. Section 5.3.2 describes the resolution steps that have to be taken if the reference is a qualified name. A lot of the resolution steps taken involve trying to match the name to the members of accessible modules, section 5.3.3 discusses how this is done. How accessible modules are found is discussed in section 5.3.4. Often it is not deductable from the syntax alone if something is a call to a member of a type passing arguments or that the result of a call to a member is indexed using these arguments, section 5.3.5 discusses how such a situation is handled. Most steps described in these sections need a way to find a member of a type most fitting to a reference, how this is done is discussed in section 5.3.6.

5.3.1 Classifying a simple named identifier

If a simple named identifier which is not a type reference is classified, the same resolution steps can be taken regardless of if there are arguments or type arguments passed in the reference.

When a variable or member is referenced, there are often clues that give away the type of the entity referenced. For instance if type arguments are provided, the entity referenced can't be a variable; when no argument list is provided, it cannot be a reference to a method which has non-optional parameters. Those clues are not enough though to know which kind of member, variable or type is referenced though. For example:

```
Something(0)
```

In this example 'Something' can several things. The first guess probably is that 'Something' is a method and the statement above is a method call[3]. This could well be correct, but there are many other things 'Something' could be. 'Something' could be a variable or a property containing an array or an object with a default property. In the next example there are no arguments anymore:

```
Something
```

In this example 'Something' also could be a property, a variable or even a method without parameters. So clues aside, both cases need similar resolution steps to find out what 'Something' really is.

To be able to classify all cases with a simple name, i.e. no qualified name, one algorithm is created which uses the name, argument types and type arguments of a reference.

The reference is classified using the following resolution steps, if in one of the steps a match is found, the result is returned and no other steps are tried:

- Try to find a variable by searching scopes of the type itself.
- Try to find it as a member of the type.
- Try to find it as a member of a module which resides within the same namespace or in a super namespace.
- Try to find it as a member of a module which is imported within the same file.
- Try to find it as a member of a module which is imported globally within the project.

In the following subsections each of these steps is discussed.

5.3.1.1 Searching in the scope

The first try is to assume a variable is referenced which is declared within the current member. The reason the local scope is searched first is the following. If two variables are declared with the same name, one in a sub scope of the scope the other directly is in, then if a variable is referenced with that name on a place where both variables are in scope, the one in the sub scope is referenced. Therefore the inside of a member is searched first. For example:

```
1   Public Class Scope
2      Private var As Integer
3      Public Sub ScopeSub(ByVal var As Integer)
4         DoSomething(var)
```

[3] It is important to note that such ambiguities are due to the design of the parser used. More sophisticated parsers do additional processing, at the border of syntax and semantic analysis, and thus are able to resolve such context-dependent ambiguities during the parsing process, resulting in more detailed syntax trees. However, this complicates the design of the parser. Given that our chosen parser does not resolve such ambiguities, we are left with the task of doing it ourselves during our own semantic analysis. The same situation occurs in many programming languages, C and C++ being notoriously (in)famous for that.

```
5       End Sub
6    End Class
```

Here ‘var’ is passed as an argument to the method ‘DoSomething’ the variable which is actually referenced is the parameter ‘var’ of the method ‘ScopeSub’ and not the variable ‘var’ in class ‘Scope’. Both the parameter ‘var’ and the variable ‘var’ are in the scope of the method call. The parameter ‘var’ however exists in a sub scope of the scope the variable ‘var’ exists in. Therefore this is the one referenced.

If a variable V with name N is declared within a scope S and some other member M also named N is defined outside this scope, it is impossible to refer to member M from within scope S, even if M would be distinguishable from V by its signature. With the precondition that the input code is correct this means when the scope tree is searched up, starting at the scope the reference is in, the first variable found with the correct name is guaranteed to be the variable referenced.

As described before every node in the AST has a reference to the scope it is directly in. To resolve the reference, first the correct variable is found by walking up the scope tree and getting the first variable with the correct name. If no variable is found within the scope of the member, this attempt fails.

If a variable is found the resolution is not completely done. If argument types were passed to the resolution method, this means the variable is indexed. This means the variable is an array type or it has a default property. If the variable is an array type, the result type will be the element type of this array type. If the variable is not is an array type, this means the variable must have a default property matching the arguments passed. This default property is resolved using member resolution which is described later in this chapter.

5.3.1.2 Searching the current type

At this point the only possibility is that the member searched is a member of the current type, or a member on a module visible at this point. Members of the current type hide members of a module because they are closer to reference in the scope tree, because of this trying to match a member of the current type is the second step in resolving a simple named reference. To do this the type the call is in is found in the scope tree and the member is resolved using the method described in 5.3.5 below.

5.3.1.3 Searching local modules

The name could reference to a member of a module which resides in the namespace the reference is in, or one of its super namespaces. The code specification states that the nested namespaces are searched from the innermost namespace going to the outermost. So the current namespace is taken and all super namespaces are constructed from it. The referenced member is searched using the method described in 5.3.3.

5.3.1.4 Searching locally imported modules

The next possibility to resolve the name is that it references a member of a module which resides in a namespace imported in the source file containing the reference. All imports are collected and the referenced member is searched using the method described in 5.3.3.

5.3.1.5 Searching globally imported modules

The last possibility to resolve the name is that it references a member of a module which resides in a namespace imported in the project file. All imports are collected and the referenced member is searched using the method described in 5.3.3.

5.3.2 Classifying a qualified named member

The same difficulties as with the classification of simple named references described above also hold for qualified named references; it's not always clear what is the kind of member searched. A qualified member introduces an extra dimension to this difficulty though; the qualifier itself can be many different things as well. So it is not only the question what is the kind of member searched for, but it also has to be determined on which type it has to be searched. The qualifying type could be a variable, a member, an uninitialized type, but also a namespace in which a module resides.

To classify a qualified named member, the following resolution steps are taken, just as with the simple name resolution, the first hit found is the correct member.

- Try to find a qualifying type and find a member on it
- Try to resolve global
- Try to find a module which path is the same
- Try to find a module using an alias import
- Try to find a module using a local import
- Try to find a module using a global import
- Try resolving the type using the type system

The next subsections will discuss each of these steps.

5.3.2.1 Searching a member on a qualified type

A qualified expression is constructed in the form "[Qualifier].Member" the dot means the member is a part of the type described by the qualifier. There is a construction however where the Qualifier can be empty and the expression looks like ".Member", this construction is the with-statement discussed in section 4.6.3.2. Within a with-statement the with-expression is implicitly used when an empty qualifier is used.

The method GetTypeFromQualifier is defined in the expression resolver to resolve a qualifier type. As one of the parameters the qualifier-expression is passed. The expression is resolved to get the qualifier-type; when this

expression is empty, the with-expression is sought in the scope and used instead.

The language specification states that when there is an empty qualifier the qualifier is substituted with the with-expression of the immediately containing with-statement, the first with-expression found when searching the scope from bottom to top will thus be the correct one.

The pre-condition the code is compilable and the fact that an empty qualifier only is allowed within a with-statement guarantees a with-statement always will be found when the qualifier-expression is empty.

If a qualifier type is found, the correct member called on the type is resolved using the algorithm described in section 5.3.5.

5.3.2.2 Searching a member of a module which is referenced with a qualifier starting with global

If a qualifier starts with the keyword 'Global', it means the rest of the qualifier should be mapped on the global namespace, which is the root of all namespaces. The rest of the qualifier should be interpreted as a fully qualified namespace. So in this namespace a module is sought with a matching member using the algorithm later in section 5.3.3.

5.3.2.3 Searching a member of a module in a partially qualified namespace

In this stage, the nested namespaces in which the call is located is searched from the inner namespace to the outer namespace for a module which contains a matching member. To do this, the algorithm described in section 5.3.3 is used.

5.3.2.4 Searching a member of a module which is referenced by an alias or an aliased namespace

For all aliased imports for which holds that the qualifier starts with its alias, the aliased part of the qualifier is replaced by the namespace of the import. In the resulting namespace is sought for a module with a matching member using the algorithm described in section 5.3.3.

The next stage is to try to find a module with a matching member by looking in the namespaces imported in the file the call is located in using the algorithm described later in section 5.3.3.

5.3.2.5 Searching a member of a module which is reachable via a global import

The last step is to try to find a module with a matching member by looking in the namespaces imported in the project the call is located in again using the algorithm described in section 5.3.3.

5.3.3 Classifying a module member

Modules in Visual Basic 8 are a particular construction. Whenever a module is accessible, all its accessible members are available without having to specify the module. For example:

```
1   Namespace Namespace1
2      Module Module1
3         Public Property ModuleProperty() As Integer
4            Get
5               …
6            End Get
7            Set(ByVal value As Integer)
8               …
9            End Set
10        End Property
11     End Module
12
13     Class Class1
14        Public Sub DoSomething()
15           ModuleProperty = 1
16        End Sub
17     End Class
18  End Namespace
```

Here 'ModuleProperty' is set to '1' on line 15. Nothing points to the fact that 'ModuleProperty' is in fact a reference to the property in 'Module1'. Just because the module 'Module1' is visible at the call, makes it possible to access its members.

The Visual Basic language specification (15 p. 52) states the following about unqualified name resolution:

> *For each nested namespace containing the name reference, starting from the innermost namespace and going to the outermost namespace, do the following:*

If the namespace contains one or more accessible standard modules, and R matches the name of an accessible nested type in exactly one standard module, then the unqualified name refers to that nested type. If R matches the name of accessible nested types in more than one standard module, a compile-time error occurs.

> *If the source file containing the name reference has one or more imports:*

If the imports contain one or more accessible standard modules, and R matches the name of an accessible nested type in exactly one standard module, then the unqualified name refers to that type. If R matches the name of accessible nested types in more than one standard module, a compile-time error occurs.

> *If the compilation environment defines one or more imports*

If the imports contain one or more accessible standard modules, and R matches the name of an accessible nested type in exactly one standard module, then the unqualified name refers to that type. If R matches the name of accessible nested types in more than one standard module, a compile-time error occurs.

Regarding qualified name resolution referring to modules, the specification further states:

Given a qualified namespace or type name of the form N.R, where R is the rightmost identifier in the qualified name, the following steps describe how to determine to which namespace or type the qualified name refers:

> *Resolve N, which may be either a qualified or unqualified name.*
>
> *If N contains one or more standard modules, and R matches the name of a type in exactly one standard module, then the qualified name refers to that type. If R matches the name of types in more than one standard module, a compile-time error occurs.*

In both the unqualified and the qualified resolution, a name can match a member of a module in one of different given namespaces. To search for module members in a set of namespaces the following algorithm is provided:

```
for all namespaces provided
        if no module with the namespace exists
                continue with the next namespace
        for each module in the namespace
                try to find the member on the module using member resolution
                if a member is found
                        return the member
return null //no module member found
```

In section 5.3.4 is discussed how given a namespace accessible modules are found. Section 5.3.5 discusses how, given a type, a correct member can be determined by applying member resolution.

In the code specification is stated that if there are more matches in one namespace, or more matches in imports, this results in a compile-time error. A precondition of the input code is that the code is compilable, therefore if the name matches a module member, exactly one match exists, so it is correct to return the first match found.

5.3.4 Finding accessible modules

A module is a static type which means no instance can be made of it. In the previous section the standard module is already partially discussed. When a module is directly reachable because it exists in the same namespace, in a super namespace or in an imported namespace, or the namespace containing the module is used in a reference, its members are directly accessible without naming the module itself.

For example, a module M with member F resides in namespace N, if N is imported, F can be accessed without qualifying it with the module name. Or for example a module M is declared in namespace NS, M contains a member F, now the reference NS.F can be used instead of NS.M.F to access F directly.

In section **Error! Reference source not found.**5.2.2 is discussed how this phenomenon is handled in the type resolution process. Later is discussed how it is handled in expression resolving.

To find member F in both examples, it is necessary to find the module without actually knowing its name. The module sought could be any module in any reachable namespace, so it is necessary to have a way to get all accessible modules in a given namespace.

As with other types, it depends on the context of the reference if a module is reachable. Although the grammar implies otherwise[4], a standard module can only have the access modifiers public and friend; this is because a module can only be nested directly in a namespace and types directly nested in a namespace cannot be private or protected. This makes it easy to determine if a module is accessible, a module is accessible if its access modifier is public, or if it is friend and the module resides in the same project as the reference.

To get a list of accessible modules the type system stores the modules in dictionaries with their namespace as a key. These dictionaries in turn are stored in a dictionary with their project or library identifier as a key. Given a namespace and the project containing the reference, the list of compiled modules is created as follows:

```
for every referenced project/assembly identifier in the project
        if the associated module dictionary doen't exist
                continue with the next identifier
        retrieve the list of modules in the given namespace from the module dictionary
        for every module in the list
        if accessible from within the given project
                add it to the result
return the result
```

5.3.5 Finding the correct member call or indexing on a type

As described before in section 5.3.1, an identifier with an argument list can be a call to a method but also could be a call to a property which has an array type or a type with a default property, which is indexed.

For Example:

```
A.DoSomething(10)
```

This could be a method call on a method called 'Something' with an argument '10'. It could also be a property with an array type as a type. This means 'A.DoSomething' returns an array and '(10)' takes the tenth item from the array. It could also be that 'A.DoSomething' returns another type with a default property which takes a number as a parameter.

To handle this situation an algorithm is provided which among other parameters takes a qualifier type, a member name, argument types and type arguments and returns the correct result. Note the list of argument types and type arguments can be empty or not provided at all.

---

[4] This is again an illustration of the semantic limitations of a context-free grammar

The correct result is found using the following steps:

```
// assume the name and arguments are a member call first
try to find the most specific one using member resolution described in section 5.3.6.
if a member is found
	return the result
//assume the name refers to a member without a parameter list
try to find the most specific one using member resolution.
if such a member is found
	if the found member returns an array type
		return the element type of the array type
	else
		// the type returned by the found should have a default property which matches the
		//arguments
		use member resolution to find the most specific matching property
		return the result
else
	if the type is an enumeration
		return the type //the member is an enumeration member call and thus results in an
				//instance of the same type
return null //no correct member is found
```

5.3.6 Member resolution

Different members in the same type can have the same name and the same number of parameters. Which members are applicable depends on the context. In some situations more than one member is applicable. Finding out which members are applicable and which one is most specific might well be the trickiest part of our semantic analysis. Different factors have influence on the applicability of a member:

- The name of the member.
- The accessibility of the member.
- The number of arguments. This sounds easy, but because of optional parameters and parameter arrays this is not that straight forward as it might seem at first.
- Are the types of the parameters correct? Often the argument types are not the same as the parameters. The argument types may be subtypes of the parameter types, or an implicit cast might exist from the argument type to the parameter type. Also the member might be a generic member, its parameters could be type parameters. To make it even more difficult, the argument can be passed as named arguments, this means the order of the arguments could be different than that of the parameters.
- The member could be shadowed by another member.

The following steps are taken to find a correct member:

```
add all “constructed members” which could be constructed with the provided type arguments and
argument types
list all matching members within the type
if none of the members found are declared shadowing
```

```
		add the matching members within the inherited types
if only one member is found
		return the member
for all members which used narrowing coercion to match
		remove the member from the list
if only one member is found
		return the member
for all members for which there is a member in the list that is more applicable
		remove the member from the list
if only one member is found
		return the member
for all members for which there is a member in the list that is less generic
		 remove the member from the list
if only one member is found
		return the member
return null //no member is found (should never be reached if the preconditions are met)
```

In section 5.3.6.1 the construction of constructed members is discussed. The section after that 5.3.6.2 how is checked if a member is applicable. Section 5.3.6.3 describes how the accessibility of a member is checked. Section 5.3.6.4 describes how of two members the most applicable is determined. Section 5.3.6.5 describes how of two members the least generic is determined.

5.3.6.1 Adding constructed members

Some members are generic members, this means type parameters can be provided. In a call these type parameters can be explicitly provided, but it is also possible to let the compiler implicitly find out what the type arguments are in an instance. All type parameters should at least be used once as a parameter type to be able to do this.

A constructed member is a member which refers to a generic member but all type parameters are replaced by the type arguments. Before a call is resolved, all matching constructed members are added. To ensure all references to a constructed member with the same type arguments are classified to the same one, there may only be one constructed instance of a generic member for every set of type arguments.

Adding the constructed members is done with the following steps:

```
if the type arguments are explicitly provided
		 find all members with the given name and the same number of type parameters as provided
		type arguments.
		for every found member
				if a constructed member for the member with the same type arguments already
				exists within the type
						continue with the next member
				construct a constructed member with given type arguments
		return
find all members with the given name that has type parameters.
for every found member
```

```
for every type parameter
        try to find the type argument by searching for instances of the type parameter in the
        list of parameters and taking the associated argument type as the type argument of
        that type parameter.
If all type arguments are found
        if a constructed member for the member with the same type arguments already
        exists within the type
                continue with the next member
        construct the constructed member using the type arguments
```

5.3.6.2 Checking for matching signatures

Checking if a member has the correct signature can be a bit complicated. A signature can contain type parameters, named parameters, optional parameters or a parameter array which don't make it easy.

When type arguments are provided, only constructed members with the right number of type parameters and the right type arguments match, so if argument types are provided, all non constructed members and members with the wrong number or type of argument types are rejected.

If there is no argument list provided, that is, a member is referred without parentheses, the referred member must be a variable or a property, if this is the case all members not a variable or a property without parameters are rejected, else there is a match.

Arguments can be named or unnamed. Unnamed arguments are mapped on parameters on the same position. Named argument explicitly pass a value to a certain named parameter. When one of the arguments is a named argument, all following arguments are mandatorily named as well.

For example

```
Sub ParameterExample(ByVal var1 As Integer, ByVal var2 As Integer)
```

Can for example be called with normal arguments

```
ParameterExample(1, 2)
```

Or with named parameters

```
ParameterExample(var2:=2, var1:=1)
```

Parameters can be optional. When using normal arguments an optional argument can be left out by leaving its place empty or completely ignored if no arguments follow. When arguments are passed using named arguments, the optional parameter can be ignored completely.

For example

```
Sub ParameterExample(Optional ByVal var1 As Integer = 1, Optional
ByVal var2 As Integer = 2)
```

Could be called with

```
ParameterExample(, 3)
```

Or

```
ParameterExample(var2:=3)
```

The last parameter of a signature can be preceded with the keyword ´ParamArray´, if this is the case the parameter type must be an array type. When a member with such a parameter is called, the argument can be passed in the form of array of the same type, but it can also be passed as arguments of the same type as the arrays element type, or even be left out completely.

For example:

```
Sub ParamArrayExample(ByVal ParamArray var() As Integer)
```

Could be called without arguments:

```
ParamArrayExample()
```

Or with one argument of the same type as the element type of the array, in this case an integer:

```
ParamArrayExample(1)
```

Or with more arguments of the same type as the element type of the array, in this case two integers:

```
ParamArrayExample(1, 2)
```

Or with one argument of the same type as the array, in this case an array of integers:

```
ParamArrayExample(New Integer() {1, 2})
```

For a signature to match, arguments and all named arguments must map to a parameter and all parameters not mapped must be optional.

Constant expressions are expressions which can be evaluated during compile time.

For example:

```
2 + 3 * 6
```

This expression evaluates to an integer with constant value 20.

Constant expressions of an integral type can be implicitly converted to a narrower type if the value falls within the range of this type. Constant expressions of type 'Double' can be implicitly converted to type 'Single' if the value falls within the range.

For example:

```
Sub ShortExample(ByVal val As Short)
```

Can be called with:

```
ShortExample(2 + 3 * 6)
```

The constant expression of type 'Integer' with value 20 is converted to the narrower type 'Short' because it falls within the range.

To check if a member matches a call, given the argument types and the type arguments the following algorithm is used, a flag determines if implicit narrowing is allowed:

```
if type arguments are provided
        if the number and type of the type arguments are not the same
                return false
if the member is a property without parameters and there areno arguments provided
        return true
if the member is not a signature member (i.e. it hasn't a list of parameters)
        if arguments where provided
                return false
        else
                return true
for every argument provided
        if the argument is a named argument
                record this fact
                exit the loop
        if the argument is an empty argument
                if the associated parameter is not  optional
                        return false
        if the index of the argument is the same or higher than the index of the last parameter of the
        member and this last parameter is a parameter array
                if the last parameter is already covered in the previous iteration
                        return false
                if the last argument matches the type of the parameter
                        record this fact
                        continue with the next argument
                if the argument matches the element type of the parameter array
                        continue with the next argument
                 else
                        return false
        if the index of the current argument is higher than the number of parameters
                return false
```

```
        if the argument type doesn't matchs the associated parameter type
                return false
if  the latter loop is exited because a named argument was encountered
        for every argument not covered in the first loop
                if a parameter having the provided name in the parameter list which is not a
                parameter array and is not covered in the last loop doesn't exists?
                        return false.
        for every parameter not covered in the first loop
                if there is a named argument with the name of the parameter
                        if the argument type doesnt match the parameter type
                                return false
                else
                        if the parameter is not optional
                                return false
if the number of arguments is the same as the number of parameters minus one and the last
parameter is a parameter array
        return true
for all parameters not yet covered in one of the previous loops
        if the parameter is not optional
                return false
return true
```

In this algorithm an argument matches a parameter in one of the following cases:

- The argument type is the same as the parameter type.
- There is a widening conversion available from the argument type to the parameter type.
- The argument type is a subtype of the parameter type.
- Implicit narrowing is allowed and the argument is a constant expression and the resulting constant fits the parameter type.

In section 5.1 is discussed how is evaluated if an expression is a constant expression of a type which can be implicitly narrowed.

5.3.6.3 Checking accessibility

Another prerequisite for a member possible matching a certain call is that it is visible to the call. To check if a member is visible, the *TypeDeclaration* of the type from where the call is made always is passed when trying to resolve a call. A member is considered visible when one of the following conditions is true:

- The calling type is the same as the type where the member resides.
- The access modifier of the member is public.
- The access modifier of the member is protected or protected friend and the calling type is a sub type of the type the member resides in.
- The access modifier of the member is friend or protected friend and the calling type resides in the same project as the type the member resides in.

5.3.6.4 Determining the most applicable member

The member that is considered most applicable is the matching member that is most specific. To determine if a member is more specific than another, the following steps are taken:

```
for all argument indexes
        if the types of the parameters matching the argument on the index in each member are the
        same
                if exactly one of the parameters is a parameter array
                        save the fact the member of which the parameter is not a parameter array
                        has a more applicable parameter
                continue with the next pair of arguments.
        if the parameter of the first member more is applicable than the one of the second member
                save this fact.
        if the parameter of the second member is more applicable than the one of the first member?
                save this fact.
if the first member contained a parameter which was more applicable then the equivalent parameter
in the second member and the second member did not contain any parameter which was more
applicable then the equivalent parameter in the first member
        return true
else
        return false
```

A parameter P1 is considered more applicable than parameter P2 if one of the following statements hold:

- The type of P1 is a subtype of the type of P2
- There exists a widening conversion from type of P1 to the type of P2
- The argument is a literal 0, the type of P1 is numeric and the type of p2 is an enum
- The type of P1 is 'Byte' and the type of P2 is 'SByte'
- The type of P1 is 'Short' and the type of P2 is 'UShort'
- The type of P1 is 'Integer' and the type of P2 is 'UInteger'
- The type of P1 is 'Long' and the type of P2 is 'ULong'

5.3.6.5 Determining the least generic member

In compilable code two accessible members of a type can only have the same signature if their signature is some specification of a generic member.

For example:

```
Sub GenericTest(Of T)(ByVal a As T, ByVal b As T, ByVal c As
Integer)
```

And:

```
Sub GenericTest(Of T)(ByVal a As T, ByVal b As Integer, ByVal c As
Integer)
```

If the next call is done:

```
GenericTest(1, 1, 1)
```

Then the signature is seen as the same:

```
Sub GenericTest(Of T)(ByVal a As Integer, ByVal b As Integer, ByVal
c As Integer)
```

If this is the case, the least generic member should be chosen. If a least generic member can be determined with respect to the members type parameters, than this is the least generic member, if not, the least generic member is determined with respect to the containing types type parameters.

A member is considered less generic than another member if it has less type parameters. If the number of type parameters is equal, a member is considered less generic if for all parameter indexes holds that the parameter on the index is equally or less generic than the parameter on that index of the other member.

A parameter is considered less generic than another parameter if it is not a type parameter in the members or types type parameters list and the other parameter is.

Given two members M1 and M2, the least generic member is determined with the following algorithm:

```
if not both members are constructed
        if the first member is constructed
                return the second member is less generic
        else
                return the first member is less generic
if the numbers of type parameters differ
        if the first member  has less type parameters as the second member
                return the first member is less generic
        else
                return the second member is less generic
for every argument
        get the pair of parameters matching the argument
        if the first parameter is more generic than the second with respect to its members type
        parameters
                save the fact the first member has a parameter which is more generic
        if the second parameter is more generic than the first with respect to its members type
        parameters
                save the fact the second member has a parameter which is more generic
if exactly one of the members has a parameter which is more generic
        return the other member is less generic
for every argument
        get the pair of parameters matching the argument
        if the first parameter is more generic than the second with respect to its types type
        parameters
                save the fact the first member has a parameter which is more generic
        if the second parameter is more generic than the first with respect to its types type
        parameters
```

```
                save the fact the second member has a parameter which is more generic
if exactly one of the members has a parameter which is more generic
        return the other member is less generic
```

5.4 Expression classification

Next to identifiers that classify to types, members or variables, *expressions* also classify to a type. Expressions are often arithmetic operator expressions or literal type expressions. If an expression is an operator expression, the operator defines which the resulting type of the expression is. Intrinsic operator definitions are built in; their resulting type is predefined. Next to built in operators, operators can also be user-defined; in such cases, the operator declaration then defines the result type of the expression. Section 5.4.1 discusses how the correct operator is found in section 5.4.2 all classifiable expressions are discussed.

5.4.1 Operator resolution

Whenever an expression is resolved which involves an operator, the correct operator has to be found to get result type of the expression. There are two types of operators, user-defined operators and predefined operators. The user-defined operators are considered first and after that the built in operators.

5.4.1.1 Resolving user-defined operators

To resolve the correct user-defined operator the following three steps are used:

- Get all matching operators
- Try to find an exact match in the resulting list, if found return it
- If there is no exact match return the *most specific* operator from the list.

User-defined operators should be declared within one of the operand types or in one of their super types. An operator matches when the operands match the parameter types of the operator. Matching operators are found using the following steps:

```
for all operand types
        collect all operators declared in the type that are of the right operator type, have the correct
        number of parameters (one for unary, two for binary operators) and of which the operand
        types match the parameter types
        if the type has a base type
                get the base type of the operand and recursively collect all matching operators
if no matching operators are found
        return null
for all operators
        if the operand types exactly match the types of the operator
                return the operator
save the first of the operators as the most specific operator
for all operators
        for each parameter
                if the type of parameter is a subtype of the one of the currently most specific
                operator or thereis a widening to it
                        save the operator as the most specific operator
```

```
                        continue with the next operator
return the most specific operator
```

To determine if one operator is more specific than another, their parameters are considered one by one. An operator is more specific if the type of the first parameter that has a different type than the parameter on the same index of the other operator, is a sub type of that other type or if there exists a widening to that type.

If one operator is most specific, for all other matching operators will hold that the most specific operator is more specific than that operator. Thus if operator MSO is the most specific operator, for every other matching operator MO_n, for the first parameter MO_nP_i, of which the type differs with that of the parameter $MSOP_i$ on the same index in MSO the following will hold; the type of $MSOP_i$ is a sub type of the type of MO_nP_i or there is a widening from $MSOP_i$ to MO_nP_i.

If such an operator cannot be found in the set of matching operators and there is no exact match, the compiler will fail. A precondition in this project is that input code will compile, therefore the method described above will always find the correct operator.

5.4.1.2 Resolving built in operators

If no user-defined operator is found, this means the operator is a predefined operator. Predefined operators cannot be extracted from the type definitions, they are part of the language itself. In the Visual Basic language specification all predefined operators are listed. To resolve expressions using a predefined operator, tables are made for every such operator. The resulting type of the expression is looked up by using the operands as keys in the mapping tables.

5.4.2 Expressions

As described before, there are multiple types of expressions. Each expression implies a different way of classifying. Expressions which classify to a type are the following:

- Dictionary member access expressions
- GetType expressions
- Cast expressions
- Parenthesized expressions
- Unary operator expressions
- Binary operator expressions
- Instance expressions
- New expressions
- Literal expressions
- Call and index expressions
- Nothing expressions
- Type reference expressions
- Simple name expressions

- Qualified name expressions

In the next subsections these expressions will be discussed.

5.4.2.1 Dictionary member access

As stated in the language specification a dictionary member access expression is used to look up a member of a collection. A dictionary member access takes the form of E!I, where E is an expression that is classified as a value and I is an identifier. The expression E!I is transformed into the expression E.D("I"), where D is the default property of E. Within a With block E can be left out, instead of E the With expression is used.

```
DictionaryAccessExpression ::= [ Expression ] ! IdentifierOrKeyword
```

In the AST the statement is represented with an *DictionaryLookupExpression object*, which contains a qualifier expression and a name. To find the resulting type of the lookup expression, the most specific default property with one parameter of the type "System.String" has to be found. To get this the qualifier expression is resolved to get the qualifier type. The name is just the value used to pas as an argument to the default property, at this point it is only interesting to know its type, this is always "System.String" so the name can be ignored. Member resolving as described before is used to find the default property.

5.4.2.2 Get Type expression

The get type expression is used to return the type object of a given type.

```
GetTypeExpression ::= GetType ( GetTypeTypeName )
```

The representation of this expression in the AST is a *GetTypeExpression* object. The content of this object is ignored, the expression always returns object of the type "System.Type", so a *ResolveResult* with an initialized representation of "System.Type" is always correct .

5.4.2.3 Cast expressions

To cast an object from one type to another a couple of cast methods are defined in the language. The DirectCast, TryCast and CType expression all cast an expression to the type described by a given typename. They have a couple of different restrictions which are not interesting in this context. Next to these three there is also a set of methods defined to cast to intrinsic types, these methods basically work the same except every one of them has its own type it casts to.

```
CastExpression ::=
	DirectCast ( Expression , TypeName ) |
	TryCast ( Expression , TypeName ) |
	CType ( Expression , TypeName ) |
	CastTarget ( Expression )

CastTarget ::=
	CBool | CByte | CChar | CDate | CDec | CDbl | CInt | CLng | CObj | CSByte |
	CShort | Sng | CStr | CUInt | CULng | CUShort
```

In the casts the resulting type is only dependent on to which type the expression is cast is used and not on the expression itself, the expression therefore can be safely ignored.

The first three cast methods are represented in the AST as *CTypeExpression*, *DirectCastExpression* and *TryCastExpression* objects. They contain an expression and a target type name. The cast expression will produce an object of the type described by the type name, so the type name is resolved to the correct type using the type system and returned.

The set of intrinsic cast expressions are represented in the AST as an *IntrinsicCastExpression* object. This object contains an intrinsic type enumeration which states the type of the cast and the expression which is the subject of the cast. Depending on which cast is used, a *ResolveResult* with the correct intrinsic type is returned.

5.4.2.4 Parenthesized expression

In the code specification a parenthesized expression is described as follows. A parenthesized expression consists of an expression enclosed in parentheses. A parenthesized expression evaluates to the value of the expression within the parentheses.

```
ParenthesizedExpression ::= ( Expression )
```

This means also the resulting type evaluates to the type of the expression within the parentheses. In the AST it is represented by a *ParentheticalExpression* object. It contains the expression that was placed within the parentheses, this expression is resolved and the result returned.

5.4.2.5 Unary operator expression

There are three unary operator expressions.

```
UnaryPlusExpression ::= + Expression
UnaryMinusExpression ::= - Expression
NotExpression ::= Not Expression
```

The resulting type of a unary operator expression is dependent on the operator definition, which operator definition has to be chosen is dependent on the resulting type of the operant expression. So to resolve the unary operator expression, first the type of the expression is resolved. This type is used to find the most specific matching overloaded operator if one exists. If one exists, the result type of this operator is the result type of the unary operator expression. If such an operator is not found, this implies the expression type is an intrinsic type. The result of unary operator expressions on intrinsic types are implicit, the correct resulting type is looked up in a mapping which contains the resulting types for all operators for all intrinsic types.

5.4.2.6 Binary operator expression

The language contains a wide range of binary operator expressions.

```
AdditionOperatorExpression ::= Expression + Expression

SubtractionOperatorExpression ::= Expression - Expression

MultiplicationOperatorExpression ::= Expression * Expression

DivisionOperatorExpression ::=
        FPDivisionOperatorExpression |
        IntegerDivisionOperatorExpression

FPDivisionOperatorExpression ::= Expression / Expression

IntegerDivisionOperatorExpression ::= Expression \ Expression

ModuloOperatorExpression ::= Expression Mod Expression

ExponentOperatorExpression ::= Expression ^ Expression

RelationalOperatorExpression ::=
        Expression = Expression |
        Expression <> Expression |
        Expression < Expression |
        Expression > Expression |
        Expression <= Expression |
        Expression >= Expression

LikeOperatorExpression ::= Expression Like Expression

ConcatenationOperatorExpression ::= Expression & Expression

LogicalOperatorExpression ::=
        Expression And Expression |
        Expression Or Expression |
        Expression Xor Expression

ShortCircuitLogicalOperatorExpression ::=
        Expression AndAlso Expression |
        Expression OrElse Expression

ShiftOperatorExpression ::=
        Expression << Expression |
        Expression >> Expression
```

As with the unary operator expressions, the result of the binary operator expression is dependent on the operator definition, which operator definition is the most specific is dependent on the resulting type of the operands. First the types of both operands are resolved. Using the operant types, the most specific matching operator is searched. If no matching operator is found, a second try is done in which literals are allowed to narrow. See literal expressions for that. The result type of the resulting operator is the result type of the binary operator expression. Is no operator found, this implies both operands are intrinsic and the result is found in a mapping.

5.4.2.7 Instance expression

The code specification describes the instance expression as the keyword Me, MyClass or MyBase. The keyword 'Me' represents an instance of the type containing the method or property accessor being executed. The Keyword MyClass is equivalent to Me, but all method invocations are treated as if the method being invoked is non-overridden. Thus, the method called will not be affected by the run-time type of the value on which the method is being called. The keyword MyBase represents the instance of the type containing the method or property accessor being executed cast to its direct base type.

```
InstanceExpression ::= Me | MyBase | MyClass
```

The AST representation of an instance expression is an *InstanceExpression* object. It contains an instance type which describes which of the three types it represents. If the instance expression is MyBase, the base type of the type the current scope is in is returned. If the instance expression is Me or MyClass, the type the current scope is in is returned.

5.4.2.8 New expression

To initialize a type the new expression is used.

```
NewExpression ::=
        ObjectCreationExpression |
        ArrayCreationExpression |
        DelegateCreationExpression

ObjectCreationExpression ::=
        New NonArrayTypeName [ ( [ ArgumentList ] ) ]

ArrayCreationExpression ::=
        New NonArrayTypeName ArraySizeInitializationModifier ArrayElementInitializer

DelegateCreationExpression ::= New NonArrayTypeName ( Expression )
```

The object creation expression and the delegate creation expression are represented in the AST with a *NewExpression* object. The array creation expression is represented by an *NewAggregateExpression* object. Both objects contain a type name, this type name is used to resolve the type using the type system.

5.4.2.9 Literal expression

A literal expression is a value expressed in code. The following grammar is heavily simplified, see the appendixes for the complete grammar.

```
Literal ::=
        BooleanLiteral |
        IntegerLiteral |
        FloatingPointLiteral |
```

```
        StringLiteral |
        CharacterLiteral |
        DateLiteral
```

In the AST literal expressions are represented by *StringLiteralExpression*, *CharacterLiteralExpression*, *DateLiteralExpression*, *IntegerLiteralExpression*, *UnsignedIntegerLiteralExpression*, *FloatingPointLiteralExpression*, *DecimalLiteralExpression* and *BooleanLiteralExpression*.

When the *StringLiteralExpression*, *CharacterLiteralExpression*, *DateLiteralExpression*, *DecimalLiteralExpression* or *BooleanLiteralExpression* is encountered in the AST the corresponding types simply can be returned, which are in the same order "System.String", "System.Char", "System.DateTime", "System.Decimal" and "System.Boolean".

The types of the other three AST literal representations are dependent on their type character, their value and if narrowing of literals is allowed.

If a type character is supplied, the type character decides the resulting type.

If there isn't a type character supplied, the *IntegerLiteralExpression* defaults to the type "System.Int32", which is the same as the intrinsic type Integer. When the literal value is bigger than the maximum value of an integer, it is considered a Long, so the type "System.Int64" which is the same is returned. Is it allowed to narrow literals, the type is considered a "System.Int16" or Short if the value is smaller than the maximum value of Short.

With no type character supplied the *UnsigendIntegerLiteralExpression* standard results in "System.UInt32". It results in an "System.UInt64" if the value is too big for an UInt32 . If narrowing literals is allowed, the type "System.UInt16" is returned if the value is smaller than the maximum value of UInt32.

With no type character supplied the *FloatingPointLiteralExpression* standard results in a "System.Double". If narrowing of literals is allowed and the value is smaller than the maximum value of Single, "System.Single" is returned.

5.4.2.10 Call or index expression

An invocation expression consists of an invocation target and an optional argument list. An indexed expression also consists of an invocation target and an argument list.

```
InvocationStatement ::= [ Call ] InvocationExpression StatementTerminator

InvocationExpression ::= Expression [ ( [ ArgumentList ] ) ]

IndexExpression ::= Expression ( [ ArgumentList ] )

ArgumentList ::=
        PositionalArgumentList , NamedArgumentList |
        PositionalArgumentList |
        NamedArgumentList
```

PositionalArgumentList ::=
  Expression |
  PositionalArgumentList , [*Expression*]

NamedArgumentList ::=
  IdentifierOrKeyword := *Expression* |
  NamedArgumentList , *IdentifierOrKeyword* := *Expression*

From syntax alone it is not possible to make the difference between a member call and a type index. Because of this there is the type *CallOrIndexExpression*, in cases where it is clear the type is *CallStatement*. Both of these types have a target expression and an argument list. The method described in section 5.3.6 is used to resolve the expression.

5.4.2.11 Nothing expression

A special case of literal is Nothing, which is the empty value.

Nothing ::= `Nothing`

In the AST nothing is represented with an *NothingExpression*. As a resolve result a special type of type declaration is returned, a *NullTypeDeclaration*.

5.4.2.12 Type reference expression

Type references are recognized as such by the syntactical analyzer, they are represented by an *TypeReferenceExpression* Object.

TypeName ::=
  ArrayTypeName |
  NonArrayTypeName

NonArrayTypeName ::=
  SimpleTypeName |
  ConstructedTypeName

SimpleTypeName ::=
  QualifiedIdentifier |
  BuiltInTypeName

BuiltInTypeName ::= `Object` | *PrimitiveTypeName*

PrimitiveTypeName ::= *NumericTypeName* | `Boolean` | `Date` | `Char` | `String`

NumericTypeName ::= *IntegralTypeName* | *FloatingPointTypeName* | `Decimal`

IntegralTypeName ::= `Byte` | `SByte` | `UShort` | `Short` | `UInteger` | `Integer` | `ULong` |
  `Long`

FloatingPointTypeName ::= `Single` | `Double`

ArrayTypeName ::= *NonArrayTypeName* *ArrayTypeModifiers*

ArrayTypeModifiers ::= *ArrayTypeModifier+*

ArrayTypeModifier ::= ([*RankList*])

```
RankList ::=
        , |
        RankList ,

QualifiedIdentifier ::=
        Identifier |
        Global . IdentifierOrKeyword |
        QualifiedIdentifier . IdentifierOrKeyword

IdentifierOrKeyword ::= Identifier | Keyword
```

The type system is used to resolve type references using the method described in section 5.2.

5.4.2.13 Simple name expression

A simple name expression is a standalone identifier with possibly a list of type arguments. It can point to a member or even a type.

```
SimpleNameExpression ::= Identifier [ ( Of TypeArgumentList ) ]
```

The method described in section 5.3.1 is used to resolve such an expression.

5.4.2.14 Qualified expression

A qualified expression is an identifier which is prefixed with another expression.

```
MemberAccessExpression ::=
        [ [ MemberAccessBase ] . ] IdentifierOrKeyword

MemberAccessBase ::=
        Expression |
        BuiltInTypeName |
        Global |
        MyClass |
        MyBase
```

The method described in section 5.3.2 is used to resolve such an expression.

5.5 Justification of the classification against the coding specification

The complete behaviour of the Visual Basic is described in the language specification (15). In this section all classification algorithms described before are justified against the classification as described in the code specification. The classification and resolution steps taken from the language specification are printed in gray.

5.5.1 Type resolution justification

Type resolution is the process by which a type reference is classified (or bound to) the type it refers to. This is discussed in section 5.2. In the language specification (15), a difference is made between qualified name resolution and simple name resolution. In this project type resolution is done in a different way than described in the language specification. In the language specification namespaces can be resolved. In our namespaces don’t exist as entities, because

a namespace itself cannot be used standalone. Another difference between the specification and our implementation is that unqualified names and qualified names are resolved in the same routine instead of two separate ones as the language specification would naturally imply.

First let us state the qualified name resolution taken from the language specification (15 p. 52):

Given a qualified namespace or type name of the form N.R, where R is the rightmost identifier in the qualified name, the following steps describe how to determine to which namespace or type the qualified name refers:

1. *Resolve N, which may be either a qualified or unqualified name.*

The resolution in the language specification describes here that N.R could be a namespace or type. As described in section 5.2; in our implementation there is no namespace entity, types qualified with a namespace are resolved in one step, by sequentially constructing and trying all possible fully qualified names until the type is found. Hence, type resolution is not done recursively.

2. *If resolution of N fails, resolves to a type parameter, or does not resolve to a namespace or type, a compile-time error occurs. If R matches the name of a namespace or type in N, then the qualified name refers to that namespace or type.*

If R matches the name of a type in N, it is found by looking up the fully qualified name N.R; as explained above, it will never match a namespace.

3. *If N contains one or more standard modules, and R matches the name of a type in exactly one standard module, then the qualified name refers to that type. If R matches the name of types in more than one standard module, a compile-time error occurs.*

For every Q.T within the fully qualified name where Q is a namespace containing modules, for every module with name M within namespace Q, the fully qualified name Q.N.T is a candidate. These candidates are tried from the longest Q to the shortest that can be constructed with the fully qualified name. A module cannot be nested, therefore a type's fully qualified name can maximally "hide" one module; thus this method described in section 5.2.2 will find every type nested in a module.

The unqualified name resolution from the language specification (15 p. 52), reads as follows:

Given an unqualified name R, the following steps describe how to determine to which namespace or type an unqualified name refers:

1. *For each nested type containing the name reference, starting from the innermost type and going to the outermost, if R matches the name of an accessible nested type or a type parameter in the current type, then the unqualified name refers to that type or type parameter.*
2. *For each nested namespace containing the name reference, starting from the innermost namespace and going to the outermost namespace, do the following:*
    a. *If R matches the name of an accessible type or nested namespace in the current namespace, then the unqualified name refers to that type or nested namespace.*

b. *If the namespace contains one or more accessible standard modules, and R matches the name of an accessible nested type in exactly one standard module, then the unqualified name refers to that nested type. If R matches the name of accessible nested types in more than one standard module, a compile-time error occurs.*

The qualified path of the reference is taken and from the innermost to the outermost location, the name is looked up. The solution is indiscriminate about if name parts in a qualified location are nested types or nested namespaces. In the language specification types and namespaces are handled separately one after the other in step 1 and 2. It is not a problem though to do it in one step; within a namespace, namespaces and types are not allowed to have the same name and types cannot have sub namespaces, thus a conflict will never occur.

If no type is found in step 1 and 2a, in step 2b, R is sought in the accessible modules. In our implementation this is handled in the lookup algorithm as described in section 5.2.2.

As described in section 5.2.3.2, all possible fully qualified names which can be constructed by concatenating the current fully qualified location and super fully qualified locations are constructed. Part of this fully qualified name can be namespaces and part can be type names.

For example a type named T which is a directly nested type in a type with the fully qualified name N.S, its fully qualified name then will be N.S.T.

This way, nested types and types in nested namespaces will both be found. Because the constructed locations are constructed from the most inner location to the outer, nested types will be found before types in nested namespaces. Type parameters are added to the type list, so they will be found in this same way.

3. *If the source file has one or more import aliases, and R matches the name of one of them, then the unqualified name refers to that import alias.*

As described in section 5.2.3.3; when an aliased import is found of which the alias matches with the start of the name sought, the matching part is replaced by the import. If the name sought completely matches the alias, this means the fully qualified name candidate will be the import itself.

4. *If the source file containing the name reference has one or more imports:*
    a. *If R matches the name of an accessible type in exactly one import, then the unqualified name refers to that type. If R matches the name of an accessible type in more than one import and all are not the same entity, a compile-time error occurs.*
    b. *If R matches the name of a namespace in exactly one import, then the unqualified name refers to that namespace. If R matches the name of a namespace in more than one import and all are not the same entity, a compile-time error occurs.*
    c. *If the imports contain one or more accessible standard modules, and R matches the name of an accessible nested type in exactly one standard module, then the unqualified name refers to that type. If R matches the name of accessible nested types in more than one standard module, a compile-time error occurs.*

As described in section 5.2.3.4: for every import in the source file containing the name, the namespace of the import is concatenated with this name to construct a candidate. This covers the situation that the import is an import of a namespace.

If the last name part of the import is the same as the first name part of the name, in the case of a simple name this is the whole name, then the last name part of the import is replaced by the name to construct a second candidate. This covers the situation the import imports a type.

If the resolution is ambiguous and both candidates described above will match a type, the code will not compile; with the precondition the code input is compilable, this guarantees the correct type.

The lookup algorithm will ensure that if the name sought matches a member of a standard module, it will be found.

5. *If the compilation environment defines one or more import aliases, and R matches the name of one of them, then the unqualified name refers to that import alias.*
6. *If the compilation environment defines one or more imports:*
    a. *If R matches the name of an accessible type in exactly one import, then the unqualified name refers to that type. If R matches the name of an accessible type in more than one import, a compile-time error occurs.*
    b. *If R matches the name of a namespace in exactly one import, then the unqualified name refers to that namespace. If R matches the name of a namespace in more than one import, a compile-time error occurs.*
    c. *If the imports contain one or more accessible standard modules, and R matches the name of an accessible nested type in exactly one standard module, then the unqualified name refers to that type. If R matches the name of accessible nested types in more than one standard module, a compile-time error occurs.*

The imports defined in the compilation environment are handled exactly the same as the imports defined in the source file.

Step 5 and 6 are the same as 3 and 4 except the imports defined in the compile environment (the project file) are used instead of the imports in the source file itself, the solution therefore also is the same.

It might be unclear at this point if types in modules and out of modules are resolved in the same order as described in the code specification. For this take for example unqualified name C which is referred within the namespace A.B. Presume every namespace contains a module named M. The order in which the solution tries out the candidates will be as follows:

- First a list of nested namespaces is constructed, C resides in namespace A.B, so the list will be A.B and A, it is impossible to create types at the same level as the root namespace so there is no empty namespace to look in.
- Lookup with the first implicit nested namespace name part A.B and explicit name part C:
- Lookup as is: **A.B.C**
- Add potentially hidden module names to the explicit part: **A.B.M.C**
- Lookup with the second implicit nested namespace name part A and explicit name part C:

- Lookup as is: **A.C**
- Add potentially hidden module names to the explicit part: **A.M.C**
- Lookup the referenced name as is **C**

If the same resolution is done with the steps described in the code specification the result is as follows:

Search the inner most namespace A.B for C: **A.B.C**

For all modules available in A.B try to find C: **A.B.M.C**

Search the next nested namespace A for C: **A.C**

For all modules available in A try to find C => **A.M.C**

The resolution in both is the same A.B.C, A.B.M.C, A.C and A.M.C. If A.B.C would be fully qualified in the reference the resolution would look be as follows:

- As before first the list of nested namespaces is constructed; A.B and A.
- Lookup with the first implicit nested namespace name part A.B and explicit name part A.B.C:
- Lookup as is: **A.B.A.B.C**
- Add potentially hidden module names to the explicit part: **A.B.A.B.M.C**, **A.B.A.M.B.C** and **A.B.M.A.B.C**
- Lookup with the second implicit nested namespace name part A and explicit name part A.B.C:
- Lookup as is: **A.A.B.C**
- Add potentially hidden module names to the explicit part: **A.A.B.M.C**, **A.A.M.B.C** and **A.M.A.B.C**
- Lookup the referenced name as is **A.B.C**
- Add potentially hidden module names to the explicit part: **A.B.M.C**, **A.M.B.C** (M.A.B.C will never be tried because there cannot exist a standard module at the same level as the root namespace)

The order in which the resolution described in the code specification is as follows. The resolution is a recursive process where the qualifier is resolved first. This example starts at the bottom of this recursive process where the simple name is resolved and works up again processing the result of the last step. Some of the branches end in 'impossible', these branches will never happen because it is impossible to have a standard module to be nested in another standard module.

- Search the inner most namespace A.B for A: A.B.A.*
    - Search B in the result: A.B.A.B.*
        - Search C in the result: **A.B.A.B.C**
        - Search C in modules in the result: **A.B.A.B.M.C**
    - Search B in modules in the result: A.B.A.M.B.*
        - Search C in the result **A.B.A.M.B.C**
        - Search C in modules in the result: A.B.A.M.B.M.C - impossible

- Search A in modules in namespace A.B: A.B.M.A.*
    - Search B in the result: A.B.M.A.B.*
        - Search C in the result: **A.B.M.A.B.C**
        - Search C in modules in the result A.B.M.A.B.M.C - impossible
    - Search B in modules in the result: A.B.M.A.M.B.* - impossible
- Search the next nested namespace A for A: A.A.*
    - Search B in the result: A.A.B.*
        - Search C in the result: **A.A.B.C**
        - Search C in modules in the result: **A.A.B.M.C**
    - Search B in modules in the result: A.A.M.B.*
        - Search C in the result: **A.A.M.B.C**
        - Search C in modules in the result: A.A.M.B.M.C - impossible
- Search A in modules in namespace A: A.M.A.*
    - Search B in the result: A.M.A.B.*
        - Search C in the result: **A.M.A.B.C**
        - Search C in modules in the result A.M.A.B.M.C - impossible
    - A.M.A.M.B.* - impossible
- Search for A as a root namespace: A.*
    - Search B in the result: A.B.*
        - Search C in the result: **A.B.C**
        - Search C in modules in the result: **A.B.M.C**
    - Search B in modules in the result: A.M.B.*
        - Search C in the result: **A.M.B.C**
        - Search C in modules in the result: A.M.B.M.C - impossible

Both resolutions try to resolve the reference in the same order: **A.B.A.B.C**, **A.B.A.B.M.C**, **A.B.A.M.B.C, A.B.A.M.B.C, A.B.M.A.B.C**, **A.A.B.C**, **A.A.B.M.C**, **A.A.M.B.C, A.M.A.B.C**, **A.B.C**, which shows both resolution methods will find the same types.

5.5.2 Simple name expressions

Simple name expressions are non qualified identifiers as described in section 5.3.1.

The simple name expression resolution taken from the language specification (15 p. 217) reads as follows:

A simple name expression consists of a single identifier followed by an optional type argument list. The name is resolved and classified as follows:

1. *Starting with the immediately enclosing block and continuing with each enclosing outer block (if any), if the identifier matches the name of a local variable, static variable, constant local, method type parameter, or parameter, then the identifier refers to the matching entity. The expression is classified as a variable if it is a local variable, static variable, or parameter. The expression is classified as a type if it is a method type parameter. The expression is classified as a value if it is a constant local with the following exception. If the local variable matched is the implicit function or Get accessor return local variable, and the expression is part of an invocation expression, invocation statement, or an AddressOf expression, then no match occurs and resolution continues.*

As described in section 5.3.1.1: from the scope of the identifier is directly in, to the scope of the immediately enclosing type, a member with a matching name name is sought, if a match is found it is returned.

Our implementation there is no distinction made between a constant and a variable. A constant could be seen as a read only variable, the read only constrain limits the usage of it. Because of the precondition that the code has to be compilable, this distinction would not add insight. Constants can just as wel be handled as variables that happen to be only read. The distinction between non-initialized types (or variables and types) and initialized types is not ignored, because this knowledge can influence member resolution.

2. *For each nested type containing the expression, starting from the innermost and going to the outermost, if a lookup of the identifier in the type produces a match with an accessible member:*
    a. *If the matching type member is a type parameter, then the result is classified as a type and is the matching type parameter.*
    b. *Otherwise, if the type is the immediately enclosing type and the lookup identifies a non-shared type member, then the result is the same as a member access of the form Me.E, where E is the identifier.*
    c. *Otherwise, the result is exactly the same as a member access of the form T.E, where T is the type containing the matching member and E is the identifier. In this case, it is an error for the identifier to refer to a non-shared member.*

In the second resolution step described in section 5.3.1.2, a match is searched in the members of the immediately enclosing type which is encountered as an initialized type. After this, all other enclosing types are searched from the innermost to the outermost; these types are encountered as uninitialized types. When the identifier matches a type, the result will be a non initialized type, if it matches a member, it will be an initialized type.

3. *For each nested namespace, starting from the innermost and going to the outermost namespace, do the following:*
    a. *If the namespace contains an accessible namespace member with the given name, then the identifier refers to that member and, depending on the member, is classified as a namespace or a type.*
    b. *Otherwise, if the namespace contains one or more accessible standard modules, and a member name lookup of the identifier produces an accessible match in exactly one standard module, then the result is exactly the same as a member access of the form M.E, where M is the standard module containing the matching member and E is the identifier. If the identifier matches accessible type members in more than one standard module, a compile-time error occurs.*

Type classification is done by the type classification described in section 5.2 and justified in section 5.5.1. The approach used to classify results in that identifiers never classify to a namespace. All containing namespaces are searched for modules as described in section 5.3.1.3. If a matching member is found on one of the found modules, this will be the result.

4. *If the source file has one or more import aliases, and the identifier matches the name of one of them, then the identifier refers to that namespace or type.*

Type classification using import aliases also done by the algorithm described in section 5.2 and justified in section 5.5.1.

5. *If the source file containing the name reference has one or more imports:*
    a. *If the identifier matches the name of an accessible type or type member in exactly one import, then the identifier refers to that type or type member. If the identifier matches the name of an accessible type or type member in more than one import, a compile-time error occurs.*
    b. *If the identifier matches the name of a namespace in exactly one import, then the identifier refers to that namespace. If the identifier matches the name of a namespace in more than one import, a compile-time error occurs.*
    c. *Otherwise, if the imports contain one or more accessible standard modules, and a member name lookup of the identifier produces an accessible match in exactly one standard module, then the result is exactly the same as a member access of the form M.E, where M is the standard module containing the matching member and E is the identifier. If the identifier matches accessible type members in more than one standard module, a compile-time error occurs.*

Type classification using imports is done by the algorithm described in section 5.2 and justified in section 5.5.1. All standard modules accessible through imports in the source file are searched for matching members as described in section 5.3.1.4.

6. *If the compilation environment defines one or more import aliases, and the identifier matches the name of one of them, then the identifier refers to that namespace or type.*

Type and classification using non local alias imports is done by the algorithm described in section 5.2 and justified in section 5.5.1.

7. *If the compilation environment defines one or more imports:*
    a. *If the identifier matches the name of an accessible type or type member in exactly one import, then the identifier refers to that type or type member. If the identifier matches the name of an accessible type or type member in more than one import, a compile-time error occurs.*
    b. *If the identifier matches the name of a namespace in exactly one import, then the identifier refers to that namespace. If the identifier matches the name of a namespace in more than one import, a compile-time error occurs.*
    c. *Otherwise, if the imports contain one or more accessible standard modules, and a member name lookup of the identifier produces an accessible match in exactly one standard module, then the result is exactly the same as a member access of the form M.E, where M is the standard module containing the matching member and E is the identifier. If the identifier matches accessible type members in more than one standard module, a compile-time error occurs.*

Type classification using non local imports is done by the algorithm described in section 5.2 and justified in section 5.5.1. All standard modules accessible through imports in the compilation environment are searched for matching members as described in section 5.3.1.5.

8. *Otherwise, the name given by the identifier is undefined and a compile-time error occurs.*

5.5.3 Member Access Expressions

A member access expression is a qualified expression which does not classify to a type as described in section 5.3.2.

The member access expression classification from the language specification (15 p. 221) reads as follows:

A member access expression is used to access a member of an entity. A member access of the form E.I, where E is an expression, a built-in type, the keyword Global, or omitted and I is an identifier with an optional type argument list, is evaluated and classified as follows:

1. *If E is omitted, then the expression from the immediately containing With statement is substituted for E and the member access is performed. If there is no containing With statement, a compile-time error occurs.*

In section 5.3.2.1 is explained how an empty qualifier resolved to the 'With' expression found in the scope tree.

2. *If E is a type parameter, then a compile-time error results.*

A precondition is that the input code is compilable, so E will thus never be a type parameter where a type parameter isn't allowed.

3. *If E is the keyword Global, and I is the name of an accessible type in the global namespace, then the result is that type.*

The only thing that can exist directly in the global namespace is another namespace; as the type system doesn't have namespace entities but resolves types in one go, it means that if the first non namespace part of a qualified expression is a module member it will be resolved as described in section 5.3.2.2, if the first non namespace part of the qualified expression is a type, it will be resolved by the type system as described in section 5.3.2.5.

4. *If E is classified as a namespace and I is the name of an accessible member of that namespace, then the result is that member. The result is classified as a namespace or a type depending on the member.*

As described before, I will never be a namespace. If I is a member of a module in namespace E it will be found as described in section 5.3.2.3, if I is a type it will be resolved by the type system as described in section 5.2.3.

5. *If E is a built-in type or an expression classified as a type, and I is the name of an accessible member of E, then E.I is evaluated and classified as follows:*
   a. *If I is the keyword New, then a compile-time error occurs.*
   b. *If I identifies a type, then the result is that type.*
   c. *If I identifies one or more methods, then the result is a method group with the associated type argument list and no associated instance expression.*

d. If I identifies one or more properties, then the result is a property group with no associated instance expression.

e. If I identifies a shared variable, and if the variable is read-only, and the reference occurs outside the shared constructor of the type in which the variable is declared, then the result is the value of the shared variable I in E. Otherwise, the result is the shared variable I in E.

f. If I identifies a shared event, the result is an event access with no associated instance expression.

g. If I identifies a constant, then the result is the value of that constant.

h. If I identifies an enumeration member, then the result is the value of that enumeration member.

i. Otherwise, E.I is an invalid member reference, and a compile-time error occurs.

6. If E is classified as a variable or value, the type of which is T, and I is the name of an accessible member of E, then E.I is evaluated and classified as follows:

a. If I is the keyword New and E is an instance expression (Me, MyBase, or MyClass), then the result is a method group representing the instance constructors of the type of E with an associated instance expression of E and no type argument list. Otherwise, a compile-time error occurs.

b. If I identifies one or more methods, then the result is a method group with the associated type argument list and an associated instance expression of E.

c. If I identifies one or more properties, then the result is a property group with an associated instance expression of E.

d. If I identifies a shared variable or an instance variable, and if the variable is read-only, and the reference occurs outside a constructor of the class in which the variable is declared appropriate for the kind of variable (shared or instance), then the result is the value of the variable I in the object referenced by E. If T is a reference type, then the result is the variable I in the object referenced by E. Otherwise, if T is a value type and the expression E is classified as a variable, the result is a variable; otherwise the result is a value.

e. If I identifies an event, the result is an event access with an associated instance expression of E.

f. If I identifies a constant, then the result is the value of that constant.

g. If I identifies an enumeration member, then the result is the value of that enumeration member.

h. If T is Object, then the result is a late-bound member lookup classified as a late-bound access with an associated instance expression of E.

If E is a type T, this type will be resolved first and I will be found by applying member resolution on E as described in section 5.3.2.1. If I is a type, the type is resolved in one time as E.I by the type system as described in section 5.2.3. If I is a variable of type T, the variable is resolved as described in section 5.3.1 and 5.3.2. A method group is not an entity which can be used as is in visual basic 8, overloaded method resolution has to be done on it to find the appropriate method. In the implementation all

accessible members matching I in type T are found and I classifies to the most appropriate one as described in section 5.3.6.

7. Otherwise, E.I is an invalid member reference, and a compile-time error occurs.

5.5.4 Overloaded Method Resolution

If a type contains more methods having the same name, the correct method is found using overloaded method resolution as described in section 5.3.6.

The overloaded method resolution from the language specification (15 p. 228) reads as follows:

Given a method group, the applicable method in the group for an argument list is determined as follows:

Members which are applicable and accessible are added as described as in section 5.3.6 instead of starting with a complete list, eliminating all inaccessible members from the list.

1. Eliminate all inaccessible members from the set.

As described in the previous step, inaccessible members are not added to the matching member list; section 5.3.6.3 discusses accessibility.

2. Eliminate all members from the set that are not applicable to the argument list. If the set is empty, a compile-time error results. If only one member remains in the set, that is the applicable member.

As described in the first step, only members with an applicable argument list are added to the member list, also see section 5.3.6.2. If after the construction of the member list it only contains one member, this is considered the correct member.

3. Eliminate all members from the set that require narrowing coercions to be applicable to the argument list, except for the case where the argument expression type is Object. If the set is empty, a compile-time error results. If only one member remains in the set, that is the applicable member.

As described in section 5.3.6 all members which require the implicit narrowing coercion are removed from the list, if only one member remains in the list, it is considered the correct member.

4. Eliminate all remaining members from the set that require narrowing coercions to be applicable to the argument list. If the set is empty, the type containing the method group is not an interface, and strict semantics are not being used, the invocation target expression is reclassified as a late-bound method access. If strict semantics are being used or the method group is contained in an interface and the set is empty, a compile-time error results. If only one member remains in the set, that is the applicable member.

A precondition is that the code should be compilable with option strict on, this makes this step redundant.

5. Given any two members of the set, M and N, if M is more applicable than N to the argument list, eliminate N from the set. If only one member remains in the set, that is the applicable member. If the remaining members do not all have the same signature, a compile-time error results.

All members in the list for which there exists a member in the list which is more applicable are removed from the list as described in sections 5.3.6 and 5.3.6.4.

6. *Otherwise, it must be the case that the remaining members have the same signature because of type parameters. Given any two members of the set, M and N, if M is less generic than N, eliminate N from the set. If only one member remains in the set, that is the applicable member. Otherwise, a compile-time error results.*

All members in the list for which there exists a member in the list which is less generic are removed from the list as described in 5.3.6 and 5.3.6.5.

A member M is considered more applicable than N if their signatures are different and, for each pair of parameters Mj and Nj that matches an argument Aj, one of the following conditions is true:

In section 5.3.6.4 is described how parameters of members are compared to determine the most applicable member.

1. *Mj and Nj have identical types, or*

If the type of two parameters differs, the algorithm checks if each parameter is considered more applicable than the other. The steps below can be seen back in the algorithm.

2. *There exists a widening conversion from the type of Mj to the type Nj, or*
3. *Aj is the literal 0, Mj is a numeric type and Nj is an enumerated type, or*
4. *Mj is Byte and Nj is SByte, or*
5. *Mj is Short and Nj is UShort, or*
6. *Mj is Integer and Nj is UInteger, or*
7. *Mj is Long and Nj is ULong.*

A member M is determined to be less generic than a member N using the following steps:

In section 5.3.6.5 is described how the least generic member is determined.

1. *If M has fewer method type parameters than N, then M is less generic than N.*

One can see that in the algorithm the member with the least type parameters is considered the least generic member.

2. *Otherwise, if for each pair of matching parameters Mj and Nj, Mj and Nj are equally generic with respect to type parameters on the method, or Mj is less generic with respect to type parameters on the method, and at least one Mj is less generic than Nj, then M is less generic than N.*

In the algorithm one can see the member which has all of the parameters matching the arguments equally or less generic with respect to the type parameters on the member is considered the least generic.

3. *Otherwise, if for each pair of matching parameters Mj and Nj, Mj and Nj are equally generic with respect to type parameters on the type, or Mj is less generic with respect to type parameters on the type, and at least one Mj is less generic than Nj, then M is less generic than N.*

In the algorithm one can see the member which has all of the parameters matching the arguments equally or less generic with respect to the type parameters on the type is considered the least generic.

5.5.5 Applicable Methods

A method is *applicable* if a method call can be matched with a method call as described in 5.3.6.2.

The applicable methods definition from the language specification (15 p. 231) reads as follows**Error! Reference source not found.**:

A method is applicable to a set of type arguments, positional arguments, and named arguments if the method can be invoked using the two argument lists. The argument lists are matched against the parameter lists as follows:

The algorithm described in section 5.3.6.2 determines if a method is applicable or not.

1. *First, match each positional argument in order to the list of method parameters. If there are more positional arguments than parameters and the last parameter is not a paramarray, the method is not applicable. Otherwise, the paramarray parameter is expanded with parameters of the paramarray element type to match the number of positional arguments. If a positional argument is omitted, the method is not applicable.*

In the algorithm all positional arguments are matched in the first loop. If there are more arguments than parameters, they are matched against the parameter array if it exists. If there are more arguments than parameters which do not match a parameter array, the method is considered not applicable.

2. *Next, match each named argument to a parameter with the given name. If one of the named arguments fails to match, matches a paramarray parameter, or matches an argument already matched with another positional or named argument, the method is not applicable.*

In the second loop in algorithm for every named argument is checked if there exists a parameter with the correct name which is not a parameter array and is not covered by another positional argument in the previous loop. If not found, the method is not considered applicable. If more named arguments match a parameter this means the input code will not compile, with the precondition the input code must be compilable, checking for more arguments with the same name is not necessary.

3. *Next, if parameters that have not been matched are not optional, the method is not applicable. If optional parameters remain, the default value specified in the optional parameter declaration is matched to the parameter. If an Object parameter does not specify a default value, then the expression System.Reflection.Missing.Value is used. If an optional Integer parameter has the Microsoft.VisualBasic.CompilerServices.OptionCompareAttribute attribute, then the literal 1 is supplied for text comparisons and the literal 0 otherwise.*

After checking if all named arguments match a parameter, the algorithm checks if for all parameters not covered in the first loop a named argument exists. If one is found, the argument type is matched to the parameter type. If the type doesn't match the member is considered not matching. If no named argument is found, it is checked the parameter is optional. If it is not optional the member is considered not matching. A precondition is the code must be compilable with option strict on, option strict on prohibits optional parameters without a specified default value.

4. *Finally, if type arguments have been specified, they are matched against the type parameter list. If the two lists do not have the same number of elements, the method is not applicable, unless the type argument list is empty. If the type argument list is empty, type inferencing is used to try and*

infer the type argument list. If type inferencing fails, the method is not applicable. Otherwise, the type arguments are filled in the place of the type parameters in the signature.

In section 5.3.6.1 is described how constructed members with the correct type arguments are added before trying to find the most applicable member. The member is matched against the constructed member on which all type parameters are substituted with the corresponding type arguments.

5.5.6 Type Argument Inference

The type argument inference from the language specification (15 p. 234) reads as follows:

When a method with type parameters is called without specifying type arguments, type inference is used to try and infer type arguments for the call. This allows a more natural syntax to be used for calling a method with type parameters when the type arguments can be trivially inferred. For example, given the following method declaration:

```
Module Util
    Function Choose(Of T)(ByVal b As Boolean, ByVal first As T, _
        ByVal second As T) As T
        If b Then
            Return first
        Else
            Return second
        End If
    End Function
End Class
```

it is possible to invoke the Choose method without explicitly specifying a type argument:

```
' calls Choose(Of Integer)
Dim i As Integer = Util.Choose(True, 5, 213)
' calls Choose(Of String)
Dim s As String = Util.Choose(False, "foo", "bar")
```

Through type inference, the type arguments Integer and String are determined from the arguments to the method.

Given a set of regular arguments matched to regular parameters, type inference inspects each argument type A and its corresponding parameter type P to infer type arguments. Each pair (A, P) is analyzed as follows:

1. *Nothing is inferred from the argument (but type inference succeeds) if any of the following are true:*
    a. *P does not involve any method type parameters.*
    b. *The argument is the Nothing literal.*
    c. *The argument is a method group.*

In our implementation described in section 5.3.6.1, all type arguments must be determined to succeed. If the type argument cannot be inferred from an argument, this is no problem as long as other arguments provide enough information to determine the type arguments.

2. *If P is an array type and A is an instantiation of IList(Of T), ICollection(Of T), or IEnumerable(Of T), then replace P with the element type of P, and A with the type argument of A and restart the process.*

This step states IList(Of T), ICollection(Of T) and IEnumerable(OF T) match the array type T(). This feature is not implemented yet in our implementation. If this case occurs, it could be a member reference cannot be classified or possibly classify to the wrong member.

3. *If P is an array type and A is an array type of the same rank, then replace A and P respectively with the element types of A and P and restart this process.*

If P is an array type and A is an array type of the same rank, then the element type of P is the type parameter. If the type argument for it is sought and A is the first instance, the element type of A will be the type argument found by the algorithm.

4. *If P is an array type and A is not an array type of the same rank, then type inference fails for the generic method.*

If the type argument for P is sought and A is not an array type, the type argument inference fails in the implementation. It could be another argument still would result in a type argument but the resulting constructed member will not match the signature.

5. *If P is a method type parameter, then type inference succeeds for this argument, and A is the type inferred for that type parameter.*

If a type argument is sought for P and at the first instance P is used directly as a parameter, then in the implementation the type of the argument provided for this parameter will be the type argument.

6. *Otherwise, P must be a constructed type. If, for each method type parameter MX that occurs in P, exactly one type TX can be determined such that replacing each MX with each TX produces a type to which A is convertible by a standard implicit conversion, then inference succeeds for this argument, and each TX is the type inferred for each MX. Method type parameter constraints, if any, are ignored for the purpose of type inference. If, for a given MX, no TX exists or more than one TX exists, then type inference fails for the generic method (a situation where more than one TX exists can only occur if P is a generic interface type and A implements multiple constructed versions of that interface).*

In the implementation TX will be the type of the argument at the first occurrence of MX. No least general base type for all occurrences of MX as described in this step is found by the implementation. The situation in which this will result in a wrong classification is very rare though.

7. *If a parameter is a paramarray, then type inference is first performed against the parameter in its normal form. If type inference fails, then type inference is performed against the parameter in its expanded form.*

The type inference implementation ignores the fact a parameter can be a parameter array, this means only the normal form of the parameter is evaluated.

8. *If all of the method arguments have been processed successfully by the above algorithm, all inferences that were produced from the arguments are pooled. This pooled set of inferences must have the following properties:*

    a. *Every type parameter of the method must have a matching inferred type argument.*

    b. *If a type parameter occurred more than once, all of the inferences about that type parameter must infer the same type argument.*

In the implementation, when a generic member is referenced, a constructed member is added if for all type parameters a type argument can be inferred from the arguments. Member resolution is than applied on all members including the constructed members. If not all arguments fit the parameter types of the constructed member, the constructed member won't be regarded a match.

5.5.7 Index Expressions

The index expression resolution from the language specification (15 p. 235) reads as follows:

An index expression results in an array element or reclassifies a property group into a property access. An index expression consists of, in order, an expression, an opening parenthesis, an index argument list, and a closing parenthesis. The target of the index expression must be classified as either a property group or a value. An index expression is processed as follows:

1. *If the target expression is classified as a value and if its type is not an array type, Object, or System.Array, the type must have a default property. The index is performed on a property group that represents all of the default properties of the type. Although it is not valid to declare a parameterless default property in Visual Basic, other languages may allow declaring such a property. Consequently, indexing a property with no arguments is allowed.*

As can be read in section 5.3.5 when the expression classifies to a value which is not of an array type, a default property matching the arguments is sought.

2. *If the expression results in a value of an array type, the number of arguments in the argument list must be the same as the rank of the array type and may not include named arguments. If any of the indexes are invalid at run time, a System.IndexOutOfRangeException exception is thrown. Each expression must be implicitly convertible to type Integer. The result of the index expression is the variable at the specified index and is classified as a variable.*

In section 5.3.5 can be seen that if the expression results in an array type, the element type of the array type is regarded to be the type the index expression classifies to. If the expression results in an array type the only possibilities are that the complete index expression classifies to an element within the array of which the element type of the array is the type or the compilation will fail. Because a precondition is the input code is compilable, it is correct to state that any index argument is a match and the element type is always the correct type to classify to.

3. *If the expression is classified as a property group, overload resolution is used to determine whether one of the properties is applicable to the index argument list. If the property group only contains one property that has a Get accessor and if that accessor takes no arguments, then the property group is interpreted as an index expression with an empty argument list. The result is used as the target of the current index expression. If no properties are applicable, then a compile-time error occurs. Otherwise, the expression results in a property access with the associated instance expression (if any) of the property group.*

If the expression classifies to a property group that means the whole index expression classifies to a specific property or a compile-time error occurs. A precondition is the code is compilable, so the index expression will always classify to one property. In section 5.3.6 is described how the property will be found by applying member resolution.

4. *If the expression is classified as a late-bound property group or as a value whose type is Object or System.Array, the processing of the index expression is deferred until run time and the indexing is late-bound. The expression results in a late-bound property access typed as Object. The associated instance expression is either the target expression, if it is a value, or the associated instance expression of the property group. At run time the expression is processed as follows:*

    a. *If the expression is classified as a late-bound property group, the expression may result in a method group, a property group, or a value (if the member is an instance or shared variable). If the result is a method group or property group, overload resolution is applied to the group to determine the correct method for the argument list. If overload resolution fails, a System.Reflection.AmbiguousMatchException exception is thrown. Then the result is processed either as a property access or as an invocation and the result is returned. If the invocation is of a subroutine, the result is Nothing.*

    b. *If the run-time type of the target expression is an array type or System.Array, the result of the index expression is the value of the variable at the specified index.*

    c. *Otherwise, the run-time type of the expression must have a default property and the index is performed on the property group that represents all of the default properties on the type. If the type has no default property, then a System.MissingMemberException exception is thrown.*

Late bound expressions are not possible if the option strict is set. A precondition is that the option strict is set so this step does not apply.

5.5.8 Operator resolution

The operator resolution from the language specification (15 p. 243) reads as follows:

Given an operator type and a set of operands, operator resolution determines which operator to use for the operands. When resolving operators, user-defined operators will be considered first, using the following steps:

As discussed in section 5.4.1, user-defined operators are considered first; only if no user-defined operator is found, a built in operator is sought.

1. *First, all of the candidate operators are collected. The candidate operators are all of the user-defined operators of the particular operator type in the source type and all of the user-defined operators of the particular type in the target type. If the source type and destination type are related, common operators are only considered once.*

As discussed in section 5.4.1.1, all matching operators are extracted from the source and target types. Only operators of which the parameters match the type of the operands are collected.

2. *Then, overload resolution is applied to the operators and operands to select the most specific operator.*

Of all matching operators, an exact match is selected if available, else the most specific match is selected. An operator is more specific than another if the first parameter type that is different is more specific.

When resolving overloaded operators, there may be differences between classes defined in Visual Basic and those defined in other languages:

3. *In other languages, Not, And, and Or may be overloaded both as logical operators and bitwise operators. Upon import from an external assembly, either form is accepted as a valid overload for these operators. However, for a type which defines both logical and bitwise operators, only the bitwise implementation will be considered.*
4. *In other languages, >> and << may be overloaded both as signed operators and unsigned operators. Upon import from an external assembly, either form is accepted as a valid overload. However, for a type which defines both signed and unsigned operators, only the signed implementation will be considered.*

These rules are not implemented; all user defined operaters are resolved in the same way; it is undefined if the bitwise or logical implementation or signed or unsigned operators are selected.

If no user-defined operator is most specific to the operands, then pre-defined operators will be considered. If no pre-defined operator is defined for the operands, then a compile-time error results.

When no user-defined operator is found, the pre-defined operator is looked up. The precondition that the source is compilable guarantees an operator is available.

Each operator lists the pre-defined types it is defined for and the type of the operation performed given the operand types. The result of type of a pre-defined operation follows these general rules:

5. *If all operands are of the same type, and the operator is defined for the type, then no conversion occurs and the operator for that type is used.*
6. *Any operand whose type is not defined for the operator is converted using the following steps and the operator is resolved against the new types:*
   a. *The operand is converted to the next widest type that is defined for both the operator and the operand and to which it is implicitly convertible.*
   b. *If there is no such type, then the operand is converted to the next narrowest type that is defined for both the operator and the operand and to which it is implicitly convertible.*
   c. *If there is no such type or the conversion cannot occur, a compile-time error occurs.*
7. *Otherwise, the operands are converted to the wider of the operand types and the operator for that type is used. If the narrower operand type cannot be implicitly converted to the wider operator type, a compile-time error occurs.*

Despite these general rules, however, there are a number of special cases called out in the operator results tables.

Note For formatting reasons, the operator type tables abbreviate the predefined names to their first two characters. So "By" is Byte, "UI" is UInteger, "St" is String, etc. "Err" means that there is no operation defined for the given operand types.

The type specification provides all built in operator type results, these type results are all put in tables which provide functionality to lookup a resulting type given an operator type and two operand types.

5.6 Summary

In this chapter we discussed how the type system constructed during the first pass of our semantic analysis is used to classify expressions, or resolve expressions to a type. First we explain how the result of the process is constructed in a way to make it easy to deduct its type, origin and (when applicable) the constant value of an expression.

Next, we described how types are identified and queried from the type system. A type is identified by creating a key using the unique id of the containing project or assembly together with their fully qualified name. Finding a correct type is done by constructing different candidate keys and try to match them with the types in the type system. These candidates are constructed in such order that the first hit is the correct type according to the resolution rules described in the language specification (15).

The third step in our classification process shows how a reference to a local variable or member is resolved. Just as with searching for a correct type, the member resolution is based on trying out different ways to classify the identifier in such a way the first hit is the correct one according to the resolution rules in the Visual Basic language specification (15)**Error! Reference source not found.**.To classify a correct we need to match a signature and to decide which of the potentially multiple matching members is the most applicable one. To classify to members in a module it is necessary to find modules which are in a certain namespace.

In addition to type and member references, expressions also classify to a type. An expression can for example be an arithmetic operation. The involved operator determines the resulting type of the expression. Which operator is used is determined by the operator type of the expression together with the types the operands classify to. The operator can be a built in operator in case the operands of the expressions classify to intrinsic types, but it also can be user defined. In the latter case operator resolution steps need to be done to find the correct operator declaration.

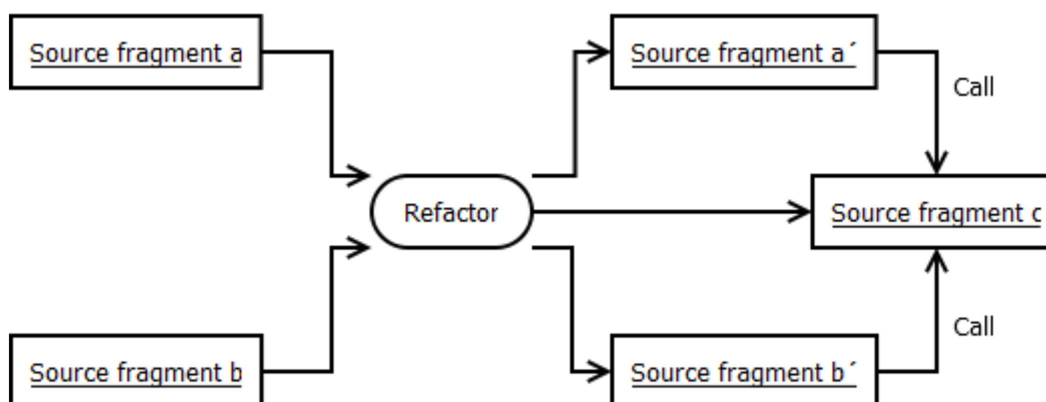
Throughout the chapter, we described how our resolution algorithms relate to the actual Visual Basic language specification (15). This description should provide a means to verify the correctness and completness of our implementation. Moreover, this also reflects on the complexity and effort required to generate a type resolution mechanism, an aspect which is much too often overlooked by the literature and practice on static analysis, but which is crucial when building language processing tools.

6 Code refactoring

In previous chapters we have shown how source code is transformed into abstract syntax trees (ASTs), how a type system is built and how type resolution can be used to determine the type and origin of values represented by expressions in the code. In this chapter we show how this functionality can be used to find semantically equally pieces of code and optionally remove them by rewriting the code using a unifying method extraction refactoring. Refactorings are a way to transform code in a behaviour preserving manner (15). One of them is the method extraction pattern described in (11 p. 110). The method extraction pattern describes how a fragment of code can be lifted out and placed in a newly created method, and how the original fragment can be replaced by a call to the extracted method.

A "simple" method extraction is based on one code fragment. In contrast the method extraction that needs to be performed in our context takes two code fragments and constructs a method to which a call can replace both fragments. For lack of a better term, we call this a *unifying* method extraction. To perform a useful unifying method extraction in the context of clone refactoring, the matching algorithm must be constructed in such a way that its results are code clones which are fit to be extracted.

The refactoring system looks like shown in the diagram below.



The input of the refactoring system is formed by two source code fragments: source fragment a and source fragment b. If these contain a semantically equal sub fragment, the output will be three code fragments. One fragment (source fragment c) will be a newly created method containing a replacement for the semantically equal sub fragments from the input. The other two fragments (source fragment a' and source fragment b') are source fragment a and source fragment b transformed in such a way that the semantically equal sub fragments are replaced by a call to the newly created method (c) and the external behaviour of source fragment a' and source fragment b' are identical to source fragment a and source fragment b.

Consider the next example, in which the next two code fragments are the bodies of some members.

```
1    Dim c As Integer
2    c = 2
3    c += 1
4    For a As Integer = 1 To 10
5        c += a
6    Next
```

```
1    Dim c As Integer = 0
2    c += 1
3    For b As Integer = 1 To 100
4        c += b
5    Next
6    c += 1
7    c += 1
```

If we want to refactor these fragments manually by creating a new method to which a call replaces the matching highlighted fragment without changing the external behaviour of the program, first we need to identify the matching sub-fragments. When the sub-fragments are identified, a new method is created. In the new method we ignore the fact the control variables of the loops 'a' and 'b' are named differently and just pick a name, say 'a'. The upper bound expressions '10' and '100' differ, so they would be replaced with a variable. The result could be as follows.

```
1    Function NewFunc(ByVal c As Integer, ByVal newVar As
Integer) As Integer
2        c += 1
3        For a As Integer = 1 To newVar
4            c += a
5        Next
6        Return c
7    End Function
```

The code in the example could be altered as follows without changing the external behaviour of the program.

```
1    Dim c As Integer
2    c = 2
3    c = NewFunc(c, 10)
```

```
1    Dim c As Integer = 0
2    c = NewFunc(c, 100)
3    c += 1
4    c += 1
```

We ignore here the fact variable 'a' and 'b' have different names; because both 'a' and 'b' are of the same type and used exactly the same way in both code fragments, they can be safely replaced with a new variable named 'a'. We chose to replace the upper bound expressions with a variable because both '10'and '100' are of the same type in the fragments, but their value differs. Implicitly the variable 'c' is replaced by a new variable as well. Hence the newly created method contains three variables; 'a', 'c' and 'newVar', two of which ('c' and 'newVar') became method parameters,and one of them ('c') is returned by the method. The values of 'c' and 'newVar' need to be passed as arguments because their values differ in both original fragments. The value of 'c' is returned because it is used again after the method call.

Chances that a programmer doing manual refactoring would next like to further transform the two fragments as follows.

```
1    Dim c = NewFunc(2, 10)
```

```
1    Dim c = NewFunc(0, 100)
2    c += 1
3    c += 1
```

In the first step of this example, except for the method extraction itself only substitutions are done, the statements are not changed. The extracted method body begins with a compound assignment and after that has a for-loop statement containing a compound assignment in its body, exactly the same as in the replaced fragments. The second step involves transformations; multiple statements are combined to one different statement.

For implementation simplicity, in this thesis only substitutions and no transformations are performed. Hence, a unifying method extraction as done in the first step of the example is done. Clean up transformations as illustrated in the second step of the example are not performed, but they can be added in future work if so desired.

How duplicate fragments are related to eachother influences how they can be removed. In (16) three cases of code duplication are identified:

- Duplication inside of a single class.
- Duplication between sibling classes.
- Duplication between unrelated classes.

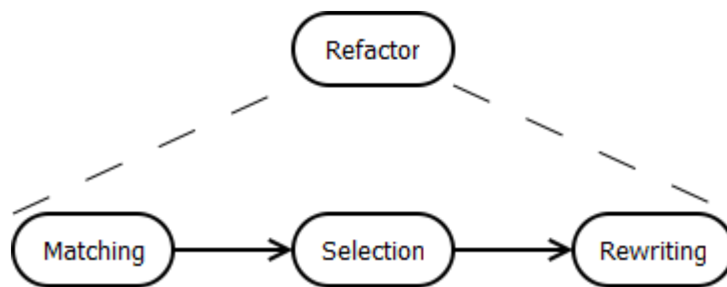
Each of these cases needs a different approach when refactored. The first case is the simplest one; no special care has to be taken about class boundaries. In the second case, where classes share the same super class, it might be possible to refactor the code clone away by placing a unifying extracted method in the super class. The most difficult case is where there are code clones in unrelated classes; this could indicate a new responsibility which could be factored out into its own class.

The scope a method is contained in influences how variables and members can be reached. If, for example, an extractable method needs to access members of an instance of type 'A', how the method is to be constructed depends on where the method resides.

For example, if the extracted method is placed into 'A' itself, then other methods of 'A' are directly accessible from the method. If the method is placed a super type of 'A,' only the methods in the super type are directly accessible. If the method is placed in another completely different type, an instance of 'A' needs to be available in some other way to access methods on it.

If a unifying method extraction needs to be done on fragments in unrelated types, first should be determined what the best place would be to place the extracted method. After the place is determined, depending on the case, the implications can be numerous. The simplest case is to extract a unifying method from two fragments if the two fragments are contained within the same type declaration. In this thesis, only the simplest case is addressed.

To do a unifying method extraction, first semantically equal fragments of code need to be identified which are suitable to extract. From these fragments a selection has to be made. After this the selected match has to be rewritten.



In section 6.1 is discussed how semantically identical statements are found by matching. In section 6.2 is discussed how code fragments are selected for rewriting. Section 6.3 discusses the rewriting itself.

6.1 Matching

In the matching stage we try to find all code fragments which are semantically equal. As described before only expression substitutions are allowed to match two statements and no transformations. This means that for two matching fragments of code every statement $a_{statement_i}$ matches statement $b_{statement_i}$. This allows for matching statements one by one without looking at the context of the statements. In the matching process it is allowed to substitute expressions to make two statements match. If different expressions substitutions are performed in matching fragments it could happen that the data flow is altered. It is necessary to check if this is the case to guarantee a proposed refactoring will not alter the external behaviour of the program.

We define two sub fragments a' and b' from fragments a and b to match if they are contiguous subsets of a and b, they contain the same number of statements, for every index i statement $a_{statement_i}$ matches statement $b_{statement_i}$ and the data flow with respect to the original fragments is not changed.

Section 6.1.1 discusses what strategy is used to find matching fragments. Section 6.1.2 discusses the matching of two statements or expressions. Section 6.1.3 discusses how is ensured a unifying method extraction won't change the dataflow with a fragment.

6.1.1 Finding matching code fragments

The goal of this stage is to find contiguous sets (runs) of matching statement pairs. As described in section 66.1, only fragments of code completely contained in the body of a member are considered, so to find fragments, only bodies of members are compared.

As discussed in section 6 only code clones are searched in which both matching pairs are contained by the same type. For every type in the analyzed code, code clones are sought by matching the body of each member with each other member within the same type, including the member itself. To find matching

code fragments first all pairs of matching statements are collected. How is determined if two statements match is discussed in section 6.1.2. These matching statement pairs are then combined into contiguous runs.

Although on each separate pair of statements a unifying method extraction can be applied without changing the dataflow and thus the external behaviour of the analyzed program, multiple matching statement pairs combined can change the dataflow. This is because the matching algorithm includes a transformation where some expressions can be substituted with variables. If there are dependencies between extracted expressions and other variables in the match, the dataflow wil change if a unifying method extraction is applied on it. Section 6.1.3 discusses how is determined if such dependencies exist in a group of matching statement pairs.

The contiguous groups of pairs of matching statements which don't change dataflow and fulfil selection criteria discussed in section 6.2 are added to a set. From this set, the best candidates are kept by removing all candidates for which in the set an overlapping match exists which has more lines of code.

6.1.2 Matching statements

As described above, to find matching code fragments, it is necessery to be able to match statements.

The statement matching algorithm is based on the assumption that every expression which results in a value of a certain type can be substituted with a variable of that type containing the resulting value of the expression.

The kind of statement is what the statement actually does. For example an assignment statement is a statement where the result of a source expression is assigned to a member referenced in a target expression; a declaration statement for example declares a local variable and a for-loop statement loops its body. As we do not perform code transformations, the kinds of two statements need to be the same for the statements to match.

So we regard two statements *first* and *second* to match if the kind of statement is the same and for each pair of properties $first_{property_i}$ and $second_{property_i}$, which resolve to types $first_{property_i_type}$ and $second_{property_i_type}$ holds; $first_{property_i_type}$ and $second_{property_i_type}$ are the same or there exists a widening from one of the types to the other, and, if the statements are block statements, for each pair of statements $first_{statement_i}$ and $second_{statement_i}$, $first_{statement_i}$ matches $second_{statement_i}$.

Not every pair of expressions which classify(resolve) to the same result type have to be substituted with a variable; if the expressions resolves to the same declaration, it only needs to be substituted if they resolve to a local variable to match.

Consider the for-loops in the following example again. Each for-loop is a statement on its own; its properties are a control variable declarator, a lower bound expression and a body which contains one statement.

```
1    For a As Integer = 1 To 10
2        c += a
3    Next
```

```
1    For b As Integer = 1 To 100
2        c += b
3    Next
```

First, the control variable declarators a and b they both classify to themselves (as declarators), which means they don't resolve to the same declaration. Both a and b are both of type Integer so they can be substituted by a variable.

The lower bound expression is in both cases '1', they are the same, so they match. The upper bound expression is '10' in the left case and '100' in the right, which is a different value but the expression classifies to the same type, so they can be substituted by a variable. The body of the for loop contains one statement, a compound assignment. The operators of both assignments are the same. The target expression 'c' resolves to a local variable declaration in both cases, which is not the same, but these local variables do classify to the same type and thus they do match. The source expression resolves in both cases to the control variable declaration, so don't resolve to the same declaration, but both do have the same type, so they can be substituted with a variable.

Section 6.1.2.1 describes how two statements are matched in more detail; section 6.1.2.2 describes the matching of expressions .

6.1.2.1 Matching statements- implementation

To find matches, the ASTs and resolving algorithms described in chapter 5 are used. Statements are built up out of keywords, operators, expressions and other statements. Each kind of statement has its own representation in the AST in the form of a different kind of AST node (not to be confused with the type in the type system; hence, we use 'kind' for AST nodes and 'type' for type system nodes). The keywords generally define the kind of statement and are abstracted away in the ASTs. The operators, statements and expressions which are part of the statement are stored in the AST node representing the statement.

Statements have semantics. To find semantically equal code fragments between which the statement kinds differ would mean transformations of the code have to be performed to match one code fragment to the other; as already mentioned, this falls out of the scope of this project and therefore for two statements to match, they have to be of the same kind.

Expressions can be among others, literals, arithmetic operations and member references. They result in a value which has a type, as explained in the classification algorithm. Expressions which are different sometimes can be lifted out of the code fragment by substituting them with variables.

To facilitate the rewriting of matches, information about the match is stored in the algorithm result. In this result, both the matching ASTs are stored together with a third version which is the unified version of both ASTs. Where an expression needs to be substituted to match two statements, the corresponding node in the unified version of the ASTs is replaced by a variable placeholder. Every substitution performed is stored in a list in the match result. Every time sub expressions are matched, the results are combined so the resulting match result contains all the substitutions. How this list of substitutions is used is discussed in section 6.2 and 6.3, this section focuses on the matching itself.

Because only statements of the same kind are considered, we first compare the kinds of the statements. If two statements are of a different kind, the match is rejected; else the match is verified by a specific algorithm for each kind of statement. Section 6.1.2.1.1 discusses how two blocks of statements encapsulated in another statement are matched. In the sections after that each kind of statement is discussed in a subsection as follows:

- Assignment statement 6.1.2.1.2
- Compound assignment statement 6.1.2.1.3
- Call statement 6.1.2.1.4
- Local declaration statement 6.1.2.1.5
- If block statement 6.1.2.1.6
- Else if block statement 6.1.2.1.7
- Else block statement 6.1.2.1.8
- For block statement 6.1.2.1.9
- For Each block statement 6.1.2.1.10
- While block statement 6.1.2.1.11
- Do block statement 6.1.2.1.12
- Return statement 6.1.2.1.13
- Try block statement 6.1.2.1.14
- Using block statement 6.1.2.1.15
- With block statement 6.1.2.1.16
- SyncBlock block statement 6.1.2.1.17
- Select block statement 6.1.2.1.18

6.1.2.1.1 Matching a statement block

In the grammar of Visual Basic, different block statement kinds exist. A block statement is a statement which encapsulates one or more blocks of statements. An example of a block statement is an 'if' statement as shown below.

```
1    If a = 1 Then
2        a = 2
3        c += a
4    Next
```

In the example the 'if' block statement covers lines 1 to 4, it encapsulates a block of statements which is on lines 2 and 3.

For a block statement to match another block statement, it means that besides all the other properties also all the statements encapsulated in the statement must match. As described before, only expression substitutions are used to make statements match, transformations are not performed. This means the number of statements must be the same and each statement must match the statement in the other block at the same position.

The matching algorithm for statement blocks is as follows:

```
if the number of statements in each statement block is not the same
        reject the match
for each pair of statements at the same position in the list of statements
        if the statement dont match
                reject the match
accept the match
```

6.1.2.1.2 Matching assignment statements

An assignment statement is a statement in the form:

Expression = *Expression*

The result of a source expression is stored in a declaration specified by a target expression.

To verify if two assignment statements match, the following algorithm is used:

```
if the two target expressions dont match
        reject the match
if the two source expressions dont match
        reject the match
accept the match
```

6.1.2.1.3 Matching compound assignment statements

A compound assignment statement is a statement in the form:

Expression *CompoundBinaryOperator* *Expression*

Where:

CompoundBinaryOperator ::= ^= | *= | /= | \= | += | -= | &= | <<= | >>=

The binary operator is applied to the result value of the source expression and the result value of the target expression. The result of the operation is stored in the declaration the target expression resolves to.

To verify if two compound assignment statements match, the following algorithm is used:

```
if the operator is not the same in both statements
        reject the match.
if the two target expressions dont match
        reject the match.
if the two source expressions dont match
```

```
        reject the match.
accept the match
```

6.1.2.1.4 Matching call statements

A call statement is a statement in the form:

[`Call`] *Expression* [`(` [*ArgumentList*] `)`]

This is a call to the declaration specified by the expression and the arguments.

Both call statements are resolved as a whole and the result is matched as described in section 6.1.2.2.2. If the match results in a complete substitution the match is rejected. If the resolve result match is rejected, the statement match is rejected.

6.1.2.1.5 Matching local declaration statements

A local declaration statement is a statement in the form:

LocalModifier VariableDeclarators

Where:

LocalModifier ::= `Static` | `Dim` | `Const`

VariableDeclarators ::=
    VariableDeclarator |
    VariableDeclarators `,` *VariableDeclarator*

VariableDeclarator ::=
    VariableIdentifiers [`As` [`New`] *TypeName* [`(` *ArgumentList* `)`]] |
    VariableIdentifier [`As` *TypeName*] [`=` *VariableInitializer*]

VariableIdentifiers ::=
    VariableIdentifier |
    VariableIdentifiers `,` *VariableIdentifier*

VariableIdentifier ::= *Identifier* [*ArrayNameModifier*]

In a local declaration statement one or more local variables are declared and potentially initialized.

A local declaration statement consist of one or more variable declarators which share a local modifier but otherwise are independent of each other. Within a variable declarator one or more variables can be declared which share the same type.

The matching algorithm is implemented in such a way all variables need to be declared in the right order to match. The order in which variables are declared does not influence the semantics of the declaration, although the order of initialization could in theory. However, as described in section 6, we decided not to apply transformationsto make statements match.

To verify if two local declaration statements match, the following algorithm is used:

```
if the number of variable declarations is not the same
        reject the match
for each pair of variable declarations on the same position
        if the types of the declarations dont match
                reject the match
        if the initializers dont match
                reject the match
        if the number of variable identifiers is not the same
                reject the match
        for each pair of identifiers on the same position
                try to substitute them
                if the substitution fails
                        reject the match
        if the number of variable identifiers differs
                reject the match
        for each pair of arguments on the same position
                if the argument expressions dont match
                        reject the match
accept the match
```

6.1.2.1.6 Matching if block statements

An if block statement is a statement of the form:

```
If BooleanExpression [ Then ]
        [ Block ]
        [ ElseIfStatement+ ]
        [ ElseStatement ]
End If
```

If the result of the Boolean expression is true, the statements in the block are executed, if not, statements in the first elseif statement of which the Boolean expression is true are executed. If none of the Boolean expressions where true, the statements in the else statement are executed.

To verify if two if block statements match, the following algorithm is used:

```
if the Boolean expressions dont match
        reject the match
if the statement blocks dont match
        reject the match
if one of the if block statements contain elseif statements and the other doesn't
        reject the match
if both of the if block statements contain elseif statements
        if the elseif statements dont match
                reject the match
if one of the if block statements contains an else block statement and the other doesn't
        reject the match
if both of the if block statements contain an else statement
        if the else statements dont match
```

```
          reject the match
accept the match
```

6.1.2.1.7 Matching elseif block statements

An elseif block statement is a statement of the form:

```
ElseIf BooleanExpression [ Then ]
[ Block ]
```

The elseif block statement is part of the if block statement described in section 6.1.2.1.6. If all Boolean expressions preceding this statement in the if block statement result in false, the Boolean expression of this statement is evaluated, if the result is true the statements in the statement block are executed.

The matching algorithm for two else if block statements is as follows:

```
if the Boolean expressions dont match
          reject the match
if the statement blocks dont match
          reject the match
accept the match
```

6.1.2.1.8 Matching else block statements

An else block statement is a statement of the form:

```
Else
[ Block ]
```

The else block statement is a part of the if block statement described in section 6.1.2.1.6. If all Boolean expressions in the if statement return false, the statements in the statement block are executed.

The verify if two else block statements match, the following algorithm is used:

```
if the statement blocks dont match
          reject the match
accept the match
```

6.1.2.1.9 Matching for block statements

A for block statement is a statement of the form:

```
For LoopControlVariable = Expression To Expression [ Step Expression ]
[ Block ]
Next [ NextExpressionList ]

NextExpressionList ::=
          Expression |
          NextExpressionList , Expression

LoopControlVariable ::=
          Identifier [ ArrayNameModifier ] As TypeName |
          Expression
```

The for block statement is a loop. A control variable is initialized with the result value of the lower bound expressions and increased every iteration with the result value of the step expression, or 1 if no step expression is provided. The loop loops until the control variable reaches the result value of the upper bound expression.

As can be seen in the grammar above, the control variable can be declared in the for block statement itself, but could also just be referred to in an expression and be declared elsewhere. Because only expression substitutions and no transformations are used, two for statements only will match if both control variables are declared in the for block statement or outside the for block statement.

To verify if two for block statements match, the following algorithm is used:

```
if one of the for block statements has its control variable declared in the statement itself and the
other hasn’t
        reject the match
if the control variable is declared in the statement itself in both for block statements
        if the variable declarators dont match
                reject the match
else
        if the expression dont match
                reject the match
if the lower bound expressions dont match
        reject the match
if the upper bound expressions dont match
        reject the match
if one of the for loop statements has a step expression and the other hasn’t
        reject the match
if both of the for loop statements have a step expression
        if the step expressions dont match
                reject the match.
if the contained statements dont match
        reject the match
accept the match
```

6.1.2.1.10 Matching for each block statements

A for each block statements is a statement of the form:

```
For Each LoopControlVariable In Expression
[ Block ]
Next [Expression ]

LoopControlVariable ::=
        Identifier [ ArrayNameModifier ] As TypeName |
        Expression
```

The for each block statement is a loop. The expression must result in a collection. The loop for every item in the collection the loop is iterated one time, with the item assigned to the control variable.

Just as the for block statement the control variable can be declared inside or outside of the for each block statement. Because only substitutions and no transformations are used, two for each statements only will match if both control variables are declared in the for each block statement or outside the for each block statement.

To verify if two for each block statements match, the following algorithm is used:

```
if one of the for each statements has its control variable declared in the statement itself and the
other hasn't
        reject the match
if the control variable is declared in the statement itself in both for each block statements
        If the variable declarators dont match
                reject the match
else
        if the expression dont match
                reject the match
if the collection expressions dont match
                reject the match
if the contained statements dont match
                reject the match
accept the match
```

6.1.2.1.11 Matching while block statements

The while block statement is a statement of the form:

```
While BooleanExpression
[ Block ]
End While
```

The while block statement is a loop. The loop is iterated as long as the Boolean expression results in true.

To verify if two for each block statements match, the following algorithm is used:

```
If the Boolean expressions dont match
        reject the match
if the contained statements dont match
        reject the match
accept the match
```

6.1.2.1.12 Matching do block statements

The do block statement is a statement of the form:

```
DoTopLoopStatement | DoBottomLoopStatement

DoTopLoopStatement ::=
        Do [ WhileOrUntil BooleanExpression ]
        [ Block ]
        Loop

DoBottomLoopStatement ::=
        Do
```

```
        [ Block ]
        Loop WhileOrUntil BooleanExpression
```

To verify if two do block statements match, the following algorithm is used:

```
if not both Boolean expressions are at the top or the bottom of the loop
        reject the match
if the Boolean expressions dont match
        reject the match
if the contained statements dont match
        reject the match
accept the match
```

6.1.2.1.13 Matching return statements

The 'Return' statement is a statement in the form:

```
Return [ Expression ]
```

To verify if two 'Return' statements match, the following algorithm is used:

```
if both return statements have an expression
        if the Boolean expressions dont match
                reject the match
else if one of the statements has an expression
        reject the match
accept the match
```

6.1.2.1.14 Matching try block statements

The 'Try' block statement is a statement in the form:

```
Try
        [ Block ]
        [ CatchStatement+ ]
        [ FinallyStatement ]
End Try

FinallyStatement ::=
        Finally
        [ Block ]

CatchStatement ::=
        Catch [ Identifier As NonArrayTypeName ] [ When BooleanExpression ]
        [ Block ]
```

To verify if two 'Try' block statements match, the following algorithm is used:

```
if the contained statements dont match
        reject the match
for every pair of catch statements
        if the expressions dont match
                reject the match
        if the containing statements dont match
                reject the match
if both 'Try' blocks have 'Finally' blocks
        if the containing statements in the 'Finally' blocks dont match
```

```
                    reject the match.
if one of the 'Try' blocks has a 'Finally' block and the other doesn't
          reject the match
accept the match
```

6.1.2.1.15 Matching using block statements

The 'Using' statement is a statement in the form:

```
Using UsingResources
          [ Block ]
End Using

UsingResources ::= VariableDeclarators | Expression
```

To verify if two 'Using' block statements match, the following algorithm is used:

```
if the contained statements dont match
          reject the match
accept the match
```

6.1.2.1.16 Matching with block statements

The 'With' statement is a statement in the form:

```
With Expression
        [ Block ]
End With
```

To verify if two 'With' block statements match, the following algorithm is used:

```
if the contained statements dont match
          reject the match
accept the match
```

6.1.2.1.17 Matching syncblock block statements

The 'SyncBlock' statement is a statement in the form:

```
SyncLock Expression
         [ Block ]
End SyncLock
```

To verify if two 'SyncLock' block statements match, the following algorithm is used:

```
if the contained statements dont match
          reject the match
accept the match
```

6.1.2.1.18 Matching select block statements

The 'Select' block statement is a statement in the form:

```
Select [ Case ] Expression
          [ CaseStatement+ ]
          [ CaseElseStatement ]
End Select
```

```
CaseStatement ::=
        Case CaseClauses
        [ Block ]

CaseClauses ::=
        CaseClause |
        CaseClauses , CaseClause

CaseClause ::=
        [ Is ] ComparisonOperator Expression |
        Expression [ To Expression ]

ComparisonOperator ::= = | <> | < | > | => | =<

CaseElseStatement ::=
        Case Else
        [ Block ]
```

To verify if two 'Select' block statements match, the following algorithm is used:

```
for each pair of case statements
        for each pair of case clauses
                if the type of the case clause is not the same
                        reject the match
                if the case clauses comparison case clauses
                        if the comparison operators are not the same
                                reject the match
                        if the expressions dont match
                                reject the match
                if the case clauses are range case clauses
                        if the expressions dont match
                                reject the match
                        if the statements in the case statements dont match
                                reject the match
                        if both 'Select' blocks contain a 'Case Else' statement
                                if the statements in the 'Case Else' Block dont match
                                        reject the match
                        if one of the 'Select' blocks contains a 'Case Else' statement
                                reject the match
                        if the contained statements dont match
                                reject the match
accept the match
```

6.1.2.2 Matching expressions

To match two expressions represented by ASTs, first they both are classified as described in chapter 5. This results in a resolve result tree which exactly describes the type and origin of the expression results. This allows for semantic matching.

Regard the following fragment of code:

```
1   Public Class Class1
2       Public a As New Integer
3
4       Public Sub Sub1()
5           Dim class1Instance1 As New Class1
6           Dim class1Instance2 As New Class1
7
8           class1Instance1.a = 1
9           class1Instance2.a = 1
10      End Sub
11  End Class
```

If the target expressions in the assignment statements on line 8 and 9 'class1Instance1.a' and 'class1Instance2.a' are classified. The result for both expressions will be a MemberCallResult which refer to the same variable declaration 'a'. The qualifier of the MemberCallResult will be a LocalVariableReferenceResult, referring to 'classInstance1' and 'classInstance2', which are different but do have the same type. Because 'a' resolves to the same declaration, that part of the expression is a match as is. The qualifiers classify to different declarations which are however of the same type, they are thus as described in section 6.1.2.2.1 substitutable and also a match. Matching of different classify results are discussed in the following sections.

- Member call 6.1.2.2.2
- Instance type 6.1.2.2.3
- Local variable declaration 6.1.2.2.4
- Literal type 6.1.2.2.5
- Type reference 6.1.2.2.6
- Built in binary operation 6.1.2.2.7
- Built in unary operation 6.1.2.2.8
- User defined binary operation 6.1.2.2.9
- User defined unary operation 6.1.2.2.10
- Array index 6.1.2.2.11
- Enumeration member 6.1.2.2.12
- Implicit instance type 6.1.2.2.13

6.1.2.2.1 Expression substituting

As explained in section 6.1.2 two expressions which are different but do resolve to the same result type are still considered a match. This is because the resulting value of both the expressions could be passed using a parameter of that type, which lifts out the different expression from the code fragment.

Consider the following two statements:

```
c = a + 3 * g                    c = a + b
```

Assume that 'c' and 'a' in both statements classify to the same declarations and all variables are of the type integer. The only syntactic and semantic difference

between the statements are the highlighted expressions. If the highlighted expressions were to be replaced with a variable of the same type, they would make both statements exactly the same, as follows.

```
c = a + newVar                    c = a + newVar
```

So when different expressions are substituted with a variable containing the resulting value of those expressions, effectively the difference are lifted out and moved to the place the variable gets assigned its value.

In the matching algorithms described in the following sections it is tried to match as much of the expressions as possible, but if an algorithm fails to match two expressions an attempt is made to substitute the expression. If a substitution succeeds, the expressions are regarded to match anyway, if not the expressions match is rejected.

A substitution succeeds if the resulting types of both expressions are the same, or for one of the types there exists a widening to the other one. A widening is a means to implicitly convert one type to another without any loss of information. This is the case when one of the types is a subtype of the other or if a widening cast operation is declared in one of the types.

6.1.2.2.2 Matching member call expressions

As described in chapter 5 when an expression which refers to a member is classified, a result of type MemberCallResult is returned. It contains a reference to the member declaration, a ResolveResult for the qualifier and a ResolveResult for each argument in the member call.

If the expressions don't classify to the same member, the two expressions are no match. The qualifier describes the type and the origin of the instance of that type the member is called on. If the qualifier cannot be matched, the whole expression cannot be matched. If the referred members match, this implies the qualifiers will match or can be substituted, if the same member is referred in two places, the qualifiers can only be of the same type or one of them a subtype of the other. In both cases this means they can be substituted and the qualifier will match.

If the number of arguments in two member calls differs, this can be caused because of optional parameters or because the last parameter of the member is a parameter array. To match two member calls which have a different number of arguments because of optional parameters would mean that the substitute variables for the optional arguments need to be optional parameters themselves. If the different number of parameters are caused by a parameter array this would mean the extracted method needs to have a parameter array as well. A method only can have one parameter array, so only one list of arguments passed to a parameter array could be substituted. In the implementation is chosen not

to support these kind of substitutions and reject the match if the number of arguments differ. Also the arguments at each position should match.

If the member call expression cannot be matched, it is tried to substitute it completely.

To verify if two MemberCallResults match, the following algorithm is used:

```
if the two members are not the same
        try to substitute the complete expression
if the qualifier doesnt match
        try to substitute the complete expression
if the number of argument is not equal?
        try to substitute the complete expression
for each pair of arguments
        if the arguments dont match
                try to substitute the complete expression
accept the match
```

6.1.2.2.3 Matching instance type expressions

An instance type could be Me or MyBase, which refer to the containing type or its base type. If an instance type expression is resolved as described in chapter 5, this results in an InstanceTypeResult. In this result the type of instance type is stored and also the type the instance type refers to. To match, they both have to be of the same instance type and the type referred has to be the same to. If the expression cannot be matched, it is tried to substitute it completely.

To verify if two InstanceTypeResults match, the following algorithm is used:

```
if the instance types are not the same
        try to substitute the complete expression
If the types are not the same
        try to substitute the complete expression
accept the match
```

6.1.2.2.4 Matching local variable reference expressions

A local variable is a variable which is only available within the scope of a member or a statement. If an expression which refers to a local variable is resolved as described in chapter 5, this results in a LocalVariableReferenceResult. Because we want to eliminate matching code fragments using the method extraction pattern, matching local variable references always need to be substituted with a variable. This is because the code fragment won't be in the same scope as the local variable anymore after the method extraction. Therefore there is no use to check if the two local variables referenced are the same or not, only the type has to match.

To verify if two LocalVariableReferenceResults match, the algorithm will thus be very simple:

```
        try to substitute the complete expression
```

6.1.2.2.5 Matching literal type expressions

A literal type expression is a written number, character or string. Their type is implied by their value. If a literal type expression is resolved as described in chapter 5, this results in a LiteralTypeResult. They only match if their value is exactly the same. If the value is not the same, it is tried to substitute the expression.

To verify if two LiteralTypeResults match, the following algorithm is used:

```
If the value of the expression not the same
        try to substitute the complete expression
accept the match
```

6.1.2.2.6 Matching type reference expressions

A type reference expression is an expression which refers to a type itself. When a type reference expression is resolved as described in chapter 5, this results in a TypeReferenceResult. Because type reference expressions refer to an uninitialized type they cannot be substituted by a variable. If two type reference expressions refer to the same type, they match.

There are situations where the types are not the same but still could be regarded semantically the same. For example if two type references expression are part of qualifiers for member call expressions which resolve to the same member. Than one of the types referred by the type references expressions could be a subtype of the other.

It is however not valid to accept two type references as matching when one of the types referred is a subtype of the other type referred. For example in the case the type reference expression is used in a conditional statement to validate the type of a variable.

Thus the situations in which type references match, besides the case they both resolve to the same type, are context dependant. Although it is possible to look at this context, the situation where two different referenced types could be considered semantically equal is rare, therefore only type reference expressions referring exactly the same type are regarded matching.

To verify if two TypeReferenceResults, the following algorithm is used:

```
if the types are not the same
        reject the match
accept the match
```

6.1.2.2.7 Matching built in binary operator expressions

A binary operator expression is an expression in which an operation is done on two operand expressions. Which operation is defined by the operator type. An operator type can for example be arithmetic like '-', '+' or '*' or logical operators like 'AndAlso' or 'OrElse' but also a string operator as '&' (concatenation) or 'Like'. Built in refers to the fact the operator resolved to a non user defined

operator. Built in operators are clearly defined and are semantically the same if the type is the same so if the operands match, comparing the operator type is enough to check if two built in binary operators match.

Often arithmetic and logical operators are commutative or associative. If an operator is commutative means the order of the operands does not influence the outcome of the expression. For example 1 + 2 has the same result as 2 + 1. If an operator is associative, this means that if in an expression the operator is used multiple times in a row, the order of evaluation does not influence the outcome of the expression. For example 1 + (2 + 3) has the same result as (1 + 2) + 3.

If two binary equal commutative operators are matched, they can match if the left operand of one of them matches the left operand of the other and the right operand matches the right of the other, but they also match if the left operand of one matches the right of the other and the right operand the left operand.

If two binary equal associative operators are matched, which have operator expressions of the same type among their operands, multiple transformations can be tried out to try and match the operators.

As stated before, only substitutions and no transforms are performed to find matches, therefore the matching described above is not implemented. Two binary operators are considered matching if the operator type is equal, the right operands match and the left operands match.

To verify two built in binary operators, the following algorithm is used:

```
if the operator types are not the same
        try to substitute the complete expression
if the left operands dont match
        try to substitute the complete expression
if the right operands dont match?
        try to substitute the complete expression
accept the match
```

6.1.2.2.8 Matching built in unary operator expressions

A unary operator expression is an expression in which an operation is done on one operand expression. As with the binary operator the operator type defines the operation. There only three types of built in unary operators; '-', '+' and 'Not'.

Just as with the binary operators the built in unary operator expressions are operator expressions which resolved to non user defined operators, which are clearly defined and semantically equal when the type is the same.

Thus two built in unary operator expressions match if the operands match and the operator type is the same.

To verify two built in unary operators, the following algorithm is used:

```
if the operator types are not the same
        try to substitute the complete expression
if the operands dont match
        try to substitute the complete expression
accept the match
```

6.1.2.2.9 Matching user defined binary operator expressions

A user defined operator expression is basically the same as the built in binary operator expression. The difference is that the operator resolves to an operator which is declared in the source code. This means it is not clearly specified what the operator exactly does and what side effects occur when it is used. Thus unless the body of the operator declarations is examined, it cannot be guaranteed two operators of the same type which resolve to user defined operators are semantically the same, unless they resolve to the same operator declarator. Examining the body of the operands to determine if they are semantically the same falls out of the scope of this project.

To match two user defined binary operators, not the operator types, but the actual operator declarations to which the operator expression resolves are compared.

To verify two user defined binary operators, the following algorithm is used:

```
if the operators are not the same (note: not only the type of the operators)
        try to substitute the complete expression
if the left operands dont match
        try to substitute the complete expression
if the right operands dont match?
        try to substitute the complete expression
accept the match
```

6.1.2.2.10 Matching user defined unary operator expressions

Just as the user defined binary operator expression is basically the same as the built in binary operator expression, the user defined unary operator is basically the same as the built in unary operator. The user defined unary operator expression resolves to an operator defined in the source code which need to be the same for the unary operator expression to match.

To verify two user defined unary operators, the following algorithm is used:

```
if the operators are not the same (note: not only the type of the operators)
        try to substitute the complete expression
if the operands dont match
        try to substitute the complete expression
accept the match
```

6.1.2.2.11 Matching array index expressions

An array index expression is an expression in which an element in an array is indexed with one or more index expressions. The type of an array consists of an element type and the number of dimensions. For two array index expressions to match, only the type of the array has to be the same, this because the array itself and the index arguments could be substituted.

To verify if two array index expression match, the following algorithm is used:

```
if the arrays dont match
        try to substitute the complete expression
for each pair of arguments
        if the arguments dont match
                try to substitute the complete expression
accept the match
```

6.1.2.2.12 Matching enumeration member expressions

An enumeration member expression is an instance of an enumeration. If the expression resolves to the same member, this means the value and enumeration type is the same and thus they match. If they do not resolve to the same member, they can resolve to another member of the same enumeration or to a member of a different enumeration.

If two enumerations member expressions have different values but are of the same enumeration type, they can be substituted and thus they are a match as well.

An enumeration is in fact a different representation of an intrinsic numerical type, its underlying type. Two different enumerations can have the same underlying type, if this is the case they could be substituted. However, enumerations are used to prevent the use of magic numbers and their enumeration type has a meaning, not necessary for a compiler but it has documenting meaning for the human reader. If two different enumerations with the same underlying type were to be matched and thus in a later stage merged, this would remove the documenting effect. The fact that enumerations can have the same underlying type is thus ignored in the matching algorithm.

To verify if two enumeration member expressions match, the following algorithm is used:

```
if the enumeration members are not the same
        try to substitute the complete expression
accept the match
```

6.1.2.2.13 Implicit instance types

When a member is referenced in the source code without qualifier, the instance or type the member is called upon is implicit. If in two member call expressions the same member is called, they are only semantically the same if the instance of the type where it is called upon is the same. In the case the member is called

from within the type it is a member of, the type is passed as an implicit instance type in the qualifier of the result as an implicit instance type.

The implicit instance type result is basically the same as the normal instance type with the difference that the normal instance type has a type property.

To verify if two implicit instance types match, the following algorithm is used:

```
if the types not the same
        try to substitute the complete expression
accept the match
```

6.1.3 Checking dataflow consistency

As explained in section 6.1.2 two expressions are considered matching if they classify to the same type, because in the unifying method extraction process these expressions can be substituted with a variable of that type. If this variable is a parameter of the extracted method, the result value of the original evaluated expressions can be passed to the method as an argument.

A consequence of this approach is that the expression now is evaluated just before the extracted method is called instead of the moment the expression at the original position was evaluated. If there is a dependency between the expression and a variable or property of which the variable is altered between the original and new evaluation position of the expression this means the data flow is changed, the expression will use another value then it would before refactoring.

Regard the following code fragments.

```
1   Dim c As Integer
2   c = 2
3   c += 1
4   For a As Integer = 1 To 10
5       c += c * 2
6   Next
```

```
1   Dim c As Integer = 0
2   c += 1
3   For b As Integer = 1 To 10
4       c += c + 2
5   Next
6   c += 1
7   c += 1
```

The matching algorithm would regard the highlighted fragments matching. The source expressions in the statement on line 5 in the left fragment and line 4 in the right are different, but are of the same type. The expression therefore can be substituted out and the resulting value can passed as an argument to the extracted method as follows.

```
1   Dim c As Integer
2   c = 2
3   c = NewFunc(c, c * 2)
```

```
1   Dim c As Integer = 0
2   c = NewFunc(c, c + 2)
3   c += 1
4   c += 1
```

```
1   Function NewFunc(ByVal c As Integer, ByVal newVar As
Integer) As Integer
2       c += 1
3       For a As Integer = 1 To 10
4           c += newVar
5       Next
6       Return c
7   End Function
```

It is obvious this refactoring does change the behaviour of the program. The expressions are dependent on the value of variable ‘c’. The value of ‘c’ changes in lines 2 and 4 in the method, which is after ‘NewFunc’ is called and thus after the moved expressions are evaluated, but before the original position of the expressions. ‘c’ changes between the evaluation of the moved expression and the original position of the expression. The original expressions will thus result in a different value than the expressions lifted out; this extraction would thus be invalid.

To guaranty the method extraction does not alter the external behaviour of the program, it must be ensured that when evaluated the lifted out expressions result in the same value as they would before they were lifted out. This means the expressions cannot be dependent on variables or properties which are changed between the time the lifted out expression is evaluated and any time the expression originally would be evaluated. If a match contains any substituted expression which isn’t independent from the rest of the code in fragments, the match must be rejected.

As is discussed in section 6.3.2 equal expressions are substituted by one variable, so equal substituted expressions can be regarded as being the same. To ensure the dataflow of the extracted method stays the same as in the original expressions, the following condition must hold; if a variable gets assigned a value in a fragment, then none of its following occurrences in the fragment can be in an substituted expression or all occurrences after the assignment including the one in the assignment must be substituted as equal expressions.

Regard the matching area of the left fragment from the example above.

```
3   c += 1
4   For a As Integer = 1 To 10
5       c += c * 2
6   Next
```

In the example above in which ‘c’ and ‘c * 2’ are substituted, the substitution ‘c * 2’ in line 5 has a dependency to variable ‘c’. ‘c’ is assigned a value in line 3, the variable is substituted which means that that all expressions containing ‘c’ must be substituted in such a way they are equal to the expression ‘c’. ‘c’ is not equal to ‘c * 2’ so the match must be rejected.

To simplify implementation a more conservative approach is taken. If a value is assigned to a variable or property in one of the fragments, then all instances of variable or property must be substituted by the same variable or none of the instances may be substituted.

6.2 Selection

In 6.1 was discussed how fragments of matching code are found. Now needs to be determined which of the code clones to select for refactoring with a unifying method extraction. Not all code fragments suitable for code refactoring using the method extraction pattern. Section 6.2.1 discusses the general preconditions a fragment of code should meet to be suitable for method extraction.

In this project a method does not need to be extracted from one code fragment but two fragments are to be removed using one unifying method extraction. As explained before, some expressions in the matching fragments are to be substituted with variables which can be parameters or return variables in the extracted method. A method can only have one return value though, section **Error! Reference source not found.** discusses this constrain.

As discussed before in section 6.1.3, lifting out expressions from a fragment using substitution variables and passing the result value of the expressions as an argument may change the order or even the number of times an expression is evaluated. If a substituted expression contains a member call, it is difficult to determine if this changes the outside behaviour of the program. Section 6.2.4 discusses this issue.

How the final code clone to be refactored is chosen is discussed in section 6.2.5.

6.2.1 Refactoring preconditions

In (17) preconditions for a set of refactorings are described which a selection of code needs to meet to guarantee the external behavior of the program is not changed by the refactoring. From this the preconditions regarding the extract method refactoring are compiled in (18) as follows (slightly adapted to suit Visual Basic):

1. The selected code must be a list of statements.
2. Within the selections, there must be no assignments to variables that might be used later in the flow of execution. This can be relaxed to a maximum of one variable because in Visual basic the value of one variable can be returned.
3. Within the selection, there must be no conditional returns. In other words, the code in the selection must either always return, or always flow beginning to end.
4. Within the selection, there must be no branches to code outside of the selection. This means no exit(break), continue or goto statements in Visual Basic unless the selection contains their corresponding targets.

In the matching phase, complete statements are compared and added to a clone, the first precondition is thus always implicitly fulfilled by the manner the code clones are sought.

As explained in section 6.1.2.2.1 all expressions which are different between two fragments and all local variables are substituted. A method can only return one variable, there can thus be at most one assignment to a substituted variable which substitutes a variable that might be used later in the flow of execution in one of the fragments . Section **Error! Reference source not found.** how is determined which variables should be returned so the clones which would require more than one return value and thus don't meet precondition two, are rejected .

If a fragment of code contains a return statement, it means the execution of the current member ends there. If this statement is moved to an extracted method, replacing the fragment of code with a call to this method, the return statement will end the extracted method. To ensure the original fragment still ends at the same point, the call to the extracted method can be made in the form of a return statement. Is the return conditional though, it is impossible to use the extract method pattern. Section 6.2.2 discusses how is ensured the third precondition is met.

The last precondition states there must be no branches in the fragments. To find matching fragments, only statements directly in method bodies are compared. The targets of 'Exit' and 'Continue' statements are loops and other conditional blocks. These blocks are always compared completely. If a break or continue statement is included in a fragment, their target thus is implicitly included as well. Goto statements are ignored in the matching algorithm, they will never be included in a match. The last precondition will thus always be met.

6.2.2 Checking a fragment conditional returns

As discussed in section 6.2.1, if a code fragment returns conditionally, it is impossible to apply the extract method pattern to it. It must thus be ensured that if the fragment contains a return statement, the fragment unconditionally returns, this means all possible paths through the fragment must end in a return statement.

Regard the following code fragment:

```
1    c += 1
2    For a As Integer = 1 To 10
3        c += c * 2
4        If c > 100 Then
5           Return 100
6        End If
7    Next
8    c += 20
9    Return c
```

This fragment could be extracted into the next method:

```
1    Protected Function ExtractedFunction(ByVal c As Integer)
2       c += 1
3       For a As Integer = 1 To 10
4          c += c * 2
5          If c > 100 Then
6             Return 100
7          End If
8       Next
9       c += 20
10      Return c
11   End Function
```

The original fragment then can be replaced with the following call:

```
Return ExtractedFunction(c)
```

In this example the flow of execution is not altered, the call to the method will always return the value the original fragment would before the refactoring.

If line 8 and 9 of the fragment would be left out, it is impossible to refactor it with a method extraction. The method extraction would result in the next method:

```
1    Protected Function ExtractedFunction(ByVal c As Integer)
2       c += 1
3       For a As Integer = 1 To 10
4          c += c * 2
5          If c > 100 Then
6             Return 100
7          End If
8       Next
9       Return c
10   End Function
```

The extracted fragment could now be replaced by one of the following two fragments:

```
1    c = ExtractedFunction(c)
2    c += 20
3    Return c
```

Or:

```
1    Return ExtractedFunction(c)
2    c += 20
3    Return c
```

In case in the value of variable ‘c’ will never pass 100 the first fragment will return the value would be returned before the refactoring, in case ‘c’ does go

past 100 the second replacement fragment returns thevalue as would be returned before the refactoring. But none of the replacements fragment return the value that would be returned before the refactoring in all cases.

To ensure a fragment meets the precondition it may not contain a conditional return, the fragment is analyzed. The fragment passes the check if no return statement is present in the fragment or the fragment unconditionally returns. If a fragment does not pass the check it is rejected.

To analyse a fragment it is assumed that every path through it which is not returned before the last statement of the fragment will pass through the last statement of the fragment. If the last statement in a fragment unconditionally returns, the whole fragment thus will unconditionally return. It is possible the last statement does not unconditionally return while the whole fragment does uncondionally return, this can only happen when the last statement is unreachable. Often compilers reject code which contains unreachable code, in Visual Basic however, it is possible the last statement in a fragment is unreachable. The situation in which unreachable code exists and matches another piece of unreachable code is assumed to be rare. Therefore the possibility the last statement in a fragment does not unconditionally return but still the whole fragment will is ignored. The implications of this decision is that in these rare situations fragments are rejected while they technically should have passed.

A statement is regarded to unconditionally return in the following cases:

- It is itself a 'Return' statement.
- It is an 'If' statement having an else branch of which the body of each branch unconditionally returns.
- It is a 'Try' in which the body of each branch ('Catch', 'Finally') unconditionally returns.
- It is an 'With', 'SyncLock', 'Case' or 'Using' statement of which the body unconditionally returns.

6.2.3 Return values of an extracted method

As discussed in section 6.2.2 fragments of code can contain 'Return' statements. Next to the existing 'Return' statements, often it is necessary to return the value of substituted variables as well. A method however can return a maximum of one value.

If more than one value should be returned by an extracted method, there are a few strategies that come to mind to handle the situation:

- Using out parameters.
- Adding fields to the containing type to pass the values.
- Creata a new type containing fields for all the variables that should be returned and return an instance of this type containing all necessary values.

- Reject the refactoring.

References to local variables can be passed using the 'ByRef' modifier on parameters. This would mean local variables can just be used as they were in the original fragment and no values need to be returned. It is however considered bad practice to use 'ByRef' parameters. It does not encapsulate what the method does from the rest of the program because it would be able to alter the value of a variable from outside its scope.

The second solution, adding fields to the type containing the method to pass values, can be elegant in some situations, but in other situations it would mean adding a field just to have some temporary variables to pass some values from a method to some local variables. This would produce obscure code and is not elegant at all.

If multiple values must be returned by a method, the most elegant solution would be the creation of a new type; a class or a structure, to hold multiple values. An instance of this type, holding the values of all the variables to be returned, could be returned by the method.

In our solution however we have chosen the fourth option, which is to reject the cases in which more than one variable should be returned. If the fragment already contains 'Return' statements returning a value, than no extra values can be returned. Section 6.3.3 discusses how is determined which variables should be returned.

6.2.4 Side effects

As described in section 6.1.3 lifting expressions out of fragments of code and using a parameter to pass the resulting value of the evaluated expression, can change the order of evaluation of expressions. In the example in the section can be even seen that the refactoring performed changes the number of times an expression is evaluated. The substituted expression in the original fragments was evaluated 10 times, after the refactoring they are only evaluated 1 time. Expressions can contain member calls, if the number of times and position expressions are evaluated changes, this also happens whit the member calls within. Often this is not a problem, but it can be. Member calls are not always idempotent and independent. Members could have different results if they are called different times or in a different order.

It is not possible to determine automatically if such side effects exist. All potentially infinite possible paths through the source code should be followed to check if side effects can occur. Because side effects often not occur, it was decided not to reject matches that have potential side effects, but mark the match as a potential risk instead. A match is regarded a potential risk when one or more of the substituted expressions contains a member call. It is left to the user to check if clones with potential risks don't have side effects and can be refactored.

6.2.5 Selecting a code clone to rewrite

At this point there is a list of matching code fragment which meet the preconditions for refactoring by method extraction. The fragments now can all be refactored, but it does not make sense for all of them to be refactored. A lot of the fragments will be very short or a lot of parameters would be needed in the extracted method.

Seven is a magic number for people's comprehension (19), so the maximum number of parameters which is desirable is seven. If the number of parameters exceeds this number, someone who reads the code loses his overview and thus this will make the code less readable. Matches that require more than seven parameters are for this reason rejected.

It is more desirable to remove a code clone if it covers more lines of code. If there are less lines of code in a code clone it means it is less desirable, at some point it becomes undesirable. Empirically the bottom boundary is set to 4 lines of code; code clones with fewer lines are rejected.

The remaining code clones are sorted on the number of lines they cover. The user gets the option to choose a code clone he likes to have rewritten. Section 6.2.4 discusses how some code clones could after refactoring result in change of external behaviour because of side effects. If this is the case the user gets a notification he should check that no external behaviour changes will occur before continuing.

6.3 Rewriting

In the matching phase fragments of code where found which where semantically similar. Semantically different expressions, which result in a value of the same type, were considered matching because they can be substituted with a variable containing the resulting values of the expressions. The original expressions then can be left out of the matching fragments. From these matches a match was selected which can be rewritten without changing the external behaviour of the program.

Now the match can be rewritten; the original matching fragments of code need to be removed from the source code, a new method needs to be constructed and added to the source code. Method calls calling this method need to be placed at the original position of the fragments, having the correct arguments and storing the result in the correct variable. First a method body is created, containing placeholders for all substituted expressions, this is discussed in section 6.3.1. Equal substituted can be taken together to be substituted by one variable, section 6.3.2 discusses how this is done. Some of the substitution variables need to get their values from where the extracted method is called and some of the variables need to be returned, section 6.3.3 discusses how this is determined. To be able to determine this, knowledge about variable dependancies is needed, how this knowledge is gained is discussed in section 6.3.4. After this the final extracted method can be constructed, this is discussed in section 6.3.5. Finally

section 6.3.6 discusses how the original code fragments are replaced with a call to the extracted method.

6.3.1 Creating the method body

To create a method which can replace two fragments of matching code, first the statements from the original fragments are unified to create the body of the method.

As explained in section 6.1.2.2.1 different expressions which result in a value of the same type are considered matching because they can be substituted with a variable of the same type, containing the result value of the original expressions. The expression then can be left out of the matching fragments and the resulting value of the expressions can be passed to the extracted method as arguments.

Next to different expressions also local variables in matching fragments need to be substituted when extracted to a method. The body of the extracted method will be a different scope then the original fragments were in, so the local variables in the original fragment contexts are not directly reachable from the new method body.

Two fragments of code are regarded matching if all statements and expressions are interchangeable or substitutable. If two fragments of code match, the nodes in the AST's representing the code fragments are also interchangeable or substitutable. Thus to create the body for the method the AST's of one of the code fragments can be copied. To make the new AST's applicable for both fragments the nodes which are not interchangeable are substituted with a placeholder. The substitutions are saved together with the expressions they replace in the first and second fragment.

Consider the next two matching fragments.

```
1    c += 1
2    For a As Integer = 1 To 10
3        c += a
4    Next
```

```
1    c += 1
2    For b As Integer = 1 To 100
3        c += b
4    Next
```

The highlighted expressions are not interchangeable, this because they are different expressions as '10' and '100' or a local variable as 'c'. One of the fragments is copied and the expressions which are not interchangeable are replaced with placeholders as follows:

```
1    Placeholder1 += 1
2    For Placeholder2 As Integer = 1 To Placeholder3
3        Placeholder4 += Placeholder5
4    Next
```

The expressions replaced by the placeholders are saved, which is the following list for the example above:

Placeholder	First expression	Second expression
Placeholder1	c	c
Placeholder2	a	b
Placeholder3	10	100
Placeholder4	c	c
Placeholder5	a	b

In section 5.5.2 is discussed how these placeholders are replaced with variables.

6.3.2 Determining variables

In the previous section was shown how the interchangeable part of two matching fragments are copied and the non interchangeable but substitutable nodes are replaced by placeholders. To unite the fragments into a new method, the next step is to determine which substituting variables are needed.

The number of substitution variables don't have to be equal to the number of substitutions made. If an equal expression is substituted multiple times, the same variable should be used to maintain the consistency of the data flow and minimize the number of variables and parameters used. In some cases it would be incorrect to use different substitution variables for equal substitutions. For example the case of a value is assigned to a variable after that its value is retrieved again. If the variable in both instance were to be substituted by two different variables, the value assigned to the first variable would not be retrieved from the second variable, the behaviour of the program thus would be changed.

As described in the previous section, substitutions made in the unified fragments are saved together with expressions they substitute. Placeholders are substituted with the same variable if the expressions substituted by the placeholders within each of the fragments are equal.

It is assumed local variables in the original fragments have useful names. So to increase readability original variable names are preserved as much as possible. If one or both of the expressions substituted by a variable is a local variable, the first name of the replaced local variables which is not yet used for another substitute variable, variable or property is used to name the substitute variable. If no name can be extracted from the original fragments, the name will be 'newVarx' where x is the first number that makes the name unique.

The placeholders from the example in the previous section can be replaced by three variables. One variable for 'Placeholder1' and 'Placeholder4', one for 'Placeholder2' and 'Placeholder5' and one for 'Placeholder3' as follows.

Variable	First expression	Second expression
c	c	c
a	a	b
newVar0	10	100

The method for the body example looks now as follows.

```
1    c += 1
2    For a As Integer = 1 To newVar0
3        c += a
4    Next
```

6.3.3 Determining which variables should be parameters and which should be returned

In the previous section is explained how two fragments are unified. This resulted in ASTs representing the unified code and a list of the variables which are used to substitute the differences between the fragments away. Each substitute variable is associated with the two expressions it replaces. The resulting values of the evaluated expressions can be passed to the extracted method as an argument to substitute the original expressions. If needed the value of the substitution variable can be returned by the extracted method and assigned to the substituted variable.

Not all substitute variables need to be parameters and have the result value of the original expressions passed as an argument. For example if a variable substitutes local variables, it could be the original local variables were declared within the fragments themselves. Because all instances of local variables are substituted, this means the instance within the declaration and thus the declaration itself is also substituted. The substitute variable will thus also be declared within the extracted method and doesn't need to be a parameter.

It could be the value of a substitution variable has to be returned, for example when it substitutes a local variable of which the value is used after the code fragments. It is however not necessary to return all substituted variables, if for example a new value is assigned to the substituted variable before the value assigned in the fragments is retrieved, the value is never used and there is no need to return it.

To determine if a substitution variable should be a parameter or if it should be returned by the extracted method, a difference can be made between variables substituting expressions which are only readable and variables which substitute expressions which are readable and writable. Expressions which are both readable and writable are variables and properties, other expressions are only readable.

Regard the following example.

```
a = 1 + b
```

Here 'a' and 'b' are variables, thus the expressions 'a' and 'b' are both readable and writable. The expression '1 + b' however is only readable, of course the

value of the expression can be changed by changing the value of ‘b’, but it’s not possible to assign a value to the expression as a whole.

Variables which substitute expressions which are not writable always should be parameters. The only reason that non-writable expressions are substituted is because they differ between the two code fragments in the match. For each of the two fragments, the substitute variable thus has a potentially different value which has to be passed as an argument to the extracted method.

The value of variables substituting non-writable expressions don’t change within the extracted method and thus never needs to be returned by the extracted method.

Which of the variables which are both readable and writable need to be parameters and which of these variables need to be returned is depending on the context of the original fragments. As explained above; in the case the substitution variable substitutes a local variable, it doesn’t need to be a parameter if it’s declared within the fragments itself. Furthermore it is only necessary to pass the value of a variable trough a parameter if that value is actually used. If the value of a parameter is changed before it is read, the initial value is discarded and there is no reason why the parameter couldn’t just be a local variable.

The same principle holds for the determination of which variables should be returned by the extracted method. If after the code fragments a substituted variable or property is never used again or is overwritten before it is read, there is no need for the extracted method to return the value of the variable.

If a substituted variable never is read after the code fragments it is obviously not necessary to return the value of the substituting variable. If the substituted variable is a local variable, it is easy to determine if it is used after the fragment, only the containing block of code needs to be examined, this is discussed in section 6.3.4. If the variable is not local, it is very difficult to determine and improbable it will not be used in other code. So if one of the variables substituted is not a local variable, it’s considered necessary to return the value of the substituting variable by the extracted method.

Regarding the example from the previous section the variables ‘c’ and ‘newVar0’ should be parameters, ‘c’ is declared before the matching fragments and its value is retrieved before a new value is assigned to it within the fragments and ‘newVar0’ is substituting an expression which is only readable. ‘a’ doesn’t need to be a parameter because it is declared within the method itself. If one assumes ‘c’ is read again after the fragments before it is overwritten, it needs to be a return value.

The list of variables now looks as follows.

Variable	First	Second	Is	Needs to be

	expression	expression	parameter	returned
c	c	c	X	X
a	a	b		
newVar0	10	100	X	

Thus a substitution variable thus is considered not to be a parameter if one of the following conditions hold:

- Both of the variables are declared within the extracted method.
- The variable is changed before it is read in all code paths possible within the extracted method.

A substitution variable doesn't need to be returned if one of the following conditions hold:

- One of the expressions substituted is not a variable or property.
- The variables substitute local variables which both are altered after the code fragments before they are read in all code paths possible.
- The variables substitute local variables which are not read after the code fragments.

6.3.4 Determining variable dependencies

As discussed above, to determine if a variable needs to be a parameter in an extracted method or if it needs to be returned, it is necessary to determine if the value in the variable is used again. The value of a variable is not used again when the following condition is true; the variable is not referred after the fragment or a new value is assigned to the variable before the value of the variable is retrieved in all paths possible through the code.

To determine if this condition is true, the code must be examined. If at the first occurrence of the variable a value is assigned to it, the condition is true, if its value is retrieved, the condition is false.

There are a few kinds of statements in which a value is assigned to a variable after it is declared. A value is assigned to a variable when it is the target expression of an assignment statement, in a compound assignment statement or in a mid assignment statement. A value is assigned to a variable when it is the loop control variable in a for loop or an for each loop. In an erase statement the value 'Nothing' is to an array. In the 'ReDim' statement an array is re-declared. If a variable is passed by reference to a member, the variable can be accessed within that member. In any other expression the variable occurs its value is retrieved or not accessed at all.

6.3.4.1 Variables in assignment statements

One of the kinds of statements in which a value is assigned to a variable is the assignment statement. In an assignment statement the resulting value of a source expression is assigned to a variable or property classified by the target

expression. If the first occurrence of a variable is in an assignment statement. The condition is true if the variable occurs in the target expression and not in the source expression. If the variable occurs in the source expression, it will be first retrieved. Only after the source expression is evaluated its result value is assigned to the variable classified in the target expression.

6.3.4.2 Variables in compound assignment and mid assignment statements

Just as in the regular assignment statement, a value is assigned to a variable classified by its target expression in a compound assignment and a mid assignment statement.

A compound assignment is a different way of writing a binary operation with as operands the variable classified by the target expression and a source expression, the result is stored in the variable classified by the target expression.

The mid assignment statement assigns a string into another string. The target expression classifies a variable of the type string. At the position specified by arguments a string specified by the source expression is inserted in the string.

In these two types of assignment statements, a value is thus assigned to the variable classified by the target expression after it is retrieved. Is the first occurrence of a variable in the target expression, it is thus first retrieved. No variable is altered in the source expression or other arguments in both statements.

Thus if the first instance of a variable occurs in a compound assignment or a mid assignment statement, the condition is false.

6.3.4.3 Variables as control variable in a *for* loop or *for each* loop

In the for loop and for each loop statement a block of code is iterated, in every iteration a value is assigned to the loop control variable. In the for loop the value assigned is specified by bounds and a step expression, in the for each loop the value is an item from a specified list.

The expressions specifying the bounds and list are evaluated at the beginning of the loop before the control variable gets assigned its first value.

If the first occurrence of a variable is as a control variable in a for loop or for each loop, the condition thus is true, but only when the variable does not occur in the expressions specifying the bounds or the input list.

6.3.4.4 Variables in an erase statement

The erase statement resets the value of an array typed variable. The erase statement is equivalent to assigning the value 'Nothing' to the variable. If the first occurrence of a variable is in an erase statement, a value is thus first assigned to it and the condition is true.

6.3.4.5 Variables in a redim statement

In a 'ReDim' statement, an array typed variable is re-declared. The 'ReDim' statement is equivalent to assigning a new instance of an array to the variable. In the 'ReDim' statement the preserve keyword can be specified, if this is the case, the values of the original array are copied to the new array. So if the preserve keyword is specified the value of the variable is retrieved before a new variable is assigned to the variable and the condition is false. If the preserve keyword is not specified, the first occurrence of a variable is as the target in the 'ReDim' statement and the variable does not occur in the initializing expression, a value is thus assigned to the variable before it is retrieved and condition is true.

6.3.4.6 Variables passed as argument

A variable could be passed by reference as an argument to a member, if this is the case, the body of the member determines if a value first is assigned to the variable or retrieved from it. One could state that a value is probably assigned to a variable passed as reference because else there would not be a reason to pass it by reference. It is however considered bad practice to pass a variable by reference to a member, it is therefore assumed not to occur often and treated in the same way as an argument passed as a value, i.e. the value is considered to be retrieved before a value is assigned to the variable.

6.3.4.7 Variables in other expressions

If the first occurrence of a variable is in an expression not described above, it is only retrieved and the condition is thus false. This means the variable needs to be a parameter or returned depending on which code is analyzed.

6.3.4.8 Conditional code blocks

The order in which statements are evaluated is not always straightforward, there are a couple of constructions which make the evaluation of blocks of code conditional. If a block of code is evaluated or not could mean the difference between if first the value of a variable is retrieved or a value is assigned to the variable. For example the bodies of loops are evaluated n times where n could be 0, the body of an if statement is only evaluated if its expression results in true. Because of these constructions different paths through the code are possible.

In most of the cases it is not known exactly which conditional blocks will be evaluated until runtime, so for every conditional block of code the condition must hold with and without the block evaluated. This means the condition also must hold without any of the blocks evaluated, so the condition thus can be said to be true if at the first occurrence of the variable outside any block the variable gets assigned a value and within every block before this occurrence the condition is true.

So an retrieval of the value of the variable before an assignment within a block of code will render the condition false and an assignment of a value to a variable will only render the condition true within the block.

6.3.5 Creating the unifying extracted method

In the previous sections is discussed how a body is created for an extracted method and how was decided which substitute variables need to be parameters of the method or need to be returned. Now a complete method can be created of it.

Visual Basic knows two types of methods, 'Sub' and 'Function'. The difference between those is that 'Function' returns a value and 'Sub' doesn't. In section 5.4.1 is shown how matches which require more than one return variables were rejected, so all matches now have one or zero return variables. If a return value is required or the fragments already contain return statements, a 'Function' is created else a 'Sub'.

If the method is a 'Function' and one of the substituted variable is marked as return variable, the return type of the method is set to the same type as that variable. If the method is a 'Function' and no substituted variable is marked as return variable, this means the fragment already must contain return statements, the return type of the method then is determined by taking the least specific type the expression in all return statements in the fragments classify too.

In section 6.3.3 was shown that not all substitution variables need to be parameters, sometimes this is because of the substitution process the declaration itself was substituted as well and thus is included in the method body and sometimes this is because the values in the variables are not used within the fragments and thus not need to be passed. Declarations for variables which are not parameters nor are declared in another way in the fragments are added before the earlier created method body within the created method.

In the case a variable needs to be returned then if the method body does not already contains one or more return statements, a return statement is added at the at the end of the method body. If the method body already contains return statements, but they don't return any value, then all return statements are replaced with return statements returning the variable.

For every substitute variable indicated as a parameter, a parameter is added to the method signature.

The method is created by creating an instance of the appropriate AST node and adding the correct children for it. Only code duplications within types themselves are sought, this means the extracted method can be added to this type. This is easily done by adding the AST node representing the method to the list of declarations within the AST representation of the type.

Regard the example in section 5.5.3, the list of parameters is as follows.

Variable	First expression	Second expression	Is parameter	Needs to be returned
c	c	c	X	X

a	a	b		
newVar0	10	100	X	

The list contains one variable which needs to be returned 'c', thus the method created is a 'Function', the return type is that of the returned variable 'c' which is 'Integer'. After this the two variables marked as parameter 'c' and 'newVar0' are added to the signature. Because the method is a 'Function' a return statement is added after the method body returning the variable marked as return variable. The result looks as follows.

```
1    Function NewFunc(ByVal c As Integer, ByVal newVar0 As
Integer) As Integer
2      c += 1
3      For a As Integer = 1 To newVar0
4        c += a
5      Next
6      Return c
7    End Function
```

6.3.6 Replacing the code fragments with a call to the extracted method

In the previous section was explained how an extracted method is added to the AST. Now a replacement method is available, the matching code fragments can be replaced by calls to this method.

The first step is to create the call statements. In section 6.3.3 is discussed how was determined which substitute variables should be parameters and how with every parameter the two substituted expressions are saved. These expressions can now be used as the arguments in the method call.

In section 6.3.3 is explained variables which are declared within the fragment, don't have to be a parameter. If such a declaration is extracted, this means the variable does not exist in the original scope anymore. If however the variable is used after the method, a declaration for the variable needs to be added to the replacing code of the original fragment. Is the value of this variable used after the fragment, then the return value of the extracted method must be assigned to this variable.

Regard the next two fragments:

```
1    Dim a As Integer
2    For a = 1 To 10
3        SomeProperty += a
4    Next
5    a = SomeProperty
```

```
1    Dim b As Integer
2    For b = 1 To 10
3        SomeProperty += b
4    Next
5    SomeProperty = b
```

The first four lines match and can be unified to the following method:

```
1    Protected Function ExtractedFunction() As Integer
2       Dim a As Integer
3       For a = 1 To 10
4          SomeProperty += a
5       Next
6       Return a
7    End Function
```

Because the declaration of the variables ‘a’ and ‘b’ are extracted, they are no longer available in the original scope when the fragments are replaced with a call to the new method, the declaration thus has to be added as follows:

```
1    Dim a As Integer = ExtractedFunction()
2    a = SomeProperty
```

And:

```
1    Dim b As Integer = ExtractedFunction()
2    SomeProperty = b
```

It is possible the original fragments contain return statements. In section 6.2.2 is explained how is ensured return statements in the fragments are only accepted if the fragment unconditionally returns. If the original fragments unconditionally return, it means the method the fragments originally reside in, should after the refactoring still return at this position. A new return statement thus needs to be inserted.

If the expressions in the return statements in the fragments are not substituted, or they originally returned local variables, the result of a call to the extracted method can directly be returned.

Regard the following example:

```
1    Dim a As Integer
2    a = 10
3    Return a
```

```
1    Dim b As Integer
2    b = 10
3    Return b
```

These fragments can be extracted to the following method:

```
1    Protected Function ExtractedFunction() As Integer
2       Dim a As Integer
3       a = 10
4       Return a
5    End Function
```

Here the original expressions in the return statements ‘a’ and ‘b’ are local variables, the extracted method call can thus directly returned, both fragments can be replaced with the following statement:

```
1    Return ExtractedFunction()
```

Is one of the substituted expression in the return statements a non local variable or a property, the value first needs to be saved in the original variable or

property before it can be returned. This has to be done to ensure the state of the program will be the same as it would be before the refactoring.

Regard the following expression:

```
1    SomeProperty = 10
2    Return SomeProperty
```

```
1    SomeOtherProperty = 10
2    Return SomeOtherProperty
```

In both fragments 10 is assigned to a property, after this the property is returned. These fragments would be too short to refactor but are used for explanatory reasons, the extracted method for these fragments would be as follows:

```
1    Protected Function ExtractedFunction(ByVal SomeProperty As
Integer) As Integer
2        SomeProperty = 10
3        Return SomeProperty
4    End Function
```

Note that in the extracted method 'SomeProperty' is a substitution variable. In the execution of the original fragments properties 'SomeProperty' and 'SomeOtherProperty' get assigned the value 10, after that their value is returned. To preserve the external behaviour of the fragment after refactoring the result value of the extracted method must be assigned to the properties. Because in the original fragments the containing method returned, the replacement should return as well. The replacement call is thus formed as follows:

```
1    SomeProperty = ExtractedFunction(SomeProperty)
2    Return SomeProperty
```

And:

```
1    SomeOtherProperty = ExtractedFunction(SomeOtherProperty)
2    Return SomeOtherProperty
```

If the original fragments did not contain a return statement, and the extracted function does return a value, it means this value should be assigned to the appropriate variable or property. In section 6.3.4 is shown that substitute variables which are returned by the extracted method always substitute an expression which classifies to a variable or a property. This means the substituted expression replaced by the substitute variable which is returned can be used as the target expression of an assignment statement having the method call as a source expression.

For the example shown in section 6.3.5 the method is a 'Function', an assignment statement is thus created. The target expressions are the substituted expressions substituted by the returned variable, this expression is in both cases 'c'. The arguments for the method call in the source expression are the substituted expressions which are substituted by the parameters of the extracted

method. This is ‘c’ for as the first argument in both cases and ‘10’ and ‘100’ as the second argument. The statements will now be as follows.

```
c = NewFunc(c, 10)                          c = NewFunc(c, 100)
```

After the replacement statements are created, statements in the fragments can be removed from the lists of statements in the AST containing the fragments, after this the new statements can be inserted.

6.4 Summary

In this chapter was shown how matching statements are found by comparing the kind of statement and the resulting type of expressions. An expression is considered a match if the resulting type is the same because this means a substitution variable of that type can be used to hold the resulting value of the expression and leave the different expressions out of the matching code.

Runs of matching statements are combined to a matching code clone to be removed using the extract method refactoring. Because the substitutions in the statements are not always compatible with each other, it can be that some code clone matches cannot validly be removed by method extraction, such code clones are rejected. Because methods can only return one value, method extractions which require more than one return value are rejected as well.

Because seven parameters is the maximum before a method becomes less readable, code clones which require more than seven parameters are rejected. If there are less than four lines of code in a match, the extraction of a method to remove the code clone is regarded undesirable; code clones less the four lines of code big are therefore rejected.

Because the substitution of expressions can result in that member calls are evaluated in a different order, theoretically side effects can occur when this happens. If there is change side effects occur the user is notified.

The code clones are ordered on size and the user can choose a code clone to remove. For the selected code clone first a method body is constructed by copying the AST of one of the matching fragments and substituting the different expressions with placeholders. The placeholders are then substituted with substitution variables in such a way that all placeholders that substitute equal expressions are replaced by the same substitution variable.

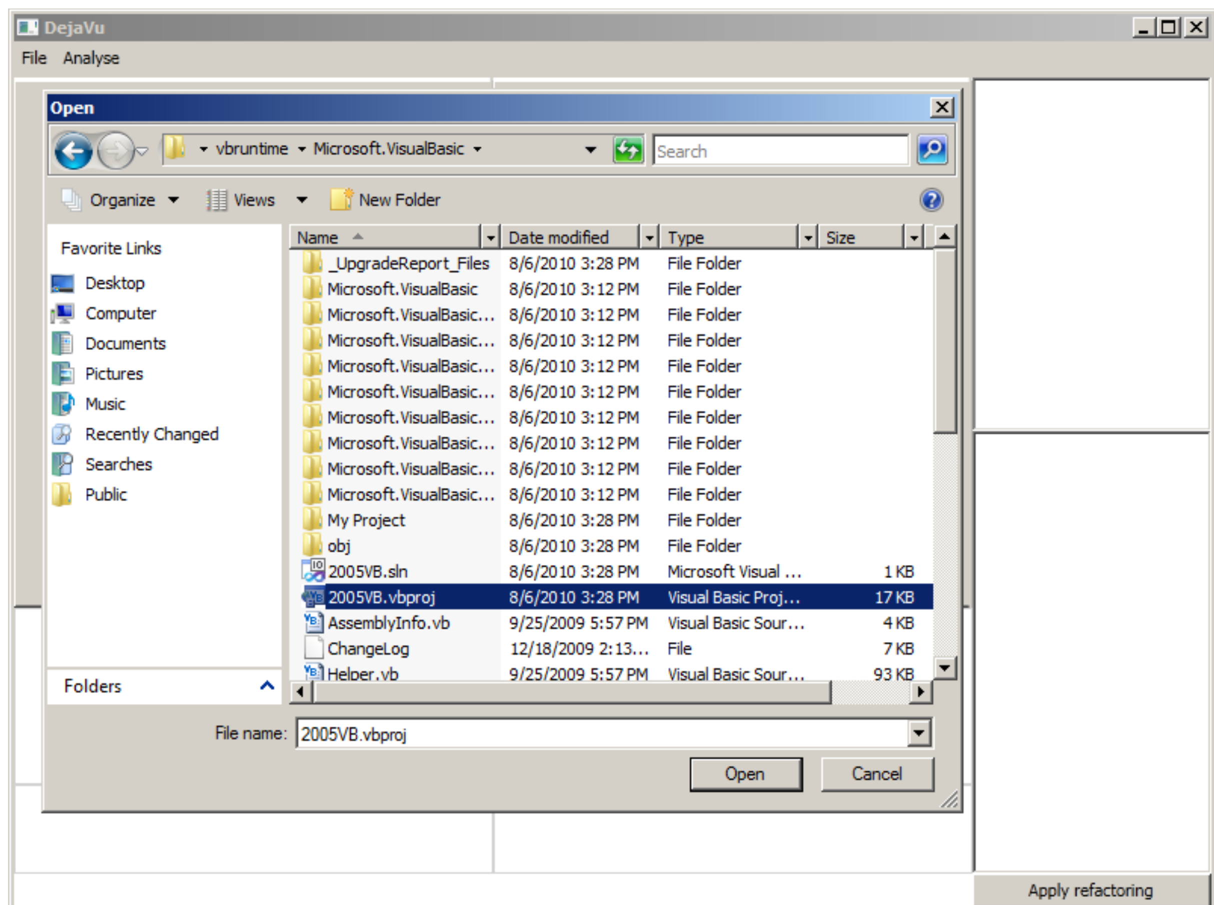
For every variable is determined if it needs to be a parameter or a return variable. A variable needs to be a parameter if it is not declared within the method body and the value of the variable is retrieved before assigned. A variable needs to be returned if it substitutes a variable or property, except the substituted variable is local and used anymore after the method.

7 Application

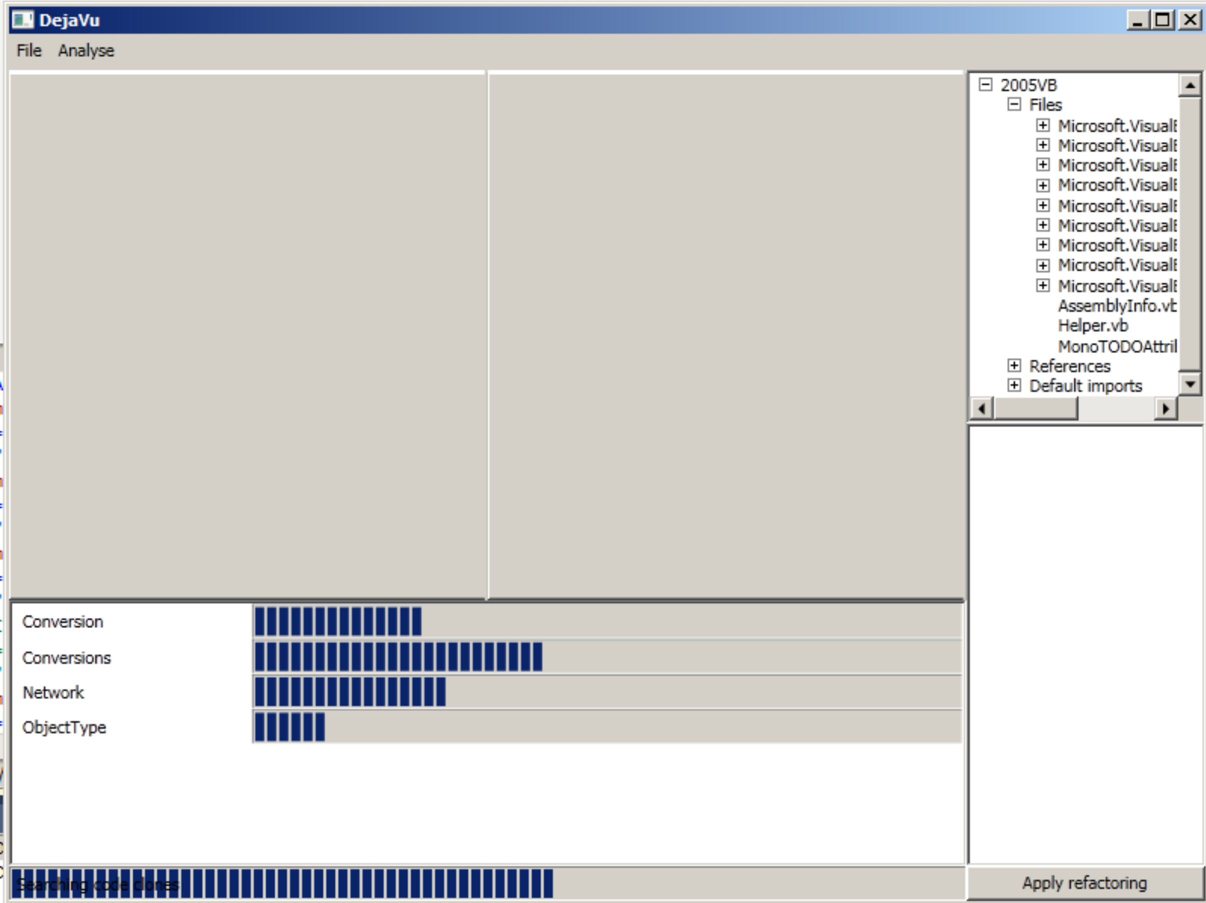
In the previous chapters was explained how a project was analyzed and after that code clones were detected so that they can be automatically removed by applying a unifying method extraction. All this functionality was implemented in the tool *DejaVu* presented in this chapter. Section 7.1 present the tool built. In section 7.2 two case stusies are presented to show the practical use of the tool.

7.1 Show case

To start analyzing a project, a project file can opened as shown below:

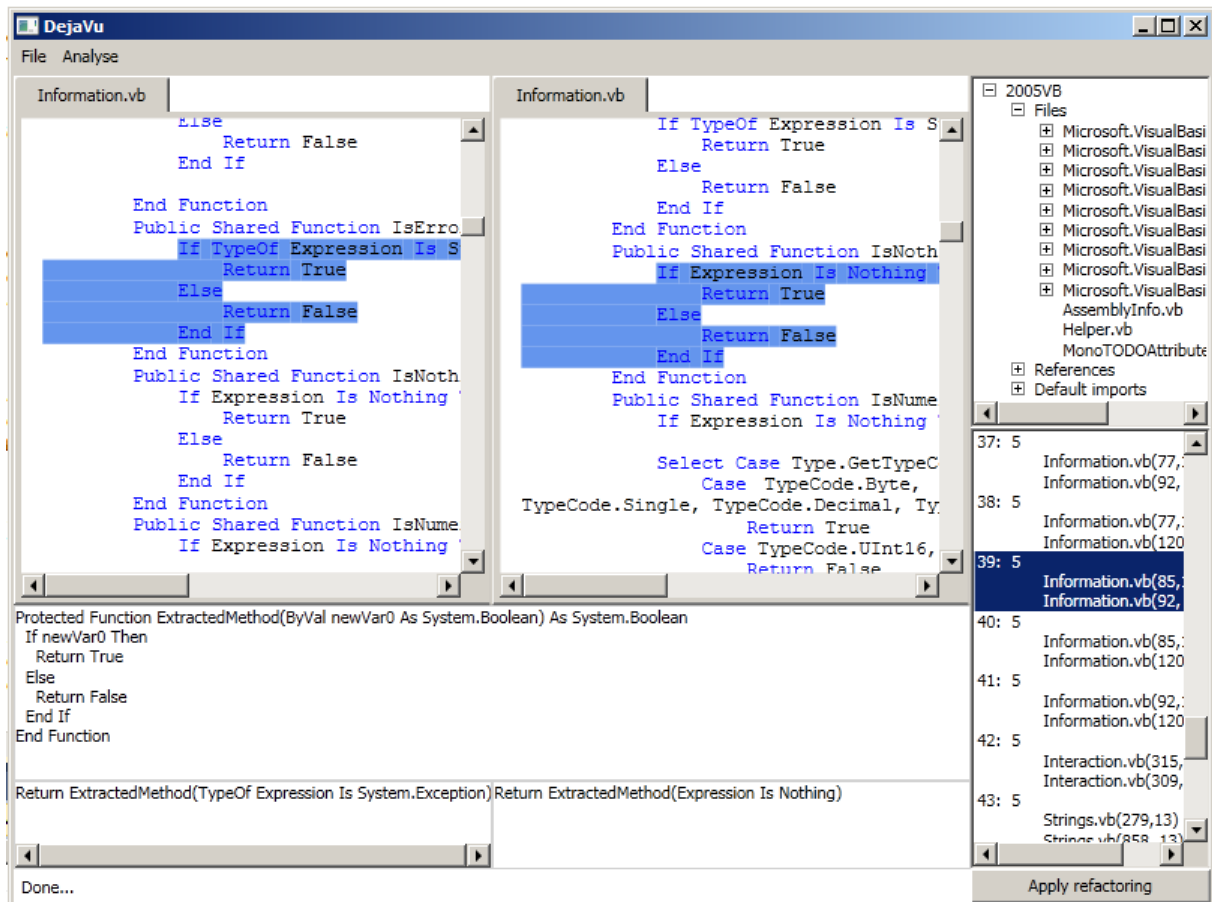


Opening a project file in the tool triggers the complete code analysis described in section 4. After the code analysis is complete, all loaded projects and their contained source code files are displayed in a tree in the right top pane. Now code clone detection can be done on all these projects, one of the projects or on one file. A file or project can be selected and with 'Find all code clones', 'Find code clones in selected project', or 'Find code clones in selected file' in the 'Analyse' menu can be selected to start the detection as described in chapter 6.1, where necessary expressions are classified using the process in section 5. Results from the classification process are buffered so they do not have to be reclassified every time the expression needs to be compared to another expression. The code clone detection process is very time consuming, so to utilize the multiple cores most computers nowdays have, for each type a new thread is started so multiple instances of the algorithm run parallel. The progress of each currently running thread is shown in a list of progress bars below that the total progress of the batch, as can be seen below:

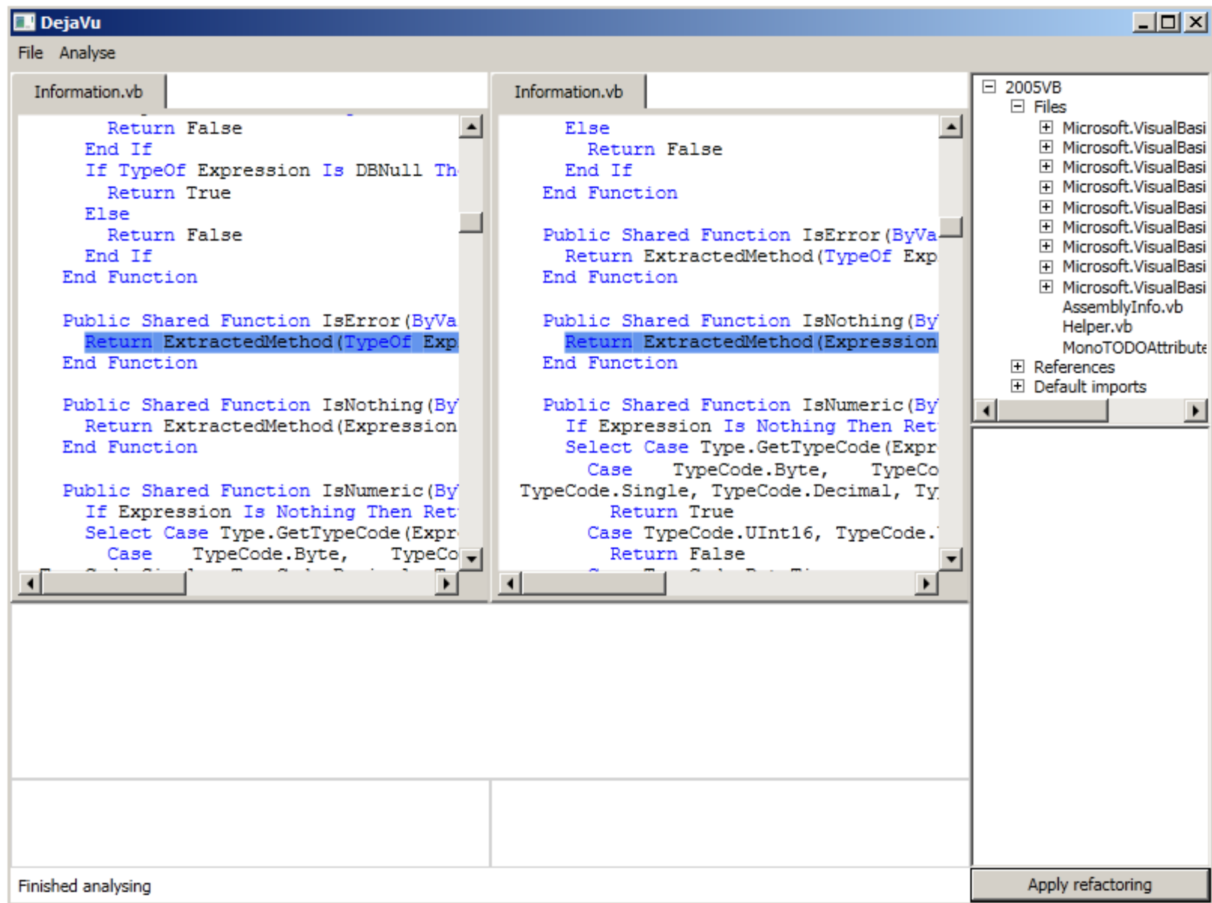


When the code clone detection process is finished, the found code clones which meet the preconditions discussed in section 6.2 are displayed in the right bottom pane ordered by size. In the list, from each match the length and the two files containing the matching fragments and the start position of the fragments in shown.

If a match is selected, for each of the fragments, the containing file is opened next to eachother and the fragments are highlighted. Below that, the unifying method extraction refactoring proposed is shown. This proposition consists of an extracted method and two method calls which should replace the highlighted fragments. The tool with a selected match is shown below:



If the user agrees with the proposed refactoring, the button 'Apply refactoring' can be clicked to apply the refactoring, shown below:



7.2 Case Study

To show the practical use of the tool, we run it on an approximal 100.000 LOC project known to contain a lot of copy-pasted code. The tool found about 500 code clones with an average length of 5 lines.

One example of the code duplicates found by the tool are the following two fragments:

```
1    readr.Close()
2
3    ProgressFormMax = colItems.Count
4    ProgressFormValue = 0
5
6    For Each objItem As ABControls.ListRow In colItems
7       ProgressFormValue = ProgressFormValue + 1
8       Dim personId As Integer =
           GetPersonIdByAzr(Convert.ToString(objItem.Col03),
           Convert.ToString(objItem.Col02))
9       objItem.Col02 = MCon.GetEntityName(EntityType.eClient,
           personId)
10      objItem.Tag = objItem.Tag & "<personID " & personId & "/>"
11
12      commnd = GetDBCommandQuery( _
13          "SELECT DAGTEKENING_VERZENDING " & _
14          "FROM AZR_AW36_01 " & _
15          "WHERE tabelID = " & CInt(objItem.Col04), _
16          conn)
17      readr = commnd.ExecuteReader()
18      If readr.Read Then
19         objItem.Col03 =
          DateNumToDateString(CInt(readr("DAGTEKENING_VERZENDING")))
20      End If
21      readr.Close()
22   Next
23   conn.Close()
```

```
1    readr.Close()
2
3    ProgressFormMax = colItems.Count
4    ProgressFormValue = 0
5
6    For Each objItem As ABControls.ListRow In colItems
7       ProgressFormValue = ProgressFormValue + 1
8       Dim personId As Integer =
           GetPersonIdByAzr(Convert.ToString(objItem.Col03),
           Convert.ToString(objItem.Col02))
9       objItem.Col02 = MCon.GetEntityName(EntityType.eClient,
           personId)
10      objItem.Tag = objItem.Tag & "<personID " & personId & "/>"
11
12      commnd = GetDBCommandQuery( _
13          "SELECT DAGTEKENING_VERZENDING " & _
14          "FROM AZR_AW310_01 " & _
15          "WHERE tabelID = " & CInt(objItem.Col04), _
16          conn)
17      readr = commnd.ExecuteReader()
18      If readr.Read Then
19         objItem.Col03 =
          DateNumToDateString(CInt(readr("DAGTEKENING_VERZENDING")))
20      End If
21      readr.Close()
22    Next
23    conn.Close()
```

It is obvious these two fragments are copy-pasted. The only difference is the part of the SQL query in which the name of the table is specified. When we proceed by automatically refactoring the code duplicate we get the following extracted method.

```
1    Protected Sub ExtractedSub(ByVal readr As DbDataReader, ByVal
colItems As Collection, ByVal newVar0 As String, ByVal conn As
DbConnection)
2       Dim commnd As DbCommand
3       readr.Close()
4
5       ProgressFormMax = colItems.Count
6       ProgressFormValue = 0
7
8       For Each objItem As ABControls.ListRow In colItems
9           ProgressFormValue = ProgressFormValue + 1
10          Dim personId As Integer =
              GetPersonIdByAzr(Convert.ToString(objItem.Col03),
              Convert.ToString(objItem.Col02))
11          objItem.Col02 =
```

```
DateNumToDateString(CInt(readr("DAGTEKENING_VERZENDING"))),
12          objItem.Tag = objItem.Tag & "<personID " & personId & "/>"
13
14          commnd = GetDBCommandQuery("SELECT DAGTEKENING_VERZENDING
" & newVar0 & "WHERE tabelID = " & CInt(objItem.Col04), conn)
15          readr = commnd.ExecuteReader()
16          If readr.Read Then
17             objItem.Col03 =
         DateNumToDateString(CInt(readr("DAGTEKENING_VERZENDING")))
18          End If
19          readr.Close()
20       Next
21       conn.Close()
22    End Sub
```

The two fragments are replaced by a call to this method with the following statements.

```
ExtractedSub(readr, colItems, “FROM AZR_AW36_01”, conn)

ExtractedSub(readr, colItems, “FROM AZR_AW310_01”, conn)
```

The novelty here is not the detection of the code duplicate, even a text based code clone detection algorithm would have found this duplicate. Method extraction is not new either. But to our knowledge this tool is the first to combine the two and perform a unifying method extraction on a detected code clone without the need for the user to check the semantic invariance of the refactoring operation. Also the fact that the method extraction is not done on one piece of code, but on two pieces combined, crafting it in such a way it is suitable to replace both fragments at the same time is novel, although (12) describes a (different) method to do so, it was only partially implemented. Also the method describes in (12) only allowes for exact matching statements although the order may differ, the implications of such a method extraction do not differ with a “normal” method extraction. A “normal” method extraction would not have to handle the differences between code fragments like in the example the name of the table.

In a second case the tool was applied to the source of the open source project mono (20) a project which consists of 20.000 LOC. Fifthy matches where found with an average length of 9 lines.

One of the matching fragment pairs found in the mono source code is the following:

```
1    If CBool(Me.TraceOutputOptions And TraceOptions.DateTime) Then
2       builder.Append(m_Delimiter)
3       builder.Append(eventCache.DateTime.ToString("u",
           System.Globalization.CultureInfo.InvariantCulture))
4    End If

1    If CBool(Me.TraceOutputOptions And TraceOptions.Callstack) Then
2       builder.Append(m_Delimiter)
```

```
3       builder.Append(eventCache.Callstack)
4    End If
```

This is a typical example of a near code clone none of the methods discussed in section 3 would find. The difference between the two fragments is the different expressions in line 3:

```
eventCache.DateTime.ToString("u",
           System.Globalization.CultureInfo.InvariantCulture)
```

And:

```
eventCache.Callstack
```

These expressions differ in such a way other methods, which only abstract away variable names and literal values, would not find them. Both expressions result into a value of the type string, because of this they can be substituted by a variable and the value of the evaluated expressions can be passed to the unifying extracted method as an argument. The expressions do however contain member calls, the match was thus marked as a potential risk for the semantic invariance when the refactoring as applied. Reviewing shows the refactoring in this case is safe.

When proceeding with automatic refactoring, the following method is extracted:

```
1    Protected Sub ExtractionSub(ByVal builder As StringBuilder,
ByVal newVar0 As String)
2       If CBool(Me.TraceOutputOptions And TraceOptions.Callstack)
           Then
2          builder.Append(m_Delimiter)
3          builder.Append(newVar0)
4       End If
5    End Sub
```

The original fragments are replaced with the following method calls:

```
ExtractedSub(builder, eventCache.DateTime.ToString("u",
           System.Globalization.CultureInfo.InvariantCulture))
```

And:

```
ExtractedSub(builder, eventCache.Callstack)
```

8 Discussion

The case studies show the tool has potential for practical use; it finds interesting code clones and is able to refactor the clones away. A big advantage of the approach used in this thesis in respect to other approaches is that code clones found by this approach always can be refactored using the method extraction pattern. The tool finds code clones which can be automatically refactored without the user's need of checking the consistency of the refactoring.

The performance of the tool though, makes the tool unpractical to use as is in a real life application. There are different approaches to make the tool more usable. In section 8.1 discusses what could be done to increase the usability of the tool. Although it is not deductable from code clones if they originate from copy-pasting, the code clones found by the tool give the impression they originated by copy-pasting, for the same reasons it is useful to remove code clones, it is useful to find and remove code fragments which is semantically similar but did not originate from copy-pasting, section 8.2 discusses what could be done to find more of these cases. The tool supports the detection and removal of pairs of code clones, more than two instances of clones can exist though, section 8.3 discusses how these could be handled.

8.1 Increasing usability

As can be seen in chapter **Error! Reference source not found.** the performance of the tool is not really practical. To really be of any practical dynamic use, it would be preferable the performance of the tool would be increased considerable. One way of increasing performance is to use faster but less reliable methods to pre select potential clones. Code clone detection using lexicographic information does not find code clones which guaranteed to be semantically the same, but will find code clones much faster. In Aries **Error! Reference source not found.** lexicographic information is used to find code clones, syntactical information then is used to suggest which code clones are appropriate to remove by using method extraction. A similar approach could be applied to this project; a lexicographic or other hash based approach could be used to find code clones or fragments of code which suggest they could be semantically the same; the match detection currently implemented could be applied to these suggested fragments instead of all the code. This way performance should go up dramatically, but it still would be possible to automatically remove the code clones with the use of method extraction.

One of the differences with other code clone detections is that because of the semantic analysis the algorithm can identify matching statements containing different expressions of resulting in the same type, therefore code clones which would not be found by other code clone detection algorithms. These faster algorithms abstract away variable names and literal values, but generally these are the only differences between fragments which are allowed in a match. Using these algorithms to preselect code clone candidates would thus mean losing a part of the code clones found.

The algorithm is based on matching continuous runs of statements where pairs of statements are the same kind. Because of this, in practice the code clones

found will be mostly results of copy-pasting. It could be researched if a faster algorithm could be found which finds matches as the algorithm in the thesis does or it could be decided that the performance increase gained by using a faster algorithm justifies the loss of a few code clones found.

A different approach to make the tool more usable is to change the intendend usage. Instead of searching code clones in a complete project, the tool could be rewritten in such a way the user could select a fragment of code it intents to change so the tool can search for code clones for just that fragment. Instead of comparing all statements with all other statements, only the statements in the selected code would have to be compared with other code thus decreasing the time the tool needs to perform its task.

8.2 Increasing the number of code clones found

As already mentioned in section 8.1 most code clones found by the algorithm seem to originate from copy-pasting. Fragments of code which are incidental semantically similar generally differ too much to be found by the algorithm presented in this thesis because at the matching and rewriting phase, only substitutions are applied and no transformations.

It is just as useful to remove code clones which arise by coincidence as it is to remove code clones which arise by copy-pasting, because of the same reasons. In (12) a technique called program slicing is used. Program dependency graphs are used to find statements which are dependant of each other, if groups of dependant statements match in two fragments, they are considered a clone. Because the order of dependency is used instead of the physical order of the statements, the groups of matching statements don't have to be continuous or even in the same order. In (12) some interesting results were reported, combining this method with the expression substitution presented in this thesis potentially results in even more interesting matches.

8.3 Finding more than two instances of a clone at once

The matching algorithm and the unifying method extraction algorithm only match two fragments and remove the code duplicates. Often though, more than two clones of a fragment exist. If code clones are removed, it is obviously preferable to remove all instances clone. It is therefore useful to adapt the match and refactor algorithm in such a way it can handle more than two instances of a clone. An approach which could be used is the following. Pairs of fragments can be matched as is done by the current algorithm. If one fragment of code has more than one match, the fragment and all permutations of the matches are candidates for refactoring. The matching and rewriting algorithms could easily be adopted, instead of that the kind of two statements need to be the same, the kind of N statements need to be the same. Instead of that two expressions need to be the same or substitutable to be a match, N expressions need to be the same or substitutable to be a match. The biggest set of fragments that can be

matched this way can be regarded as a set of code clones and be refactored away.

9 Conclusions

When coding large projects, chances are that existing code fragments will be re-written over and again. This can be done unconsciously but also by copy-pasting, this is unfortunately a widespread practice. In some case studies of commercial software 5% of the code were clones. It is well-known that code duplication decreases the maintainability of the code rapidly. Detecting clones and remove them using method extraction will improve maintainability and understandability because only one copy of the clone needs to be maintained and understood.

In this thesis a tool is presented which can detect code duplicates in programs written in Visual Basic. The tool offers the possibility to automatically remove these duplicates by using a method extraction refactoring. The novel part of this tool is that it can ensure that the code clones presented for refactoring actually can be refactored without changing the external behaviour of the refactored program. To our knowledge this is tool is the only one existent which can perform a fully automated code extraction which is constructed on the bases of more than one fragment of code.

The code clone algorithm used is AST based, but unlike other implementations using this approach, it does not ignore variable names or literal values. The code clone algorithm is designed in such a way it searches code clones which can be removed using the method extraction refactoring. A semantic analysis is used to classify each expression, the algorithm compares the types and origin expressions classify to. Parts of expressions that are not the same but do classify to the same type can be substituted by a variable holding the result value of the evaluated original expressions or if the original expression is a variable or property act as a temp variable. These variables can be parameters via which these values are passed to the extracted method or be returned if the value is used after the fragments the extracted method is based on.

To regard two fragments a match, other methods only allow for different variable names or literal values and is some novel cases a different order of statements. In the method presented in this thesis, the statements in matching fragments must be in the same order and of the same kind, but expressions can be substituted with variables as long as the data flow is not affected. Because of this it is possible to find code clones which other methods know won't find.

To make it possible to compare nodes in the way needed for the matching algorithm and the nature of the target language; Visual Basic 8, deep semantic analysis is needed. This is because the language contains different constructions which make classification non trivial. Because none was available for the target language, a semantic analyzer had to be implemented. In this thesis a semantic analyzer is presented, which can classify any expressions to the type and origin of the contained value.

The matching algorithm first finds pairs of matching statements, allowing expressions to be substituted where necessary and possible. These matching

statement pairs are combined in such a way that the resulting fragments are maximized but the original data flow stays intact and the selection criteria are fulfilled.

To construct an extracted method, one of the fragments is used as the method body with the appropriate expressions substituted as variables. Data flow analysis is used to determine which of the substitute variables need to be parameters and which one needs to be returned if any. Declarators are added to the method body for the substitute variables which are not already implicit declared in the body and are not parameters.

To increase performance and usability, future work on the tool could include using a crude clone detection algorithm, for example token based, to find potential code clones much faster, the matching algorithm currently used could then only be applied on the clones found by this algorithm.

We feel that using slicing techniques in the matching and rewriting algorithm would increase the number of interesting clones found and removed. The core elements that are necessary to implement slicing are already in place; the dependency and data flow consistency check. The only thing that needs to be altered is the algorithm which combines matching statement pairs.

10 References

1. *Clone Detection Using Abstract Syntax Trees.* **Baxter, Ira D, et al.** 1998. 14th IEEE International Conference on Software Maintenance (ICSM'98). p. 368.

2. *On Finding Duplication and Near-Duplication in Large Software Systems.* **Baker, Brenda S.** 1995. IEEE Second Working Conference on Reverse Engineering. pp. 86-95.

3. Does code duplication matter? *SolidSource BV.* [Online] 15 January 2010. http://www.solidsourceit.com/blog/?p=4.

4. *CCFinder: A Multilinguistic Token-Based Code Clone Detectino System for Large Scale Source Code.* **Kamiya, Toshihiro, Kusumoto, Shinji and Inoue, Katsuro.** 7, July 2002, IEEE Transactions on Software Engineering, Vol. 28, pp. 654-270.

5. *Aries: Refactoring Support Environment Based on Code Clone Analysis.* **Higo, Yoshiki, Kamiya, Toshihiro and Kusumoto, Shinji.** Cambridge, MA, USA : s.n., 2004. Proceedings of the Eight IASTED International Conference Software Engineering and Applications.

6. *A Language Independent Approach for Detecting Duplicate Code.* **Ducasse, Stéphane, Rieger, Matthias en Demeyer, Serge.** 1999. 15th IEEE International Conference on Software Maintenance (ICSM '99). p. 109.

7. SolidSDD Software Duplication Detector. *SolidSource BV.* [Online] http://www.solidsourceit.com.

8. *Pattern Matching for Clone and Concept Detection.* **Kontogiannis, K A, et al.** 1-2, 1996, Automated Software Engineering, Vol. 3, pp. 77-108.

9. *Clone Detection using Abstract Syntax Suffix Trees.* **Koschke, Rainer, Falke, Raimar en Frenzel, Pierre.** Benevento : sn, 2006. 13th Working Conference on Reverse Engineering. (WCRE '06). pp. 253-262.

10. **Komondoor, Raghavan V en Horwitz, Susan.** Using Slicing to Identify Duplication in Source Code. [boekaut.] Patrick Cousot. *Static Analysis.* sl : Springer Berlin/Heidelberg, 2001, pp. 40-56.

11. **Fowler, Martin.** *Refactoring: Improving the Design of Existing Code.* sl : Addison Wesley, 1999.

12. **Komondoor, Raghavan V.** Automated Duplication-Code Detection and Procedure Extraction. Madison, WI, USA : Springer, 2003.

13. **Grune, Dick, et al.** *Modern Compiler Design.* s.l. : Wiley, 2000.

14. **Vick, Paul.** VBParser. *Panopticon Central.* [Online] 27 March 2006. http://panopticoncentral.net/archive/2006/03/27/11531.aspx.

15. —. Visual Basic Language Specification 8.0. *MSDN.* [Online] 15 11 2005. http://msdn.microsoft.com/en-us/library/ms234437(VS.80).aspx.

16. *Tool Support for Refactoring Duplicated OO Code.* **Ducasse, Stéphane, Rieger, Matthias en Golomingi, Georges.** Lisbon : sn, 1999. Preceedings of the ECOOP '99 Workshop on Experiences in Object-Oriented Re-Engineering.

17. **Opdyke, William F.** Refactoring Object-Oriented Frameworks. sl, Illinois : University of Illinois, 1992.

18. *Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method.* **Murphy-Hill, Emerson and Black, Andrew P.** Leipzig, Germany : ACM, 2008. Proceedings of the 30th international conference on Software engineering. pp. 421-430.

19. *The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capicity for Processing Information.* **Miller, G A.** 2, March 1956, The Psychological Review, Vol. 63, pp. 81-97.

20. Mono. [Online] http://www.mono-project.com/VisualBasic.NET_support.

11 Appendix 1: Visual Basic 8.0 grammar

11.1 Lexical Grammar

Start ::= [*LogicalLine+*]

LogicalLine ::= [*LogicalLineElement+*] [*Comment*] *LineTerminator*

LogicalLineElement ::= *WhiteSpace* | *LineContinuation* | *Token*

Token ::= *Identifier* | *Keyword* | *Literal* | *Separator* | *Operator*

11.1.1 Characters and Lines

Character ::= < any Unicode character except a *LineTerminator* >

LineTerminator ::=
    < Unicode carriage return character (0x000D) > |
    < Unicode linefeed character (0x000A) > |
    < Unicode carriage return character > < Unicode linefeed character > |
    < Unicode line separator character (0x2028) > |
    < Unicode paragraph separator character (0x2029) >

LineContinuation ::= *WhiteSpace* `_` [*WhiteSpace+*] *LineTerminator*

WhiteSpace ::=
    < Unicode blank characters (class Zs) > |
    < Unicode tab character (0x0009) >

Comment ::= *CommentMarker* [*Character+*]

CommentMarker ::= *SingleQuoteCharacter* | `REM`

SingleQuoteCharacter ::=
    `'` |
    < Unicode left single-quote character (0x2018) > |
    < Unicode right single-quote character (0x2019) >

11.1.2 Identifiers

Identifier ::=
    NonEscapedIdentifier [*TypeCharacter*] |
    Keyword *TypeCharacter* |
    EscapedIdentifier

NonEscapedIdentifier ::= < *IdentifierName* but not *Keyword* >

EscapedIdentifier ::= `[` *IdentifierName* `]`

IdentifierName ::= *IdentifierStart* [*IdentifierCharacter+*]

IdentifierStart ::=
    AlphaCharacter |
    UnderscoreCharacter *IdentifierCharacter*

IdentifierCharacter ::=
    UnderscoreCharacter |
    AlphaCharacter |
    NumericCharacter |

CombiningCharacter |
FormattingCharacter

AlphaCharacter ::=
< Unicode alphabetic character (classes Lu, Ll, Lt, Lm, Lo, Nl) >

NumericCharacter ::= < Unicode decimal digit character (class Nd) >

CombiningCharacter ::= < Unicode combining character (classes Mn, Mc) >

FormattingCharacter ::= < Unicode formatting character (class Cf) >

UnderscoreCharacter ::= < Unicode connection character (class Pc) >

IdentifierOrKeyword ::= *Identifier* | *Keyword*

TypeCharacter ::=
IntegerTypeCharacter |
LongTypeCharacter |
DecimalTypeCharacter |
SingleTypeCharacter |
DoubleTypeCharacter |
StringTypeCharacter

IntegerTypeCharacter ::= `%`

LongTypeCharacter ::= `&`

DecimalTypeCharacter ::= `@`

SingleTypeCharacter ::= `!`

DoubleTypeCharacter ::= `#`

StringTypeCharacter ::= `$`

11.1.3 Keywords

Keyword ::= < member of keyword table in 2.3 >

11.1.4 Literals

Literal ::=
BooleanLiteral |
IntegerLiteral |
FloatingPointLiteral |
StringLiteral |
CharacterLiteral |
DateLiteral |
Nothing

BooleanLiteral ::= `True` | `False`

IntegerLiteral ::= *IntegralLiteralValue* [*IntegralTypeCharacter*]

IntegralLiteralValue ::= *IntLiteral* | *HexLiteral* | *OctalLiteral*

IntegralTypeCharacter ::=
ShortCharacter |
UnsignedShortCharacter |

IntegerCharacter |
UnsignedIntegerCharacter
LongCharacter |
UnsignedLongCharacter |
IntegerTypeCharacter |
LongTypeCharacter

ShortCharacter ::= `S`

UnsignedShortCharacter ::= `US`

IntegerCharacter ::= `I`

UnsignedIntegerCharacter ::= `UI`

LongCharacter ::= `L`

UnsignedLongCharacter ::= `UL`

IntLiteral ::= *Digit+*

HexLiteral ::= `&` `H` *HexDigit+*

OctalLiteral ::= `&` `O` *OctalDigit+*

Digit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

HexDigit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` | `A` | `B` | `C` | `D` | `E` | `F`

OctalDigit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`

FloatingPointLiteral ::=
FloatingPointLiteralValue [*FloatingPointTypeCharacter*] |
IntLiteral *FloatingPointTypeCharacter*

FloatingPointTypeCharacter ::=
SingleCharacter |
DoubleCharacter |
DecimalCharacter |
SingleTypeCharacter |
DoubleTypeCharacter |
DecimalTypeCharacter

SingleCharacter ::= `F`

DoubleCharacter ::= `R`

DecimalCharacter ::= `D`

FloatingPointLiteralValue ::=
IntLiteral `.` *IntLiteral* [*Exponent*] |
`.` *IntLiteral* [*Exponent*] |
IntLiteral *Exponent*

Exponent ::= `E` [*Sign*] *IntLiteral*

Sign ::= `+` | `-`

StringLiteral ::=
DoubleQuoteCharacter [*StringCharacter+*] *DoubleQuoteCharacter*

DoubleQuoteCharacter ::=
" |
< Unicode left double-quote character (0x201C) > |
< Unicode right double-quote character (0x201D) >

StringCharacter ::=
< Character except for DoubleQuoteCharacter > |
DoubleQuoteCharacter DoubleQuoteCharacter

CharacterLiteral ::= DoubleQuoteCharacter StringCharacter DoubleQuoteCharacter C

DateLiteral ::= # [Whitespace+] DateOrTime [Whitespace+] #

DateOrTime ::=
DateValue Whitespace+ TimeValue |
DateValue |
TimeValue

DateValue ::=
MonthValue / DayValue / YearValue |
MonthValue - DayValue - YearValue

TimeValue ::=
HourValue : MinuteValue [: SecondValue] [WhiteSpace+] [AMPM]

MonthValue ::= IntLiteral

DayValue ::= IntLiteral

YearValue ::= IntLiteral

HourValue ::= IntLiteral

MinuteValue ::= IntLiteral

SecondValue ::= IntLiteral

AMPM ::= AM | PM

Nothing ::= Nothing

Separator ::= (|) | { | } | ! | # | , | . | : | :=

Operator ::=
& | * | + | - | / | \ | ^ | < | = | > | <= | >= | <> | << | >> |
&= | *= | += | -= | /= | \= | ^= | <<= | >>=

11.2 Preprocessing Directives

11.2.1 Conditional Compilation

Start ::= [CCStatement+]

CCStatement ::=
CCConstantDeclaration |
CCIfGroup |
LogicalLine

CCExpression ::=
LiteralExpression |

```
        CCParenthesizedExpression |
        SimpleNameExpression |
        CCCastExpression |
        CCOperatorExpression

CCParenthesizedExpression ::= ( CCExpression )

CCCastExpression ::= CastTarget ( CCExpression )

CCOperatorExpression ::=
        CCUnaryOperator CCExpression
        CCExpression CCBinaryOperator CCExpression

CCUnaryOperator ::= + | - | Not

CCBinaryOperator ::= + | - | * | / | \ | Mod | ^ | = | <> | < | > |

        <= | >= | & | And | Or | Xor | AndAlso | OrElse | << | >>

CCConstantDeclaration ::= # Const Identifier = CCExpression LineTerminator

CCIfGroup ::=
        # If CCExpression [ Then ] LineTerminator
        [ CCStatement+ ]
        [ CCElseIfGroup+ ]
        [ CCElseGroup ]
        # End If LineTerminator

CCElseIfGroup ::=
        # ElseIf CCExpression [ Then ] LineTerminator
        [ CCStatement+ ]

CCElseGroup ::=
        # Else LineTerminator
        [ CCStatement+ ]
```

11.2.2 External Source Directives

```
Start ::= [ ExternalSourceStatement+ ]

ExternalSourceStatement ::= ExternalSourceGroup | LogicalLine

ExternalSourceGroup ::=
        # ExternalSource ( StringLiteral , IntLiteral ) LineTerminator
        [ LogicalLine+ ]
        # End ExternalSource LineTerminator
```

11.2.3 Region Directives

```
Start ::= [ RegionStatement+ ]

RegionStatement ::= RegionGroup | LogicalLine

RegionGroup ::=
        # Region StringLiteral LineTerminator
        [ LogicalLine+ ]
        # End Region LineTerminator
```

11.2.4 External Checksum Directives

```
Start ::= [ ExternalChecksumStatement+ ]

ExternalChecksumStatement ::=
        # ExternalChecksum ( StringLiteral , StringLiteral , StringLiteral ) LineTerminator
```

11.3 Syntactic Grammar

```
AccessModifier ::= Public | Protected | Friend | Private | Protected Friend

QualifiedIdentifier ::=
        Identifier |
        Global . IdentifierOrKeyword |
        QualifiedIdentifier . IdentifierOrKeyword

TypeParameterList ::=
        ( Of TypeParameters )

TypeParameters ::=
        TypeParameter |
        TypeParameters , TypeParameter

TypeParameter ::=
        Identifier [ TypeParameterConstraints ]

TypeParameterConstraints ::=
        As Constraint |
        As { ConstraintList }

ConstraintList ::=
        ConstraintList , Constraint |
        Constraint

Constraint ::= TypeName | New
```

11.3.1 Attributes

```
Attributes ::=
        AttributeBlock |
        Attributes AttributeBlock

AttributeBlock ::= < AttributeList >

AttributeList ::=
        Attribute |
        AttributeList , Attribute

Attribute ::=
        [ AttributeModifier : ] SimpleTypeName [ ( [ AttributeArguments ] ) ]

AttributeModifier ::= Assembly | Module

AttributeArguments ::=
        AttributePositionalArgumentList |
        AttributePositionalArgumentList , VariablePropertyInitializerList |
        VariablePropertyInitializerList
```

AttributePositionalArgumentList ::=
  AttributeArgumentExpression |
  AttributePositionalArgumentList `,` *AttributeArgumentExpression*

VariablePropertyInitializerList ::=
  VariablePropertyInitializer |
  VariablePropertyInitializerList `,` *VariablePropertyInitializer*

VariablePropertyInitializer ::=
  IdentifierOrKeyword `:=` *AttributeArgumentExpression*

AttributeArgumentExpression ::=
  ConstantExpression |
  GetTypeExpression |
  ArrayCreationExpression

11.3.2 Source Files and Namespaces

Start ::=
  [*OptionStatement+*]
  [*ImportsStatement+*]
  [*AttributesStatement+*]
  [*NamespaceMemberDeclaration+*]

StatementTerminator ::= *LineTerminator* | `:`

AttributesStatement ::= *Attributes* *StatementTerminator*

OptionStatement ::=
  OptionExplicitStatement |
  OptionStrictStatement |
  OptionCompareStatement

OptionExplicitStatement ::= `Option Explicit` [*OnOff*] *StatementTerminator*

OnOff ::= `On` | `Off`

OptionStrictStatement ::= `Option Strict` [*OnOff*] *StatementTerminator*

OptionCompareStatement ::= `Option Compare` *CompareOption* *StatementTerminator*

CompareOption ::= `Binary` | `Text`

ImportsStatement ::= `Imports` *ImportsClauses* *StatementTerminator*

ImportsClauses ::=
  ImportsClause |
  ImportsClauses `,` *ImportsClause*

ImportsClause ::= *ImportsAliasClause* | *ImportsNamespaceClause*

ImportsAliasClause ::=
  Identifier `=` *QualifiedIdentifier* |
  Identifier `=` *ConstructedTypeName*

ImportsNamespaceClause ::=
  QualifiedIdentifier |
  ConstructedTypeName

NamespaceDeclaration ::=
    `Namespace` *QualifiedIdentifier* *StatementTerminator*
    [*NamespaceMemberDeclaration+*]
    `End` `Namespace` *StatementTerminator*

NamespaceMemberDeclaration ::=
    NamespaceDeclaration |
    TypeDeclaration

TypeDeclaration ::=
    ModuleDeclaration |
    NonModuleDeclaration

NonModuleDeclaration ::=
    EnumDeclaration |
    StructureDeclaration |
    InterfaceDeclaration |
    ClassDeclaration |
    DelegateDeclaration

11.3.3 Types

TypeName ::=
    ArrayTypeName |
    NonArrayTypeName

NonArrayTypeName ::=
    SimpleTypeName |
    ConstructedTypeName

SimpleTypeName ::=
    QualifiedIdentifier |
    BuiltInTypeName

BuiltInTypeName ::= `Object` | *PrimitiveTypeName*

TypeModifier ::= *AccessModifier* | `Shadows`

TypeImplementsClause ::= `Implements` *Implements* *StatementTerminator*

Implements ::=
    NonArrayTypeName |
    Implements `,` *NonArrayTypeName*

PrimitiveTypeName ::= *NumericTypeName* | `Boolean` | `Date` | `Char` | `String`

NumericTypeName ::= *IntegralTypeName* | *FloatingPointTypeName* | `Decimal`

IntegralTypeName ::= `Byte` | `SByte` | `UShort` | `Short` | `UInteger` | `Integer` | `ULong` | `Long`

FloatingPointTypeName ::= `Single` | `Double`

EnumDeclaration ::=
    [*Attributes*] [*TypeModifier+*] `Enum` *Identifier* [`As` *QualifiedName*]
StatementTerminator

```
        EnumMemberDeclaration+
        End Enum StatementTerminator

EnumMemberDeclaration ::= [ Attributes ] Identifier [ = ConstantExpression ]
StatementTerminator

ClassDeclaration ::=
        [ Attributes ] [ ClassModifier+ ] Class Identifier [ TypeParameterList ]
StatementTerminator
        [ ClassBase ]
        [ TypeImplementsClause+ ]
        [ ClassMemberDeclaration+ ]
        End Class StatementTerminator

ClassModifier ::= TypeModifier | MustInherit | NotInheritable | Partial

ClassBase ::= Inherits NonArrayTypeName StatementTerminator

ClassMemberDeclaration ::=
        NonModuleDeclaration |
        EventMemberDeclaration |
        VariableMemberDeclaration |
        ConstantMemberDeclaration |
        MethodMemberDeclaration |
        PropertyMemberDeclaration |
        ConstructorMemberDeclaration |
        OperatorDeclaration

StructureDeclaration ::=
        [ Attributes ] [ StructureModifier+ ] Structure Identifier [ TypeParameterList ]
                StatementTerminator
        [ TypeImplementsClause+ ]
        [ StructMemberDeclaration+ ]
        End Structure StatementTerminator

StructureModifier ::= TypeModifier | Partial

StructMemberDeclaration ::=
        NonModuleDeclaration |
        VariableMemberDeclaration |
        ConstantMemberDeclaration |
        EventMemberDeclaration |
        MethodMemberDeclaration |
        PropertyMemberDeclaration |
        ConstructorMemberDeclaration |
        OperatorDeclaration

ModuleDeclaration ::=
        [ Attributes ] [ TypeModifier+ ] Module Identifier StatementTerminator
        [ ModuleMemberDeclaration+ ]
        End Module StatementTerminator

ModuleMemberDeclaration ::=
        NonModuleDeclaration |
```

```
        VariableMemberDeclaration |
        ConstantMemberDeclaration |
        EventMemberDeclaration |
        MethodMemberDeclaration |
        PropertyMemberDeclaration |
        ConstructorMemberDeclaration

InterfaceDeclaration ::=
        [ Attributes ] [ TypeModifier+ ] Interface Identifier [ TypeParameterList ]
StatementTerminator
        [ InterfaceBase+ ]
        [ InterfaceMemberDeclaration+ ]
        End Interface StatementTerminator

InterfaceBase ::= Inherits InterfaceBases StatementTerminator

InterfaceBases ::=
        NonArrayTypeName |
        InterfaceBases , NonArrayTypeName

InterfaceMemberDeclaration ::=
        NonModuleDeclaration |
        InterfaceEventMemberDeclaration |
        InterfaceMethodMemberDeclaration |
        InterfacePropertyMemberDeclaration

ArrayTypeName ::= NonArrayTypeName ArrayTypeModifiers

ArrayTypeModifiers ::= ArrayTypeModifier+

ArrayTypeModifier ::= ( [ RankList ] )

RankList ::=
        , |
        RankList ,

ArrayNameModifier ::=
        ArrayTypeModifiers |
        ArraySizeInitializationModifier

DelegateDeclaration ::=
        [ Attributes ] [ TypeModifier+ ] Delegate MethodSignature StatementTerminator

MethodSignature ::= SubSignature | FunctionSignature

ConstructedTypeName ::=
        QualifiedIdentifier ( Of TypeArgumentList )

TypeArgumentList ::=
        TypeName |
        TypeArgumentList , TypeName
```

11.3.4 Type Members

```
ImplementsClause ::= [ Implements ImplementsList ]
```

ImplementsList ::=
    InterfaceMemberSpecifier |
    ImplementsList `,` *InterfaceMemberSpecifier*

InterfaceMemberSpecifier ::= *NonArrayTypeName* `.` *IdentifierOrKeyword*

MethodMemberDeclaration ::= *MethodDeclaration* | *ExternalMethodDeclaration*

InterfaceMethodMemberDeclaration ::= *InterfaceMethodDeclaration*

MethodDeclaration ::=
    SubDeclaration |
    MustOverrideSubDeclaration |
    FunctionDeclaration |
    MustOverrideFunctionDeclaration

InterfaceMethodDeclaration ::=
    InterfaceSubDeclaration |
    InterfaceFunctionDeclaration

SubSignature ::= *Identifier* [*TypeParameterList*] [`(` [*ParameterList*] `)`]

FunctionSignature ::= *SubSignature* [`As` [*Attributes*] *TypeName*]

SubDeclaration ::=
    [Attributes] [ProcedureModifier+] Sub SubSignature [*HandlesOrImplements*]
LineTerminator
    Block
    `End Sub` *StatementTerminator*

MustOverrideSubDeclaration ::=
    [*Attributes*] [*MustOverrideProcedureModifier+*] *Sub SubSignature* [
HandlesOrImplements]
        StatementTerminator

InterfaceSubDeclaration ::=
    [*Attributes*] [*InterfaceProcedureModifier+*] *Sub SubSignature StatementTerminator*

FunctionDeclaration ::=
    [*Attributes*] [*ProcedureModifier+*] `Function` *FunctionSignature* [
HandlesOrImplements]
        LineTerminator
    Block
    `End Function` *StatementTerminator*

MustOverrideFunctionDeclaration ::=
    [*Attributes*] [*MustOverrideProcedureModifier+*] `Function` *FunctionSignature*
        [*HandlesOrImplements*] *StatementTerminator*

InterfaceFunctionDeclaration ::=
    [*Attributes*] [*InterfaceProcedureModifier+*] `Function` *FunctionSignature*
StatementTerminator

ProcedureModifier ::=
    AccessModifier |
    `Shadows` |

```
        Shared |
        Overridable |
        NotOverridable |
        Overrides |
        Overloads

MustOverrideProcedureModifier ::= ProcedureModifier | MustOverride

InterfaceProcedureModifier ::= Shadows | Overloads

HandlesOrImplements ::= HandlesClause | ImplementsClause

ExternalMethodDeclaration ::=
        ExternalSubDeclaration |
        ExternalFunctionDeclaration

ExternalSubDeclaration ::=
        [ Attributes ] [ ExternalMethodModifier+ ] Declare [ CharsetModifier ] Sub Identifier
                LibraryClause [ AliasClause ] [ ( [ ParameterList ] ) ] StatementTerminator

ExternalFunctionDeclaration ::=
        [ Attributes ] [ ExternalMethodModifier+ ] Declare [ CharsetModifier ] Function
Identifier
                LibraryClause [ AliasClause ] [ ( [ ParameterList ] ) ] [ As [ Attributes ]
TypeName ]
                StatementTerminator

ExternalMethodModifier ::= AccessModifier | Shadows | Overloads

CharsetModifier ::= Ansi | Unicode | Auto

LibraryClause ::= Lib StringLiteral

AliasClause ::= Alias StringLiteral

ParameterList ::=
        Parameter |
        ParameterList , Parameter

Parameter ::=
        [ Attributes ] ParameterModifier+ ParameterIdentifier [ As TypeName ] [ =
ConstantExpression ]

ParameterModifier ::= ByVal | ByRef | Optional | ParamArray

ParameterIdentifier ::= Identifier [ ArrayNameModifier ]

HandlesClause ::= [ Handles EventHandlesList ]

EventHandlesList ::=
        EventMemberSpecifier |
        EventHandlesList , EventMemberSpecifier

EventMemberSpecifier ::=
        QualifiedIdentifier . IdentifierOrKeyword |
        MyBase . IdentifierOrKeyword |
        Me . IdentifierOrKeyword
```

```
ConstructorMemberDeclaration ::=
        [ Attributes ] [ ConstructorModifier+ ] Sub New [ ( [ ParameterList ] ) ]
LineTerminator
        [ Block ]
        End Sub StatementTerminator

ConstructorModifier ::= AccessModifier | Shared

EventMemberDeclaration ::=
        RegularEventMemberDeclaration |
        CustomEventMemberDeclaration

RegularEventMemberDeclaration ::=
        [ Attributes ] [ EventModifiers+ ] Event Identifier ParametersOrType [
ImplementsClause ]
                StatementTerminator

InterfaceEventMemberDeclaration ::=
        [ Attributes ] [ InterfaceEventModifiers+ ] Event Identifier ParametersOrType
StatementTerminator

ParametersOrType ::=
        [ ( [ ParameterList ] ) ] |
        As NonArrayTypeName

EventModifiers ::= AccessModifier | Shadows | Shared

InterfaceEventModifiers ::= Shadows

CustomEventMemberDeclaration ::=
        [ Attributes ] [ EventModifiers+ ] Custom Event Identifier As TypeName [
ImplementsClause ]
                StatementTerminator
                EventAccessorDeclaration+
        End Event StatementTerminator

EventAccessorDeclaration ::=
        AddHandlerDeclaration |
        RemoveHandlerDeclaration |
        RaiseEventDeclaration

AddHandlerDeclaration ::=
        [ Attributes ] AddHandler ( ParameterList ) LineTerminator
        [ Block ]
        End AddHandler StatementTerminator

RemoveHandlerDeclaration ::=
        [ Attributes ] RemoveHandler ( ParameterList ) LineTerminator
        [ Block ]
        End RemoveHandler StatementTerminator

RaiseEventDeclaration ::=
        [ Attributes ] RaiseEvent ( ParameterList ) LineTerminator
        [ Block ]
        End RaiseEvent StatementTerminator
```

```
ConstantMemberDeclaration ::=
        [ Attributes ] [ ConstantModifier+ ] Const ConstantDeclarators StatementTerminator

ConstantModifier ::= AccessModifier | Shadows

ConstantDeclarators ::=
        ConstantDeclarator |
        ConstantDeclarators , ConstantDeclarator

ConstantDeclarator ::= Identifier [ As TypeName ] = ConstantExpression StatementTerminator

VariableMemberDeclaration ::=
        [ Attributes ] VariableModifier+ VariableDeclarators StatementTerminator

VariableModifier ::=
        AccessModifier |
        Shadows |
        Shared |
        ReadOnly |
        WithEvents |
        Dim

VariableDeclarators ::=
        VariableDeclarator |
        VariableDeclarators , VariableDeclarator

VariableDeclarator ::=
        VariableIdentifiers [ As [ New ] TypeName [ ( ArgumentList ) ] ] |
        VariableIdentifier [ As TypeName ] [ = VariableInitializer ]

VariableIdentifiers ::=
        VariableIdentifier |
        VariableIdentifiers , VariableIdentifier

VariableIdentifier ::= Identifier [ ArrayNameModifier ]

VariableInitializer ::= RegularInitializer | ArrayElementInitializer

RegularInitializer ::= Expression

ArraySizeInitializationModifier ::=
        ( BoundList ) [ ArrayTypeModifiers ]

BoundList::=
        Expression |
        0 To Expression |
        UpperBoundList , Expression

ArrayElementInitializer ::= { [ VariableInitializerList ] }

VariableInitializerList ::=
        VariableInitializer |
        VariableInitializerList , VariableInitializer

VariableInitializer ::= Expression | ArrayElementInitializer
```

```
PropertyMemberDeclaration ::=
	RegularPropertyMemberDeclaration |
	MustOverridePropertyMemberDeclaration

RegularPropertyMemberDeclaration ::=
	[ Attributes ] [ PropertyModifier+ ] Property FunctionSignature [ ImplementsClause ]
		LineTerminator
	PropertyAccessorDeclaration+
	End Property StatementTerminator

MustOverridePropertyMemberDeclaration ::=
	[ Attributes ] [ MustOverridePropertyModifier+ ] Property FunctionSignature [
ImplementsClause ]
		StatementTerminator

InterfacePropertyMemberDeclaration ::=
	[ Attributes ] [ InterfacePropertyModifier+ ] Property FunctionSignature
StatementTerminator

PropertyModifier ::= ProcedureModifier | Default | ReadOnly | WriteOnly

MustOverridePropertyModifier ::= PropertyModifier | MustOverride

InterfacePropertyModifier ::=
	Shadows |
	Overloads |
	Default |
	ReadOnly |
	WriteOnly

PropertyAccessorDeclaration ::= PropertyGetDeclaration | PropertySetDeclaration

PropertyGetDeclaration ::=
	[ Attributes ] [ AccessModifier ] Get LineTerminator
	[ Block ]
	End Get StatementTerminator

PropertySetDeclaration ::=
	[ Attributes ] [ AccessModifier ] Set [ ( ParameterList ) ] LineTerminator
	[ Block ]
	End Set StatementTerminator

OperatorDeclaration ::=
	UnaryOperatorDeclaration |
	BinaryOperatorDeclaration |
	ConversionOperatorDeclaration

OperatorModifier ::= Public | Shared | Overloads | Shadows

Operand ::= [ ByVal ] Identifier [ As TypeName ]

UnaryOperatorDeclaration ::=
	[ Attributes ] [ OperatorModifier+ ] Operator OverloadableUnaryOperator ( Operand
)
		[ As [ Attributes ] TypeName ] LineTerminator
```

```
        [ Block ]
        End Operator StatementTerminator

OverloadableUnaryOperator ::= + | - | Not | IsTrue | IsFalse

BinaryOperatorDeclaration ::=
        [ Attributes ] [ OperatorModifier+ ] Operator OverloadableBinaryOperator
                ( Operand , Operand ) [ As [ Attributes ] TypeName ] LineTerminator
        [ Block ]
        End Operator StatementTerminator

OverloadableBinaryOperator ::=
        + | - | * | / | \ | & | Like | Mod | And | Or | Xor |
        ^ | << | >> | = | <> | > | < | >= | <=

ConversionOperatorDeclaration ::=
        [ Attributes ] [ ConversionOperatorModifier+ ] Operator CType ( Operand )
                [ As [ Attributes ] TypeName ] LineTerminator
        [ Block ]
        End Operator StatementTerminator

ConversionOperatorModifier ::= Widening | Narrowing | ConversionModifier
```

11.3.5 Statements

```
Statement ::=
        LabelDeclarationStatement |
        LocalDeclarationStatement |
        WithStatement |
        SyncLockStatement |
        EventStatement |
        AssignmentStatement |
        InvocationStatement |
        ConditionalStatement |
        LoopStatement |
        ErrorHandlingStatement |
        BranchStatement |
        ArrayHandlingStatement |
        UsingStatement

Block ::= [ Statements+ ]

LabelDeclarationStatement ::= LabelName :

LabelName ::= Identifier | IntLiteral

Statements ::=
        [ Statement ] |
        Statements : [ Statement ]

LocalDeclarationStatement ::= LocalModifier VariableDeclarators StatementTerminator

LocalModifier ::= Static | Dim | Const

WithStatement ::=
        With Expression StatementTerminator
```

```
        [ Block ]
        End With StatementTerminator

SyncLockStatement ::=
        SyncLock Expression StatementTerminator
        [ Block ]
        End SyncLock StatementTerminator

EventStatement ::=
        RaiseEventStatement |
        AddHandlerStatement |
        RemoveHandlerStatement

RaiseEventStatement ::= RaiseEvent IdentifierOrKeyword [ ( [ ArgumentList ] ) ]
        StatementTerminator

AddHandlerStatement ::= AddHandler Expression , Expression StatementTerminator

RemoveHandlerStatement ::= RemoveHandler Expression , Expression StatementTerminator

AssignmentStatement ::=
        RegularAssignmentStatement |
        CompoundAssignmentStatement |
        MidAssignmentStatement

RegularAssignmentStatement ::= Expression = Expression StatementTerminator

CompoundAssignmentStatement ::= Expression CompoundBinaryOperator Expression
StatementTerminator

CompoundBinaryOperator ::= ^= | *= | /= | \= | += | -= | &= | <<= | >>=

MidAssignmentStatement ::=
        Mid [ $ ] ( Expression , Expression [ , Expression ] ) = Expression
StatementTerminator

InvocationStatement ::= [ Call ] InvocationExpression StatementTerminator

ConditionalStatement ::= IfStatement | SelectStatement

IfStatement ::= BlockIfStatement | LineIfThenStatement

BlockIfStatement ::=
        If BooleanExpression [ Then ] StatementTerminator
        [ Block ]
        [ ElseIfStatement+ ]
        [ ElseStatement ]
        End If StatementTerminator

ElseIfStatement ::=
        ElseIf BooleanExpression [ Then ] StatementTerminator
        [ Block ]

ElseStatement ::=
        Else StatementTerminator
        [ Block ]
```

```
LineIfThenStatement ::=
	If BooleanExpression Then Statements [ Else Statements ] StatementTerminator

SelectStatement ::=
	Select [ Case ] Expression StatementTerminator
	[ CaseStatement+ ]
	[ CaseElseStatement ]
	End Select StatementTerminator

CaseStatement ::=
	Case CaseClauses StatementTerminator
	[ Block ]

CaseClauses ::=
	CaseClause |
	CaseClauses , CaseClause

CaseClause ::=
	[ Is ] ComparisonOperator Expression |
	Expression [ To Expression ]

ComparisonOperator ::= = | <> | < | > | => | =<

CaseElseStatement ::=
	Case Else StatementTerminator
	[ Block ]

LoopStatement ::=
	WhileStatement |
	DoLoopStatement |
	ForStatement |
	ForEachStatement

WhileStatement ::=
	While BooleanExpression StatementTerminator
	[ Block ]
	End While StatementTerminator

DoLoopStatement ::= DoTopLoopStatement | DoBottomLoopStatement

DoTopLoopStatement ::=
	Do [ WhileOrUntil BooleanExpression ] StatementTerminator
	[ Block ]
	Loop StatementTerminator

DoBottomLoopStatement ::=
	Do StatementTerminator
	[ Block ]
	Loop WhileOrUntil BooleanExpression StatementTerminator

WhileOrUntil ::= While | Until

ForStatement ::=
	For LoopControlVariable = Expression To Expression [ Step Expression ]
StatementTerminator
```

```
        [ Block ]
        Next [ NextExpressionList ] StatementTerminator

LoopControlVariable ::=
        Identifier [ ArrayNameModifier ] As TypeName |
        Expression

NextExpressionList ::=
        Expression |
        NextExpressionList , Expression

ForEachStatement ::=
        For Each LoopControlVariable In Expression StatementTerminator
        [ Block ]
        Next [Expression ] StatementTerminator

ErrorHandlingStatement ::=
        StructuredErrorStatement |
        UnstructuredErrorStatement

StructuredErrorStatement ::=
        ThrowStatement |
        TryStatement

TryStatement ::=
        Try StatementTerminator
        [ Block ]
        [ CatchStatement+ ]
        [ FinallyStatement ]
        End Try StatementTerminator

FinallyStatement ::=
        Finally StatementTerminator
        [ Block ]

CatchStatement ::=
        Catch [ Identifier As NonArrayTypeName ] [ When BooleanExpression ]
StatementTerminator
        [ Block ]

ThrowStatement ::= Throw [ Expression ] StatementTerminator

UnstructuredErrorStatement ::=
        ErrorStatement |
        OnErrorStatement |
        ResumeStatement

ErrorStatement ::= Error Expression StatementTerminator

OnErrorStatement ::= On Error ErrorClause StatementTerminator

ErrorClause ::=
        GoTo - 1 |
        GoTo 0 |
```

```
        GotoStatement |
        Resume Next

ResumeStatement ::= Resume [ ResumeClause ] StatementTerminator

ResumeClause ::= Next | LabelName

BranchStatement ::=
        GotoStatement |
        ExitStatement |
        ContinueStatement |
        StopStatement |
        EndStatement |
        ReturnStatement

GotoStatement ::= GoTo LabelName StatementTerminator

ExitStatement ::= Exit ExitKind StatementTerminator

ExitKind ::= Do | For | While | Select | Sub | Function | Property | Try

ContinueStatement ::= Continue ContinueKind StatementTerminator

ContinueKind ::= Do | For | While

StopStatement ::= Stop StatementTerminator

EndStatement ::= End StatementTerminator

ReturnStatement ::= Return [ Expression ]

ArrayHandlingStatement ::=
        RedimStatement |
        EraseStatement

RedimStatement ::= ReDim [ Preserve ] RedimClauses StatementTerminator

RedimClauses ::=
        RedimClause |
        RedimClauses , RedimClause

RedimClause ::= Expression ArraySizeInitializationModifier

EraseStatement ::= Erase EraseExpressions StatementTerminator

EraseExpressions ::=
        Expression |
        EraseExpressions , Expression

UsingStatement ::=
        Using UsingResources StatementTerminator
                [ Block ]
        End Using StatementTerminator

UsingResources ::= VariableDeclarators | Expression
```

11.3.6 Expressions

```
Expression ::=
        SimpleExpression |
```

```
        TypeExpression |
        MemberAccessExpression |
        DictionaryAccessExpression |
        IndexExpression |
        NewExpression |
        CastExpression |
        OperatorExpression

ConstantExpression ::= Expression

SimpleExpression ::=
        LiteralExpression |
        ParenthesizedExpression |
        InstanceExpression |
        SimpleNameExpression |
        AddressOfExpression

LiteralExpression ::= Literal

ParenthesizedExpression ::= ( Expression )

InstanceExpression ::= Me

SimpleNameExpression ::= Identifier [ ( Of TypeArgumentList ) ]

AddressOfExpression ::= AddressOf Expression

TypeExpression ::=
        GetTypeExpression |
        TypeOfIsExpression |
        IsExpression

GetTypeExpression ::= GetType ( GetTypeTypeName )

GetTypeTypeName ::=
        TypeName |
        QualifiedIdentifier ( Of [ TypeArityList ] )

TypeArityList ::=
        , |
        TypeParameterList ,

TypeOfIsExpression ::= TypeOf Expression Is TypeName

IsExpression ::=
        Expression Is Expression |
        Expression IsNot Expression

MemberAccessExpression ::=
        [ [ MemberAccessBase ] . ] IdentifierOrKeyword

MemberAccessBase ::=
        Expression |
        BuiltInTypeName |
        Global |
        MyClass |
        MyBase
```

DictionaryAccessExpression ::= [*Expression*] `!` *IdentifierOrKeyword*

InvocationExpression ::= *Expression* [`(` [*ArgumentList*] `)`]

ArgumentList ::=
    PositionalArgumentList `,` *NamedArgumentList* |
    PositionalArgumentList |
    NamedArgumentList

PositionalArgumentList ::=
    Expression |
    PositionalArgumentList `,` [*Expression*]

NamedArgumentList ::=
    IdentifierOrKeyword `:=` *Expression* |
    NamedArgumentList `,` *IdentifierOrKeyword* `:=` *Expression*

IndexExpression ::= *Expression* `(` [*ArgumentList*] `)`

NewExpression ::=
    ObjectCreationExpression |
    ArrayCreationExpression |
    DelegateCreationExpression

ObjectCreationExpression ::=
    `New` *NonArrayTypeName* [`(` [*ArgumentList*] `)`]

ArrayCreationExpression ::=
    `New` *NonArrayTypeName* *ArraySizeInitializationModifier* *ArrayElementInitializer*

DelegateCreationExpression ::= `New` *NonArrayTypeName* `(` *Expression* `)`

CastExpression ::=
    `DirectCast` `(` *Expression* `,` *TypeName* `)` |
    `TryCast` `(` *Expression* `,` *TypeName* `)` |
    `CType` `(` *Expression* `,` *TypeName* `)` |
    CastTarget `(` *Expression* `)`

CastTarget ::=
    `CBool` | `CByte` | `CChar` | `CDate` | `CDec` | `CDbl` | `CInt` | `CLng` | `CObj` | `CSByte` |
`CShort` |
    `CSng` | `CStr` | `CUInt` | `CULng` | `CUShort`

OperatorExpression ::=
    ArithmeticOperatorExpression |
    RelationalOperatorExpression |
    LikeOperatorExpression |
    ConcatenationOperatorExpression |
    ShortCircuitLogicalOperatorExpression |
    LogicalOperatorExpression |
    ShiftOperatorExpression

ArithmeticOperatorExpression ::=
    UnaryPlusExpression |
    UnaryMinusExpression |

```
        AdditionOperatorExpression |
        SubtractionOperatorExpression |
        MultiplicationOperatorExpression |
        DivisionOperatorExpression |
        ModuloOperatorExpression |
        ExponentOperatorExpression

UnaryPlusExpression ::= + Expression

UnaryMinusExpression ::= - Expression

AdditionOperatorExpression ::= Expression + Expression

SubtractionOperatorExpression ::= Expression - Expression

MultiplicationOperatorExpression ::= Expression * Expression

DivisionOperatorExpression ::=
        FPDivisionOperatorExpression |
        IntegerDivisionOperatorExpression

FPDivisionOperatorExpression ::= Expression / Expression

IntegerDivisionOperatorExpression ::= Expression \ Expression

ModuloOperatorExpression ::= Expression Mod Expression

ExponentOperatorExpression ::= Expression ^ Expression

RelationalOperatorExpression ::=
        Expression = Expression |
        Expression <> Expression |
        Expression < Expression |
        Expression > Expression |
        Expression <= Expression |
        Expression >= Expression

LikeOperatorExpression ::= Expression Like Expression

ConcatenationOperatorExpression ::= Expression & Expression

LogicalOperatorExpression ::=
        Not Expression |
        Expression And Expression |
        Expression Or Expression |
        Expression Xor Expression

ShortCircuitLogicalOperatorExpression ::=
        Expression AndAlso Expression |
        Expression OrElse Expression

ShiftOperatorExpression ::=
        Expression << Expression |
        Expression >> Expression

BooleanExpression ::= Expression
```