An Object-Oriented Semantic Model for .NET

10 November 2005

Blair Neate
Supervisors: Mr. Warwick Irwin and Dr. Neville Churcher

Abstract

Software engineering is a complex activity requiring software engineers to understand an intricate structure of components, with many different inter-relationships. In object-oriented software, these components include classes, interfaces, fields, methods and other entities. Relationships include inheritance, implementation, containment, invocation and many more. Static analysis of source code can be used by tools to convey this information to software developers, to increase their understanding of the software they are designing. In this research we present the design of a semantic model that exposes the semantic structure (particularly components and their relationships) of software that is written in a language that compiles to the .NET Common Language Runtime (CLR). The Microsoft .NET Framework is a significant change in programming technology because it allows source code written in different languages to interoperate within the same execution environment (the CLR). By modelling the intermediate language that .NET languages compile to, our model is independent of any specific programming language. We have designed and implemented a semantic model that explicitly exposes the semantic concepts in the CLR and also show its usefulness by presenting an application that would not be possible without the model: calculating the CodeRank metric.

Contents

Chapter 1

Introduction

Software engineering is a complex activity requiring software engineers to understand an intricate structure of components, with many different inter-relationships. In object-oriented software, these components include classes, interfaces, fields, methods and other entities. Relationships include inheritance, implementation, containment, invocation and many more. When designing object-oriented software, developers must understand relevant components and relationships and interpret them in the light of a wide variety of design forces, principles and patterns. The forces at work in a software design can be extraordinarily complex and conflicting, as the designer seeks to optimise competing concerns of simplicity, power, extendability and understandability.

Tools can be used by developers to help increase their understanding of the structure of the software. However, in practice, these tools are often limited to source code editors and UML diagramming. One of the difficulties in developing tools to better inform software developers is gathering the data that the tool needs to work with. In this research, we design a model that explicitly exposes all the components and relationships within .NET software systems, for use by other tools.

The .NET Common Language Runtime (CLR) represents a significant advance in programming technology by allowing different programming languages and libraries to execute and inter-operate within the same execution environment. This is achieved by programming languages (of which there are currently about twenty) compiling to the Microsoft Intermediate Language (MSIL), instead of compiling directly to native code. This intermediate language is then run on the .NET virtual machine. The intermediate language defines common object-oriented semantics that programming languages (particularly object-oriented languages) can map their specific syntax and semantics on to. By modelling explicitly the semantics of the CLR, we can represent the deep structure of source code written in a .NET language. There are a wide variety of languages available to run on the .NET CLR from languages that directly reflect the semantics of the CLR (like C#) to a functional language like Mondrian. The more distant the source language semantics are from the CLR, the more interpretation is needed when using the model.

There are many applications that can use a semantic model to convey useful information to developers in a meaningful way. These include metrics, software visualisation, detecting adherence to design heuristics and in development tools. Using our semantic model, many of these applications can be developed independent of programming language. Our model also exposes new relationships in source code, particularly generics, and allows easy development of applications (like the CodeRank metric) that were previously not possible

1.1 Report Outline

The main contribution of this research is the design of a model that explicitly captures all of the semantics in the CLR. The rest of this document is structured as follows:

- Chapter 2 provides background information about semantic modelling, the .NET Framework and the motivation for our approach.
- Chapter 3 describes in detail the model we have designed to represent the semantic concepts in the CLR.
- Chapter 4 discusses how the semantic model we designed can be populated from source code.
- Chapter 5 presents an application that makes use of our semantic model to demonstrate its value by calculating a metric (CodeRank) that was previously not possible.
- Chapter 6 contains a discussion about our semantic model and its applications, as well as future work.
- Chapter 7 summarises the what we have achieved in this research.

Chapter 2

Background and Motivation

2.1 Object-Oriented Programming

Object-Oriented Programming (OOP) is a computer programming paradigm in which the software is modelled as objects interacting with each other. The first programming language that contained these ideas is generally considered to be Simula 67, which was developed during the mid 1960s. The Smalltalk language (designed in the early 1970s) was influenced by Simula and based on the design goal of creating a high-level language suitable for children and is recognised as the first pure OO language. Smalltalk's designer, Alan Kay, was the first to use the term "object-oriented". However, it wasn't until the C++ extension of C was developed in the early 1980s that object-oriented programming started to become widely used, when C++ was used by a large number of programmers and organisations. At this point, many programmers did not use C++ for its object-oriented features, but rather for basic abstraction and improved type checking compared with C (Bruce 2002).

As developers began to understand and use the object-oriented paradigm, the benefits of object-oriented programming were more widely experienced. Many programming languages were extended to include object-oriented features with varying degress of success, including ADA, Pascal and BASIC. This led to languages being designed to include object-oriented features from the ground up. These languages include Eiffel, Java and C#, as well as dynamically typed object-oriented languages like Python and Ruby.

2.1.1 Object-Oriented Convergence

The benefits of object-oriented programming are now well understood and object-oriented programming has become the dominant programming paradigm. Bruce (2002) observes that there seem to be "clear advantages of the object-oriented style in organizing and reusing software components." Although there are still some debates about exactly what concepts should be included in a modern object-oriented language, there has been enough convergence to lead one author to state there is "wide agreement on what fundamental features a modern programming language should have and how it should behave" (Chappell 2002). Object-oriented languages all support similar concepts of classes, inheritance, fields, methods and method invocation. This convergence of fundamental features can be seen in how similar the recent releases from Microsoft and Sun are, C# 2.0 and Java 1.5 respectively. Meyer (2000) provides a comprehensive list of criteria for an object-oriented language, which is not without controversy due to his inclusion of multiple inheritance as a desirable object-oriented language feature.

No single object-oriented programming language has become dominant. There is still argument over which of the many object-oriented languages is best. However, the major differences are often merely syntactic, rather than in the semantics of the language. Chappell (2002) observes that while language syntaxes differ, the semantic abstractions underlying most popular programming languages are almost identical. This convergence of object-oriented concepts across different languages led Chappell (2002) to ask "why not define a standard implementation of those semantics, then allow different syntaxes to be used to express those semantics?" Microsoft have addressed exactly this question as part of their .NET Framework

(described further in Section 2.3) by providing a runtime environment that is syntax and language independent. This runtime environment provides a common set of semantic underpinnings for a wide variety of languages, such as C# and VB.NET.

2.2 Static Analysis of Software

This section explains the motivation for static analysis and describes the main concepts of static analysis. Semantic modelling, a form of static analysis, is introduced and an existing tool for building a semantic model of Java is presented, along with its limitations and general applications.

2.2.1 Software Development is Difficult

Software engineering is difficult because of the nature of software. Writing software is a mental activity—it is not possible to see the product that is being built. Other challenges in industrial software development are size and complexity, while the software is continually changing. Often software projects are so large and complex that no one developer can understand it all. Of these challenges, McConnell (2004) believes "managing complexity is the most important technical topic in software development."

Software can not be understood solely by reading the source code. A higher-level *mental model* of the software's structure is needed. This mental model includes understanding what the components are and the relationships and dependencies between them. In an object-oriented programming language, there are a multitude of these component and relationship types. There also design forces at work and the developer needs to consider design principles and patterns as well.

Tools and techniques are needed to convey this structural information to the developer, to help them quickly build their mental model of the software. Unified Modeling Language (Object Management Group 2005*b*) is one such diagramming technique that helps developers understand the structure of their software. Such tools are useful for all developers and tasks, whether it is an experienced developer considering opportunities for refactoring or developers new to the project, seeking to understand the overall structure of the system.

Tools to show the structure of software need to have high quality data about the underlying source code. This requires some form of static analysis to extract a model of the software, which can then be used by tools to expose this structure to developers.

2.2.2 Static Analysis

Static analysis of software involves analysing the software (normally source code) to obtain information relevant to developers and tools, allowing them to better understand and manipulate the software they are writing. Static analysis happens at compile time and does not execute the source code, as opposed to dynamic analysis, which is performed at runtime. Static analysis is conventionally decomposed into the phases of *lexical analysis* to break the source code into tokens, *parsing* to group the tokens into a legal parse tree, followed by *semantic analysis* to understand the meaning of the parse trees. The focus of this research is on semantic analysis.

2.2.3 Semantic Modelling

The syntax of a language is the way the symbols are allowed to be combined (Slonneger & Kurtz 1995). Given source code, parsing can be used to create a syntax tree, which represents the structure of the symbols used in the code. Although this information is useful, it is not sufficient for helping software developers understand their code. Programmers do not think primarily in terms of the groupings of symbols they use to write their code.

The semantics of a programming language define the meaning of the syntactically-valid sentences in a language (Slonneger & Kurtz 1995). Examples of semantic concepts present in a typical object-oriented programming language are classes, methods, fields, inheritance, implementation and method invocations.

These are the concepts developers know and work with when writing object-oriented software and are the concepts we include in our model. For example, the declaration:

```
Foo f;
```

contains legally arranged tokens as part of a larger program. Syntactically, we know only that a variable identifier `f` follows a type identifier `Foo`. We do not know the scope of `f`, or what operations may meaningfully be performed on it. Similarly, we do not know where (or if) `Foo` is defined, and what the definition allows. Semantic modelling knows this is variable `f` of type `Foo` (which has been declared somewhere else) and models this relationship to make it explicit.

Semantic is an overloaded term in computer science. When we talk about semantic analysis and modelling, we mean the analysis of the structure of the software in terms of the programming language (for example, Java), rather than the semantics of the domain (for example, an inventory system). Semantic modelling of software is not related to the semantic web.

2.2.4 Java Symbol Table (JST)

This research project improves on earlier work, which shared the goal of building a semantic model to represent the structure of asoftware system. Irwin & Churcher (2003) developed a semantic model for the Java 1.4 programming language, named Java Symbol Table (JST). This tool recognises and builds a model of all the semantic concepts present in a Java parse tree including concepts like packages, classes, methods, constructors, fields, parameters and local variables. The relationships between these concepts (like inheritance, interface implementation and method invocation) are also included in the semantic model. The resulting model is a comprehensive model of a Java program, as shown in Figure A.1.

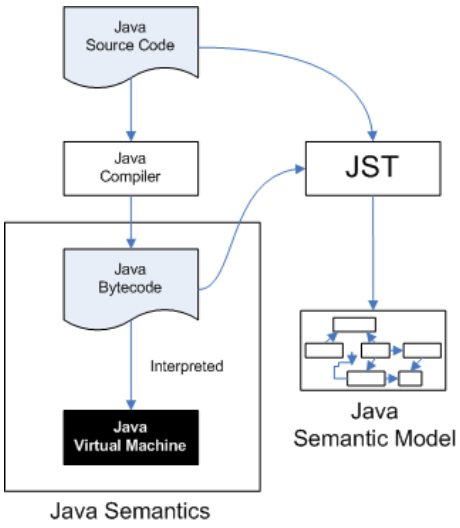


Figure 2.1: The Java Compilation and Execution Process

Java source code compiles to bytecode, which is then run on the Java Virtual Machine (JVM), as shown in Figure 2.1. JST builds the semantic model for a Java program using Java source code if possible and the Java reflection API otherwise. The semantic model produced by JST has proven useful for a range of applications, some of which are discussed in Section 2.2.5.

Limitations of JST

Although JST models all of the semantic concepts for a program written in Java 1.4, there are some useful relationships that are not modelled due to limitations with what can be expressed using Java 1.4. These limitations show opportunities where JST can be improved. A good example of this is code that uses the

Java Collections API. When a class `A` contains a `List` of class `B`, this is an important relationship between components `A` and `B` that JST cannot detect or represent because there is no way to write this relationship using Java, a `List` contains `Objects`. However, when a language contains generics, this relationship can be modelled.

The primary limitation of the semantic model produced by JST is that it only models the semantic concepts present in Java 1.4 and is highly specific to Java 1.4. Since JST was developed, Sun has released Java 1.5. This language includes new concepts that JST does not represent, for example, generics, enumerations and annotations (meta-data available at runtime).

2.2.5 Applications of Semantic Modelling

A semantic model of a software system does not have value to a programmer unless the data it contains is conveyed in a meaningful way. There are many applications for a semantic model including visualisation, metrics, heuristics and IDEs.

Metrics and Visualisation

Software metrics measure some aspect of software to increase understanding and "effectively manage the software development and maintenance process" (Moller & Paulish 1993). Although software metrics were first developed in the early 1970s for procedural code, some metrics have been adapted, while others have been created specifically for object-oriented software. The most well-known of these is the object-oriented metrics set proposed by Chidamber & Kemerer (1994).

Software visualisation is a useful technique to help developers understand large, complex software by applying information visualisation tools and techniques to software. Two useful techniques are abstraction (to highlight one particular aspect of a system) or the use of a metaphor, for example a city (Charters, Knight, Thomas & Munro 2002).

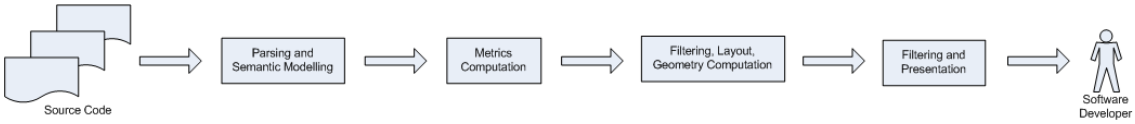


Figure 2.2: The Visualisation Pipeline

Metrics and visualisation both need a good source of data if they are going to benefit developers. The visualisation pipeline approach (Irwin, Cook & Churcher 2005), as shown in Figure 2.2, is a flexible and extensible technique for producing quality metrics and visualisations. This project fits into the semantic modelling phase of the pipeline, exposing the semantic structure of .NET source code for use by other tools, where previously only Java source code could be used. Although it is possible to calculate some metrics directly from syntax trees (for example, Weighted Methods per Class (Chidamber & Kemerer 1994)), a metric like Response for Class (Chidamber & Kemerer 1994) requires semantic knowledge because information is needed about what other objects are in scope and what methods can be invoked on them.

In order to demonstrate the value of our semantic model for software metrics and visualisation, we used the data from our semantic model for calculating a metric: CodeRank. CodeRank illustrates the value of a common a semantic model because by defining the metric once for the semantic model, we can calculate CodeRank for source code written in any .NET language. This application is described further in Chapter 5.

Heuristics

In object-oriented programming, there are many design maxims and heuristics for helping developers improve the quality of their code. One list of these is provided by Riel (1996). Some of these heuristics can't be quantified, for example *model the real world*. However, it is possible to detect, using a semantic model, whether many of the heuristics have been broken. For example *limit inheritance hierarchies to a depth of six* or a more general principle like *avoid cyclic dependencies*. These more concrete heuristics

combine well with software metrics. Automatically informing developers when their code conflicts with known design guidelines is beneficial for helping software developers write better code.

Integrated Development Environments (IDE)

Modern Integrated Development Environments (IDE) need some form of semantic model to represent the structure of the software being developed within them. This allows the IDE to provide different views of the source code, for example, UML. It also allows the IDE to be "smart" when performing tasks like refactoring. For example, if a method is renamed, the semantic model knows everywhere where that method is used. One example of such a model is the Java Development Tool API (JDT) (Eclipse Foundation 2005) which provides an API for accessing the structure of a Java program being developed within the Eclipse IDE. Commercial IDE development companies generally do not release information about the design of their underlying models.

Another example of an application of a semantic model is the collaborative software engineering environment (CAISE) developed by (Cook, Churcher & Irwin 2004). This uses an enhanced version of JST, shared using a client/server architecture among all tools in CAISE. One of the primary benefits of having a comprehensive semantic model for a collaborative project is that it can be used to find when developers are working closely together at a semantic level.

2.2.6 Related Approaches for Modelling the Structure of Software

Our idea of a semantic model for software is not the only approach for representing the structure of source code. Some other approaches are described in this section including Abstract Syntax Trees (AST), reflection and XMI.

Abstract Syntax Trees

When parsing source code, a parse tree can be used to show how the structure of the tokens conform to the underlying grammer of the language. However, parse trees are not normally in a "useful format for further processing" (Grune, Bal, Jacobs & Langendoen 2000). Abstract Syntax Trees (AST) approximate parse trees by removing unimportant "superficial distinctions of form" (Aho, Sethi & Ullman 1986). Semantic information can be attached to syntax tree nodes (Grune et al. 2000) and links can be made between nodes to represent semantic relationships between the syntactic components. Although the main use of an AST is in compilers, it can also be used to represent the structure of software. For example, JDT (Eclipse Foundation 2005) is based on binding semantic information to an AST. However, an AST is still based on the syntax, which is of more interested to compilers, rather than the semantic structure, which is of direct interest to software developers. This is why we have chosen to design a single model that is focused solely on semantics, not syntax.

Reflection

Reflection is a useful way to find out about types and objects in a program at runtime. The .NET Framework 2.0 reflection library has been updated to include support for generics and also allows local variable information to be extracted. Although reflection is often used by tools to create objects and invoke methods dynamically, it can be used solely to extract information about the types and relationships within a .NET assembly for use by other tools. The UML class diagram showing the design of the .NET reflection framework is shown in Figure 2.3.

In this model, every type is represented by `Type`, including classes, interfaces, arrays, value types, enums, type parameters, generic types, open and closed constructed generic types. Consequently, a multitude of methods are available for helping the programmer figure out exactly what type they are dealing with. It is also possible to call methods that are illegal, if you don't have a reference to the type you expected.

Our semantic model is similar to reflection, in that it captures the same concepts. However, reflection was not designed for general static analysis and has a number of limitations when used for this purpose:

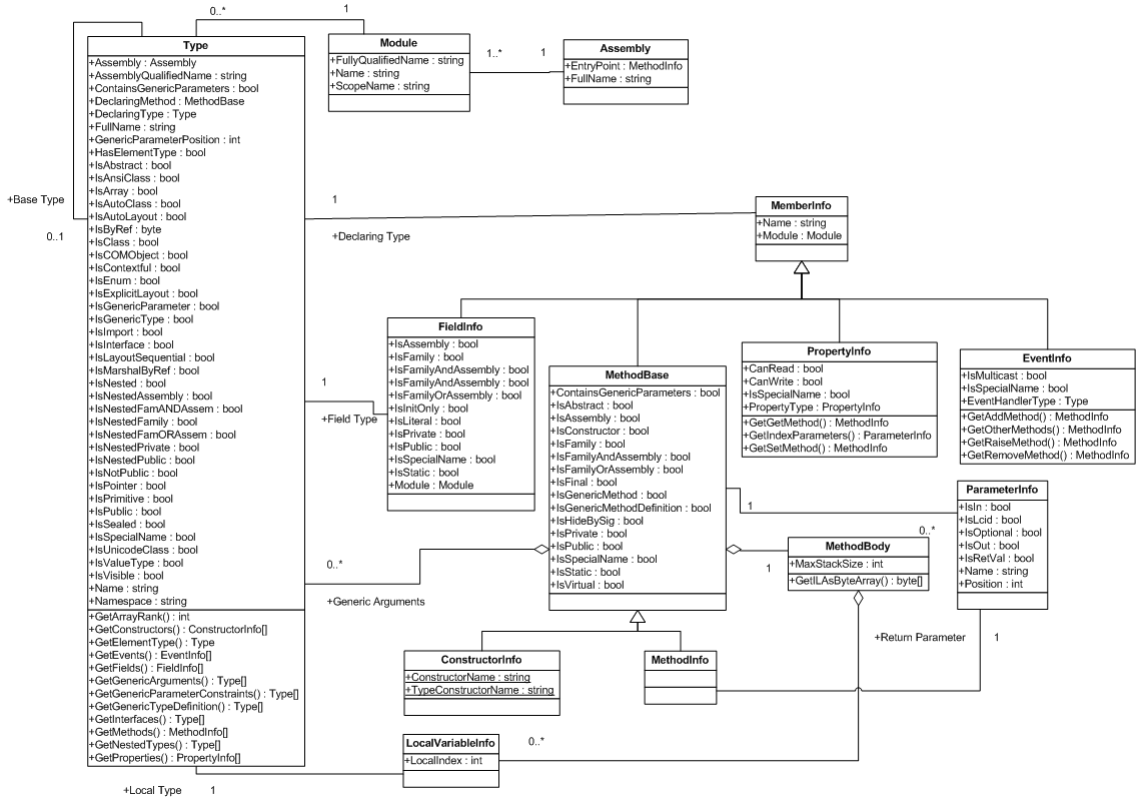


Figure 2.3: Class Diagram of the .NET Reflection Framework

- It is not possible to get information about private types and members. A lot of code written by software developers is private and if a semantic model is going to be informative in helping developers understand software, then it needs to model *all* code.
- It is not possible to "see" inside methods. Although .NET 2.0 improves this by providing information about local variables, this is still not enough for a semantic model. Apart from local variables, we are interested in what methods each method invokes and what fields are accessed. This information is of interest to software developers when analysing dependencies of coupling and cohesion in their code.
- By representing everything as a `Type`, reflection does not fully expose the semantic differences of the source code. This needs to be constructed first by any tool that uses reflection directly.

XMI

XML Metadata Interchange (XMI) (Object Management Group 2005*a*) is a standard originally intended for exchanging the structure of software (particularly UML diagrams) between development tools. This is different motivation from our goal of modelling the semantic structure of source code. Like reflection, XMI does not model some of the data we are interested in, like the internals of methods.

2.3 The Microsoft .NET Framework

The semantic model produced by JST has some limitations that were discussed in Section 2.2.4. Some of these limitations could be improved by modelling Java 1.5 (rather than Java 1.4) source code. Instead, we

have chosen to use a similar approach by designing a semantic model to reflect the semantic concepts in the .NET Common Language Infrastructure (CLI). There are many reasons why we have chosen to model .NET, the most important of which is that the .NET CLR explicitly defines common semantics that are independent of language or syntax. The following sections give a brief overview of the important parts of the .NET Framework relevant to this research project.

2.3.1 Common Language Runtime (CLR)

The CLR is Microsoft's implementation of the CLI. The CLI is an ECMA (ECMA 2005*b*) and ISO standard that allows "applications written in multiple high-level languages can be executed in different system environments without the need to rewrite those applications to take into consideration the unique characteristics of those environments" (ECMA 2005*b*). The CLR includes a virtual machine for running Microsoft's Intermediate Language (MSIL). Although the CLR is similar to Sun's JVM and MSIL is similar to Java bytecode, there are a number of major differences that make .NET a "fundamental change" in the state of the art of software development (Meyer 2002).

Miller & Ragsdale (2003) state that the .NET CLR started with the goal of a multi-language standard, allowing programmers to be able to choose the most suitable programming language for a given module. Furthermore, they thought that these modules should not only be able to interoperate with each other but also that that should be able to run on any platform without needing to be rewritten or recompiled.

2.3.2 Meta-Data and Microsoft Intermediate Language (MSIL)

When source code written to run on the CLR is compiled, a Windows Portable Executable (PE) file is produced. This file contains meta-data describing the types and members in the program and MSIL instructions, contained within the method definitions. The MSIL instructions are low-level, assembly instructions that are executed in a stack-based environment. They include a "Turing complete" (ECMA 2005*b*) set of basic instructions like arithmetic and branching operations. However, unlike other assembly languages, MSIL also includes object model instructions to provide a "common, efficient implementation of a set of services" used by many higher-level languages. This includes instructions to deal with creating new objects, loading and storing fields in an object and calling virtual methods on objects. This means these object-oriented semantics are built into the MSIL and CLR itself.

The compilation and execution process for .NET is shown in Figure 2.4. Language independence is achieved by the compiler for each language compiling the source code to meta-data and MSIL, rather than native code.

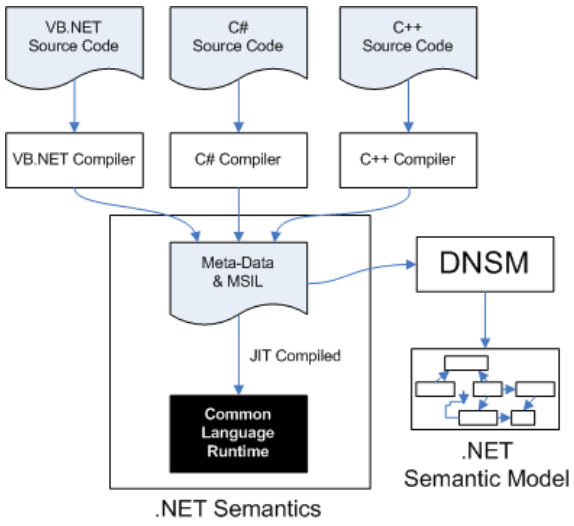


Figure 2.4: The .NET Compilation and Execution Process

This diagram looks similar to the Java compilation and execution process, shown in Figure 2.1, however, there are some important differences. The most important of these is that Java's bytecode was design to support Java only, while .NET was designed with the goal of being able to support multiple languages. The JVM is excellent for compiling Java to, but has been shown to be suboptimal other languages (Meijer & Gough 2001) because it does not have the ability to include languages features not present in Java. Microsoft have been working with many different language implementors so that the CLR "supports many features that have surfaced in only a few programming languages, for example, pointers" (Venners & Eckel 2003*a*). Figure 2.4 also shows that our .NET Semantic Model tool (DNSM) builds the semantic model solely from the MSIL and meta-data and does not use the original source code like JST.

2.3.3 .NET Semantics

The CLR is designed primarily to run statically, strongly typed object-oriented languages. However, even for these similar languages, there are a number of subtle differences. The .NET designers have attempted to define a "suitable common base that can serve as a bridge" (Meyer 2002) between mainstream object-oriented languages, as well as "novelty" languages. The semantics are encapsulated within the MSIL and CLR as shown in Figue 2.4. These semantics are described in detail in the 550 page standard in natural language. However, we want to explicitly and more formally model these common .NET semantics using object-oriented concepts, so that the model is accessible to other tools.

Despite the convergence of object-oriented languages to the point where defining the common semantics is possible, there are still points of disagreement from some commentators. For example, Meyer (2002) thinks that while .NET object model is an improvement on C++ and Java, there are some mistakes including single inheritance instead of multiple inheritance and allowing methods to be overloaded.

2.3.4 Common Type System (CTS)

As well as allowing developers to define their own classes, all programming languages have some data-types already built into the language. These include types like integers, floating point types and string. Before .NET, each language defined its own type system. In .NET, the CTS establishes a framework that enables cross-language integration, type safety, and efficient code execution (ECMA 2005*b*) by defining common types, for example, how many bits each integer has. All of these types are syntax-independent. This is an important part of being able to allow languages to interoperate. (Chappell 2002) says

> Suppose you choose to define the core abstractions for a programming model without mapping them to any particular syntax. If the abstractions were general enough they can then be used in many programming languages. Rather than inextricably mingling syntax and semantics, they could be kept separate, allowing different languages to be used with the same set of underlying abstractions. This is exactly what's done in the CLR's CTS.

2.3.5 Common Language Specification (CLS)

The Common Language Specification (CLS) is a set of rules that restricts the CTS to help language interoperability. For example, rule 4 states that any publicly accessible types and members have to be case insensitive. This allows languages that are not case sensitive to interoperate with languages that are case sensitive. Eric Gunnerson (Venners & Eckel 2004*b*) summarises the CLS are a description of "how languages should behave so they can play nicely together I think of the CLS as an interoperability spec."

2.3.6 C# Language

C# is a statically-typed object-oriented programming language designed to run on the CLR. The link between C# and MSIL is very close, and it is not clear exactly which language came first. C# "directly reflects the .NET object model" (Meyer 2002).

Although official literature from Microsoft states that C# was derived from C and C++ (with no mention of Java), it is clear that Java language features have had an influence on the design of C# (Mok 2003).

C# has selected some of the successful features of Java, as well as combining new language features in designing C#. This means that in our semantic model for .NET, there are a lot of similarities with the JST semantic model.

2.3.7 Motivation for using .NET 2.0

The CLI is continuing to evolve. The third edition (version 2.0) was ratified in June 2005 and the .NET Framework 2.0 is due for release by Microsoft in November 2005. Although there have been a lot of changes to C#, the only major change in the CLI standard is the additional of generics and related generics libraries. We have chosen to build a semantic model for .NET 2.0 because by including generics we have a more comprehensive model of the structure of the software. By modelling the semantics of the CLI, our model can be used for source code written in any language that compiles to .NET.

2.4 Generics

This section describes the new generics language feature in .NET and Java and explains why generics improves the power of static analysis when modelling the structure of source code.

2.4.1 Description

In a programming language, generics allows types and methods to be "parameterized by the types of data they store and manipulate" (ECMA 2005*a*). A simple example of where this is useful is when you want to store a collection of objects, for example `string`s. Anders Hejlsberg (Venners & Eckel 2004*a*) points out that in a language with no generic support, there is a tension between using an array or a collection like `List`. Using an array gives you strong typing because you can declare an array of strings:

```
private string[] names;
```

However, if you don't know in advance the size of the collection, you need to write the code to adapt the size of the array. Using a `List` is more convenient because you don't need to be concerned with the number of objects.

```
private List names;
```

Unfortunately, you lose type safety because the list can contain any `object` (not necessarily a `string`) and you also need to downcast when getting objects back out of the `List`.

Generics solves this problem by allowing types to be declared with type parameters. For example, a `List` has one type parameter (`public class List<T> { ... }`), which represents the type of objects allowed in inside it. So, a list of strings can instantiate the generic type like:

```
private List<string> names;
```

The methods on the `List` also include the type parameter as part of their signature. For example, some of the most useful methods in `List<T>` look like:

```
public void Add(T item)
public bool Remove(T item)
```

The type parameters declared can be used almost anywhere throughout the type definition, just like any other type.

When a generic type is used, it needs to be *instantiated*. That is, each type parameter needs to be given an argument. This argument could well be another type parameter. From this point of view, generic type declarations are similar to functions in that they take arguments for each parameter and return new types (Bruce 2002). More accurately, generic types can be seen as functions from types to types (Bruce 2002). Once a generic type has been instantiated, the type parameters are substituted with the given arguments, so for example, the methods on the instantiated type `List<string>` would become:

```
public void Add(string item)
public bool Remove(string item)
```

The advantage of this is that there is now strong type checking and no downcasting is necessary. Other benefits of using generics include (especially for large programs) improved readability and robustness (Bracha 2004) and in .NET, there is also a performance gain when using generics.

2.4.2 History

Although generics is a new language feature in .NET and Java, the concept of generics is not new. LISP was the first programming language to have type parameterization (Eckel 2005). Eiffel was one of the early commercial compilers that supported "bounded polymorphism" in an object-oriented language (Bruce 2002).

C# generics appear similar to C++ templates on the surface, but the way they are implemented is completely different. Anders Hejlsberg summarises the differences well (Venners & Eckel 2004*a*):

> C# generics are really just like classes, except they have a type parameter. C++ templates are really just like macros, except they look like classes. C++ templates are actually untyped, or loosely typed. Whereas C# generics are strongly typed.

The strong typing of C# generics means there are some restrictions on how generic types can be used, compared with C++ templates. Microsoft (2005) provides a comprehensive list of the differences between C# generics and C++ templates. An example of a difference relevant to this research is that C# does not allow the type parameter to be used as the base class for the generic type like this:

```
public class Foo<T, S> : T { ... }
```

2.4.3 Generics in the CLI

One of the best things about the way generics are implemented in .NET is that there is "native support for generics in the MSIL and the CLR itself" (Lowy 2005). When a generic class in C# is compiled to MSIL, it is represented like a normal class, but with type parameters and some extra metadata. This makes it easy for us to extract out the generic information when populating our semantic model. This approach contrasts directly with Sun's implementation of generics for Java 1.5.

2.4.4 Java 1.5 Generics

The Java implementation of generics was constrained by the design goal that the JVM remained unchanged, to allow backward compatability with legacy code. It also leads to the tricky situation of being able to write code that it is a mix of generic and non-generic code. This means that unlike the CLR, the JVM knows nothing about generics. To get around this, Java uses a technique called *type erasure*. At compile time, the compiler removes any type parameters and replaces them with `java.lang.Object` and automatically does any casting that is needed. There are a number of disadvantages of this approach including no performance gains possible when using generics and at runtime, you don't know what the original type parameters were. In .NET, if you have a `List`, you can use the reflection library to find out what type of objects the list contains. In Java, this is not possible because all type information is erased at compile time. This makes static analysis of Java difficult because unless the model is built directly from source code, the generics information is not available.

2.4.5 Generics in the Semantic Model

The main advantage of generics in .NET is that it gives static type safety. However, this also allows a big improvement in the power of static analysis. Using a non-generic collections makes static analysis of the software's structure difficult because there is no simple approach possible to detect composition and aggregation relationships between types. Generics allows us to model these relationships in our semantic model, which overcomes one of the main limitations of the JST semantic model.

2.5 Research Objective

The objective of this research project is to take the common semantics described by the CLI standard (and implemented by Microsoft in the CLR) and model them explicitly, so that other tools can get access to the data. Although this is a complex data modelling problem, the benefits are that new relationships between components can be modelled and any tools that use the semantic model can be developed once and used for source code written in multiple .NET programming languages

Chapter 3

Design of a Semantic Model for .NET

This chapter describes the primary contribution of this research: a new semantic model that captures and formalises the common semantic underpinnings of .NET languages. The primary benefit of such a model is that it explicitly exposes the semantic constructs of a program to software engineering tools. The complete semantic model for .NET is shown in Figure A.2. Some .NET concepts echo those of Java, so we first describe parts of the model that are conceptually similar to the JST semantic model. The remainder of the chapter describes entirely new extensions to the model. Generics provided the most complex and challenging modelling task and so is presented in its own section.

3.1 Java and .NET Common Language Features

The JST tool that produces a complete semantic model for Java 1.4 was introduced in Section 2.2.4. A simplified version of this model is shown in Figure A.1. As mentioned previously, .NET 1.1 is very similar to Java 1.4 and so many of the semantic concepts are similar. All the basic semantic concepts like single inheritance of classes and multiple implementation of interfaces, classes containing methods and fields, methods containing parameters and exception handling are almost identical. Major differences include the access type and modifiers, arrays and packages.

The similarities meant that we used JST as the starting point for designing a semantic model for .NET. The basic structure of the JST semantic model was kept. Every language concept is a declaration (`Decl`) and almost everything else is either a type (`TypeDecl`) or has a type (`TypedDecl`). This is shown in Figure 3.1[1]. Each declaration has an owner and a unique name.

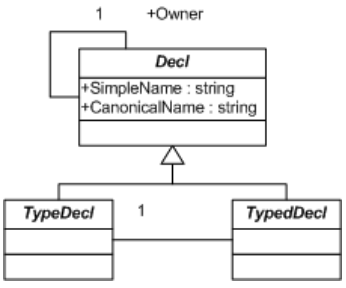


Figure 3.1: The Root Classes of the Semantic Model

[1]The UML diagrams presented in this section are simplified part of a larger and more complex design. Note that C# properties are represented as public fields and all collections are represented as arrays (which is not necessarily how they are implemented.

3.1.1 Classes

The most fundamental concept in object-oriented programming and therefore our semantic model is the idea of a class. JST uses the concept of a `UserType`, that is specialised into either a `ClassType` or `InterfaceType`. The class and interface concepts both exist in .NET, however .NET also includes structs. A struct is similar to a class, with the following exceptions:

- Structs do not support inheritance.
- Structs are allocated space on the stack, rather than the heap. This means there are performance benefits for using structs instead of classes.

In the CLI, there is no explicit difference between interfaces, classes and structs. These three are all declared as a `.class` in the meta-data. The difference between these semantic concepts is in the class header. Interfaces have the attribute `interface`, structs extend from the .NET Framework library class `System.ValueType`, while everything else is just a normal class. We chose to explicitly model interfaces, structs and classes as separate entities because these entities differ semantically and are of direct interest to software developers. This approach allows us model the semantic constraints of these entities better and expose the semantics in the CLI. For example, a class can implement multiple interfaces, (each of which is a class with the `interface` modifier).

Classes in .NET can have four kinds of members: fields, methods, properties and events. Java does not include properties or events. Classes can also contain nested types. Our design for representing classes, interfaces and structs is shown in Figure 3.2. This diagram does not include generics, which will be discussed in Section 3.2.

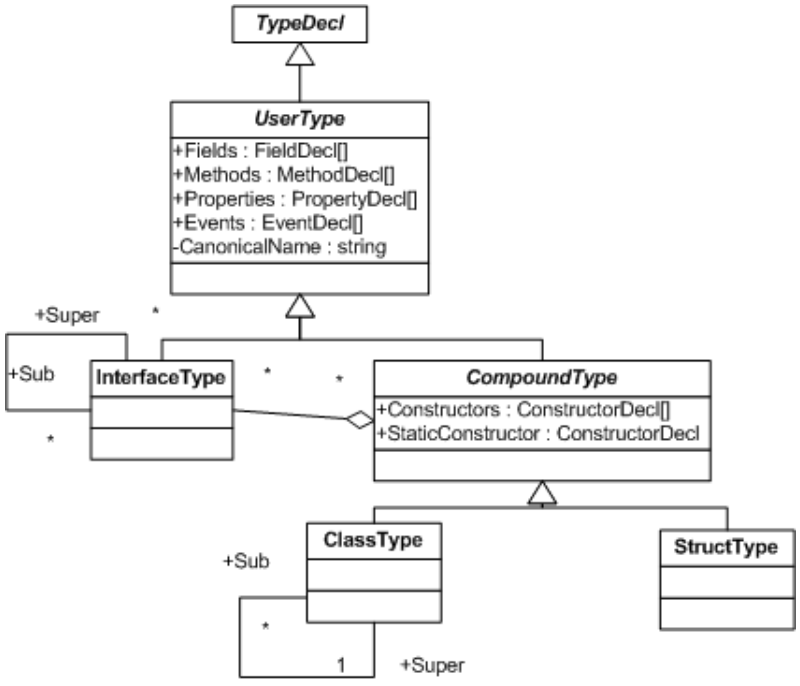


Figure 3.2: Design for Classes, Interfaces and Structs

3.1.2 Methods and Constructors

Methods are a fundamental part of any object-oriented programming language. A method is a "member that implements a computation or action that can be performed by an object or class" (ECMA 2005*a*). In the

CLI, the concept of an instance constructor (which gets called to initialise a new instance of a class) is represented in the meta-data as a method with the special name `.ctor`. We followed the JST semantic model by explicitly modelling constructors separately from methods. Again, this better reflects the semantics of these two language concepts because there are significant differences. For example, constructors cannot be overriden. Common features of methods and constructors are encapsulated in the `OperationDecl` class, as shown in Figure 3.3.

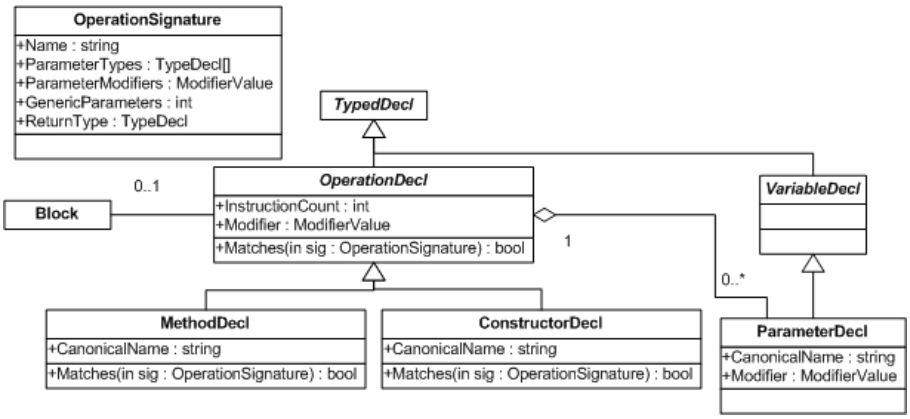


Figure 3.3: Design for Methods and Constructors

A major difference between methods in Java and .NET is that in Java, all instance methods are virtual. In other words, sub classes can override a method, unless it has been declared with the `final` modifier. In .NET, by default all methods are non-virtual. Programmers must declare a method virtual if they want to allow it to be overridden. This was done for performance and versioning reasons (Venners & Eckel 2003*c*). In our semantic model, a virtual method has `virtual` as part of its modifier. This information needs to be used when resolving method calls for determining what methods are inherited. In both Java and .NET, constructors are not inherited by subclasses.

Static Constructors

Static constructors get called automatically before any instance of the class is created or before any static members of the class are referenced. This is just like the static initialiser blocks in Java (which are not modelled by JST). The main difference between static constructors and static initialisers is that Java allows multiple static initialiser blocks to be declared in each class, while .NET only allows only one static constructor.

In the CLI, static constructors are represented as methods with the special name `.cctor`. Static constructors can not have any parameters, so there can only be one per class (no overloads are possible). In our semantic model, a static constructor is represented as an instance of `ConstructorDecl`. Each `CompoundType` contains zero or one static constructor.

Operation Signatures

The CLI allows method names to be overloaded on the number, types of the parameters, modifiers on the parameters and the return type (ECMA 2005*b*)[2]. We need to know the method signature when looking up a method within a class. `OperationSignature` is a class that represents everything that comprises a method signature. An object of this type can then be passed around when looking up methods and constructors.

[2]The CLS only allows methods to be overloaded on the number and types of the parameters and the return type for two specific operator overloads.

Parameters

Our design for parameters is shown in Figure 3.3. In the CLI, each parameter has a type and so we have modelled parameters as a subclass of `VariableDecl`, like in the JST model. Parameters also have a name and modifier.

Operator Overloading

Operator overloading allows programmers to redefine the behaviour of built-in operators such as `+`, `--` etc. Although they are convenient as a shortcut for programmers, they can be a source of confusion because it is not clear what method is being called. For this reason, many languages do not support operator overloading, for example, Java. The CLI specification states that because most languages do not have operator overloading, languages that do support it need to provide another way for that operation to be performed. An overloaded operator is just a static method with special syntax and rules about the number and type of parameters, as well as the return type. For this reason, we have not explicitly modelled operator overloading in our semantic model, they will appear as normal static methods.

3.1.3 Fields

A field is a member that "represents a variable associated with an object or class" (ECMA 2005*a*). Each field has a name and a type. In the JST semantic model, classes and interfaces contain fields, however, the CLI allows fields to be globally within an assembly, within a module or inside a class declaration (class, interface or struct). Apart from access modifiers, possible modifiers for a field are `initonly`, `literal` and `static`.

We have put fields in the same place as the JST semantic model, as a subclass of `VariableDecl`. This is shown in Figure 3.4.

3.1.4 Blocks and Statements

The JST semantic model goes down to the block level, which is the smallest scope in Java. Each block knows any local variables declared inside it and references made in the block to fields and methods or other `TypedDecl`s. Any further detail needed can be found by going down to the parse tree level, which is stored as part of the semantic model. In our .NET semantic model, we don't have the full parse trees. However, we have still decided to only go down to the block level and not model any lower constructs, like statements. This is because the statements a developer writes are translated into the stack-based assembly instructions of MSIL. These are of little use to developers because they are often different from what the developer originally wrote due to the instruction set available and compiler optimisations. Also, we are more interested in modelling the structure of the software, than the very low-level details. Our design for block is shown in Figure 3.4.

The major design change from the JST semantic model is that although all scopes are semantic concepts of interest, the CLI does not allow blocks to be nested. When a programming language like C# is compiled to MSIL, all local variables are moved to the top of the operation declaration and all loop statements are converted to assembly level branch instructions.

Local Variables

As described in the previous section, local variable declarations are moved to one place at the top of the method declaration and given a unique identifier (an index) instead of the name used in the original code. Our design for local variables in our semantic model are shown in Figure 3.4.

Exception Handling

C# has very similar exception handling to Java built into the language. This structured exception handling is also built into the CLI. We have modelled this very similar to JST with a `CatchBlock` class which

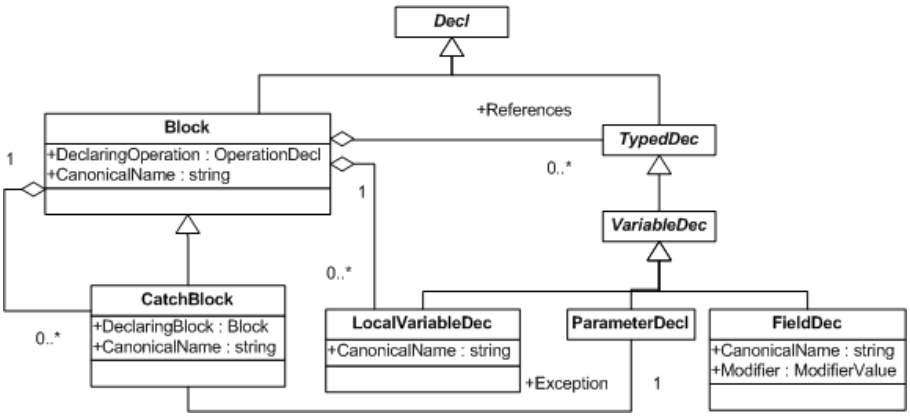


Figure 3.4: Design for Blocks

is a normal block with a parameter containing the exception type. This exception type must be of type `System.Exception`.

3.1.5 Arrays

The CLI standard makes the distinction betwen two types of arrays (ECMA 2005*b*):

Vectors "single-dimension arrays with a zero lower bound." Vectors have good performance because they are directly supported by instructions in MSIL.

Arrays all other arrays are a subclass of `System.Array`. These *rectangular* arrays have a *rank* which is the number of dimensions, a type and the lower and upper bound for each dimension (arrays don't to have start at zero).

There is little semantic difference between vectors and arrays. Rectangular arrays can be represented as arrays of arrays, where each array is the same size. We chosen to follow the way JST modelled arrays by modelling all arrays as being jagged. This uses the decorator pattern (Gamma, Helm, Johnson & Vlissides 1995) to wrap up a type with an array. This allows us to easily model arrays of arrays.

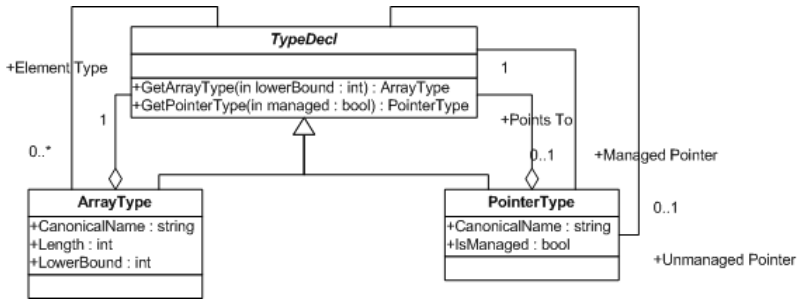


Figure 3.5: Design for Arrays and Pointers

The design for arrays in our semantic model is shown in Figure 3.5. The main change from JST is that we have added the lower bound because unlike Java, arrays don't have to start from zero. `TypeDecl` contains a method `GetArrayType` for getting an array of a specific type and lower bound. This method is implemented using lazy instantiation, an `ArrayType` is only created if it is needed.

3.1.6 Modifiers

Like Java, the CLI allows types and member declarations to have modifiers, which specify, for example, access levels and inheritance details (e.g. `abstract` and `virtual`). In JST, a java library class (`java.lang.reflect.Modifier`) was used to represent modifiers. In the .NET reflection library, there is an enumeration for each type or member that can have a modifier. For example:

- `EventAttributes`
- `FieldAttributes`
- `GenericParameterAttributes`
- `MethodAttributes`
- `ParameterAttributes`
- `TypeAttributes`

In our semantic model, we have only one `ModifierValue` enumeration that stores all combinations of modifers. It may be a better design to split this enumeration up like the .NET reflection library so only legal modifiers are allowed on each declaration. Each type or member has a `ModifierValue` field that stores the modifer. There is also a `Modifier` class that provides static methods that take a `ModifierValue` as a parameter, for example, `bool IsPublic(ModifierValue mv)`.

3.2 Design of Generics

An overview of generics in .NET was given in Section 2.4. The parts of our semantic model relevant for generics is shown in Figure 3.6. The semantic concepts of generics in the C# language and MSIL are almost identical. This section describes our design for modelling the semantics of generics defined in the CLI.

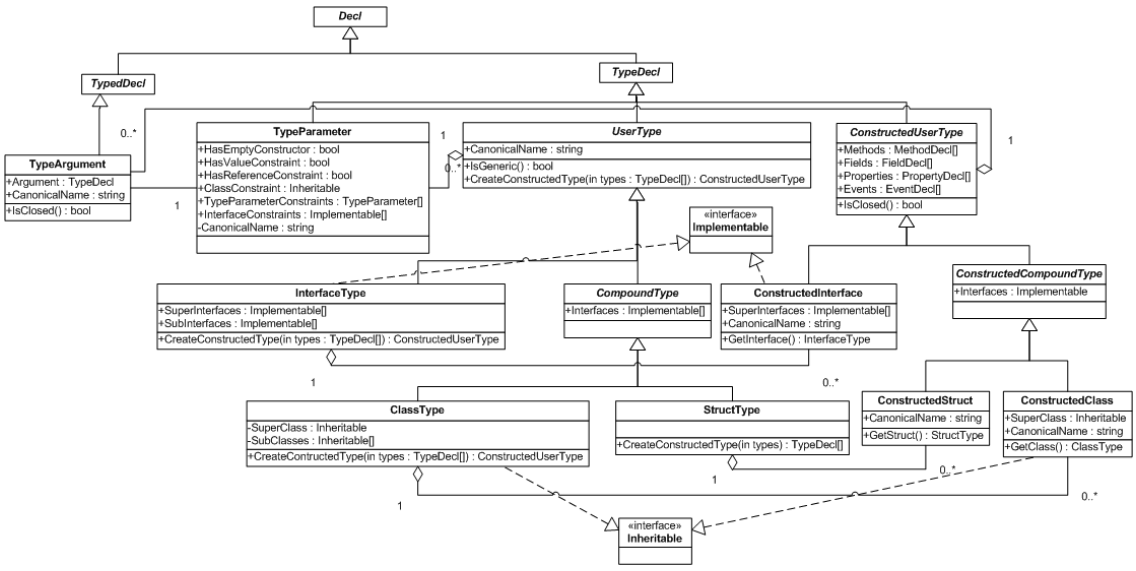


Figure 3.6: Design for Generics

3.2.1 Generic Types

"A generic type definition is one that includes generic parameters." (ECMA 2005*b*) We chose not to model generic and non-generic type definitions separately. Any class in the CLI can have generic parameters declared. A non-generic class is simply one with zero type parameters. In our semantic model, a `.class` corresponds to `UserType`, that is, any class, struct or interface. Generic types are represented as a `UserType`, which have a collection of `TypeParameters`. For non-generic `UserTypes`, this collection will be empty. The `IsGeneric()` method is also provided for testing whether a type is a generic type declaration.

3.2.2 Generic Parameters

Each generic parameter declared for a type has "name and an optional set of constraints" (ECMA 2005*b*). This has been modelled as `TypeParameter` in our model. There are a number of different types of constraints that can be placed on a generic parameter. The C# specification (ECMA 2005*a*) groups these constraints into three groups:

Primary Constraint (optional) can be one of:

- A `ClassType`. When instantiated, the argument will have to be this type or a subclass of it.
- The keyword `class` which only allows the parameter to be instantiated as a reference type.
- The keyword `struct` which only allows the parameter to be instantiated as a value type.

Secondary Constraints can be zero or more of either:

- An `InterfaceType`.
- A `TypeParameter`.

Constructor Constraint (optional):

- The keyword `new()`. This constraint guarantees that the type parameter will have a default constructor.

Our design for `TypeParameter` is shown in Figure 3.6. The `class`, `struct` and `new()` keywords have been modelled as boolean properties, while the other constraints are modelled as relationships. Note that it is possible to have more than one primary constraint in our model, however, as we will always be building the model from compiled source code, we do not see this as a problem. A type parameter can be used in the generic declaration almost anywhere a type can, so we made `TypeParameter` a subclass of `TypeDecl`.

3.2.3 Generic Methods

Like type definitions, methods are also allowed to have type parameters. We have taken a similar approach to types by giving `MethodDecl` a collection of type parameters. For non-generic methods, this collection will be empty. The `IsGeneric()` method is also provided for testing whether a method declaration is a generic method declaration. This design is shown in Figure 3.7.

3.2.4 Type Arguments

Generic type declarations are instantiated with a type. For example, `List<string>` instantiates the `List<T>` type declaration with `string`. Each type argument knows what type parameter it is instantiating and what type it has been instantiated as.

Type parameters can be instantiated with any type (including another type parameter). The CLI standard makes the distinction between open and closed types (ECMA 2005*b*):

Open the type contains at least one generic parameter e.g. `List<S>`

Closed the type contains no generic parameters e.g. `List<int>`

This has been modelled as the `IsClosed()` method..

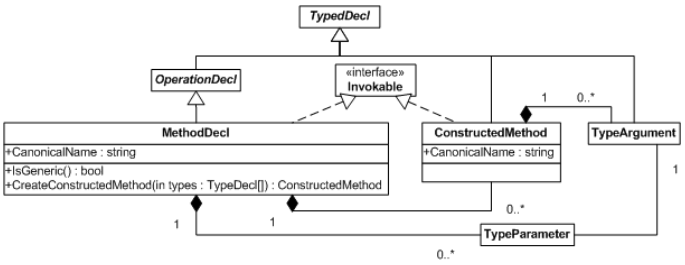


Figure 3.7: Design for Generic Methods

3.2.5 Instantiating Generic Types

A generic type is instantiated with an argument for each generic parameter declared. These may be any type, including type parameters and other instantiated types. This presents a difficult modelling problem. We initially created a `ConstructedUserType` class to represent instantiated types. Each `ConstructedUserType` knows what user type it instantiates. Each (generic) `UserType` has a collection of instantiated types. Each instantiated type has a collection of arguments (Section 3.2.4).

We found that simply having one constructed type to represent constructed types was insufficient. For example, you still need to look up members (methods, fields etc.) on these constructed types. Different constructed types have different members, for example, constructed interfaces don't have constructors. For this reason, we chose to make a parallel hierarchy reflecting the `UserType` hierarchy as shown in Figure 3.6.

One disadvantage of our parallel hierarchy is that there is no super type for similar semantic concepts. For example, the base class of `ClassType` is another `ClassType` object. However, the base class of a type could also be an instance of `ConstructedClassType` like:

```
public class A : B<int> { ... }
```

To overcome this, we introduced the interfaces `Inheritable`, `Implementable` and `Invokable`. This allows each `ClassType` to have a base class that is an `Inheritable`, which in this case can either be a `ClassType` or `ConstructedClassType`.

Garcia, Jarvi, Lumsdaine, Siek & Willcock (2003) provide a good description of the two possible approaches for a compiler to implement generic code:

Heterogeneous Translation specialised code is generated for each instantiation. Ada and C++ use this approach to produce specialised code for each instantiation of a generic type. This translation produces efficient code but can result in a large amount of generated code if there are a lot of instantiations.

Homogenenous Translation general code is shared for all instantiations. Modula 3, ML and Java use this approach where the same piece of code is used by all instantiations. In Java's situation, the code is kept general by using `Object` instead of the actual generic instantiation.

C# generics uses both translation methods by specialising *and* sharing code. For type instantiations that are instantiated with value types, a specialised copy of executable native code is generated. This allows efficient execution of the code. When instantiated with a reference type, one copy of the code is generated which can be shared because they are "representationally identical" (Venners & Eckel 2004*a*). All of this code generation is done at runtime by the JIT compiler. The system checks if the code for an instantiated type has been generated yet, if not, it generates it.

Our initial approach to modelling instantiated types was homogenous translation. Using this method, each instantiated type knew what generic type it was instantiating and with what arguments. When looking up something about a member in an instantiated type (for example, the return type of the `Foo()` method) the instantiated type would look up the method in the generic declaration and then dynamically substitute for any type parameters with the appropriate type argument.

However, we found when using the semantic model, either populating it or using it in an application, this approach was not satisfactory because we often needed a reference to an instantiated member. Although it could be possible to use a decorator pattern (Gamma et al. 1995) to decorate a method, we have chosen to explicitly generate all the members when instantiating a type. This is a simpler design and sufficient for modelling instantiated types. This is similar to the heterogeneous translation approach. When an instantiated type is created, new instantiated members (methods, constructors, properties etc.) are also created and instantiated where necessary. This design is shown in Figure 3.6.

Instantiation of generic types is controlled by a factory method (Gamma et al. 1995) in each generic type declaration. This allows subclasses of `UserType` to control what `ConstructedUserType` type to create. Also, instantiated types are only created if they don't already exist.

3.2.6 Instantiating Methods

Methods are instantiated in a manner similar to generic types. Each generic method has a factory method (`CreateConstructedMethod` to control instantiation. Each instantiated method contains a deep copy of the method it is instantiating, with the type parameters replaced with the given type arguments.

3.3 Design of .NET Specific Language Features

3.3.1 Assembilies and Modules

Assemblies are the unit of deployment in .NET. An assembly has a *manifest* that specifies the version, name, culture and the modules the assembly contains and uses. A module is a single file containing "executable content in the format specified here" (ECMA 2005*b*). Our design for assemblies and modules is shown in Figure 3.8. We have only modelled concepts important to the software's structure. That is, an assembly has a name and associated modules while a module has a name and associated types.

3.3.2 Enumerations

An enumeration type "defines a type name for a related group of symbolic constants" (ECMA 2005*a*). C# uses the C style syntax for declaring enums e.g.

```
enum Weekday {Mon, Tue, Wed, Thu, Fri, Sat, Sun}
```

In C#, each enumeration has an accessibility modifier, a name, an optional type (any primitive integer type), and finally the members of the enumeration (optionally initialised). Each enumeration declaration is a class and derives from the `System.Enum` class, which gives enumerations useful methods.

The CLI does not contain an explicit enumeration type. Enums are represented as a class with an immediate base type of `System.Enum`. These enum types also have some restrictions (compared with normal value types) (ECMA 2005*b*):

- Enums classes are only allowed fields as members, no methods, constructors etc.
- Enums classes can not implement any interfaces.
- Eums can have only one instance field (which is to be the the underlying type of the enum).
- All other fields have to be static and literal.

The standard states these restrictions mean that enum types can be implemented very efficiently. We have captured these semantics of enum types separate from a `ClassType`, as shown in Figure 3.8.

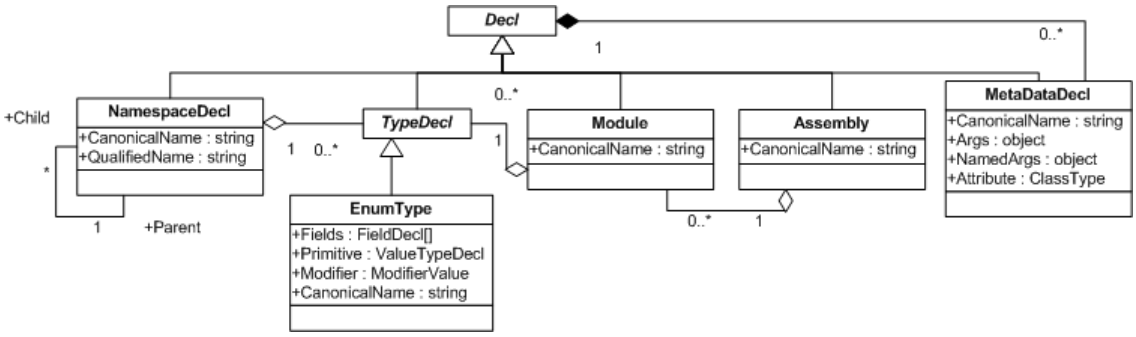


Figure 3.8: Design for Assemblies, Modules, Namespaces, Attributes and Enumerations

Criticism of .NET Enumerations

One of the advantages of using .NET CLR is that it is type-safe. However, the way enums are implemented, they are basically just integers grouped together in a class. This means there is no more type safety than if static constants were used. For example, the following code is legal:

```
enum Day { MON, TUE, WED, THU, FRI, SAT, SUN };
Day d = (Day) 10;
```

Any code that is expecting an enum of type `Day` can now receive an illegal value.

The Java 1.5 implementation of enum types is type-safe, although not as efficient as .NET. Java enums are based on the Typesafe Enum pattern documented by Bloch (2001). In this pattern, each enum is also represented as a class. Each of these classes has a private constructor, so that number of instances of the class in existence can be tightly controlled. Each enum value is represented as public, static field that is an instance of that class. Java 1.5 provides the syntax to make it easy for a developer to write this.

3.3.3 Attributes

Attributes (or annotations in Java 1.5) allow programmers to attach declarative information to entities in the program. Built-in attributes can be used, or the programmer can define their own custom attributes. Unlike existing documentation features, such as Java's `javadoc`, this information is available at run-time.

Our design of attributes within our semantic model is shown in Figure 3.8. An attribute can be attached to almost any type or member so each `Decl` has a collection of `MetaDataDecl`s. Each attribute knows what type of attribute it is, which we have represented as a `ClassType`. Attribute values consist of a list of arguments and a list of optional key and value pairs.

3.3.4 Delegates

Delegates may be thought of as a "type-safe mechanism for implementing the "function pointers" of C." (Gough 2001*a*). A delegate declaration associates the delegate name with a method signature. For example

```
delegate int D1(int i, double d);
```

declares a delegate type `D1`. Methods compatible with this signature (taking an `int` and `double` and returning an `int`) can be assigned to a variable of this delegate type. The `+=` operator is usually used when assigning to delegate types because a variable of a delegate type can refer to more than one method at a time. If this is the case, then each method is invoked synchronously. For each method in the invocation list, that is an instance method (as opposed to a static method) then the delegate stores the object instance as well. In .NET, delegates are closely linked with events (Section 3.3.5).

There has been considerable debate as to whether the concept of a delegate belongs in an object-oriented language. Sun maintains that delegates are unnecessary and detrimental to the language (Java Language

Team 2001) and so have not included delegates in Java. Microsoft disagrees with this and believe that delegates do fit nicely into an object-oriented language (Microsoft 2001). Anders Hejlsberg (Venners & Eckel 2003*b*) argues that delegates add expressive power you can't get using just classes or interfaces and that most programming languages have something semantically similar to a delegate (for example, function pointers in C++ or closures in LISP).

In the CLI specification, delegate types are represented as a class that extends `System.Delegate`. They have two private fields (Miller & Ragsdale 2003):

- one is an object
- one is the method that can be called on that object

Each delegate class also has a number of compiler generated methods as well for invoking the method(s) it contains. Early in our semantic model design, we explicitly modelled delegates as their own class. However, as we continued to model them (for example adding generics) they looked more and more like classes. The extra complexity to model delegates as separate entities when they are really just classes was not worth the benefit that only delegate types can be associated with events. This gives a complete representation of the code at the CLI level (which can later be manipulated to how a programmer views delegates in a language like C#).

3.3.5 Events

Events are used in .NET to "enable an object or class to provide notifications." Events are closely linked with delegate types. Gough (2001*a*) sees event types as "delegates with some additional semantics." In C#, event declarations appear like field declarations. For example:

```
public delegate void EventHandler(object sender, System.EventArgs e);
public event EventHandler click;
```

The difference is that the type of the event declaration must be a delegate. When compiled to MSIL, events look very similar to properties. An event has a name and type. The compiler can generate up to four methods for dealing with events: add, remove, fire and any number of other methods. In our semantic model, events are modelled as a subclass of `TypedDecl`. Like a property, it contains references to four possible types of methods associated with the event, as shown in Figure 3.9.

3.3.6 Namespaces

Namespaces help developers structure their software into components by providing a "logical organizational system" (ECMA 2005*a*). They do so by providing a higher level of scoping than simple type names (Allen, Avent, Gilani, Reid & Templeman 2002). This also helps to prevent name clashes between types when using different libraries.

The CLI specification does not have any understanding of namespaces: "while some programming languages introduce the concept of a namespace, the only support in the CLI for this concept is as a metadata encoding technique" (ECMA 2005*b*). Types are always fully qualified within each assembly. We have chosen to model namespaces explicitly because this matches the mental model the developer has of the system by converting the fully qualified "dotted" names in the CIL.

Although essentially the same semantic concept, Java packages are subtly different from .NET namespaces. This required us to adapt the JST approach. More specifically:

- Each package declaration statement in Java is for the entire source file, while in .NET multiple namespaces can be declared within the same source file.
- The package structure has to match the directory structure of the project in Java, while in .NET there is no such requirement.

Our design of namespaces is shown in Figure 3.8.

3.3.7 Properties and Indexers

Properties are a new .NET feature, so were not in JST. They are a convenient shortcut for writing the accessors (getters and setters) for a field within a type. For example, if you were wanting to write a class that contains a name (in C#):

```
private string name;
public string Name
{
  get
  {
    return name;
  }
  set
  {
    name = value;
  }
}
```

This property can be accessed just like a field, however, a method is really being called. When the above property is compiled, the getter and setter methods get generated by the compiler and when the property is accessed, the underlying method is invoked. For example, the above property generates two methods: `get_Length()` and `set_Length(string value)`. These two methods are then grouped together as a property, in the meta-data:

```
.property instance string Length()
{
  .get instance string get_Length()
  .set instance void set_Length(string)
}
```

The CLI allows a property to group methods together: a `get` method, a `set` method and zero or more `other` methods. Every property has a type, so a property fits in to our semantic model well as a subclass of `TypedDecl`. It has references to three methods declarations: `getter`, `setter` and `other`, as shown in Figure 3.9.

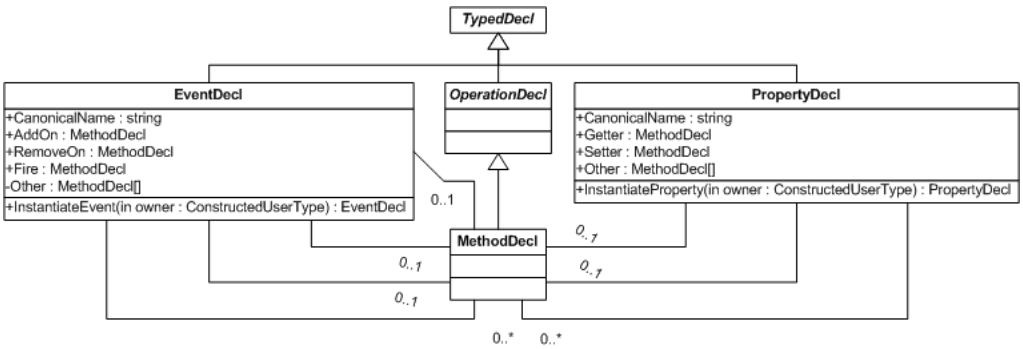


Figure 3.9: Design for Properties and Events

Properties do not add anything to a language, the same effect could be achieved by writing two methods. Therefore, properties and events (Section 3.3.5) are just syntactic sugar. Anders Hejlsberg (Venners & Eckel 2003*b*) points out however that a lot of OOP is just syntactic sugar. Programmers could write their own VTABLE and macros and write object-oriented software using C. However, providing these languages features to software developers allows them to be more efficient when writing code.

C# also contains indexers that "enables an object to be indexed in the same way as an array" (ECMA 2005*a*). Indexers are a special type of property: "whereas properties enable field-like access, indexers enable array-like access" (ECMA 2005*a*). When compiled, indexers are declared as a `.property` just like a normal property, except the signatures of the methods that get generated contain an integer parameter for the index. This means that we have not explicitly modelled indexers in our semantic model because they are just properties in the CLI.

3.3.8 Pointers

In the CLI specification, there are two types of pointers:

managed pointers are controlled by the garbage collector and are restricted in their use.

unmanaged pointers have no restrictions on how they are used and therefore cannot be verified by the CLR.

We have modelled managed and unmanaged pointer types using the Decorator (Gamma et al. 1995) design pattern. This is similar to how we modelled arrays. This pattern is useful when you need to "attach additional responsibilities to an object dynamically" (Gamma et al. 1995). The .NET documentation for the `System.Reflection.Pointer` class describes the class as "providing a wrapper class for pointer", which is consistent with our design approach. In this context, we want to wrap a type up as a pointer and change functionality like the name of the type. The other important advantage of this design solution is that these pointer types can be "chained" together to represent pointers to pointers. This design is shown in Figure 3.5. The `TypeDecl` class has a `GetPointerType` method that returns a pointer to that type. We distinguish the different between managed and unmanaged pointers using a `bool` field.

3.3.9 Other .NET Language Features

In C# 2.0, as well as generics, a number of other new language features were added. However, for these languages features, there was no change in the underlying CLI. These features are more syntactic sugar: reducing the amount of typing the software developer has to do and increasing the work the compiler does. A good example of this is anonymous methods with special syntax for declaring the method for a delegate type "in-line." The compiler generates a new method and connects this method to the delegate.

3.4 Known Limitations

There are a few limitations or improvements that can be made to our semantic model, all of which would be relatively simple to implement.

- Overriding is a relationship between methods that is useful to developers. This relationship has not been explicitly modelled in our semantic model, although all the information and methods are available to derive it.
- Methods and fields can be declared globally in modules and assemblies. This can be modelled by adding a global class to modules and assemblies that contain these global methods and fields.
- Our model only loads from correct, compiled code. It does not handle broken references, such as invoking non-existent methods. JST also has this limitations, however, CAISE, which uses an enhanced version of JST can handle uncompiled code. This would be a simple to implement in our if necessary.
- The CLI allows method pointer types. These could be included in our design by creating a pointer to a `OperationSignature`.

- There is a trade-off in our design as to whether we model every small semantic constraint. For example, a meta-data declaration is associated with a class that must extend `System.Attribute`. We have no way to model this semantic constraint in our design. However, for these minor details, we do not think it is worth the extra complexity required in the semantic model for it to be able to model these details.

3.5 Implementation

Although influenced by the design of JST, our .NET semantic model is implemented from scratch in C#. When implementing and using our semantic model, we were better able to determine exactly what methods each class needed.

Chapter 4

Populating the Semantic Model

The preceding chapter described in detail our design for a semantic model for .NET. In this chapter, we address the problem of populating this model in order to represent actual programs. There are a number of possible ways to do this, two of which we have implemented as part of this research. The approaches discussed in this chapter are: parsing, reflection and PERWAPI. Each of these approaches varies in difficulty, and more importantly, the quality of data that can be extracted when using them.

As discussed in the introduction to .NET (Section 2.3.2), .NET programs are not compiled directly to native code. Rather, they are compiled to MSIL and associated metadata, which is stored inside a portable executable (PE) file. On Windows, this can either be an `exe` or `dll`. If needed, the MSIL can be extract from a PE file using the Microsoft disassembler (`ildasm`). This intermediate language is used in some of these approaches, while others use the .NET Framework libraries or source code directly.

4.1 Parsing

The JST tool used complete XML parse trees (generated by `yakyacc` (Irwin & Churcher 2001)) to build the semantic model for Java 1.4. JST can fully resolve overloaded methods using the correct type promotions (taking into account overriding and access control). We could similarly use parsing to populate our semantic model, either from MSIL, or from the source code itself.

4.1.1 Parsing MSIL and Meta-Data

Parsing the meta-data and MSIL directly to populate the semantic model is a complex programming task. However, .NET has the advantage that by writing the code once to parse the meta-data and MSIL, the semantic model can be used for any languages that compile to the CLR. However, we didn't need to use this approach because the PERWAPI (Section 4.3) library provides an API for accessing the same data.

4.1.2 Parsing Source Code

Any language that targets the CLR has to be mapped onto the language concepts of the CLR. In this process, there is some loss of information for every language. The degree of loss depends on how extensively concepts of the source language must be transformed to fit the CLR. Even for a language like C# that maps closely to the MSIL, some loss occurs. For example, all conditional statements get broken down into simple branch instructions. Similarly, local variables declared inside blocks are all moved to the top of the method declaration.

Populating the semantic model directly from the source code is ultimately the most powerful approach because there is no loss of information. However, our semantic model is designed specifically for MSIL and not for any actual programming language. Extending the model for actual languages and populating them from source code is part of the future work for this project (discussed in Section 6.4).

4.2 Reflection

Using reflection to model software was discussed in Section 2.2.6. It has two major disadvantages for this purpose: no data is available for private types and members or for the code inside methods. Despite reflection's limitations, it is still useful, especially if the source code is not available or not needed (for example, .NET Framework libraries). In our first attempt to populate our semantic model, we used only reflection to populate as much of our semantic model as possible.

The code to populate the semantic model using reflection is a greedy algorithm that goes through each type declared in the source code, determines what type it is (using reflection), and then loads it and all of its members. If it reaches a type it doesn't yet know about then that type is fully loaded before continuing.

4.3 Program Executable – Reader/Writer – Application Programming Interface (PERWAPI)

PERWAPI (Gough & Corney 2005) is a tool for programatically reading and writing .NET executable files (either `.exe` or `.dll`). For populating our semantic model, we need only the reading features of the API. The interface that PERWAPI presents reflects exactly the structure of the PE file. It gives more information than reflection because it can access private code and methods internals. A simplified version of the PERWAPI class hierarchy used to extract information from the PE files is shown in Figure A.3

This design looks similar to our semantic model. However, there a number of important differences. Firstly, PERWAPI represents exactly the syntactic structure of the PE file containing MSIL. This does not match the semantic concepts a programmer works with, nor does it reflect the semantics of MSIL. There are a number of instances where the model does not reflect the semantic concepts:

- Constructors are methods with the special name `.ctor`. Each method can have generic parameters, and so this means constructors can have generic parameters.
- Each class can have at most one static constructor. However, classes have a collection of constructors, any number of which could have the static modifier.
- The constraint on a generic type parameter is of type `Type`. However, this allows lots of possible illegal constraints.
- A class contains a collection of interfaces that it implements. However, because interfaces are represented as special classes, each class actually contains a collection of classes.

Our model improves on this by exposing the actual semantics of the meta-data and MSIL, rather than presenting exactly what is in the PE file at the syntax level. We have used PERWAPI to completely populate our semantic model from any PE file. The code to populate the semantic model loops through all the types in a given .NET assembly, mapping the PERWAPI types to the types in our semantic model. When a type is referenced (for example, as a base class), if it is not already loaded, that type is fully loaded before continuing. Reflection is used for loading types in the .NET Framework libraries because we don't need access to private components or the internals of methods. Our semantic model currently doesn't support uncompiled code with broken references, so types and members need to be carefully loaded in the right order to make sure they are created before they are used. This requires two passes, one to load all the types and members and one to load all operator invocations.

4.4 Final System

The final software system includes the implementation of the semantic model in C#, the code to populate the model using reflection and PERWAPI, and a number of metrics, which use the semantic model to calculate their values. This is approximately 9,000 lines of code and is all contained inside one assembly, which can be executed on any .NET virtual machine. The semantic model can be built for itself (and the PERWAPI library) in less than one second on a Pentium 4 2.0GHz computer. This is 25,000 lines, which is comprised of 194 classes.

Chapter 5

Applications

We have built a semantic model and can populate it. It provides a richer basis for metrics, visualisations and other applications than is available elsewhere. This chapter discusses an application that directly uses our semantic model intended to demonstrate the value and capability of our model. No tool provides the data we need for this application, a complete model of the structure of a software system.

5.1 CodeRank

CodeRank is a set of new metrics that we have developed previously (Neate, Irwin & Churcher 2005) for informing software developers about one aspect of the code they are writing. The metric is based on applying Google's PageRank (Page, Brin, Motwani & Winograd 1998) concept to software. Calculating this metric requires a semantic graph that spans the program. ClassRank measures software using classes as the level of aggregation. Classes are the nodes and there are a number of different relationships that can exist between classes, including:

Inheritance a class *inherits* from another class

Implementation a type can *implement* multiple interfaces

Contains a type can *contain* a field of a type

Invocation a type can *invoke* a method declared in another type

Instantiation a type *instantiates* a generic type

Contains Many a type can *contain many* of another type

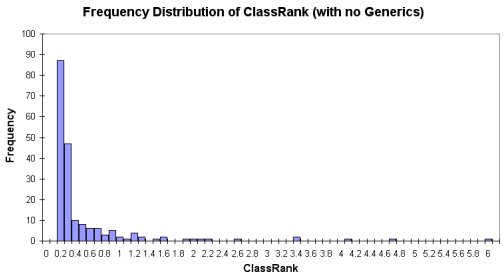
It is not yet clear which of these relationships is the most important, or how each relationship show be weighted.

The development of CodeRank is not an integral part of this project. Instead, it serves as a case study and proof of the value of our semantic model.

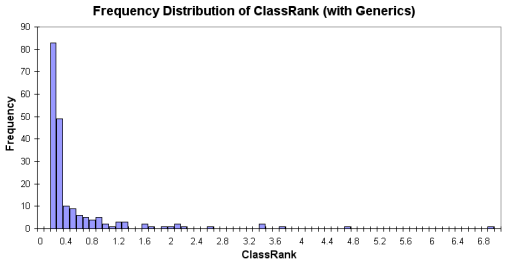
5.1.1 Implementation

The semantic model we have designed can be seen as a fully connected, directed graph. Starting at the assembly of interest, it is possible to traverse to any type or member represented in the model. We have implemented ClassRank using the Visitor (Gamma et al. 1995) design pattern. This pattern is useful for performing operations over a object structure without changing the object structure when adding new operations. In our design, different metrics are the operations which can be added without changing the semantic model in any way.

In our semantic model, there is a natural traversal starting at an assembly. Each assembly groups modules, which contain types, which contain members etc. This default navigation is encapsulated in an



(a) Without Generics



(b) With Generics

Figure 5.1: Frequency Distributions for ClassRank

abstract class (`ModelVisitor`). A metric can be implemented by extending this class and overriding the relevant methods. As a simple example, to count the number classes in a project, the method to visit a class is overridden so that it includes a counter.

When implementing ClassRank, we used classes, structs and interfaces, as well as instantiated types as the nodes. The six types of relationships described above were used as the links between the nodes.

5.1.2 Results

We ran the ClassRank algorithm on the source code written for this research, combined with the PERWAPI library. This software is approximately 25,000 lines of code and contains 194 classes. PERWAPI is written in C# 1.1 and so does not contain any generic code, while the software we have constructed is written in C# 2.0 and so contains generics where appropriate. We ran the ClassRank metric with and without the relationships obtained using generics (*instantiation* and *contains many*). The results for the top five classes are shown in Table 5.1 and the frequency distributions are shown in Figure 5.1.

Type Name	ClassRank without Generics	ClassRank with Generics
HonoursProject.Model.Decl	5.93	6.82
PERWAPI.MetaDataElement	4.62	4.62
HonoursProject.Visitor.ModelVisitor	4.06	3.65
PERWAPI.PEReader	3.37	3.37
PERWAPI.MetaDataOut	3.32	3.32

Table 5.1: ClassRank of the Top 5 Ranked Types

The class with the highest ClassRank is `Decl`. This the root of our semantic model and the most important class in the system. `MetaDataElement` is near the top of the PERWAPI type hierarchy (as shown in Figure A.3) and `ModelVisitor` is the base class for any new operations defined over the semantic model. Like many metrics, the frequency distribution for ClassRank is skewed right.

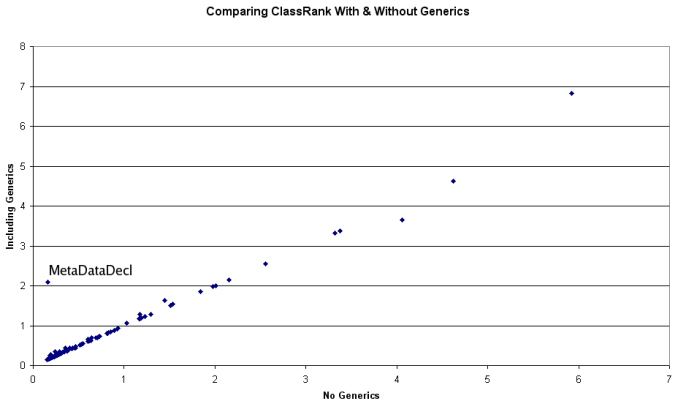


Figure 5.2: Scatter Plot of Generic and Non-Generic ClassRank

Figure 5.2 shows a scatter plot of the ClassRank metric including and excluding the generic relationships. There is a very strong correlation, except for one obvious outlier. This class is `MetaDataDecl`. As shown in Figure 3.8, each `Decl` has a collection of meta data objects (`List<MetaDataDecl>`). This relationship can only be detected when we include generic relationships. `Decl` is the highest rank class and it does not have many outbound links, so `MetaDataDecl` receives a large amount of rank, indicating that it is an important class. This is an indication of the value of complete, high-quality data, in this case the ability to extract generic type information.

5.1.3 Value of CodeRank

CodeRank is an extremely promising new metric that captures a dimension of software inaccessible to other metrics. Using our .NET semantic model improves on the original implementation using JST because we can use generic relationships and the metric is defined independent of language, meaning that the rank can flow around classes in a system written in multiple languages.

Chapter 6

Discussion

6.1 Applications of our Semantic Model

By compiling all source code to an intermediate language and executing it on a virtual machine, .NET allows software projects to be written in multiple languages. By modelling this intermediate language, our semantic model expose the semantics for a software project written in object-oriented languages that compile to .NET.

An application that uses our semantic model can gain valuable structure information about any .NET language. For example, it is difficult to compare metrics across source code written in different object-oriented languages because it is difficult to define precise definitions for metrics written in one language (Churcher & Shepperd 1995) and more difficult for multiple languages. However, as we have shown in Chapter 5, it is possible to define rigorous and language-indendent metrics using the semantic concepts exposed by our semantic model. In our example, ClassRank can be calculated and compared for source code written in any .NET language. In a multiple language software project, rank can flow across components written in different languages. This is a significant advance over current practice.

6.2 Future Versions of .NET

Microsoft recently released the specification for version 3.0 of C#. This includes a number of new language features including implicitly typed arrays and local variables, anonymous types and extension methods. However, there is no corresponding update in the CLI specification. All the new language features compile to the existing CLI standard that we have modelled. The compiler is doing more work to map these features onto existing languge features in the CLI, which means our semantic model can represent C# 3.0 code.

It would appear that the CLI is now stable. The only feature to be added to the third edition of the standard was generics and there do not appear to be any plans for any more major changes. This gives us confidence that our semantic model will be useful in the forseeable future for representing software compiled for .NET.

6.3 Limitations of our Semantic Model

Obviously, although MSIL represents the semantic concepts common across many programming languages, it does not actually reflect the exact semantic concepts of any one programming language. The closest actual programming languages used by developers are C# and VB.NET, which were designed in conjunction with the CLI, however, there is still a semantic gap between these programming languages and the CLI. This semantic gap is larger for other object-oriented languages like C++ and Eiffel, where the compiler needs to perform a number of clever “tricks” to overcome the mismatch in semantics. Non object-oriented languages like Mondrian and Haskell have an even larger gap from the CLI semantics.

One of the benefits of modelling the CLI is that our semantic model can handle any language that compiles to the CLR. It could be argued that if we were to build a semantic model of a physical (rather than virtual) machine like the Pentium, then we would be able to model every programming language. However, this model would be too low-level and not capture and expose the semantics that programmers are interested in.

Each programming language has too many specific language features to allow a common model, while the physial machine is too low level to be of use. By using a virtual machine as an intermediate form the "full richness of the source language is reduced to a simplified set of operations that are still able to express all the operations of the source" (Gough 2001*a*). Microsoft has defined such a virtual machine and intermediate language between the source language and machine language that "bridges the semantic gap" (Gough 2001*a*). This is why we have modelled the CLI, because it captures the core common semantics of object-oriented programming languages, and yet is high-level enough that it can provide useful information to developers.

6.4 Future Research

This research has set the foundation for future research to build upon the semantic model we have developed for the .NET 2.0 Framework. There are a number of directions this research could take, including developing applications that use the data from the semantic model to inform software developers about the structure of their software or extending the model and populating it from source code.

6.4.1 Applications using the Semantic Model

The goal of this project was to build a rigorous model of the semantic concepts of the CLR. By itself, this is not much use for helping developers understand the structure of their code. Applications need to be built to make use of the rich data that the semantic model provides. We have already shown the many useful applications (in Section 2.2.5) that have been made using output from JST. There is no reason why these applications and more couldn't be created using the output from our .NET semantic model and immediately improved by language independence. Using a pipeline as a metaphor, our semantic model fits in after the compiler has produced the intermediate language. By conforming to the pipeline, our model can extend the work documented in (Churcher & Irwin 2005).

We have shown one such application which uses the output from our semantic model: CodeRank. We are unaware of any other tool that directly produces the data needed to be able to calculate this metric, and so this shows that our semantic model is useful for new applications.

One area of code analysis that has not had a lot of attention elsewhere is generics. Our semantic model provides a comprehensive view of that way generic types are declared, instantiated and used throughout the project. Effective metrics and visualisation need to be researched that can use this data provide useful information to software developers.

6.4.2 Populating from Source Code

Our semantic model reflects the semantic concepts of the CLI and closely related languages like C# and VB.NET. However, there is always a loss of information when a language is compiled into MSIL. A simple example of this is the way conditional statements are compiled. MSIL has no knowledge of `if` or `while` statements, all it knows about are `branch` instructions and labels. When inspecting MSIL, we have no idea how the developer originally wrote the code.

For languages that don't map exactly into MSIL, there are some subtle, yet important semantic differences between the original source code and the MSIL representation. A good example of this is a language like Eiffel, which supports multiple inheritance. In this case, the compiler has to perform some tricks to map Eiffel on the CLR and our tools downstream don't know the original semantic concepts present in the code.

A possible solution to this would be to populate the semantic model directly from source code, rather than an intermediate language. This would firstly require the model to be extended in some way to include

the semantic concepts specific to each language. The best way to do this would be to keep the original semantic model unmodified and write “plugin” components for each separate language. Unfortunately, there is a major tradeoff involved in this improved semantic model. Instead of writing the code once to build the model (like we do when modeling MSIL), it needs to be written for every source language you want to have a complete semantic model for. Although valuable, this is a time consuming process.

Chapter 7

Conclusion

In this research, we have shown the need for tools to help software developers understand the structure of their software. To be useful, these tools need complete and accurate data about the underlying source code. We introduced the approach of constructing a semantic model that can represent all the semantics explicitly and formally using object-oriented concepts.

We presented our design for a semantic model that can represent the semantic concepts in the CLI. Some of the .NET language concepts are close to the concepts in Java, which meant that some parts of our .NET semantic model were similar to the JST semantic model. The most complex and challenging design task was incorporating generics into our model. We also showed how all other .NET specific language features can be represented in our semantic model. The final design (shown in Figure A.2) is a comprehensive model that explicitly exposes the semantics of the CLI, which can be used by other tools for informing developers about their software.

The semantic model has been implemented in C#. The PERWAPI library was used to populate the model, given software that has been compiled to execute on the CLR. Our semantic model presents an API that can be used by other tools to extract the semantic information and use it to help software developers better understand and manage the complexity of the software they are developing.

By choosing to model the .NET intermediate language, we can leverage the language independence of the CLI in our semantic model. By modelling Version 2.0 of .NET (which includes generics), we have been able to represent new relationships that are useful to developers, particularly aggregation and composition relationships when the generic collections are used. The effectiveness of our semantic model was demonstrated in Chapter 5, where we presented the CodeRank metric, which could not be calculated without using our semantic model.

Further research would be useful to develop tools that can use the data from the semantic model to present to software developers, particularly tools that make use of the new data available from modelling generics. The semantic model could also be extended so that it can completely represent the semantics of software written in a programming language used by developers (like C#).

Bibliography

Aho, A. V., Sethi, R. & Ullman, J. D. (1986), *Compilers, Principles, Techniques, and Tools*, Addison-Wesley.

Allen, K. S., Avent, N., Gilani, S. F., Reid, J. D. & Templeman, J. (2002), *Fast Track C#*, Wrox Press.

Bloch, J. (2001), *Effective Java: Programming Language Guide*, Java series, Addison-Wesley, Reading, MA, USA.

Bracha, G. (2004), 'Generics in the Java Programming Language'.
`http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`

Bruce, K. B. (2002), *Foundations of Object-Oriented Languages: Types and Semantics*, Massachusetts Institute of Technology.

Chappell, D. (2002), *Understanding .Net: A Tutorial and Analysis*, Pearson Education.

Charters, S. M., Knight, C., Thomas, N. & Munro, M. (2002), Visualisation for informed decision making; from code to components, *in* 'SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering', ACM Press, New York, NY, USA, pp. 765–772.

Chidamber, S. R. & Kemerer, C. F. (1994), 'A metrics suite for object oriented design', *IEEE Transactions on Software Engineering* **20**(6), 476–493.

Churcher, N. I. & Shepperd, M. J. (1995), 'Comments on "A metrics suite for object oriented design"', *IEEE Transactions on Software Engineering* **21**(3), 263–265.

Churcher, N. & Irwin, W. (2005), *Informing the Design of Pipeline-Based Software Visualisations*, Vol. 45 of *Conferences in Research and Practice in Information Technology*, ACS, Sydney, Australia. Asia Pacific Symposium on Information Visualisation (APVIS2005).
`http://crpit.com/confpapers/CRPITV45Churcher.pdf`

Cook, C., Churcher, N. & Irwin, W. (2004), Towards synchronous collaborative software engineering, *in* 'APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)', IEEE Computer Society, pp. 230–239.

Eckel, B. (2004), 'Java vs. .NET'.
`http://www.mindview.net/WebLog/log-0035`

Eckel, B. (2005), 'Generics Aren't'.
`http://www.mindview.net/WebLog/log-0050`

Eclipse Foundation (2005), 'JDT'.
`http://www.eclipse.org/jdt/`

ECMA (2005*a*), *ECMA-334: C# Language Specification*, third edn, ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland.

ECMA (2005*b*), *ECMA-335: Common Language Infrastructure (CLI)*, third edn, ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc.

Garcia, R., Jarvi, J., Lumsdaine, A., Siek, J. & Willcock, J. (2003), A comparative study of language support for generic programming, *in* 'OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications', ACM Press, New York, NY, USA, pp. 115–134.

Gough, J. & Corney, D. (2005), Reading and Writing PE-files with PERWAPI.
`http://www.plas.fit.qut.edu.au/perwapi/Default.aspx`

Gough, K. J. (2001*a*), *Compiling for the .Net Common Language Runtime*, Prentice Hall PTR, Upper Saddle River, NJ, USA.

Gough, K. J. (2001*b*), Stacking them up: a comparison of virtual machines, *in* 'ACSAC '01: Proceedings of the 6th Australasian conference on Computer systems architecture', IEEE Computer Society, Washington, DC, USA, pp. 55–61.

Grune, D., Bal, H. E., Jacobs, C. J. H. & Langendoen, K. G. (2000), *Modern Compiler Design*, John Wiley and Sons, Ltd.

Irwin, W. & Churcher, N. (2001), XML in the visualisation pipeline, *in* 'CRPITS '11: Proceedings of the Pan-Sydney area workshop on Visual information processing', Australian Computer Society, Inc., pp. 59–67.

Irwin, W. & Churcher, N. (2003), Object oriented metrics: Precision tools and configurable visualisations, *in* 'METRICS '03: Proceedings of the 9th International Symposium on Software Metrics', IEEE Computer Society, Washington, DC, USA, p. 112.

Irwin, W., Cook, C. & Churcher, N. (2005), Parsing and semantic modelling for software engineering applications, *in* 'Proceedings of the Australian Software Engineering Conference'.

Java Language Team (2001), About Microsoft's "Delegates", Technical report, Sun Microsystems, Inc.
`http://java.sun.com/docs/white/delegates.html`

Lowy, J. (2005), 'An Introduction to C# Generics'.

McConnell, S. C. (2004), *Code Complete*, 2nd edn, Microsoft Press.

Meijer, E. & Gough, J. (2001), 'Technical Overview of the Common Language Runtime'.

Meyer, B. (2000), *Object-Oriented Software Construction*, Prentice Hall.

Meyer, B. (2002), 'The significance of .NET'.
`http://archive.eiffel.com/doc/manuals/technology/bmarticles/sd/dotnet.html`

Microsoft (2001), The Truth About Delegates, Technical report, Microsoft.
`http://msdn.microsoft.com/vjsharp/productinfo/visualj/visualj6/technical/articles/general/tru`

Microsoft (2005), 'Differences Between C++ Templates and C# Generics'.
`http://msdn2.microsoft.com/en-us/library/c6cyy67b`

Microsoft Corporation (2005), 'Visual c# 3.0'.
`http://msdn.microsoft.com/vcsharp/future/`

Miller, J. S. & Ragsdale, S. (2003), *The Common Language Infrastructure Annotated Standard*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Mok, H. N. (2003), *From Java to C#: A Java Developer's Guide*, Addison-Wesley Longman Publishing Co., Inc.

Moller, K. H. & Paulish, D. J. (1993), *Software Metrics*, IEEE Press + Champman & Hall.

MSDN .NET Framework Developer Center (n.d.). http://msdn.microsoft.com/netframework/. `http://msdn.microsoft.com/netframework/`

Neate, B., Irwin, W. & Churcher, N. (2005), CodeRank: A New Family of Software Metrics, Technical Report TR-COSC 07/05, Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand.

Object Management Group (2005*a*), 'MOF 2.0 / XMI Mapping Specification, v2.1'.

Object Management Group (2005*b*), 'Unified Modeling Language 2.0'. `http://www.uml.org/`

Page, L., Brin, S., Motwani, R. & Winograd, T. (1998), 'The pagerank citation ranking: Bringing order to the web', *Technical report, Stanford University, Stanford, CA* .

Prosise, J. (2002), *Programming Microsoft .NET*, Microsoft Press.

Riel, A. J. (1996), *Object-Oriented Design Heuristics*, Addison-Wesley Longman Publishing Co., Inc.

Slonneger, K. & Kurtz, B. L. (1995), *Syntax and Semantics of Programming Languages*, Addison-Wesley Publishing.

Tucker, A. & Noonan, R. (2002), *Programming Languages: Principles and Paradigms*, McGraw Hill.

Venners, B. & Eckel, B. (2003*a*), 'Contracts and Interoperability: A Conversation with Anders Hejlsberg'. `http://www.artima.com/intv/interopP.html`

Venners, B. & Eckel, B. (2003*b*), 'Delegates, Components, and Simplexity: A Conversation with Anders Hejlsberg', Website. `http://www.artima.com/intv/simplexity.html`

Venners, B. & Eckel, B. (2003*c*), 'Versioning, Virtual, and Override: A Conversation with Anders Hejlsberg'. `http://www.artima.com/intv/nonvirtual.html`

Venners, B. & Eckel, B. (2004*a*), 'Generics in C#, Java, and C++: A Conversation with Anders Hejlsberg'. `http://www.artima.com/intv/generics.html`

Venners, B. & Eckel, B. (2004*b*), 'Insights into the .NET Architecture: A Conversation with Eric Gunnerson'. `http://www.artima.com/intv/dotnet.html`

Appendix A

Large UML Models

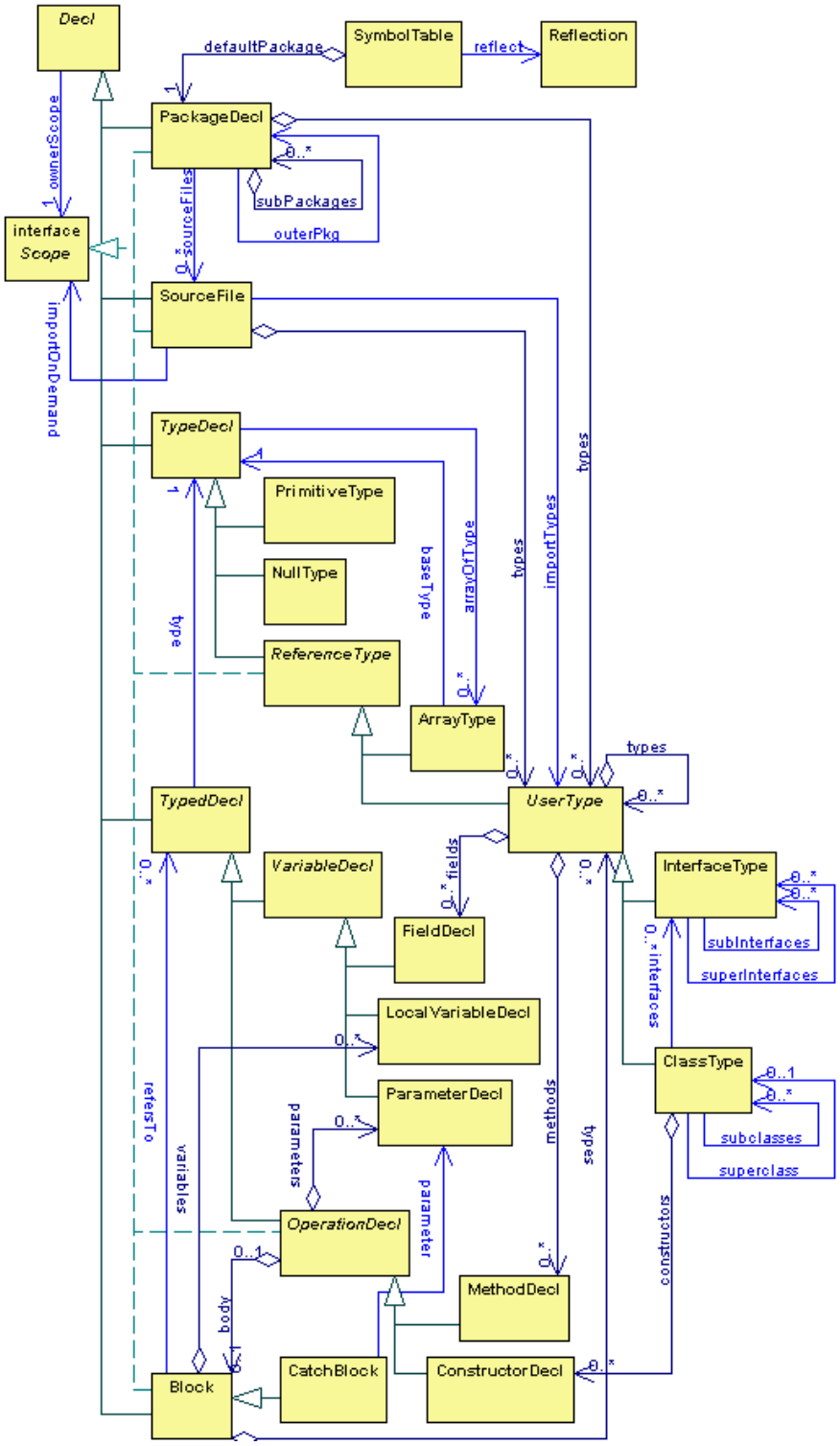


Figure A.1: JST Semantic Model

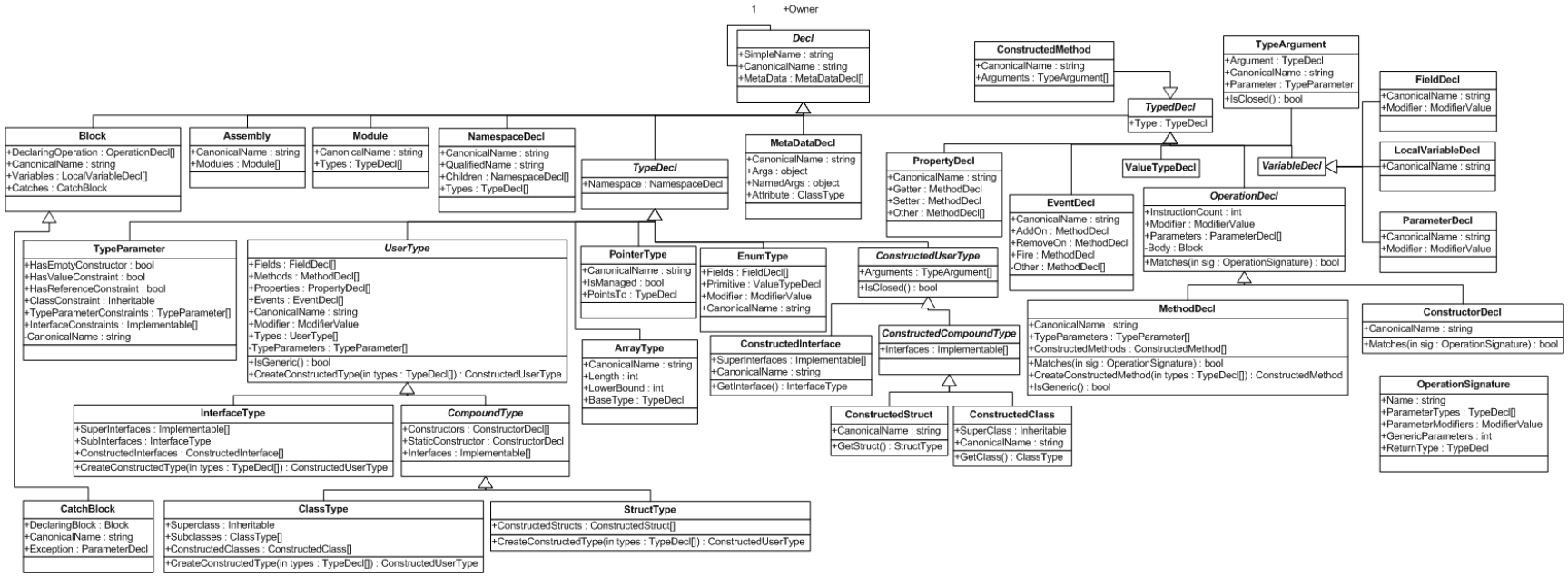


Figure A.2: Complete .NET Semantic Model

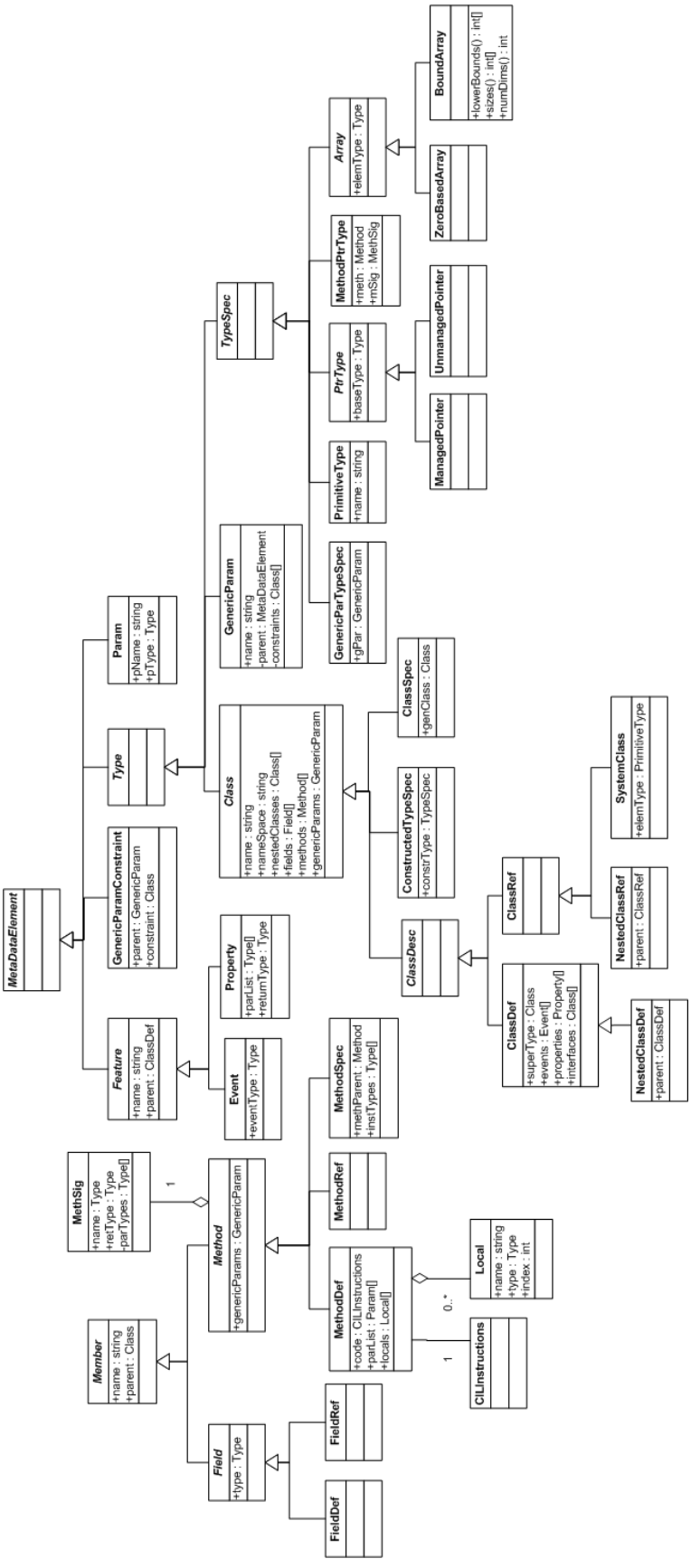


Figure A.3: Simplified Class Diagram of PERWAPI