Fachhochschul-Masterstudiengang
SOFTWARE ENGINEERING
4232 Hagenberg, Austria

A Code Generation Pipeline for .NET

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science in Engineering

Eingereicht von

BSc Julian Lettner

Betreuer: DI (FH) Dr. Fabian Schmied, rubicon IT GmbH, Wien
Begutachter: FH-Prof. DI Dr. Heinz Dobler

September 2012

software engineering
IKM Fakultät Hagenberg

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, September 3rd, 2012

Julian Lettner

Contents

Kurzfassung

Durch die Verwendung in Frameworks und anderen Entwicklungswerkzeugen avancierte Codegenerierung von einem Thema, das Compilern vorbehalten war, zu einem wesentlichen Bestandteil der modernen Softwareentwicklung. Codegenerierungstechniken haben ein breites und vielfältiges Einsatzgebiet und unterscheiden sich hinsichtlich angebotener Funktionen und Einschränkungen. Diese Techniken ermöglichen neue Ansätze in der Informatik und Softwaretechnik oder werden verwendet, um Probleme effizienter zu lösen.

Nachteilig ist, dass die meisten, auf Codegenerierung basierenden Frameworks nicht gemeinsam eingesetzt werden können. Diese Arbeit analysiert das zugrunde liegende Problem und kombiniert ausgereifte Codegenerierungstechniken mit einem neuen Ansatz für Interoperabilität. Sie beschreibt Entwurf, Implementierung und Test einer Codegenerierungs-Pipeline für die .NET-Plattform. Das Ziel der Pipeline ist es, das gemeinschaftliche Generieren von Code durch unabhängige Teilnehmer zu ermöglichen.

Die Arbeit untersucht den von der Pipeline generierten Code und vergleicht ihn mit Code, der mit Hilfe des weit verbreiteten DynamicProxy-Frameworks generiert wurde. Der Vergleich zeigt, dass die entwickelte Pipeline sowohl als Basis für andere Frameworks und Entwicklungswerkzeuge als auch in Anwendungen mit anspruchsvollen Codegenerierungsanforderungen genutzt werden kann. Das Resultat ist als quelloffenes Projekt unter typepipe.codeplex.com verfügbar.

Abstract

The use in frameworks and tools advanced code generation from a topic solely relevant for compilers, to an integral part of modern software development. Code generation techniques have a broad and diverse field of application and come with different features and constraints. They support new approaches in computer science as well as in software engineering or are used to solve problems more efficiently.

Unfortunately, most code generation-dependent frameworks and tools do not work well together. This thesis analyzes the underlying problem and combines well-understood code generation techniques with a new approach towards interoperability. It describes the design, implementation and test of a code generation pipeline for the .NET platform. The goal of the pipeline is to allow multiple independent participants to generate code collaboratively.

The thesis examines the code generated by the pipeline and compares it to code generated by the popular DynamicProxy framework. The comparison shows that the developed pipeline can be used as a basis for frameworks and tools as well as in applications with demanding code generation needs. The result is available as an open source project at typepipe.codeplex.com.

Chapter 1

Introduction

1.1 Definition

Code generation is an omnipresent topic in computer science and software engineering. It serves as a vehicle for current research as well as reliable production systems. The most prominent use of code generation can be found in compilers. A compiler essentially translates a source language into a target language. The translation process is called compilation and includes parsing, syntax analysis, semantic analysis, and code generation [ASU86]. This thesis, however, focuses on code generation by frameworks. Frameworks employ code generation to tackle difficult problems and provide specific features to their users. In contrast to a compiler whose main purpose is transformation, frameworks use it as an enabling technology.

1.2 Motivation

The field of application for code generation is broad and diverse. In research areas such as metaprogramming, it is a necessity. Subsets of this research, for example aspect-oriented programming (AOP) [JN05], have been adopted by the industry and more are to follow. Code generation is also an integral part of model-driven engineering (MDE) [GDD09], which aims to raise the level of abstraction to produce higher quality software that is easier to maintain in less time. Many current frameworks use it to provide features that would not be feasible otherwise. In addition, code generation finally starts to emerge in product development. Application developers use it to replace repetitive, error-prone tasks and for increased productivity.

As diverse as its applications are the shapes, the input data, and the output format of existing code generation techniques. Each technique has unique advantages and disadvantages, use cases, and constraints. Unsurprisingly, different frameworks choose different code generation techniques according to their needs. The problem is that most code generation techniques

have no intrinsic concept of extensibility or modularization. Most current frameworks that rely on code generation therefore lack interoperability and options for integration with other frameworks. As simultaneous use of different frameworks is rarely possible, developers are often faced with the decision which framework is the most valuable for their scenario. This thesis strives to improve the described situation.

1.3 Scope

The scope of this thesis is code generation by frameworks based on the .NET platform. Readers are assumed to be generally familiar with the C# language, the .NET platform and, in particular, with .NET reflection and Reflection.Emit (Ref.Emit). In addition, concepts like abstract syntax trees (ASTs) and design patterns are used without further explanation. Abbreviations and acronyms are spelled at their first use and are included in the list of abbreviations at the end of the document. Technical idiomatic expressions are *highlighted* on the first appearance.

1.4 Goal

The goal of this thesis is to

- investigate existing code generation possibilities for the .NET platform,
- illustrate their shortcomings in terms of interoperability,
- gather requirements for a code generation component that overcomes these shortcomings,
- design, implement, and test such a component,
- record the rationale behind key decisions and document important aspects of the design and implementation,
- examine the quality and performance of the generated code, and
- verify that the developed component can be utilized beneficially in real-world projects.

1.5 Structure

The rest of this thesis is organized as follows:

Chapter 2 presents a classification for code generation techniques, briefly introduces four popular .NET code generation frameworks, and classifies them according to the established criteria.

Chapter 3 explains the concept of proxies and investigates the shortcomings of existing code generation frameworks. Subsequently, it outlines the re-motion framework and its use cases for code generation. The chapter

concludes with a list of requirements for re-motion's future code generation component and mentions existing projects that have been evaluated.

Chapter 4 presents the design of the developed pipeline.

Chapter 5 highlights interesting implementation details, encountered issues, and their solutions.

Chapter 6 illustrates the usage of the developed application programming interfaces (APIs) by means of a simple example. Afterwards it compares the code generated by the pipeline with code generated by another popular framework.

Chapter 7 summarizes the thesis, points out areas considered for future work, and states personal experiences of the author.

The last chapter is followed by a list of used abbreviations and the bibliography.

Chapter 2

Code Generation Techniques

2.1 Classification

This section establishes criteria that can be used to classify code generation techniques. Note that not every technique can be uniquely classified with a single combination of attributes. A technique may support multiple scenarios or take a hybrid approach. Java Server Pages (JSP), for example, are typically translated into *Servlets* on demand, i.e., on the first request, during development. In production, however, most JSP are translated and compiled at deployment time [RPL03, Section 1.1.7].

2.1.1 Template vs. API-based

One characteristic for distinguishing techniques is how code generation is supported from the developer's point of view.

Most template-based approaches provide rules, control structures, and some sort of expression language for filling in dynamic values for placeholders. Rules may be invoked by matching input elements (data-driven) or called directly (program-driven). Extensible Stylesheet Language Transformation (XSLT) [Kay08] with its matched and named rules is a good example for this.

API-based approaches facilitate code generation through library functions or by providing an in-memory model representing the generated code.

In general, template-based approaches are considered to offer higher productivity while API-based approaches are thought of being more flexible.

2.1.2 Source vs. Executable Code

The shape and nature of the resulting output also represents an aspect for distinguishing code generation techniques. The distinction can be made by asking the question whether or not the generated output can be directly executed. While executable code represents a runnable program, source code

must be compiled or further processed in some other way to obtain an executable program.

For this discussion, the notion of executable code is not equivalent to machine code, which solely refers to the set of instructions understandable by a certain physical processor [Lin06]. Therefore, executable code also includes code in intermediate languages (ILs) that are not directly executed, for example byte code for the Java virtual machine (JVM) [LY12] or IL for the .NET Common Language Runtime (CLR) [GG01].

2.1.3 Compile vs. Runtime

Another criterion is the point in time when code generation happens. A broad distinction for this point is compile time versus runtime.

In this context compile time roughly means sometime before program execution. This includes design time, compile time in its strict sense, and deployment time. Code generation at compile time can be implemented as pre- or post-compile step or by directly influencing the output of the compiler in any other way.

Runtime refers to the time in that the program is actually executed. At runtime, code is most often generated either at startup or on demand, later during program execution.

2.2 Example Frameworks

There is a vast number of different frameworks, libraries, and tools available for code generation. These frameworks greatly differ in their supported concepts and capabilities, and are therefore suited for different scenarios. This section briefly introduces four popular code generation frameworks for the .NET platform. Table 2.1 classifies them according to the criteria established in section 2.1. The following sections sketch the main idea and predominant usage scenarios for the introduced frameworks.

	Basis		Code		Time	
	Template	*API*	*Source*	*Executable*	*Compile*	*Run*
T4	✓		✓		✓	✓
CodeDOM		✓	✓	✓	✓	✓
DynamicProxy		✓		✓	✓*	✓
Mono Cecil		✓		✓	✓	✓*

* Supported but less common.

Table 2.1: Classified example frameworks.

2.2.1 Text Template Transformation Toolkit (T4)

As one might infer from its name, the Text Template Transformation Toolkit (T4) [MST4] is a template-based code generation framework. The generated output can be any text such as comma-separated values (CSV), program source code, or free form text. T4 is included in Visual Studio (VS) and features a straightforward template language, which makes it very accessible. Templates are created by mixing snippets of C# or VB.NET with static content. The template language supports code fragments (`<#`) to define the program flow, expressions (`<#=`) to output values, and declarations (`<#+`) to declare template members.

After a template has been defined, source code for a .NET type representing the template is generated and compiled by the T4 template engine. The compiled template can be used for text generation both at design time and runtime. This two-step process is depicted in figure 2.1. At runtime, T4 is most often used to generate HTML, resource files or data in other formats. At design time, the most common usage scenario is the automation of repetitive and error prone programming tasks. That is the generation of source code for domain types or infrastructure concerns based on existing types, configuration data or database entries. The generated source files are automatically compiled by VS and the resulting types can be immediately used.

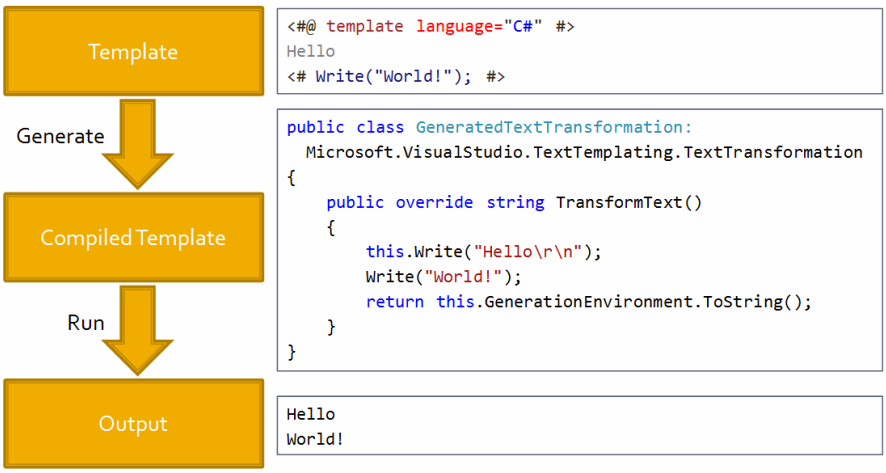

Figure 2.1: T4 template transformation process [Syc07].

2.2.2 CodeDOM

The Code Document Object Model (CodeDOM) [MSCD] is a library to generate source code for multiple programming languages. It is included in the Base Class Library (BCL) and can be found in the namespace `System.CodeDom`. Source code is represented by an in-memory model composed of interlinked

CodeDOM elements. The model and the corresponding API are language-agnostic. From an instance of this model, source code can be emitted and compiled with the help of a language specific `CodeDomProvider`. The BCL provides concrete implementations of this class for C#, VB.NET, and JScript.

Although CodeDOM supports a broad range of scenarios, it has fallen out of favor due to its deficiencies. Compared to T4 templates, CodeDOM user code is quite complex and less maintainable. Another disadvantage is code generation performance for dynamic scenarios. At runtime, source code is emitted that in turn has to be compiled for execution. This is less efficient than Ref.Emit-based approaches, which directly generate IL code. Today, CodeDOM is primarily used by tools that generate source code to be included as part of a VS solution. For example, a number of VS designer, typed WSDL messages, and LINQ to SQL classes all use CodeDOM under the covers. On the server, CodeDOM is used to compile Active Server Pages .NET (ASP.NET) at runtime similiar to JSP.

2.2.3 DynamicProxy

DynamicProxy is a .NET library for creating proxy objects. It is part of the Castle open source project [CP]. Internally, it uses Ref.Emit for code generation. DynamicProxy supports the generation of several proxy types:

- Class proxy,
- interface proxy with and without target,
- interface proxy with target interface.

The method behavior of all proxy types can be adapted by supplying implementations of the `Castle.DynamicProxy.IInterceptor` interface when the proxy is created. In addition, so-called *interceptor selectors* can be used to match interceptors to their respective target methods.

Class proxies are implemented by deriving a proxy type from the given class and overriding the intercepted methods. For interfaces, the generated proxy type simply implements the interface and delegates all methods to the specified target. If no target is provided then every method must be adapted by at least one interceptor. For non-`void` methods this interceptor also needs to provide the return value of the method. Interface proxies with target interface allow swapping the original target with another implementation of the target interface type. For an explanation of the proxy concept and its limitations, see section 3.1.

DynamicProxy also has a few other noteworthy features. Dynamic code generation is not a lightweight operation, therefore DynamicProxy caches the generated proxy types. These types can be persisted into an assembly and loaded from disk across program executions. DynamicProxy also supports a basic interface-based approach for mixins.

Originally, DynamicProxy was developed as an alternative to the intrusive proxying mechanisms offered by the CLR [Ver04]. Today, there exist

many frameworks that offer similar capabilities for creating dynamic runtime proxies. For example, there are Spring.NET, Unity, LinFu, Dynamic Decorator–just to name a few. Furthermore, there is also a considerable number of projects using DynamicProxy for their code generation needs. The most popular ones are the Castle project itself, NHibernate, Rhino Mocks, Moq, Ninject, and Autofac.

2.2.4 Mono Cecil

Cecil [MC] is an open source .NET library to inspect, modify and create managed assemblies. It is hosted as a part of the popular Mono project [Mono]. Mono started as a Linux version of the C# compiler and the CLR. Today, it is a full-fledged, open source, cross-platform implementation of .NET that is binary compatible with Microsoft.NET. It also acts as an umbrella project for many useful libraries and tools that are not included in the .NET platform, for example Cecil.

When compared with Ref.Emit, Cecil has a few distinctive characteristics. Unlike Ref.Emit, Cecil treats metadata and code contained in managed assemblies as pure data. That means that Cecil can reflect over types inside an assembly without loading the assembly into an `AppDomain` instance. In addition, Cecil allows to modify existing types and to save the modified assembly. As Cecil's object model is directly based on the Common Language Infrastructure (CLI) ECMA standard [ECMA], it supports virtually all features of the CLI, including features that are not accessible through the Ref.Emit API. Cecil also gives the user a more advanced view on the IL code of methods than Ref.Emit, which exposes method bodies as simple byte arrays.

Cecil is a powerful library but its power comes at a cost. A disadvantage of Cecil is its level of abstraction. It is comparable with Ref.Emit's abstraction level and therefore considerably lower than that of the frameworks introduced in the previous sections. A profound understanding of CLI metadata concepts and the IL [Lid06] is a necessity to work with Cecil.

Cecil can be utilized for many usage scenarios including inspecting foreign assemblies at runtime and modifying existing types and methods in a post-compile step. Therefore, many open source projects and commercial products such as static code analysis tools, aspect weavers, and code generators are based on Cecil. The most popular ones are ILSpy, .NET Reflector, Reflexil, SharpDevelop, LINQPad, EQATEC Profiler, MbUnit, NCrunch, AspectDNG, LinFu, MonoTouch, and Mono for Android [MCU]. Other libraries that provide similar capabilites are the Common Compiler Infrastructure by Microsoft Research [BF09], and the PostSharp SDK [PSS].

This chapter established a classification for code generation frameworks and introduced four popular frameworks for the .NET platform. In the next chapter, we will look at the proxy concept, the shortcomings of existing frameworks, and code generation in the re-motion framework.

Chapter 3

Problems of Traditional Approaches

3.1 Proxy Types

Many approaches for code generation and higher level frameworks are based on a mechanism called *proxying*. In order to understand the limitations of these approaches it is necessary to investigate the underlying proxy concept.

A proxy [GHJV94] is defined to be "an object which acts as a placeholder for a target object". Wherever the target is expected, the proxy can be used instead. This closely relates to the Liskov substitution principle (LSP) [LW94], which states that an object of a subtype can be substituted for one of a supertype. Therefore, proxies can be implemented with the existing subtyping mechanisms in .NET [KFS06], i.e., interfaces and inheritance.

In the following, we look at two kinds of implementations of the proxy concept: interface proxies and subclass proxies. The CLR also has built-in support for another proxy mechanism, called *transparent proxy*. It requires the proxied object to inherit from the `System.ContextBoundObject` class and therefore has an intrusive nature. Because of this and other technical limitations, it is less favorable than the approaches described below. For a detailed comparison of proxying techniques in .NET, see [KFS06].

An *interface proxy* (Fig. 3.1) implements the same interfaces as the target and holds a reference to the target object. Client code is required to access the target through these interfaces only. Upon a client call, the proxy may adapt the behavior of the call before delegating to the referenced target object.

A *subclass proxy* (Fig. 3.2) derives from the target and overrides methods in order to intercept them. Therefore, the subclass proxy approach is subject to the following limitations: The target needs to be subclass-able, meaning it must not be final[1], and must have an accessible constructor. As mem-

[1] In C# final is signaled by the keyword `sealed`.

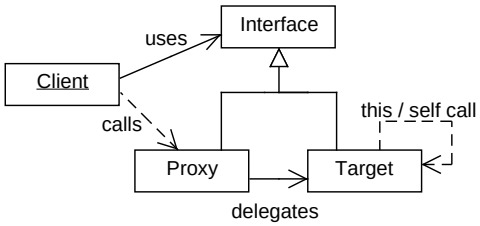

Figure 3.1: Interface proxy.

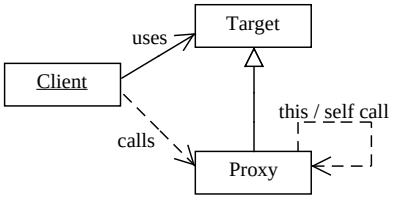

Figure 3.2: Subclass proxy.

ber interception is implemented through overriding, only virtual non-final members can be intercepted.

A disadvantage of both approaches is that they need additional care at instance creation time: Instead of directly instantiating the target, a proxy object should be created. Using class factories or dependency injection, both approaches can be made completely transparent to client code. Figure 3.1 and 3.2 show the discussed proxy types. Solid lines depict static aspects while dashed lines depict runtime behavior.

A disadvantage of interface proxies is that inside a target method, the *this* reference refers to the target itself and not to the proxy. This means that *self calls*, i.e., methods called on the *this* reference, cannot be intercepted. In addition, special care has to be taken to prevent the proxy from leaking its target. Because in the subclass proxy approach the proxy *is a* target, it does not have the same limitations. Proxy and target are one and the same object. Therefore, the *this* reference is automatically correct and self calls are properly intercepted [KFS06]. In the rest of this thesis, the term proxy stands for subclass proxy, if not specified otherwise.

3.2 Shortcomings of Existing Frameworks

This section explains three major shortcomings of existing frameworks by means of a common example. Consider the following real-life usage scenario for code generation: In a large project an object-relational mapping tool (O/R mapper) is utilized to store object entities in a relational database. The O/R mapper uses code generation to provide features such as lazy loading. Additionally, an AOP framework is employed to deal with cross-cutting concerns in a clean and localized manner. The AOP framework generates code to adapt the behavior of methods. Some of the cross-cutting concerns also affect classes that represent entities stored in the database. This means that certain classes need to be extended by both tools.

3.2.1 Deficient Tool Integration

Depending on the capabilities of the tools and their underlying code generation technique, the need to extend the same classes causes a set of problems. The following identifies three approaches:

(1) In the worst case, both tools use their own code generation facilities and do not account for the possibility of other tools extending the same classes. Such tools might not work together at all or cause severe problems related to caching of generated code, serialization of object instances, efficiency of the generated code, and ordering of modifications made to the code. Assume that the O/R mapper and the AOP framework are both based on the subclass proxy approach as described in the previous section. Such a solution is subject to most of the stated limitations. For example, the generated code is non-optimal and two additional types are generated for every extended class. Figure 3.3 depicts this situation.

(2) A better starting position is that both tools use their own code generation facilities but do account for other tools. A good example for this approach is Spring.NET [AM09], which comes with a simple AOP model and integrates with the data access tier. One supported data access framework is NHibernate [KBHK09], a popular open source O/R mapper. The AOP model in Spring.NET is used for declarative transaction handling and also supports user-defined aspects. Another example for this category of tools is NHibernate itself, which is based on DynamicProxy. NHibernate offers code generation hooks for adding DynamicProxy-based interceptors to the generated subclass proxies. These interceptors can be used by third party tools and application developers to implement infrastructure concerns.

Both examples improve upon the previous situation, but still have one major disadvantage: At least one of the involved frameworks needs to have knowledge about the other. In the first example, it is Spring.NET that depends on specific versions of NHibernate for integration. In the second example, both frameworks must be designed to work together. Firstly, the authors of NHibernate anticipated the need for code generation hooks and exposed a part of their code generation internals via an API. Secondly, third party tools and application developers directly depend on this API and therefore also on

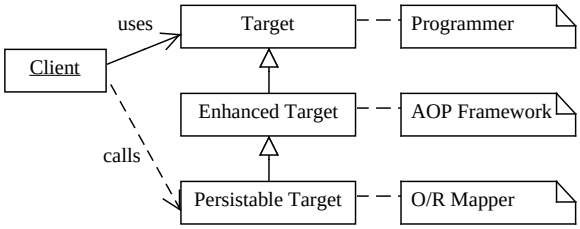

Figure 3.3: A target extended by two subclass proxies.

DynamicProxy. Note that simply using the same code generation framework is not enough. Additional effort must be made to ensure that both tools work together. For the second example, the additional effort means using the API provided by NHibernate instead of directly using DynamicProxy. This additional effort must be invested for all supported tool and framework combinations. As the frameworks have knowledge about each other, changes in one framework may trigger problems in other components. The resulting solution is not flexible, has poor maintainability, and errors might be hard to spot and debug. Therefore, the approach is unfeasible for a generalized solution that should work with many different tool and framework combinations.

(3) The third and best option is that both tools use a unified code generation facility. With this option, the tools do not generate the code themselves but instruct a separate component about what code to generate. This component aggregates all the data needed for code generation from the different tools and emits code based on it. The advantage of this approach is that the tools do not need to have knowledge about each other; instead, they can be completely independent.

3.2.2 Low Level of Abstraction

Another issue that results from every tool having its own code generation facility is duplicated effort and a low level of abstraction for tool authors. To support member interception in the AOP framework mentioned in the initial example, framework authors might use Mono Cecil to modify the involved types. This is suboptimal for two reasons: (1) For tool authors, code generation is a means to provide the intended behavior; a mere implementation detail. But nonetheless they have to deal with the intrinsic details of CLI metadata and IL code. (2) In addition, a lot of time is spent on the development of infrastructure code and services that cannot be shared between projects that use different code generation techniques.

Now consider a unified code generation facility as described in option 3 in the previous section. Because such a unified code generation facility already has all the data needed for code generation, it can provide services such as caching of generated code, serialization of object instances, and ordering of modifications made to the code. In addition, it may offer a common model for generating code on the concept layer, which shields developers from code generation details. This means that tool authors are able to work on a higher level of abstraction instead of directly emitting IL instructions with a specific code generation technique.

3.2.3 Lack of User Choice

The third shortcoming of most tools is that they are hard-wired with respect to the underlying code generation technique. This means that the decision which code generation technique to use is made by the tool. But this decision should be left to the user of the tool, that is, the application developer. Only the application developer knows the scenario, the environment and its constraints, and the other frameworks with that the tool must integrate. For example, runtime code generation with Ref.Emit might be a good choice in server environments but is not available on mobile platforms. Another constraint might be that a code generation technique requires the code to run with full trust, which is not possible in certain scenarios. Ordering of modifications made to the code by different tools is yet another issue. In the initial example, should the AOP framework or the O/R mapper be closer to the target? What if other tools and frameworks that require code generation are added? Tools cannot predict these factors and therefore should leave the decision about the underlying code generation technique to the application developer.

3.3 Code Generation in re-motion

re-motion [REM] is an open source .NET framework developed by rubicon[2]. It is licensed under the GNU Lesser General Public License (LGPL). re-motion is aimed at enterprise applications and focuses on domain-driven design (DDD) [Eva03], maintainability, and extensibility. In order to achieve these properties, some of its components, most notably re-store and re-mix, employ code generation. re-store is an O/R mapper and deeply integrates with the other re-motion components, which provide functionality like a security model or object-to-control bindings. re-mix is an elegant implementation of mixins for the .NET platform.

3.3.1 Mixins

Mixin programming [Sin11] is coined as "a style of software development where units of functionality are created in a class and then mixed in with other classes". When other classes *inherit* from a mixin they do this to collect functionality and not as a means of specialization. At rubicon, mixins are used to follow an approach named *use case slicing* [JN05]. It allows implementing the functionality needed for a certain use case or product feature in focused mixins instead of scattering the code throughout many different classes. These mixins are added incrementally to the existing domain model. The goal is to compose whole or large parts of applications from small, reusable units of functionality.

[2] www.rubicon.eu

Next to achieving a DDD-centric approach, maintainability, and extensibility, at rubicon this is also used as a means for *product line development* [Nyh02]. Related topics are *composition-based programming* [JN05] and *multi-dimensional separation of concerns* [TOHS99]. On the Java platform, Qi4j [OH12] is a popular framework for domain centric application development. It includes many of the aforementioned concepts.

3.3.2 Shortcomings

The code generation facilities in re-motion, albeit well engineered, have conceptual shortcomings. re-store and re-mix both use Ref.Emit to generate their own subclass proxies. As a consequence, two additional types are generated if a class is both persistable and enhanced by mixins. The re-motion components fall into the second category of tools characterized in subsection 3.2.1. This means that they work well together but one tool, in this case re-store, needs to have knowledge about the other, namely re-mix. One reason why this is necessary is a feature of re-mix that allows mixins to add properties to classes. The values of these properties should of course also be saved in the database when the object is persisted. Therefore, re-store needs to know if the types specified by the user are enhanced with additional properties. re-store also needs a way to examine added properties as well as, to read and write their values.

Although the parts of re-store and re-mix that are responsible for code generation are based on shared infrastructure, code generation itself is not unified. The exemplary C# code snippet in listing 3.1 sketches a user API of re-store. The illustrated method creates a persistable instance of type `T` extended by the currently configured mixins. This is accomplished by holding on to and using a reference to re-mix, which constitutes the biggest shortcoming of the approach. In addition to knowledge of re-mix in re-store, the approach is also subject to the other shortcomings described in section 3.2. Despite its issues, the approach is currently used in production.

Listing 3.1: A simplified illustration for the instantiation of a persistable object from a mixed type in re-store.

```
public T NewObject<T> () {
  Type originalType = typeof (T);
  Type mixedType = _remix.CreateTypeWithCurrentMixins (originalType);
  Type persistableMixedType = _restore.CreateTypeWithPersistence (mixedType);
  T instance = (T) Activator.CreateInstance (persistableMixedType);
  return instance;
} // method NewObject
```

The following discusses the concrete issues caused by the shortcomings of the approach. *Tangling* prevents proper separation of concerns [TOHS99], which lowers comprehensibility and increases coupling. High coupling in turn compromises reusability and extensibility. Special care has to be taken to ensure that different code generation components do not interfere with each other.

Other issues are feature duplication[3] and missing features. For example, both re-store and re-mix support the interception of properties, but only re-mix supports caching of the generated types on disk. All of these issues impair the overall maintainability. Therefore, it was decided to introduce a unified code generation facility for re-motion. The requirements for this facility are listed in the next section.

3.3.3 Requirements

The code generation component for re-motion must meet the following requirements:

(1) The component follows a pipeline approach to enable multiple independent participants to generate code collaboratively. It supports an easy way to configure the pipeline by allowing the user to add and remove participants as needed.

(2) Participants are provided with an abstract model of the code to generate, which they can use to specify their modifications. This model abstracts from the underlying code generation technique. The code is generated by pluggable code generation back-ends and the decision which back-end to use is left to the user.

(3) The component supports all functionality currently needed by re-store and re-mix. This set of features includes interception of constructors, methods, property accessors, and event accessors; introduction of new members via interfaces; and addition of inner classes.

(4) The component can be configured to cache the generated types on disk to avoid unnecessary re-generation.

(5) If requested, it ensures that instances of generated types can be serialized.

(6) Modification of code in existing assemblies is possible without requiring the source code.

(7) The generated code is highly efficient, as if it had been generated by a compiler. This excludes, for example, abstractions that represent argument lists as object arrays because converting an argument list to an object array at runtime introduces an overhead that can be avoided with a better abstraction. Note that the efficiency of the generated code greatly depends on the capabilities of the used code generation back-end.

(8) In addition to the performance of the generated code, the runtime performance of the generation process itself is not significantly slower than that of the current solution.

(9) Last but not least, the component and all of its dependencies are license-compatible with re-motion, that means, LGPL-compliant.

[3] Feature duplication should not be confused with code duplication.

The following list enumerates the described requirements for future reference.

1. Pipeline approach.
2. Abstract model.
3. Functionality required by re-store and re-mix.
4. On-disk caching.
5. Serialization.
6. Modification of code without sources.
7. High efficiency of generated code.
8. Good runtime performance of generation process.
9. License compatibility.

3.3.4 Evaluated Frameworks

The following frameworks and tools have been considered as a replacement for re-motions code generation facilities: DynamicProxy, LinFu, PostSharp, and Roslyn.

DynamicProxy was already introduced in section 2.2.3.

LinFu [Lau08] is a framework that offers similar proxying capabilities as DynamicProxy. Both tools use Ref.Emit to generate subclass proxies and therefore are subject to the limitations described in section 3.1. LinFu also offers a simple AOP model based on Mono Cecil.

PostSharp [PS] is a popular commercial AOP framework based on a custom Portable Executable (PE) writer. It is known for its simplicity and sophisticated VS integration.

Roslyn [MSR] is a long term Microsoft project to open the C# and VB.NET compilers as APIs. Through these APIs users and tools can access the data that the compilers build up during the compilation process and use it to gain an understanding about the compiled code. Roslyn offers functionality for working with the syntax and semantics of code as well as dynamic compilation. At the time of writing, Roslyn is only available in the form of a Community Technology Preview (CTP).

The numbers in the following paragraph map to the list of requirements presented in the previous section. Unfortunately, none of the considered projects meets all the requirements. The main shortcoming is that none of the projects follows a consistent pipeline approach (1). DynamicProxy allows for configuration of an interceptor chain but the approach is not sophisticated enough as a substitute for a code generation pipeline. Another point that none of the projects offers is a model that abstracts from the underlying code generation technique (2). Next to these two main points, the individual projects lack additional requirements: DynamicProxy (7), LinFu (3, 4, 7), PostSharp (6, 8), and Roslyn (6). Note that this mapping of missing functionality represents only a rough guideline.

Apart from the evaluated projects, the author does not have knowledge about projects that better meet the stated requirements. As a result, the decision was taken to develop a new code generation component.

In addition to the original requirements, the in-house development must fulfill the following: The developed software component is a general purpose code generation pipeline that can not only be used by re-motion but also by other projects. Development is test-driven. Note that the current code generation infrastructure in re-motion is based on internal parts of DynamicProxy. Replacing that infrastructure with the newly developed component removes the dependency on DynamicProxy, which is a nice side effect. The rest of this thesis deals with the design, implementation, and test of a code generation pipeline that fulfills the stated requirements.

Chapter 4

Pipeline Design

4.1 Goal

The main goal of the pipeline is to allow multiple independent components to generate code collaboratively. A component participates in the code generation process, hence we call it *participant*. The key characteristic about participants is that they are independent. This means that they do not have knowledge about each other or might not even know about each other's existence. A participant may be an O/R mapper, an AOP framework, an inversion of control (IoC) container, a mixin library or any other tool that needs to generate code.

The pipeline has two kinds of users: tool authors and application developers. Tool authors implement participants. They use the APIs offered by the pipeline to specify the code to generate. Application developers configure the pipeline by adding participants to it. They use the generated types in their application. The pipeline itself usually does not provide useful features to the application developer. It is rather an enabling technology that allows different tools to work together.

4.2 Architectural Overview

Figure 4.1 shows an architectural overview of the pipeline. Gray shapes and the tube labeled "TypePipe" represent pipeline components. TypePipe is the project name of the developed code generation pipeline. Blue shapes depict participants, which are developed by tool authors. Note that the illustrated participants are just an example for a set of configured participants. Green shapes constitute elements provided or used by the application developer. Dashed shapes represent components that have yet to be developed. For details on possible future work, see section 7.2.

The following gives a coarsely grained outline of the pipeline's code generation process. The process is based on CLR types and starts with the

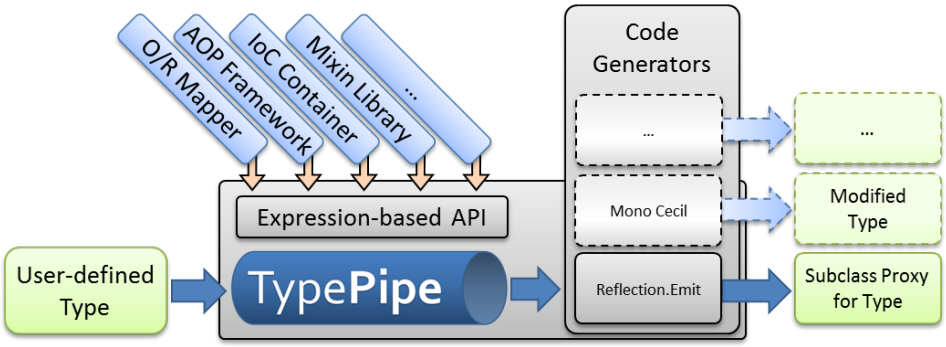

Figure 4.1: Architectural overview of the pipeline with example participants.

application developer. An application developer provides a type for modification or requests the generation of a new type. If a new type is requested, the pipeline creates it. In the second phase, the pipeline invokes the configured participants in the specified order and supplies them with the type. Each participant gets the chance to analyze the type and specify modifications via an expression-based API. Next, the pipeline hands the modified type to one of the pluggable code generators. The code generator emits code based on the modifications made by the participants. The concrete form of the resulting type greatly depends on the used generator. The Ref.Emit-based generator, for example, creates a subclass proxy while the generator based on Mono Cecil directly modifies the original type. Regardless of the concrete form of the generated type, in the end, the application developer receives a type with behavior as specified by the participants.

The key characteristic of the described process is that the pipeline orchestrates the participants. Instead of generating code themselves, participants are invoked by the pipeline and specify which code to generate through the provided APIs. This is the concept of IoC [Fow02] applied to code generation.

4.3 Data Flow between Participants

The flow of modification data from one participant to the next is an important concept in the design of the pipeline. It determines the way and the possibilities for a participant to react to modifications made by previous participants. As noted in the previous section, an application developer may provide an existing type or request the generation of a new type. If an existing type is provided, it is wrapped in a mutable representation of the type. We call the concept of this mutable type representation *mutable type*. Every mutable type has an *underlying type*, that is, the type it is based on. In this case the wrapped existing type is the underlying type. Participants can access the underlying type via the mutable type. If the generation of a new

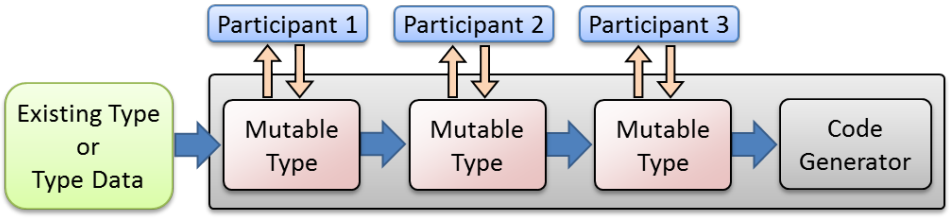

Figure 4.2: Data flow between pipeline participants.

type is requested, the application developer specifies the properties of the type with the request. Type properties include, among others, type name, base type, and implemented interfaces. From these properties the pipeline again creates a mutable type. The underlying type of a mutable type that is not based on an existing type is the mutable type itself.

In the next phase, the pipeline invokes the participants one after the other and hands them the mutable type. This process is depicted in figure 4.2. The participants and the order in which they are invoked can be configured by the application developer. The purpose of the mutable type in this process is twofold: On the one hand, it acts as a query model. Participants can inspect type properties and investigate the type's members. On the other hand, it is a mutable representation of a type, hence allowing participants to specify modifications. This is challenging because both functions depend on each other and must be kept in sync. For example, adding a member with the same name and signature as an already existing member is not allowed. Furthermore, after adding or modifying a member, the change is immediately reflected in the query model. This ensures that subsequent participants see the modifications made by previous participants. It also prevents modifications by subsequent participants conflicting with modifications by previous participants.

After each participant has been invoked, the mutable type is handed to the code generator. Note that no actual code is generated when participants specify their modifications. The mutable type simply aggregates the modifications and updates the query model accordingly. These aggregated modifications are made available to the code generator via the Visitor pattern [GHJV94]. Based on the modification data, the code generator emits a type that can be used by the application developer.

The initial design of the pipeline did not have the concept of a mutable type that is both a query model and a modification model. Rather, the initial design featured a specification-based approach. These specifications modeled the type modifications as first class citizens. Participants were supplied with a separate query model and created specifications to request modifications. The pipeline's responsibility was to combine these specifications before code

generation. This specification-based approach was not feasible for two reasons. First, it did not provide a clean way to deal with conflicting specifications. Second, participants could not see and therefore could not account for modifications made by previous participants. The mutable type concept evolved organically from the need to enable these scenarios.

4.4 Mutable Reflection

The notion of *mutable reflection* refers to the mutable type and its members. The implementation class of the mutable type concept derives from the abstract `System.Type` base class. It provides access to a type's members, which are also mutable. The implementation classes of the mutable members derive from the abstract base classes in the `System.Reflection` namespace. For example, the class representing a mutable method derives from `MethodInfo`. The mutable reflection classes constitute the participant-facing API of the pipeline. This is shown in figure 4.3.

Inheriting from the abstract base classes in the `System.Reflection` namespace has far-reaching consequences. These base classes declare a broad API but do not provide an implementation for it. This means that deriving from them imposes a significant implementation overhead. In addition, the `System.Reflection` API is quite aged and can be cumbersome to work with. On top of that, some of the declared operations are not applicable at code generation time at all. Standard reflection is based on framework-internal

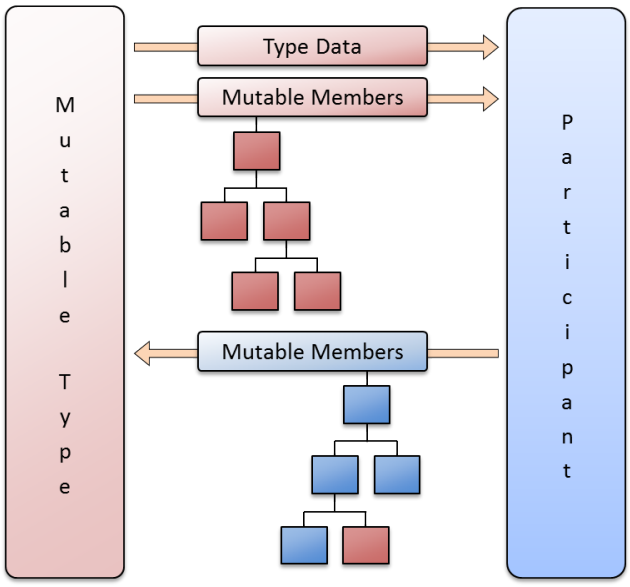

Figure 4.3: Mutable reflection concept.

implementation classes[1], which represent types and their members. These classes are heavily intertwined with the CLR and access internal CLR data structures through unmanaged APIs. With standard reflection, it is possible, for example, to invoke a method dynamically. At code generation time however, a mutable method just represents a method that should be generated but no code was yet emitted. Therefore, a mutable method cannot be invoked. The same holds for all runtime-like operations of the mutable reflection classes.

But deriving from the classes in the `System.Reflection` namespace also has advantages. The mutable reflection classes are designed to be directly usable in the representation of method bodies. This means that method bodies can refer to types and members that have yet to be generated. No mapping or intermediary objects are needed. This is only possible by deriving from the standard reflection base classes. For more details and a description of the representation of method bodies, see the next section. Another advantage is that many developers and especially tool authors are familiar with the `System.Reflection` namespace. This means that working with the mutable reflection classes causes low friction. Eventually, it was decided that the advantages of this design outweigh its disadvantages.

Another detail of the mutable reflection classes is the range of offered operations. Note that different code generation techniques have different capabilities. However, the range of offered operations is not determined by the lowest common denominator of those capabilities. This means that the mutable reflection classes offer operations that cannot possibly succeed with certain code generation techniques. An example for this is adapting the behavior of non-virtual methods when using the Ref.Emit-based code generator. Without further work, the problem would not be noticed until code generation. For this reason, it would be hard to identify the participant responsible for a given error. In the pipeline, this problem is solved by applying the Strategy pattern [GHJV94]. The mutable reflection classes are injected with a code generation-dependent strategy, which is used to validate that an operation is viable with the used code generator. This allows the pipeline to catch most errors related to code generation capabilities at specification-time. When the pipeline encounters such an error, it raises an appropriate exception. Hence, the participant that caused an error can be immediately identified from the resulting *stack trace*. Participants may even handle the exception themselves and provide a fallback, if possible. Also note that through the use of the Strategy pattern the mutable reflection classes are independent from code generation. See section 4.6 for details on this topic.

[1] For example, `RuntimeType`, `RuntimeMethodInfo`, `RuntimePropertyInfo`, ...

4.5 Representation of Method Bodies

In order to represent the bodies of mutable methods, a powerful model is needed. Ref.Emit and Mono Cecil both represent method bodies as streams of IL instructions. For the pipeline, this form of representation is unsuitable. The main problem is that IL instruction streams cannot be easily composed. This is an issue because the individual parts of a method body might be specified by different participants. Participants need the ability to investigate method bodies and compose new ones by combining new parts with parts of the existing body. Another argument against modeling method bodies with IL instructions is their low level of abstraction. Tool authors should not need to be IL experts [Lid06] to develop participants.

ASTs are a natural way to represent method bodies on a higher level. ASTs can be processed elegantly using the Visitor pattern and can be easily composed from other or parts of other ASTs. Figure 4.3 illustrates a participant that queries the body of a mutable method. Subsequently, the participant replaces the original body with a newly created one. Note that the new body (blue nodes) incorporates the original body (red node).

The development of an AST, powerful enough to represent method bodies would take a big effort. Fortunately, there are a few readily usable ASTs available in the .NET ecosystem. The VS Code Model and CodeDOM both use an AST as their primary data structure and API surface. DynamicProxy contains an AST internally and the namespace `System.Linq.Expressions` constitutes yet another one. The VS Code Model is a so-called *design time language model*. Its purpose is to represent source code at design time. CodeDOM is used to create and compile source code. That means that those ASTs are source-centric. Therefore they are language-specific and have requirements like explicit representation of errors and syntactic fidelity to the source code[2]. These requirements do not match with the requirements of the pipeline, that is, semantic representation of method bodies. The AST of DynamicProxy is such a semantic representation. It is used in re-motions current code generation facilities. From this experience, it was concluded that the DynamicProxy AST is not powerful enough and not of high enough quality to use it for the pipeline. This rules out all but one considered AST implementations.

The AST defined by the classes in the namespace `System.Linq.Expressions` was added to the BCL in version 3.5 of the .NET framework. As one might infer from the namespace, it originally was designed to represent Language Integrated Query (LINQ) expressions. Initially, it supported only expressions, hence the name expression tree (ET) [Chi11]. In version 4.0 of the .NET framework, ETs have been extended to also support statements. ETs

[2]It can be argued that those data structures resemble *concrete syntax trees* or *parse trees* [ASU86] rather than ASTs.

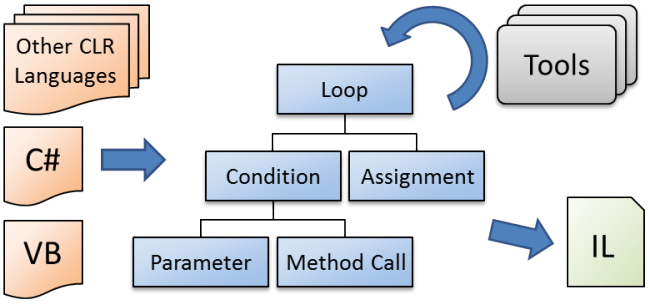

Figure 4.4: Expression trees are semantic models of code.

do not exactly model the syntax of the language but rather the semantics of the formed expressions. They provide the right level of abstraction by representing common constructs found in CLR languages. Examples for such constructs are loops, conditions, and method calls. ETs enable the representation of any static method in a language-independent way and shield the developer from the underlying IL instructions. On top of that, ETs offer an API to compile and run the code represented by the tree. Today, ETs are known as a generally useful semantic code model. They can serve as a "common currency" in metaprogramming tools, i.e., tools that treat programs as data. Figure 4.4 illustrates this concept. Because of the characteristics described in this section, we chose ETs to represent the bodies of mutable methods. For implementation details and a description of encountered issues, see section 5.2.

4.6 Abstraction of Code Generation Techniques

Apart from the pluggable code generators, the pipeline is independent from the underlying code generation techniques. A code generator encapsulates all details specific to a code generation technique. Depending on the underlying capabilities of the technique, the pipeline can be used for different scenarios. The participants, however, are shielded from these differences. Table 4.1 compares two important code generation techniques: Ref.Emit and Mono Cecil. The Ref.Emit-based generator dynamically creates subclass proxies at runtime, as depicted in section 3.1. The Cecil-based generator directly modifies existing types inside assemblies and is commonly used in a post-compile step.

Section 4.4 mentioned that the operations offered by mutable reflection are not constrained by the lowest common denominator of the different code generation capabilities. Operations that are not supported by the current code generator *fail fast* by raising a descriptive exception. This is achieved by using the Strategy pattern to check if the requested operation is available.

Characteristic	*Reflection.Emit*	*Mono Cecil*
Generation time	Code is generated at runtime.	Code is generated in post-compile step (requires additional build step).
Runtime overhead	Time is spent during application startup or runtime.	None, time is spent up front.
Limitations	Limited to subclass proxy generation.	None, types can be modified at will.
Instantiation	Object creation via factory. `factory.Create<MyObject>();`	Object creation via constructor. `new MyObject();`
Granularity	Specific modifications on type and instance level (through factory).	Modifications only on type level.
Flexibility	Can use runtime data.	All decisions must be made at compile time.

Table 4.1: Comparison of code generation techniques.

Instances of the validation strategy are created by the used code generator and injected into the mutable reflection implementation classes. The second point where the pipeline interacts with the code generator is just after the last participant finished and the mutable type is handed over to the generator. At this stage, the generator can process the modifications recorded by the mutable type with the help of the Visitor pattern or simply use the query model to analyze the mutable type. This procedure is described in section 4.3.

As part of this thesis, a Ref.Emit-based code generator was developed. For a description of the implementation see section 5.3. The development of additional code generators based on Mono Cecil, CodeDOM or Roslyn is regarded as future work. Section 7.2 provides a brief overview of areas considered for future work.

Chapter 5

Implementation

5.1 Mutable Reflection

The mutable reflection domain is an integral part of the pipeline. It serves as both, the query and the modification model for participants. Another, less obvious, responsibility of mutable reflection objects is their usage inside ETs. To enable this, the mutable reflection objects are derived from their counterparts in the `System.Reflection` namespace, which requires reimplementing large parts of the reflection API. The reasoning behind this design is described in section 4.4. The design, however, imposes different responsibilities onto the mutable type implementation and therefore violates the *single responsibility principle* [Mar00]. To mitigate this problem and reduce complexity, the mutable type implementation has been broken up into two classes as shown by figure 5.1.

On top of the mutable type hierarchy is the abstract class `System.Type`, which serves as an entry point to .NET reflection. An instance either represents a class, a value type, an interface or a generic parameter. The internal class `RuntimeType` is the concrete implementation of `Type` used for standard reflection. `TypeBuilder` and `GenericTypeParameterBuilder` are located in the namespace `System.Reflection.Emit` and are part of the Ref.Emit API. The former is used to create new types while the latter is used to define generic parameters for newly created types and methods.

For use in the pipeline, the class `CustomType` was derived from `Type`. The responsibility of this class is to bridge the gap between standard reflection and the mutable reflection model. It implements all abstract methods defined in `Type`. For example, there are six `GetMethod` overloads that all delegate to the abstract method `GetMethodImpl`. The full signature of this method is shown in listing 5.1. We will not discuss the meaning of the individual parameters here, but note that it is a rather complex method. In addition, there is a similar set of methods for fields, constructors, properties, events, and nested types. Also note that `Type` declares a number of operations that cannot be

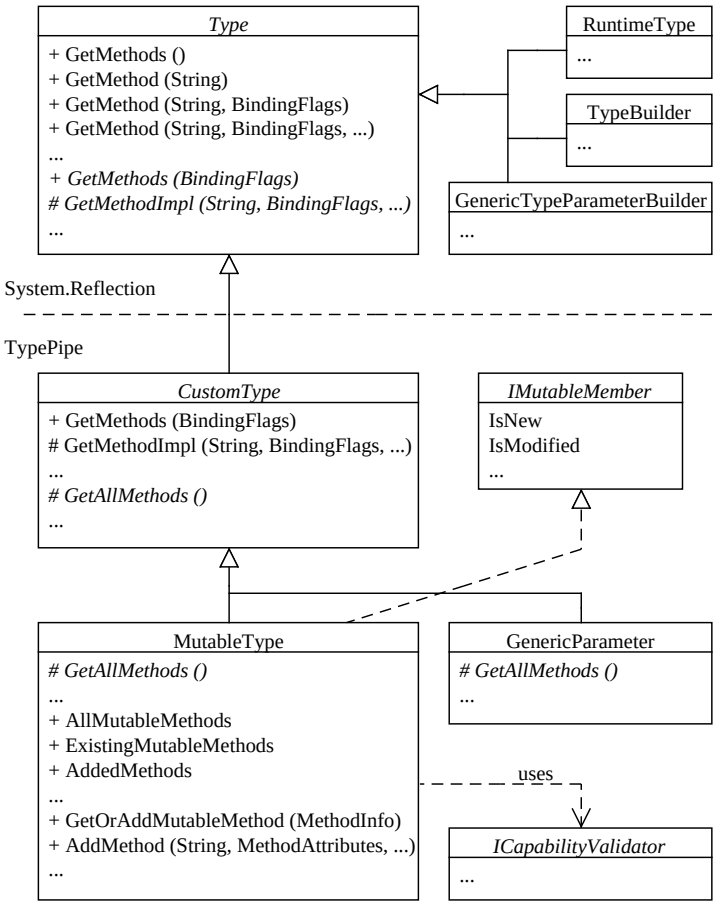

Figure 5.1: Mutable type hierarchy.

supported in the context of the pipeline. Examples of such operations are the methods `InvokeMember` and `MakePointerType`. In short, `CustomType` implements the semantics of standard reflection where possible and allows other classes to inherit this implementation. This also enables code sharing between the mutable type implementation and the `GenericParameter` class, which is the pipeline's equivalent of Ref.Emits `GenericTypeParameterBuilder`.

Listing 5.1: Signature of method `GetMethodImpl` of class `Type`.

```
protected abstract MethodInfo GetMethodImpl (
    string name,
    BindingFlags bindingAttr,
    Binder binder,
    CallingConventions callConvention,
    Type[] types,
    ParameterModifier[] modifiers
);
```

The concrete class `MutableType` is the implementation class for the mutable type concept. It derives from `CustomType` and adds the concept of mutability. For example, it allows the addition of new methods and provides convenience properties for typed access to mutable methods. Requested modifications, of course, must also be reflected in the query methods implemented by `CustomType`. This means that `CustomType` needs a way to access state from its subclasses. This is accomplished by using the Template Method pattern [GHJV94] in a similar way to how it is used by `Type`. Listing 5.2 shows the signature of the template method for accessing methods as declared by `CustomType`. Unlike the method `GetMethodImpl` of class `Type`, it is easy to implement.

Listing 5.2: Signature of method `GetAllMethods` of class `CustomType`.

```
protected abstract IEnumerable<MethodInfo> GetAllMethods ();
```

The last unmentioned elements of the diagram in figure 5.1 are the interfaces `IMutableMember` and `ICapabilityValidator`. The `MutableType` class implements `IMutableMember`, which declares characteristics that are shared by all mutable members. These characteristics include properties that let participants determine if a member was modified and whether it was created from scratch or based on an existing member. The interface `ICapabilityValidator` constitutes the strategy used to check if a requested modification is viable with the current code generator. It is injected into `MutableType` via the constructor. Besides the strategy, the constructor takes a parameter of type `UnderlyingTypeDescriptor`. An object of this type holds all the data that is needed to fully initialize a `MutableType` instance. This includes type name, type attributes, base type, implemented interfaces, fields, constructors, methods and so forth. If the mutable type is based on an existing type, the descriptor also contains this existing type to allow the initialization of the mutable type's underlying type property. Through the usage of the descriptor, the actual `MutableType` class does not need to distinguish whether it represents a newly created type or a type that is based on an existing one. For fields, methods and all other mutable members there are similar descriptors that have the same purpose.

There is one more interesting aspect involving the `MutableType` constructor. Two general guidelines for constructors are that they should (1) *do no real work* and (2) *yield a fully usable object* [HWR08].

The first point suggests that a constructor should not contain complex initialization logic. In particular, it should not be responsible for instantiating collaborators or creating complex object graphs. Violation of this guideline results in inflexible design and code that is hard to test.

The second point represents a cardinal rule of object-oriented design: Class invariants are established by the constructor and must remain uncompromised regardless of the state of an object. Note that in this context the term *fully usable* should not be confused with *fully initialized*. Initialization

logic might be deferred, as long as the class invariants hold from the user's point of view.

Unfortunately, the mentioned guidelines conflict when applied to the `MutableType` constructor. Remember that the mutable type holds references of its mutable members. Ideally, these mutable members are injected into the `MutableType` constructor via the descriptor. The mutable members, however, have a property `DeclaringType` which is essentially a back-reference to the containing `MutableType` instance. This constitutes a circular dependency: One thing cannot be correctly created without the other. The solution we decided to use was to instantiate the mutable members inside the `MutableType` constructor. This violates the first guideline, which states that constructors should do no real work. We argue that in this situation it is the lesser evil because the `MutableType` class is tailored towards a very specific need. This means that flexibility and reusability are less of an issue. On the testing-front, although cumbersome, we spent extra effort on thoroughly testing all of `MutableType`'s members, including its constructor. The decision was reinforced by the fact that the mutable reflection classes constitute the front-facing API of the pipeline. Therefore, it is important that their instances are fully usable at any time.

5.1.1 Mutable Type API

The following sections take a closer look on the API offered by `MutableType` and `MutableMethodInfo` for working with mutable methods. In particular, we illustrate the rather complex shape of the API and explains why it cannot be simpler.

Listing 5.3 states the operations provided by `MutableType` for working with mutable methods. The method `GetOrAddMutableMethod` fulfills two functions:

(1) It can be used to retrieve the mutable counterparts for standard `MethodInfo` objects. For this to work, the underlying method must be declared on the type represented by the mutable type. The members `AllMutableMethods`, `ExistingMutableMethods`, and `AddedMethods` are convenience properties. They allow participants to query for mutable methods directly, instead of getting the according `MethodInfo` object first and then calling `GetOrAddMutableMethod` with it. Note that `AllMutableMethods` is different from the standard reflection API `GetMethods` in the way that it does not include methods inherited from base types. The reason for this is that only members that are declared by the mutable type can be altered.

(2) The second function of `GetOrAddMutableMethod` provides mutable methods for methods declared in base types. Above we mentioned that members that are not declared by the mutable type cannot be altered. To work around this, `GetOrAddMutableMethod` overrides the specified method in the mutable type and returns the mutable version of the new override. If an override already existed for the specified method, the mutable version of the exist-

ing override is returned. The described behavior is helpful for interception scenarios. Participants do not need to distinguish between overriding and modifying for method interception. `GetOrAddMutableMethod` also works nicely for two or more participants intercepting the same base method. The first call will create an override, while subsequent calls return the existing override. This enables tool authors to develop robust participants, which work regardless of their position in the pipeline with reasonable effort. Note that only methods that are both virtual and not final can be overridden. This is an intrinsic constraint of .NET and therefore holds for all code generators.

Listing 5.3: `MutableType` API for working with mutable methods.

```
public class MutableType : CustomType, IMutableMember {
  ...
  public ReadOnlyCollection<MutableMethodInfo> AllMutableMethods { get; }
  public ReadOnlyCollection<MutableMethodInfo> ExistingMutableMethods { get; }
  public ReadOnlyCollection<MutableMethodInfo> AddedMethods { get; }

  public MutableMethodInfo GetOrAddMutableMethod (MethodInfo method);

  public MutableMethodInfo AddMethod (
      string name,
      MethodAttributes attributes,
      Type returnType,
      IEnumerable<ParameterDeclaration> parameterDeclarations,
      Func<MethodBodyCreationContext, Expression> bodyProvider
  );
  public MutableMethodInfo AddGenericMethod (
      string name,
      MethodAttributes attributes,
      IEnumerable<GenericParameterDeclaration> genericParameterDeclarations,
      Func<ReturnTypeContext, Type> returnTypeProvider,
      Func<ParameterContext, IEnumerable<ParameterDeclaration>>
          parameterDeclarationsProvider,
      Func<MethodBodyCreationContext, Expression> bodyProvider
  );
  ...
} // class MutableType
```

In addition to the members already discussed, listing 5.3 shows two more method-related APIs: `AddMethod` and `AddGenericMethod`. As the names suggest, they can be used to add non-generic and generic methods to the mutable type. All but the last parameter of `AddMethod` are straightforward. Participants naturally have to specify name, attributes, return type, and parameters of the new method. The attributes determine method properties including visibility, class or instance affiliation, final/virtual/abstract flags, *virtual table* layout, and other CLI flags. Parameters are defined by providing parameter declarations. A parameter declaration contains the name, type, and attributes of the parameter. Specifying parameter attributes is only necessary for out/ref or optional parameters, and parameters that have default values.

The last parameter of `AddMethod` is more complex than the previous ones. Instead of directly providing the method body as an expression, participants have to specify a provider that returns the body. The reason for this design

is that it allows deferring the execution of the body provider. Note that the parameters are specified in form of declarations rather than instances of type `ParameterExpression`. From these declarations, the pipeline creates the parameter expressions and passes them to the body provider through the `MethodBodyCreationContext` object. Through the context object, participants have access to expressions representing parameters, the *this* reference, and other constructs that are helpful when building method bodies. Some of these constructs are discussed in section 5.1.3. Another effect of this design is that it gives control over instantiating the parameter expressions to the pipeline and therefore takes away this responsibility from the participant.

The method `AddGenericMethod` is similar to `AddMethod` but it is more complex due to the generic parameters. Name, attributes and body provider parameters have the same meaning as in the non-generic version. The third parameter specifies the generic parameters. As with normal parameters, participants do not create generic parameters themselves, but provide declarations and leave the instantiation of the actual generic parameters to the pipeline. Participants, however, need access to the instantiated generic parameters in order to specify return type and parameter declarations. The solution is similar to the one employed for specifying the method bodies: Using delegates to defer execution. Instead of specifying return type and parameter declarations directly, participants supply providers, which have access to the generic parameters through a context object. The same mechanism is also used for declaring interface constraints on generic parameters. For easier comprehension, listing 5.4 shows an example usage of the `AddGenericMethod` API. On top of the listing is the example usage and on the bottom is the definition of an equivalent method in C#.

Listing 5.4: Example usage of `AddGenericMethod`.

```
var genericParameterDeclaration = new GenericParameterDeclaration (
    "T",
    attributes: GenericParameterAttributes.DefaultConstructorConstraint,
    interfaceConstraintProvider: ctx =>
        new[] { typeof (IComparable<>).MakeGenericType (ctx.GenericParameter) }
);

mutableType.AddGenericMethod (
    "Method",
    MethodAttributes.Public | MethodAttributes.Static,
    new[] { genericParameterDeclaration },
    ctx => typeof (IEnumerable<>).MakeGenericType (ctx.GenericParameters[0]),
    ctx => new[] { new ParameterDeclaration (ctx.GenericParameters[0], "arg") },
    ctx => Expression.NewArrayInit (
        ctx.GenericParameters[0],
        new[] { ctx.Parameters[0], Expression.New (ctx.GenericParameters[0]) }
    )
);

// the code above creates a method with the same shape as the one below
public static IEnumerable<T> Method<T> (T arg)
    where T: IComparable<T>, new() {
  return new T[] { arg, new T() };
} // method Method<T>
```

5.1.2 Mutable Method API

Mutable methods are represented by the class `MutableMethodInfo`, which is derived from the standard `MethodInfo` reflection class. Listing 5.5 shows members that are introduced by `MutableMethodInfo` in addition to the members declared by `MethodInfo`.

Listing 5.5: Members introduced by the `MutableMethodInfo` class.

```
public class MutableMethodInfo : MethodInfo, IMutableMethodBase {
  ...
  public MethodInfo UnderlyingSystemMethodInfo { get; }

  public bool IsNew { get; }
  public bool IsModified { get; }

  public IEnumerable<ParameterExpression> ParameterExpressions { get; }
  public Expression Body { get; }
  public bool CanSetBody { get; }
  public void SetBody (
      Func<MethodBodyModificationContext, Expression> bodyProvider);

  public MethodInfo BaseMethod { get; }

  public ReadOnlyCollection<MethodInfo> AddedExplicitBaseDefinitions { get; }
  public bool CanAddExplicitBaseDefinition { get; }
  public void AddExplicitBaseDefinition (MethodInfo overriddenMethodBaseDefinition);
  ...
} // class MutableMethodInfo
```

The properties `IsNew` and `IsModified` are defined in the `IMutableMember` interface, which is transitively inherited through `IMutableMethodBase`. `IsNew` can be used to check whether the mutable method was newly created or is based on an existing method. `IsModified` simply returns if the mutable method was modified after creation. The property `UnderlyingSystemMethodInfo` returns the underlying method for existing methods and the mutable method itself for new methods. This behavior is analogous to the behavior of property `UnderlyingSystemType` of class `MutableType`.

The properties `ParameterExpressions` and `Body` represent the parameters and the body of the method in form of expressions. While the parameters are fixed, the body can be changed with the method `SetBody`. This method takes a similar body provider as the method `AddMethod` described in the previous section. The only difference is that the body provider is supplied with a slightly different context object. In this case, the context object is of type `MethodBodyModificationContext` instead of `MethodBodyCreationContext`. In addition, there is the property `CanSetBody` which states if changing the body of this mutable method is a viable operation with the current code generator. For example, while a code generator based on Mono Cecil should be able to modify any method body, a Ref.Emit-based code generator cannot modify mutable methods that are based on existing, non-virtual or final methods. The `CanSetBody` property is implemented by using an injected strategy as described in section 4.6. When the property returns false, the `SetBody` method

raises an exception. This behavior follows the *fail fast* approach, which is also described in the referenced section. The members outlined in this paragraph constitute the main functionality of the `MutableMethodInfo` class. They enable modification of method bodies using ETs.

The last set of members allows participants to create explicit method overrides. Before we can take a closer look at the members we have to understand the semantics of method overrides in .NET. Method overrides are a very specific and complex subject and therefore we will only touch on the subject here. For a complete illustration we refer to [Lid06, Chapter 10].

Method Overrides

Method overrides are subject to a number of general constraints regarding the *overridden* as well as the *overriding* methods. All participating methods must be virtual instance methods declared in the same type hierarchy and need to have matching signatures. A CLI method signature essentially contains the return and parameter types next to a few other things. It does not contain, however, the name of the method. In addition, the overridden method must not be final. There are two types of overrides: Implicit and explicit method overrides.

The concept of implicit method overrides is a common feature in object-oriented programming and generally well understood. However, there are nuances worth mentioning. The ingredients of an implicit override are methods that conform to the mentioned constraints and also have the same name. The left hand side of figure 5.2 depicts the implicit override hierarchy of a method `Foo`. We will look at the hierarchy from the point of view of class `M` which represents the type that is currently modified. From this point of view, method `M.Foo` is the overriding method while class `A`, `B`, and `C` declare the overridden methods. If any of these methods is called with dynamic binding, `M.Foo` is invoked instead.

The method on top of the override hierarchy is called *base definition*. In an override hierarchy the base definition is the only method that has the *newslot* flag set in its method attributes. This is because the newslot flag affects the virtual table layout. Methods that have the flag enabled are guaranteed a new slot in the *virtual method table*. Therefore, they are not part of an existing override hierarchy but instead start their own hierarchy. For example, assume that `C.Foo` is declared with the newslot flag set[1]. The single override hierarchy is cut into two and `C.Foo` becomes a base definition itself. This means that when `A.Foo` is called with dynamic binding, `B.Foo` is invoked instead of `M.Foo`. The base definition of `M.Foo` is now `C.Foo`.

A method's *base method* is simply the most derived method in the override hierarchy that is overridden by this method. In the example, the base

[1] In C# this can be accomplished by replacing the `override` keyword with the `new` keyword.

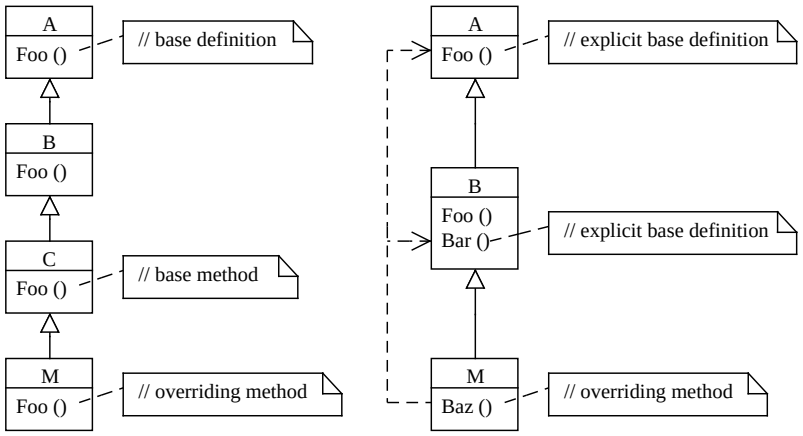

Figure 5.2: Implicit and explicit method overrides.

method of `M.Foo` is `C.Foo`, the base method of `C.Foo` is `B.Foo`, etc. Note that a base method is not necessarily defined in the direct base type of a method's declaring type. In C#, base methods can be called by using the `base` qualifier keyword. For call sites that refer to a method via the `base` keyword the C# compiler emits non-virtual calls. The reason is that base methods must be invoked without dynamic binding from code in derived types in order to avoid infinite recursion. The next section introduces an API that can be used in this situation to build correct ETs.

The member `GetBaseDefinition` of class `MethodInfo` can be used to get the base definition of a method. For methods that already constitute base definitions, `GetBaseDefinition` returns the same method. It never returns `null`. On the other side, standard reflection does not offer a straightforward way to access base methods. Therefore, a respective property has been added to the `MutableMethodInfo` class to make working with implicit method overrides more convenient. The property returns `null` for methods without a base method. Base definitions and base methods apply to implicit overrides only.

Besides implicit overrides, the CLR also supports explicit overrides. The difference is that explicit overrides do not require that involved methods have the same name. As there is no implicit matching, explicit overrides are declared with additional metadata. The right hand side of figure 5.2 illustrates an explicit override hierarchy. The dashed arrow represents the additional metadata in class `M`. This metadata specifies two explicit base definitions for method `Baz`, which therefore overrides `A.Foo` and `B.Bar`. If one of the two methods, or any method that implicitly overrides them, is called with dynamic binding, `M.Baz` is invoked instead. This includes method `Foo` of class `B`. Note that explicit overrides do not have the concept of a base method.

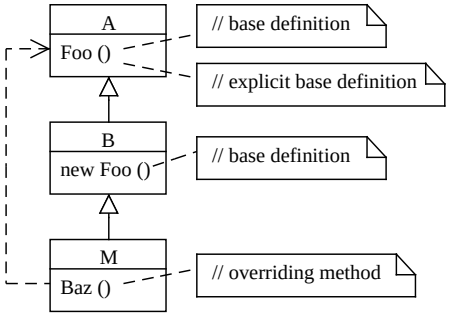

Figure 5.3: Overriding a shadowed method requires an explicit override.

A method can override both a single method implicitly as well as multiple methods explicitly. However, a certain method can be overridden at most once. Note that C# language does not support the declaration of explicit overrides by the programmer. Internally, the C# compiler uses them to generate explicit interface implementations [Coo11]. To declare an implicit method override with the pipeline, a participant needs to add a method that fulfills the mentioned constraints. This means that the method needs the right attributes and a matching signature and name. A more convenient option is to pass an overridable method to the `GetOrAddMutableMethod` method as discussed in the previous section. If possible, `GetOrAddMutableMethod` creates an implicit override and otherwise falls back to an explicit override.

Figure 5.3 illustrates a situation that requires an explicit override. The classes `A` and `B` each define a virtual instance method `Foo` with equal signatures. Both methods have the *newslot* flag set and therefore start new override hierarchies. Assume that we want to override method `A.Foo` in class `M`. From the point of view of class `M` the method `A.Foo` is *shadowed* by `B.Foo`. A method `M.Foo` would implicitly override the *shadowing* method `B.Foo` but not the shadowed method `A.Foo`. Hence, overriding `A.Foo` requires an explicit override. The name of the overriding method is different to prevent it from implicitly overriding `B.Foo`. However, it would be possible for a method `M.Foo` to override both `B.Foo` implicitly and `A.Foo` explicitly. Note that in C# it is not possible to override shadowed methods[2] because explicit overrides are not supported.

To declare explicit method overrides, participants can use the three remaining members in listing 5.5. The properties `AddedExplicitBaseDefinitions` and `CanAddExplicitBaseDefinition` can be used to query previously added explicit based definitions and to check whether adding new ones is a viable operation. Existing explicit overrides cannot be investigated through the reflection API. The method `AddExplicitBaseDefinition` allows to add explicit

[2]In the context of C# shadowing is also often referred to as *hiding*.

base definitions if it is supported by the current code generator. The shape and relationship of this member triple is similar to the triple `Body`, `CanSetBody`, and `SetBody`.

5.1.3 Method Body Context API

The purpose of the context classes mentioned in the previous section is to help participants to build method bodies. When adding new methods, participants are supplied with an object of class `MethodBodyCreationContext`. Listing 5.6 shows the most important members of this class. Note that the members are not declared by `MethodBodyCreationContext` but inherited from its base types.

Listing 5.6: `MethodBodyCreationContext` API for building method bodies.

```
public MutableType DeclaringType { get; }

public bool IsStatic { get; }
public Expression This { get; }

public Type ReturnType { get; }
public ReadOnlyCollection<ParameterExpression> Parameters { get; }

public bool IsGeneric { get; }
public ReadOnlyCollection<GenericParameter> GenericParameters { get; }

public bool HasBaseMethod { get; }
public MethodInfo BaseMethod { get; }

public MethodCallExpression GetBaseCall (string methodName,
                                         params Expression[] arguments);
public MethodCallExpression GetBaseCall (MethodInfo baseMethod,
                                         params Expression[] arguments);

public Expression GetCopiedMethodBody (MutableMethodInfo otherMethod,
                                       params Expression[] arguments);
```

The property `DeclaringType` is a simple convenience property that gives typed access to the declaring type of the method. `IsStatic` is another convenience property that can be used to check whether the method is a class or instance method. The `This` property can be used to obtain an expression that represents the *this* reference. Under the covers, the pipeline creates a `ThisExpression`, which is explained in section 5.2.2. For static methods, the `This` property raises an exception.

The property `ReturnType` holds the return type of the method, which may be `void`, a standard type, a mutable type or a generic parameter if the method is generic. Accessing and assigning parameters is an integral part when building method bodies with ETs. Therefore, the method parameters can be retrieved through the typed `Parameters` collection. The collection is populated from the parameter declarations specified by the participant.

Whether or not a method is generic can be determined with the `IsGeneric` property. Furthermore, the property `GenericParameters` gives access to the

generic parameters in a similar way `Parameters` does for normal parameters. When the property `HasBaseMethod` is true, `BaseMethod` can be used to access the base method of the method whose body is currently built. If there is no base method, the property raises an exception.

The previous section mentions that base methods must be called without dynamic binding in order to avoid infinite recursion. Remember that ETs have no notion for declaring classes or inheritance. Therefore, they provide no possibility to distinguish between normal method calls and the invocation of methods declared in base types. All virtual methods are called with dynamic dispatch semantics by default. This is not only a problem when calling base methods, but whenever a participant wants to call a method declared in a base type that is overridden in the current type. The two overloads of method `GetBaseCall` help in this situation. They create a `MethodCallExpression` that represents a call to the specified method without dynamic binding. This expression can be used to call base methods and other methods that are overridden by the current type.

The method `GetCopiedMethodBody` copies the body of another mutable method. This is helpful in situations when a method is modified but the original behavior is still needed. In an interception scenario, for example, it might be necessary to create a *delegate* for a method and then modify that method. As delegates are essentially references to methods[3], the behavior of the delegate changes with the underlying method. To retain the original behavior the method body is copied into a new method and then the delegate is created for the new method. So, subsequent modifications of the original method do not affect the behavior of the delegate.

Besides the class `MethodBodyCreationContext`, there is another context class named `MethodBodyModificationContext`. While the former is used to build bodies for new methods, the latter helps with building bodies when modifying existing methods. The modification context class has all the members of the creation context plus a few additional ones shown in listing 5.7. The values of the members are not specified by a participant but extracted from the signature of the underlying method.

Listing 5.7: Additional APIs of class `MethodBodyModificationContext`.

```
public Expression PreviousBody { get; }
public Expression GetPreviousBodyWithArguments (params Expression[] arguments);
```

The property `PreviousBody` returns an expression representing the body of the mutable method that is modified. The body of the mutable method will be replaced with the expression that is currently built, hence the name previous body. Essentially, it is a convenience property that can be used in place of property `Body` of class `MutableMethodInfo`. The possibility to include the previous body when building method bodies is useful for a wide range of scenarios. However, it is often necessary to modify the arguments of the included body.

[3] In fact, delegates are often compared to function pointers.

The method `GetPreviousBodyWithArguments` returns an expression representing the previous body and updates it to use the specified arguments. Section 6.1 illustrates the usage of some of the presented APIs by means of a simple participant.

5.2 Expression Trees

Section 4.5 described the reasons for choosing ETs [Chi11] to represent method bodies. It also mentions that since the addition of statements in version 4.0 of the .NET framework, ETs can be used to represent static methods. The reason why ETs only support static methods is that the semantics of static methods are sufficient to represent any lambda[4] or LINQ expression.

5.2.1 Characteristics

An interesting property of ETs is that the model is expression-based. This means that everything is modeled as expression nodes which have a type. Statements and references are represented the same way as values and calculations. The type of a few nodes that represent statements is `void`, indicating that no result value can be evaluated for this node. An example for such a statement is a call to a method that has no return value. Most statements, however, are represented by nodes that have a type different from `void` and a result value. The block or sequence construct (`{...}`), for example, is modeled by the block expression. By default, the block expression's type and result value are derived from the last expression in the sequence. Another example is the conditional expression, which can be used to represent the conditional operator (`? :`). The type of the conditional expression is determined by the types of the two alternatives which must match. It is also possible to specify the type of the expression explicitly. This enables the reuse of the expression for modeling the common `if-else` construct. When modeling the `if-else` construct, the type of the expression is forced to `void`.

The advantage of an expression-based model is that there is a common concept for processing and evaluating nodes where everything has a type and a result value. This fits languages such as Lisp, Ruby, F#, Nemerle; and it does no harm when modeling other languages. While a unified representation for statements and expressions is advantageous, we consider not having a distinct notion for references poor design. A reference is an expression that can be used on the left-hand side of an assignment, also referred to as *l-value*. Having a distinct notion for references would allow for a more strongly typed API. This would enable the compiler to catch more errors, which can only be

[4]Lambda expression as defined by the C# and VB.NET language specification, not as defined by the *lambda calculus*.

signaled through exceptions with the current design. Microsoft states that the main reasons for not adding references were the compatibility requirement towards .NET 3.5 as well as resource and time constraints [Chi11]. The initial plan by Microsoft was to introduce a generalized *l-value* model later, but ETs were not further developed for .NET 4.5 so this is unlikely to happen.

5.2.2 Extending Expression Trees

In the previous section, we argued that ETs are an elegant solution for representing static methods. The pipeline, of course, also needs to support instance methods. In order to support instance methods, we extended ETs by adding an expression for representing the *this* reference. The `ThisExpression` can be used inside the body of an instance method to call instance methods, access other instance members or simply to obtain a reference to the *this* object. Figure 5.4 shows the types involved in this extension. The visibility of elements is omitted for clarity.

On top of the class hierarchy are the `Expression` and `ExpressionVisitor` classes. These abstract classes are defined in the `System.Linq.Expressions` namespace among concrete expressions such as `ConditionalExpression`, `LoopExpression`, and `MethodCallExpression`. The `Expression` class acts as the base for all other expressions. It has the property `Type` and the method `Accept` taking one argument of type `ExpressionVisitor`. The class `ExpressionVisitor` defines one `Visit*` method for every concrete expression class and a single `VisitExtension` method, which is not constrained to a specific expression. It serves as base class for components that need to process ETs and provides default implementations for all of its methods. Listing 5.8 shows the default implementation of `Expression.Accept`, which simply dispatches to the `VisitExtension` method of the given visitor instance. This means that `VisitExtension` can be used as an extensibility point for processing custom expressions. Together, `Expression` and `ExpressionVisitor` constitute a typical implementation of the Visitor pattern.

Listing 5.8: Default implementation of method `Accept` in class `Expression`.

```
protected internal virtual Expression Accept (ExpressionVisitor visitor) {
  return visitor.VisitExtension (this);
} // method Accept
```

The `VisitExtension` method alone is good for small extensions, but we wanted our custom expressions to integrate with the existing visitor implementation. Therefore, the interfaces `ITypePipeExpression` and `ITypePipeExpressionVisitor` were introduced. They mirror the existing base classes: `ITypePipeExpression` defines an `Accept` method for *double dispatch* and `ITypePipeExpressionVisitor` defines `Visit*` methods for processing concrete custom expressions such as the `ThisExpression`. The class `TypePipeExpressionBase` serves as an abstract base class for custom expressions in the pipeline. It is derived from `Expression`, implements `ITypePipeExpression` and overrides `Expression.Accept` as depicted

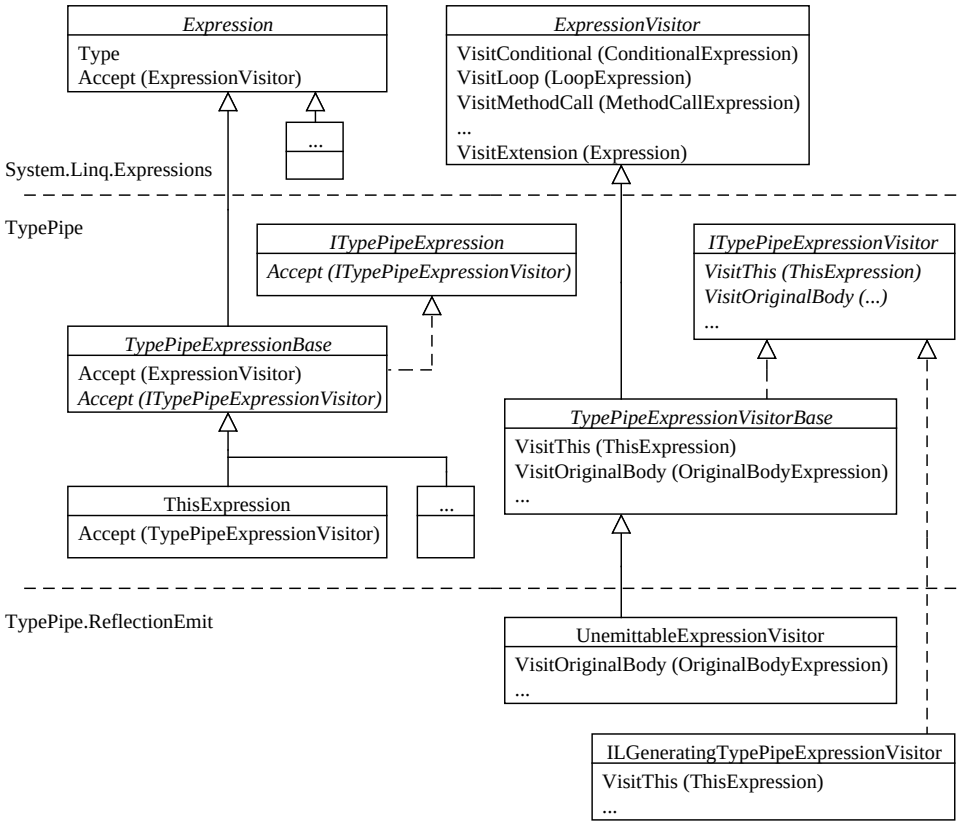

Figure 5.4: Custom expressions and visitor integration.

in listing 5.9. The implementation checks if the given visitor is of type `ITypePipeExpressionVisitor` and either delegates to the corresponding `Accept` method or simply calls the overridden base method. The second `Accept` method, which takes a `ITypePipeExpressionVisitor`, is abstract.

Listing 5.9: Method `Accept` of class `TypePipeExpressionBase` overriding the default implementation in `Expression`.

```
protected internal override Expression Accept (ExpressionVisitor visitor) {
  var typePipeExpressionVisitor = visitor as ITypePipeExpressionVisitor;
  if (typePipeExpressionVisitor != null)
    return Accept (typePipeExpressionVisitor);

  return base.Accept (visitor);
} // method Accept
```

With this infrastructure in place we can easily add new custom expressions by deriving from `TypePipeExpressionBase` and implementing the abstract `Accept` method. Listing 5.10 shows the implementation of this method for the `ThisExpression` class. The method calls the appropriate method on the visitor instance, completing the double dispatch. This finishes the integra-

tion of custom expressions with the existing visitor implementation. If a custom expression is processed with a standard visitor, the `Accept` method of `TypePipeExpressionBase` method will call its base method, which in turn calls the `VisitExtension` method of the visitor. However, if a custom expression is processed with a pipeline visitor, the standard call to `Accept` is forwarded to its pipeline counterpart. Concrete custom expressions implement this method by simply dispatching to the corresponding method on the `ITypePipeExpressionVisitor` interface.

Listing 5.10: Implementation of method `Accept` in class `ThisExpression` as defined by interface `ITypePipeExpression`.

```
public override Expression Accept (ITypePipeExpressionVisitor visitor) {
  return visitor.VisitThis (this);
} // method Accept
```

Besides the `ThisExpression`, the `OriginalBodyExpression` is another example for a custom expression. It is used to represent the original body of a method before any modifications are made to it. The `TypePipeExpressionVisitorBase` class serves as an abstract base for visitors that need to handle both existing as well as custom expressions. It implements the methods defined in `ITypePipeExpressionVisitor` by delegating to its `VisitExtension` method inherited from `ExpressionVisitor`. This means that visitor implementations that derive from `TypePipeExpressionVisitorBase` only need to override the `Visit*` methods for expressions they are interested in.

So far, all of the discussed classes are independent from the actual code generation technique. The two remaining visitor implementations are part of the Ref.Emit-based code generator and explained in section 5.3.3. The next section finally describes how code is generated from ETs.

5.3 Code Generator

The initial implementation of the pipeline includes a Ref.Emit-based code generator. This section outlines the original plan to use .NET framework functionality for code generation, the issues encountered with the approach, and the eventual solution.

5.3.1 Compiling Expression Trees with the .NET Framework

The BCL provides a simple way to generate code for ETs. The expression classes `LambdaExpression` and the derived `Expression<TDelegate>` [Chi11] form the entry point to this functionality. As the name suggests, the former class describes a lambda expression. It captures a block of code that is similar to a method body. The latter is a strongly typed version of the former for expressions whose delegate type is known at compile time. The two classes offer the methods `Compile` and `CompileToMethod`. These methods can be used to com-

pile an expression into a delegate or Ref.Emit method builder. Listing 5.11 demonstrates the use of the classes and their `Compile` method.

Listing 5.11: Compiling an ET into a delegate.

```
// 1) Creating a lambda expression
var a = Expression.Parameter (typeof (int), "a");
var b = Expression.Parameter (typeof (int), "b");
var body = Expression.Add (a, b);

LambdaExpression plusExpression = Expression.Lambda (body, new[] { a, b });
Delegate plus = plusExpression.Compile();
int result1 = (int) plus.DynamicInvoke (4, 7);   // 11

// 2) Code as data
Expression<Func<int, int, int>> minusExpression = (x, y) => x - y;
Func<int, int, int> minus = minusExpression.Compile();
int result2 = minus (4, 7);   // -3
```

Note that the presented listing does not reveal the type of the `plus` delegate instance which is `Func<int, int, int>`. The shown `Lambda` factory method creates a lambda expression by first constructing an appropriate delegate type. This means instantiating the generic delegate types `Func` or `Action` by filling in their generic arguments according to the supplied parameter expressions. Alternatively, there exist overloads of the factory method that allow the specification of the delegate type. If the delegate type is known at compile time, a generic version can be used to retrieve a more strongly typed `Expresssion<TDelegate>` instance.

Another interesting detail is the creation of the `minusExpression` instance in the second example of listing 5.11. It is created directly from a C# lambda. For such statements, the C# compiler does not compile the lambda but rather emits code that builds an ET which represents the lambda. This concept is referred to as *code as data*. Code as data is a central concept in languages such as Lisp, Clojure, and Nemerle.

5.3.2 Issues

The initial plan for code generation was to use the second mentioned method `CompileToMethod` to compile ETs into Ref.Emit method builders. This approach was not feasible due to the following issues.

As mentioned in previous sections, ETs only support static methods. This is caused by two limitations: ETs originally provide no way to reference the *this* object, and the method `CompileToMethod` can only be used to emit code into method builders for static methods. In the previous sections 5.2.2 and 5.1.3, we already described the infrastructure for extending ETs with a `ThisExpression` and the context API to provide it to participants. This solves the first limitation. The second limitation can be circumvented by compiling the ET into a static helper method with an additional parameter for the *this* reference. This helper method is then called from the original instance method by passing in the *this* object as an argument. Other parameters are

simply forwarded. This workaround requires to switch out the occurrences of the `ThisExpression` with an expression representing the additional parameter.

Another issue is that the `MethodCallExpression` class does not support calls to virtual methods without dynamic dispatch semantics. This is needed when a participant wants to call a method of the base type that is overridden by the current type. The workaround for this issue includes the declaration of an instance helper method. For the body of this helper method we manually emit a non-virtual call to the desired base method. Once again, the parameters are simply forwarded. With the help of the context API, participants can build expressions that represent non-virtual base calls. These expressions are replaced with standard method call expressions to the respective helper methods before compilation of the ET.

The last issue we encountered was that it is not possible to reference types and members that are in the process of being generated inside ETs. At code generation time, only Ref.Emit builder objects, for example instances of `TypeBuilder` and `MethodBuilder`, are available for added or modified members. Unfortunately, these builder objects cannot be used inside ETs. The reason is that the factory methods used to create expressions access members of the builder objects that are not properly implemented. For example, the `Expression.Call` factory method, which is used to create instances of `MethodCallExpression`, queries the parameters of the specified method. This fails because the `GetParameters` method raises an exception for unfinished method builders. In addition, the builder classes are sealed, so we cannot provide our own implementation. The described issue also contributed to the creation of the mutable reflection domain.

Although the suggested workarounds for the first two issues produce suboptimal code, they perform good enough to be included in a feasible solution. For the last issue, however, we could not find a suitable workaround. This is very unfortunate as it is caused by a combination of two separate aspects. Firstly, the builder classes derive from the corresponding reflection classes[5] but do not properly implement the inherited APIs. Secondly, the ET compilation process does not offer any extensibility points. This is unfortunate in particular because ETs could easily support an extension model through the `VisitExtension` method and the implemented Visitor pattern.

5.3.3 Modifying the Lambda Compiler

Due to the issues detailed in the previous section we could not use the functionality provided by the BCL to generate code from ETs. Fortunately, the ET classes and the compilation functionality are also included in the Dynamic Language Runtime (DLR) [CT09]. The DLR is a project by Microsoft that makes it easier for language developers to create dynamic languages for

[5] For example, `MethodBuilder` is derived from `MethodInfo`.

the .NET platform. Examples for such languages are IronRuby and IronPython, which are .NET implementations of Ruby and Python respectively. The DLR also provides language interoperability for dynamic operations on objects and common hosting APIs for embedding dynamic languages in .NET applications to enable scripting. Most parts of the project were added to the BCL in version 4.0 of the .NET framework. Moreover, it serves as the basis for the implementation of the C# keyword `dynamic`. Microsoft provides the source code for the DLR under the Apache License 2.0.

The component of the DLR that provides the functionality to generate code for ETs is called lambda compiler. As the lambda compiler does not offer any extensibility points, we decided to include the DLR sources in our code generator and then modify the lambda compiler directly. Microsoft, however, does not provide tests for the lambda compiler component. Modifying it therefore imposes the risk of breaking existing functionality. Additionally, the inclusion of source code from other libraries is bad in general from a maintainability standpoint. Especially updating library source code to a new version while preserving made modifications can be very costly.

However, we believe that it is still the best approach due to the following arguments: Although modifying the lambda compiler is risky and keeping it up to date might require a large amount of effort, the accumulated effort is still less than what would be necessary to build an ET compiler from scratch. This point is reinforced by the fact that the DLR lambda compiler is quite sophisticated and performs optimizations such as *inlining* inner lambda expressions or reusing local variables. By building upon the lambda compiler, our generated code also benefits from these optimizations. Another argument is the stability of the DLR project. The last commit dates back to August 2010 and Microsoft states that they do not plan to further develop the DLR. This means that maintainability and update effort are not as big of an issue. Nevertheless, we tried to minimize the necessary modifications by carefully inserting extension points into the lambda compiler.

Figure 5.5 sketches the main parts of the unmodified lambda compiler. The process of compiling an ET into a delegate starts with the user calling the `Compile` method on the ET. This creates a dynamic method that holds the emitted code. Next, the lambda compiler traverses the ET and emits appropriate IL instructions with the help of the IL generator retrieved from the dynamic method. Finally, a delegate is created from the dynamic method and returned to the user.

For traversal of the nodes in the ET, the lambda compiler simulates the Visitor pattern. Delegation is not achieved by means of double dispatch but through a monolithic `switch` statement that tests the expression type. The reason for this is that the lambda compiler shipped with version 3.5 of the .NET framework, while the Visitor pattern was retrofitted into the expression classes in version 4.0. Unfortunately, the lambda compiler was not adapted to use the pattern. In addition, the lambda compiler only supports

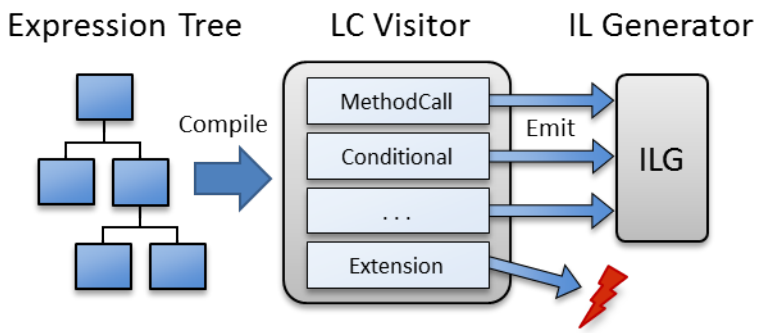

Figure 5.5: Original lambda compiler.

standard expressions. For custom expressions, it delegates to the method `EmitExtensionExpression`, which simply raises an exception.

Figure 5.6 shows the components of the Ref.Emit-based code generator. It includes a modified version of the lambda compiler. The classes of the mutable reflection domain aggregate the modifications specified by the participants. After the last participant executed, the pipeline hands the mutable type over to the code generator. The `MutableType` class makes the aggregated modifications available to the code generator via the Visitor pattern. Based on this data, the first component of the code generator, i.e., the proxy builder, creates a subclass proxy type and overrides modified members using the Ref.Emit APIs. All builder objects created in this process are stored in a dictionary that maps from mutable members to Ref.Emit builders.

The ETs that represent the bodies of mutable methods contain four kinds of expressions: Standard expressions (blue), standard expressions that contain mutable members (violet), custom expressions that can be reduced to standard expressions (red), and custom expressions that cannot be reduced to standard expressions (green). As explained in section 5.2.2, all custom expressions used by the pipeline implement the interface `ITypePipeExpression`. The class `UnemittableExpressionVisitor` (abbreviated with UEV) prepares method bodies by replacing expressions that are not understood by the modified lambda compiler. The `OriginalBodyExpression` is such an expression. As we are building a subclass proxy, it can be replaced with a non-virtual call to the overridden base method. Another example are constant expressions that contain mutable members. These mutable members must be substituted with their corresponding builder objects.

After all expressions that are not understood by the lambda compiler have been replaced, the ET is compiled into the appropriate method builder created by the proxy builder. The included lambda compiler then traverses the ET and emits appropriate IL instructions. For most expressions, this works exactly as in the original lambda compiler. However, the mutable

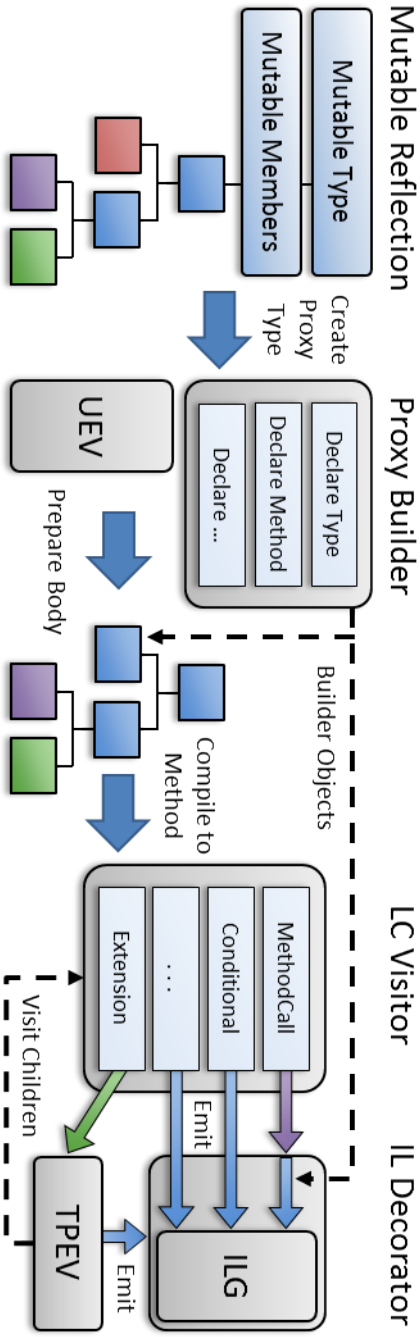

Figure 5.6: Ref.Emit-based code generator including an extended version of the lambda compiler.

members inside expressions cannot be used for emitting code. Unlike real reflection objects, they do not have access to internal CLI metadata that is required by the IL generator. Therefore, we modified the lambda compiler to use a decorated IL generator. The Decorator pattern [GHJV94] allows us to switch out mutable members for their corresponding Ref.Emit builders before delegating to the original IL generator.

To enable compilation of non-reducible custom expressions, we inserted an extension point into the method `EmitExtensionExpression` by wiring it up with the visitor infrastructure described in section 5.2.2. Instead of raising an exception, the method was modified to cast the supplied expression to the type `ITypePipeExpression`. Afterwards, the expression is double-dispatched to the `ILGeneratingTypePipeExpressionVisitor` class (abbreviated with TPEV), which uses the implemented Visitor pattern and the decorated IL generator to emit code for it. An example for a custom expression that cannot be reduced to a standard expression is the `ThisExpression`. Listing 5.12 shows the implementation of the method responsible for emitting code for the `ThisExpression`. The method simply emits an IL instruction for loading the first argument, which always refers to the *this* object within instance methods. For more complex custom expressions that themselves contain child expressions, the `ILGeneratingTypePipeExpressionVisitor` class is injected with a delegate that points back to the lambda compiler. After all method bodies have been emitted, the proxy builder completes the type and the pipeline provides it to the application developer.

Listing 5.12: Method `VisitThis` in `ILGeneratingTypePipeExpressionVisitor`.

```
public Expression VisitThis (ThisExpression expression) {
  _ilGenerator.Emit (OpCodes.Ldarg_0);
  return expression;
} // method VisitThis
```

Another interesting detail of the compilation process is that the execution of body preparation and following steps is deferred until all mutable members are declared by the proxy builder. This is necessary as method bodies might refer to mutable members, which need to be replaced. With the design stated above, we essentially limited the necessary modifications to the lambda compiler to two points: (1) We changed it to use the decorated IL generator and (2) inserted an extension point into the `EmitExtensionExpression` method using the the visitor infrastructure of the pipeline. Note that the classes `UnemittableExpressionVisitor` and `ILGeneratingTypePipeExpressionVisitor` use this infrastructure to process the ETs as illustrated by figure 5.4.

This chapter described the implementation of the mutable reflection domain, the representation of method bodies, and the Ref.Emit-based code generator. In the next chapter, we will look at an example participant and investigate the code generated by the pipeline.

Chapter 6

Results

6.1 Simple Participant

The following illustrates the usage of some of the API presented in subsections 5.1.1 through 5.1.3 by means of a simple example. Assume that we want to persist the properties of simple domain classes in a similar way to how it is done by O/R mappers. The class `Person` in listing 6.1 is an example for such a domain class. Among others it has a property `Name`. Note that the property setter executes additional logic besides updating the backing field of the property with the new value.

Listing 6.1: Simple domain class `Person`.

```
public class Person {
  private string _name;
  public string Name {
    get { return _name; }
    set {
      var oldName = _name;
      _name = value;
      Console.WriteLine ("My name was {0}, now it is {1}.", oldName, _name);
    } // setter
  } // property Name

  // More properties
} // class Person
```

Using the pipeline we want to modify domain classes so that they persist their properties whenever a new value is set. To accomplish this, it is necessary to execute additional logic in the set accessor of properties. However, it is important that the original logic of the set accessors remains intact. Listing 6.2 illustrates the C# equivalent of the `Person` class after the intended modification.

As the first part of this modification, a private method `Persist` is added to the class. The method has two parameters accepting the property name and the new value of the property. Although the implementation of the method is left out for brevity, it may be of arbitrary complexity. As the second part

of the modification, a call to the added method is included from every set accessor. The name of the property is supplied via a string literal and the new value is simply forwarded to the `Persist` method.

Listing 6.2: Modified domain class `Person`.

```
public class Person {
  private void Persist (string propertyName, object propertyValue) {
    // Persist property value
  } // method Persist

  private string _name;
  public string Name {
    get { return _name; }
    set {
      var oldName = _name;
      _name = value;
      Console.WriteLine ("My name was {0}, now it is {1}.", oldName, _name);
      Persist ("Name", value);
    } // setter
  } // property Name

  // More properties
} // class Person
```

Listing 6.3 shows a pipeline participant that modifies classes in the described way. In general, participants need to implement the `ITypeAssemblyParticipant` interface, which defines a single method named `ModifyType`. When the application developer requests a type, the pipeline creates a mutable representation for the requested type and hands it over to the `ModifyType` method of the configured participants. In our example, the requested type is `Person`, and therefore `Person` is also the underlying type of the supplied `MutableType` instance.

Lines 5 to 12 of listing 6.3 are responsible for adding the aforementioned `Persist` method. To add a method, we must specify its name (line 6), attributes (7), return type (8), parameters (9, 10), and body (11). The creation of the body is delegated to the method `BuildPersistMethodBody` declared at line 34. The method takes an argument of type `MethodBodyCreationContext`, which provides useful members to build method bodies represented by ETs. The actual implementation of the method is again left out for brevity.

The second part of the `ModifyType` method inserts calls to the newly created `Persist` method at the end of set accessors. Line 14 iterates through all public properties of the requested type. The set accessor is retrieved from the property (16) and turned into its mutable representation (17). Finally, the body of the set accessor is modified (19–30). Again, a context object is provided, which offers useful functionality for building the body (20). As we are modifying a method, it is of type `MethodBodyModificationContext`.

The outermost expression of the body is a `BlockExpression` (21). A block can be used to combine multiple other expressions. The first expression enclosed by the block represents the previous body of the set accessor (22). It is provided by the context object. The second expression represents a method

call (23) on the current object (24). The *this* reference is again acquired from the context object. The target method of the call is the previously added `Persist` method (25). The following two expressions represent the arguments of the method call. While the name of the property is supplied via a `ConstantExpression` (26), its new value is passed along from an existing `ParameterExpression` (27). Note that we modify the body of a set accessor and therefore have access to the new value through the `Parameters` collection of the context object.

Listing 6.3: Simple participant for persisting property values.

```
public class PersistenceParticipant : ITypeAssemblyParticipant {
  public void ModifyType (MutableType mutableType) {
    MutableMethodInfo persistMethod = mutableType.AddMethod (
        "Persist",
        MethodAttributes.Private,
        typeof (void),
        new[] { new ParameterDeclaration (typeof (string), "propertyName"),
                new ParameterDeclaration (typeof (object), "propertyValue") },
        ctx => BuildPersistMethodBody (ctx)
    );

    foreach (PropertyInfo property in mutableType.GetProperties()) {
      MethodInfo setter = property.GetSetMethod();
      MutableMethodInfo mutableSetter = mutableType.GetOrAddMutableMethod (setter);

      mutableSetter.SetBody (
          ctx =>
          Expression.Block (
              ctx.PreviousBody,
              Expression.Call (
                  ctx.This,
                  persistMethod,
                  Expression.Constant (property.Name),
                  ctx.Parameters[0]
              )
          )
      );
    } // foreach
  } // method ModifyType

  private Expression BuildPersistMethodBody (MethodBodyCreationContext context) {
    // Build persist method body
  } // method BuildPersistMethodBody
} // class PersistenceParticipant
```

When we analyze the presented example, we notice that the `Name` property in the original `Person` class is not marked virtual. Therefore, its accessor methods are not virtual either and cannot be adapted with the help of subclass proxies. This means that the illustrated participant only works when used together with a code generator that supports in-place modification of existing assemblies. For example, a code generator based on Mono Cecil has this capability. On the other hand, a Ref.Emit-based code generator cannot support this scenario.

Another issue with the example participant is that it is not robust. It deliberately lacks exception handling and only works for public instance

properties that have public set accessors. Furthermore, it does not support indexed properties[1],which are common in VB.NET. Note that the call to `GetProperties` in line 14 returns all public properties of the type. This includes static properties and properties defined in base types. For static properties the code fails because the `This` property of the context object raises an exception. The participant can handle properties defined in base types as long as they are marked virtual. This works because `GetOrAddMutableMethod` automatically creates overrides for methods defined in base types. The restriction that these methods need to be virtual is required by the CLI metadata rules [Lid06] and therefore independent from the used code generator.

6.2 Comparison with DynamicProxy

In this section we compare the pipeline implementation with the popular DynamicProxy framework. In particular, we investigate the shape of the generated code and its runtime performance. For a brief description of DynamicProxy, see section 2.2.3.

6.2.1 Example

The simple class `PriceCalculator` in listing 6.4 serves as foundation for the comparison. It has a single method `GetOrderPrice`, which calculates the price of a product order for the specified quantity and customer. The properties `Product.RetailPrice` and `Customer.Discount` are of type `decimal`, which is the recommended data type for monetary values.

Listing 6.4: A simple price calculator.

```
public class PriceCalculator {
  public virtual decimal GetPrice (int quantity, Product product, Customer customer)
  {
    return quantity * product.RetailPrice * (1 - customer.Discount);
  } // method GetPrice
} // class PriceCalculator
```

Assume that we want to change the price calculation to add a ten percent mark-up for all except gold customers without modifying the source code of the `PriceCalculator` class. As the `GetPrice` method is virtual, we can use both DynamicProxy and the pipeline in combination with the Ref.Emit-based code generator to accomplish our goal.

6.2.2 Implementation

This section illustrates, how we can implement the requested change using DynamicProxy and the pipeline. In order to have a baseline for later comparisons, we also manually derive a subclass. Listing 6.5 shows the code needed

[1] C# only supports indexed default properties also known as *indexers*.

for manual subclass creation. The implementation calls the overridden base method and adds a ten percent mark-up if the customer is not a gold customer. The suffix `m` on `1.1m` indicates that the literal is of type `decimal`.

Listing 6.5: Subclass adapting the behavior of the price calculator.

```
public class DerivedPriceCalculator : PriceCalculator {
  public override decimal GetPrice (int quantity, Product product,
                                    Customer customer) {
    decimal orderPrice = base.GetPrice (quantity, product, customer);
    return customer.Level != CustomerLevel.Gold ? 1.1m * orderPrice : orderPrice;
  } // method GetPrice
} // class DerivedPriceCalculator
```

Note that for manual implementation of the requested change, it would better to use the Decorator pattern [GHJV94] instead of subclassing. If possible, an interface containing the `GetPrice` method should be extracted from class `PriceCalculator` to allow adaption of parameters and return values through decorators. This results in a more flexible design and better reusable code.

Next, we take a look at the DynamicProxy-based implementation. DynamicProxy revolves around the notion of interceptors, which are represented by the interface `IInterceptor`. The interface has a single method `Intercept` taking an argument of type `IInvocation`. Instances of `IInvocation` provide access to useful runtime data about the method call. This includes, among other things, target object, arguments, and return value of the method call. Listing 6.6 illustrates an implementation of the `IInterceptor` interface that accomplishes our goal.

Listing 6.6: A interceptor adapting the behavior of the price calculator.

```
public class PriceCalculatorInterceptor : Castle.DynamicProxy.IInterceptor {
  public void Intercept (IInvocation invocation) {
    invocation.Proceed();
    var customer = (Customer) invocation.Arguments[2];
    var orderPrice = (decimal) invocation.ReturnValue;
    invocation.ReturnValue =
        customer.Level != CustomerLevel.Gold ? 1.1m * orderPrice : orderPrice;
  } // method Intercept
} // class PriceCalculatorInterceptor
```

The implementation of the interceptor in listing 6.6 is straightforward. The call to the method `Proceed` on the invocation objects invokes the next interceptor in line, and ultimately the target method. Afterwards, we retrieve the needed arguments from the invocation object and update the return value accordingly. Note that the interceptor interface is flexible enough to adapt any method but the interceptor implementation in listing 6.6 is tied to a specific method. For this scenario, DynamicProxy offers so-called interceptor selectors that can be used to configure the set of interceptors that adapt a target method. We omit the declaration of the according interceptor selector for brevity.

Listing 6.7 shows the pipeline counterpart of the two implementations presented above. This implementation is far more complex than the pre-

vious two. The reason for this is that instead of expressing our intention directly in code we need to build an ET that represents such code. As mentioned in section 6.1, all pipeline participants need to implement the interface `ITypeAssemblyParticipant`, which declares a single method `ModifyType`. For the price-adapting participant in listing 6.7, this method is implemented as follows: First, we retrieve the mutable version of the `GetPrice` method from the mutable type. In lines 8 to 28 we modify the method body by building an ET with the help of the provided context object.

Listing 6.7: A participant adapting the behavior of the price calculator.

```
1  public class PriceCalculatorParticipant : ITypeAssemblyParticipant {
2    public void ModifyType (MutableType mutableType) {
3      MutableMethodInfo mutableMethod =
4          mutableType.AllMutableMethods.Single (m => m.Name == "GetPrice");
5
6      mutableMethod.SetBody (
7          ctx => {
8            var orderPrice = Expression.Variable (typeof (decimal));
9            return Expression.Block (
10               new[] { orderPrice },
11               Expression.Assign (orderPrice, ctx.PreviousBody),
12               Expression.Condition (
13                   Expression.NotEqual (
14                       Expression.Property (ctx.Parameters[2], "Level"),
15                       Expression.Constant (CustomerLevel.Gold)
16                   ),
17                   Expression.Multiply (
18                       Expression.Constant (1.1m),
19                       orderPrice
20                   ),
21                   orderPrice
22               )
23           );
24         } // lambda bodyProvider
25     );
26   } // method ModifyType
27 } // class PriceCalculatorParticipant
```

The ET built in lines 11 to 26 is an exact representation of the code in method `GetPrice` of class `DerivedPriceCalculator` in listing 6.5. Line 11 creates an expression representing a local variable of type `decimal`. In order to use a variable we need to bind it and bring it into scope. This can be done with a `BlockExpression` (line 12, 13), which is also used to group the remaining expressions. Inside the block, we build an expression that represents an assignment (14). On the left-hand side of the assignment is the local variable declared above and on the right-hand side is the previous body, which is provided by the context object. After the assignment, we add a `ConditionalExpression` to the block (15–25). It represents the conditional operator (`? :`). For the test part of the operator, we build an expression that represents a not-equal check (16). The first operand of the check is the property `Level` of the third parameter (17), which is of type `Customer` and acquired from the context object. As the second operand, we use the constant `Gold` from the enumeration `CustomerLevel` (18). When the test evaluates to true,

the result of the conditional operator is determined by an expression that represents a multiplication (20) of the `decimal` constant `1.1` (21) with the local variable (22). Otherwise, the result is just the value of the local variable (24). As the `ConditionalExpression` is the last expression in the block, it defines the type and result value of the surrounding `BlockExpression`. Furthermore, the `BlockExpression` is the outermost expression and in turn defines the return value of the method.

When we compare the three implementations from the devlopers's point of view, we can easily see that the participant-based solution is the most complex. When working, developers usually have a solution in mind which they translate into code. This is what happened during the implementation of the manual subclass. Essentially, the same mindset can be used when implementing DynamicProxy interceptors. The only difference is that we have to use DynamicProxy's generic APIs, which slightly increases complexity. The implementation of a pipeline participant, however, requires us to work with code from a meta-level. Instead of translating the envisioned solution into code, we need to build an ET that represents code that in turn accomplishes our goal. Therefore, it is often helpful to make this a two-step process. When building complex method bodies, we can first sketch the solution code and then model it using ETs.

Another interesting aspect is how the three alternatives allow delegation to existing code and the semantics attached to the act of delegation. For the manual subclass, delegation means a simple base call. All data is statically available and known at compile time. No further modifications can be made once the code is compiled. When using DynamicProxy, delegation means calling the method `Proceed` on the invocation object. This invokes the next interceptor in line, and ultimately the target method. DynamicProxy offers a generic API that can be used to intercept any overridable method. It provides access to all kinds of runtime data and metadata, and therefore offers the greatest flexibility. Modifications are made by configuring interceptor chains during proxy creation on a per instance basis. For the pipeline, delegation means embedding an expression that represents the previous body in the expression that is used for the new body. Note that the generated code depends on the used code generator and is not necessarily a base call. When using the pipeline, modifications are made at code generation time. Accessing metadata requires building expressions that represent calls to reflection APIs as the pipeline does not provide this data. Furthermore, all three alternatives offer a simple way to prevent delegation to existing code altogether. For manual subclasses and DynamicProxy, this means to skip the base call and the call to `Proceed` respectively. Using the pipeline, we achieve the same by not embedding the previous body when building a new one.

6.2.3 Generated Code

After comparing the implementations from the developer's point of view, we take a look at the generated code. DynamicProxy and the used code generator for the pipeline both employ Ref.Emit as their underlying code generation technique to emit types into a dynamic assembly. The frameworks have been configured to save the dynamic assembly to disk for later inspection. For inspection, we use the popular .NET Reflector decompilation tool. The decompiler allows us to inspect assembly metadata, types, and members using a hierarchic tree view. In addition, method code cannot only be viewed as raw IL, but also in various other .NET languages including C#. Note that the decompiled code is not necessarily a perfect match with the original source code, if it exists. Reasons for deviations include compiler optimizations, imperfect decompilation routines, and loss of information due to translation into IL constructs that have no counterpart in the respective source language. The semantics, however, is equal to the one of the source code that was used as the input for the compiler. For our purpose, this is sufficient as there is no original source code in the traditional sense.

Figure 6.1 shows the structure of the dynamic assembly generated by DynamicProxy. The assembly contains two types named `PriceCalculatorProxy` and `PriceCalculator_GetPrice`. These types constitute the generic implementations of the proxy and invocation object respectively. The proxy class derives from `PriceCalculator` and overrides the `GetPrice` method. Invocation objects are instantiated from the class `PriceCalculator_GetPrice`, which transitively implements the `IInvocation` interface. The following paragraphs explain the most important members of the two classes.

Listing 6.8 shows the decompiled code of the generic proxy class. It overrides the `GetPrice` method to allow adaption by interceptors. The method implementation first creates an instance of the invocation object class shown in listing 6.9. The constructor is supplied with the target type, a back-reference to the proxy, the interceptors, the target method, and the arguments of the method call. The arguments are captured in an object array, which requires boxing of value types. The next step calls the `Proceed` method on the invocation object, which results in the execution of the first interceptor. After the interceptor chain finished execution, the return value is retrieved from the invocation object and returned to the caller. As the property `ReturnValue` on the invocation object is of type `object`, unboxing is required for value types.

DynamicProxy generated an additional method named `GetPrice_callback`, which simply delegates to the overridden `GetPrice` method. The purpose of this method is to provide a way to call the overridden base method from code outside of the `PriceCalculatorProxy` class hierarchy, in particular, from the invocation object class. A helper method is necessary because the CLR considers non-virtual calls of virtual methods unverifiable, unless the call represents a base call within the class hierarchy [Lid06]. This is the situation that occurs when we use the `base` keyword in C#. The keyword can be used to call a virtual base method without dynamic dispatch semantics.

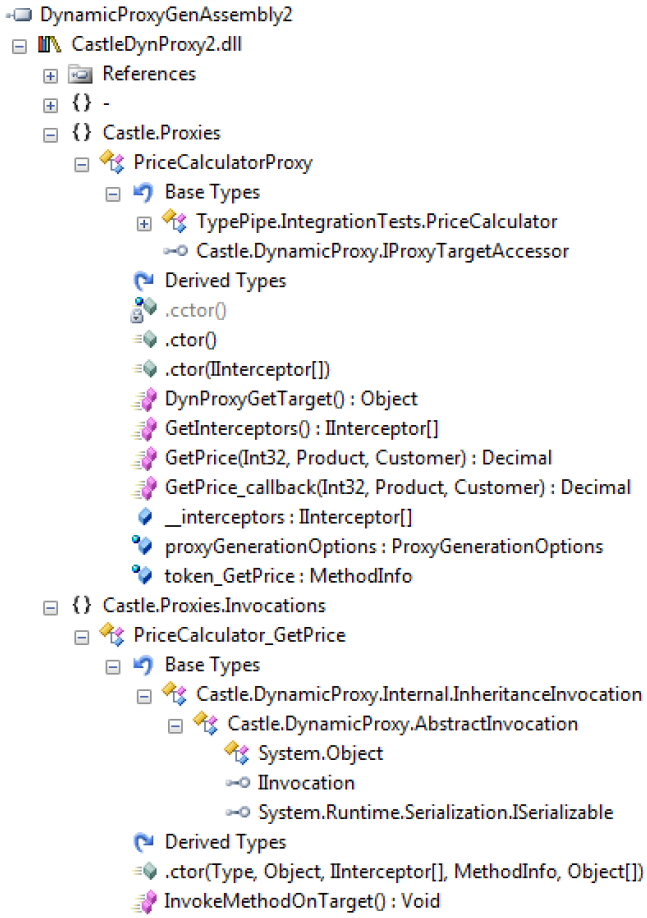

Figure 6.1: Dynamic assembly generated by DynamicProxy.

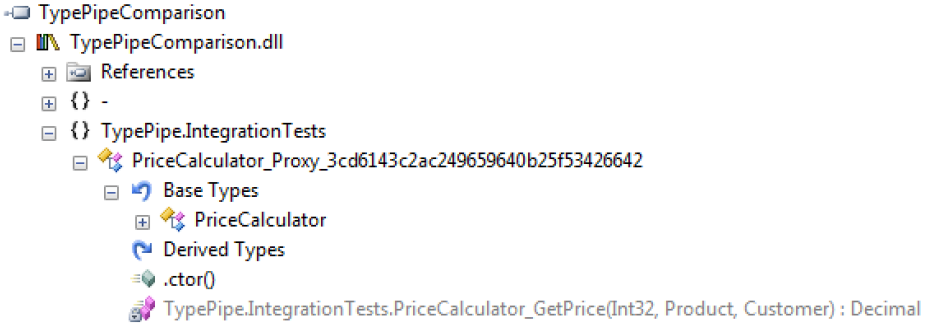

Figure 6.2: Dynamic assembly generated by the pipeline and its Ref.Emit-based code generator.

Listing 6.8: A portion of the generic proxy class generated by DynamicProxy.

```
public class PriceCalculatorProxy : PriceCalculator, IProxyTargetAccessor {
  // Fields, constructors and some methods are omitted.

  public override decimal GetPrice (int quantity, Product product,
                                    Customer customer) {
    PriceCalculator_GetPrice invocation = new PriceCalculator_GetPrice (
        typeof (PriceCalculator),
        this,
        this.__interceptors,
        PriceCalculatorProxy.token_GetPrice,
        new object[] { quantity, product, customer }
    );
    invocation.Proceed();
    return (decimal) invocation.ReturnValue;
  } // method GetPrice

  public decimal GetPrice_callback (int quantity, Product product,
                                    Customer customer) {
    return base.GetPrice (quantity, product, customer);
  } // method GetPrice_callback
} // class PriceCalculatorProxy
```

The decompiled code of the generated invocation object class is illustrated in listing 6.9. This class named `PriceCalculator_GetPrice` inherits from the abstract base class `InheritanceInvocation` and therefore transitively implements the `IInvocation` interface. As the members defined by `IInvocation` are implemented by the base types of the invocation object class, the class itself only contains two members.

Listing 6.9: Invocation object class generated by DynamicProxy.

```
public class PriceCalculator_GetPrice : InheritanceInvocation {
  public PriceCalculator_GetPrice (Type targetType, object proxy,
      IInterceptor[] interceptors, MethodInfo proxiedMethod, object[] arguments)
      : base (targetType, proxy, interceptors, proxiedMethod, arguments) {
  } // constructor PriceCalculator_GetPrice

  public override void InvokeMethodOnTarget () {
    decimal price =
        (base.proxyObject as PriceCalculatorProxy)
            .GetPrice_callback ((int) base.GetArgumentValue (0),
                                (Product) base.GetArgumentValue (1),
                                (Customer) base.GetArgumentValue (2));
    base.ReturnValue = price;
  } // method InvokeMethodOnTarget
} // class PriceCalculator_GetPrice
```

The first member of the class `PriceCalculator_GetPrice` is the constructor that has been mentioned in the description of listing 6.8. The second member is the method `InvokeMethodOnTarget`. It is declared abstract in the base class `InheritanceInvocation` and therefore requires implementation. The method is invoked when the last interceptor in the chain calls the `Proceed` method. Its purpose is to call the original target method on the proxy instance. This is achieved by casting the proxy instance to the concrete proxy type and invoking the `GetPrice_callback` method mentioned previously. The arguments for the method call are retrieved with the help of the method `GetArgumentValue`, which is declared by the `IInvocation` interface and implemented in a base

type. The arguments are provided as plain `object` instances and therefore need to be casted to their respective types before they can be used in the method call. The return value of the target method is then assigned to the property `ReturnValue` on the invocation object. For value types, casting the arguments and setting the return value again requires unboxing and boxing respectively. This concludes the investigation of the code generated by DynamicProxy.

Next, we will look at the code that is generated by the Ref.Emit-based code generator of the pipeline. The structure of the generated dynamic assembly is depicted in figure 6.2. The assembly contains a single type named `PriceCalculator_Proxy_3cd61...`. It represents the generated proxy class. Note that the type name includes a globally unique identifier (GUID) to avoid name clashes. This is necessary as the pipeline generates a separate proxy class for every set of participants even if the same type is requested. This means that the proxy class and the contained logic are specific to a certain set of participants. The logic is hard-wired and cannot be configured through interceptors or similar mechanisms.

Listing 6.10 shows the decompiled code of the proxy class generated by the pipeline. It contains a single private method with a signature equal to the one of method `GetPrice` declared in the base class. The method explicitly overrides the `GetPrice` method and hence must be virtual. Note that C# has no syntax for declaring explicit method overrides. Therefore, we use the IL override directive to illustrate the explicit override [Lid06]. The directive has the following syntax: `.override <class_ref>::<method_name>`. Besides the full type name, the `<class_ref>` component also contains an assembly reference for types that are defined in other assemblies. For our example the assembly reference is `[TypePipe.IntegrationTests]`. It was omitted from the listing for brevity.

The name of the overriding method consists of the name of the overridden method prefixed with its declaring type. This avoids name clashes when modifying or overriding different base definitions that have the same name and signature. It also preserves the possibility to add a set of methods that shadows and overrides a base method at the same time. This situation is described in section 5.1.2.

Listing 6.10: Proxy class generated by the Ref.Emit-based code generator.

```
public class PriceCalculator_Proxy_3cd61... : PriceCalculator {
  private virtual decimal TypePipe.IntegrationTests.PriceCalculator_GetPrice (
      int quantity, Product product, Customer customer) {
    .override TypePipe.IntegrationTests.PriceCalculator::GetPrice

    decimal price = base.GetPrice (quantity, product, customer);
    return ((customer.Level == CustomerLevel.Gold) ? price : (1.1M * price));
  } // method TypePipe.IntegrationTests.PriceCalculator_GetPrice
} // class PriceCalculator_Proxy_3cd61...
```

The code in listing 6.10 is straightforward. Although we look at the decompiled form, it is almost equivalent to the source code of the manual subclass in listing 6.5. The only difference is that the comparison was changed from a

not-equal to an equal test, which also required flipping the operands of the conditional operator. In the underlying IL code, this is represented by the `beq` instruction, which was emitted instead of the more obvious `bne.un` instruction[2]. The reason for this is an implementation detail of the lambda compiler, which treats not-equal comparisons as inverted equal comparisons. The rest of the code is exactly as specified by the ET in listing 6.7.

6.2.4 Runtime Performance

In this last section we compare the runtime performance of the generated code. To compare runtime performance we call the `GetPrice` method with randomized test data and measure the execution time for all three implementations. For time measurement we use the `StopWatch` class, which is part of the BCL. The generated code is called via a variable with the static type `PriceCalculator`. Therefore, the execution times include a virtual method table lookup as it is needed in a real-life scenarios. The test program itself is compiled in release mode, which enables compiler optimizations. Execution times were measured on a desktop machine with an Intel Core2 Duo E8400 processor operating at 3.0 GHz and 8 GB of memory running the 64 bit version of Windows Server 2008 R2. Note that the specification of the test system is stated for completeness only. In this comparison we highlight the relative differences rather than absolute execution times.

For the test input we use randomly generated data. For every call to the `GetPrice` method, we need an integer quantity, a product object, and a customer object. Table 6.1 depicts name, type, and constraining intervals of all elements in an input record. Element values are generated uniformly at random within the specified intervals. To guarantee fairness, all three implementations are tested with the same set of input records. This required us to keep the input data in memory, resulting in `OutOfMemoryException`s for test runs with a high number of calls. To overcome this problem, we employed the Flyweight pattern [GHJV94] for the `Product` and `Customer` test inputs. In addition, the maximum number of generated input records is capped at 10^6. Test runs with a higher number of calls simply iterate through the input data multiple times.

After generating the input data, we can turn our attention to the execution times presented in table 6.2. The first thing to note is that 1000 method calls take less than one millisecond for all three implementations. Another similarity is the growth of execution time with respect to the number of calls. Execution time grows linearly for all three implementations. We did not observe any inhibiting factors or other limitations for larger number of calls. Unsurprisingly, the manual subclass implementation compiled by the

[2] `beq`: Branch if the two topmost elements on the stack are equal. `bne.un`: Branch if not equal. Integer values are interpreted as unsigned; floating-point values are compared unordered.

Name	*Type*	*Interval*
Quantity	`int`	$[1, 100]$
Product retail price	`decimal`	[10.0, 1000.0]
Customer discount	`decimal`	[0.0, 0.15]
Customer level	`CustomerLevel`	20% Gold, 40% Silver, 40% Bronze

Table 6.1: Characteristics of the generated test data.

Calls	*Manual Subclass*	*DynamicProxy*	*TypePipe*
10^3	< 1 ms	< 1 ms	< 1 ms
10^4	5 ms	7 ms	5 ms
10^5	52 ms	74 ms	53 ms
10^6	524 ms	707 ms	531 ms
10^9	8 m 45 s	11 m 52 s	8 m 53 s

Table 6.2: Execution times of the generated code.

C# compiler with optimizations enabled yields the fastest code. While the DynamicProxy-based implementation is about 35 % slower, the code generated by the pipeline only needs 1 to 2 % more execution time. Considering the information of the two previous sections, this is unsurprising as well. The pipeline acts somewhat like a compiler that translates ETs into IL code. Only code specified by the participants is generated. Therefore, there is little overhead. DynamicProxy, however, provides a generic API and access to runtime data even when interceptors do not use it. This introduces overhead but at the same time offers greater flexibility at runtime. The overhead also depends on the shape of the intercepted method. The `GetPrice` method, for example, takes an argument of type `int` and returns a `decimal` value. These types are value types and therefore require additional boxing and unboxing operations, which increases overhead.

Keep in mind that we compared the code for a very simple example. For more complex methods with longer execution times, for example, methods accessing a database, the introduced overhead becomes neglectable. Although we are contented that the pipeline and the Ref.Emit-based code generator create near-optimal code that performs well, performance is not the main reason to choose the pipeline over other frameworks. The main advantage of the pipeline is interoperability. Enabling multiple independent participants to generate code collaboratively is what really makes it valuable.

Chapter 7

Conclusion, Future Work and Experiences

7.1 Conclusion

In this thesis, we motivated the importance of code generation and the trend of becoming even more prominent. We investigated existing code generation frameworks for the .NET platform and illustrated their shortcomings in terms of interoperability. Subsequently, the underlying issues that impede interoperability were analyzed. We documented our experiences with the re-motion framework that led to the decision to develop a new code generation pipeline for the .NET platform.

The thesis included the rationale behind key design decisions, such as representing method bodies with expression trees to enable composition. It also highlighted selected APIs, implementation details, encountered issues, and the applied solutions. The usage of the pipeline was demonstrated by means of a simple example. We examined the quality and performance of code generated by the pipeline and concluded that there are only minor differences when compared to the output of the C# compiler.

Although development is still ongoing, the pipeline is already used in an external project. The project is a new AOP framework that aims to be an open source alternative for the popular but commercial PostSharp framework. It can be found at activeattributes.codeplex.com. So far, the framework author did not encounter any conceptual shortcomings when using the pipeline. After integration in re-motion, the pipeline will reduce the complexity and cost for maintaining and developing features in re-store and re-mix. By developing the pipeline as an open source project, we are confident that we make a valuable contribution to the .NET ecosystem. The pipeline empowers framework authors as well as application developers to focus on the actual problem at hand rather than dealing with code generation issues.

7.2 Future Work

The initial design, implementation, and test of the pipeline have been finished, but development is still going on. The following highlights areas that are considered for future work.

7.2.1 Additional Code Generators

As part of this thesis a Ref.Emit-based code generator has been developed. Section 4.2 claims that the pipeline was designed in such a way that the code generator forms a pluggable component that can be switched out for other implementations. However, as part of this thesis, only a single code generator based on Ref.Emit has been developed. Development of additional code generators is regarded as future work.

There are various motivations behind the development of additional code generators. From an engineering point of view, it is interesting to show that the pipeline design is generic enough to support back-ends based on other code generation techniques. This also reveals if the code generators are truly pluggable components that can be switched at will. Of course, a part of this future work would be to refine the pipeline design if necessary. From the user standpoint, the pipeline becomes more versatile with more than one code generator available. Firstly, other code generators might lift the limitations imposed by the subclass proxy approach. Secondly, the pipeline can be used in environments where Ref.Emit is not available. Candidates for technologies that enable future code generators are Mono Cecil, CodeDOM, and Roslyn.

7.2.2 Decompilation of Existing Method Bodies

Currently, original method bodies are represented by a placeholder expression named `OriginalBodyExpression`. This means that participants do not have the option to drill into and analyze bodies of unmodified existing methods using ETs. The goal of the proposed refinement is to decompile existing IL code into ETs. This would enable participants to selectively include parts from the original body into the new body when modifying methods. Participants could also analyse and modify original method bodies at statement level. Examples for this are injecting or replacing specific statements, or finding out which members are accessed by a method implementation. As decompilation is a complex operation, it should be executed lazily on a per method basis.

Note that there exist popular IL decompilers such as .NET Reflector, ILSpy, and dotPeek. Most of these tools are capable of transforming IL code into source code of various .NET languages but none supports decompiling into ETs. However, the open source project ILSpy is based on another

open source library named NRefactory. NRefactory contains an IL decompiler whose output is a semantic AST-based model. Part of the future work is to research if NRefactory or other existing decompilation techniques such as Mono Cecil can be utilized for this task.

7.2.3 Porting DynamicProxy

The porting DynamicProxy project does not concern the pipeline implementation per se, but is still very promising. The goal is to port DynamicProxy to use the pipeline instead of its custom facilities for code generation. Note that DynamicProxy is a popular open source project and used by many tools and frameworks. Porting DynamicProxy would make these tools and frameworks compatible with other pipeline-based tools. Another promising aspect is the potential exposure of the pipeline project to a vast part of the .NET open source community due to DynamicProxy's popularity.

7.3 Experiences

While writing this thesis and developing the TypePipe project I have made valuable experiences.

The first thing that I want to positively emphasize is the working atmosphere at rubicon. My colleagues are all very helpful and open to questions and discussions. In addition, they are happy to share their knowledge and expertise. I learned a lot in a short amount of time.

This leads me to the next point: Areas, in which I was able acquire new skills or improve my existing understanding. On the technical side, the .NET IL assembly language is definitely one of these areas. A few months ago, a method body in IL seemed overwhelming to me. In the meantime, I can make sense of it. Although it is a dry read [Lid06] was definitely helpful in this matter. Other areas in which I improved are design principles and patterns, test-driven development, and working productively with tools like VS and ReSharper.

Of course I also recognized areas in which I have deficits. In particular, time management and prioritizing are two such areas I need to work on.

A lesson I learned is that the 80-20 rule, or Pareto principle, also applies to software development: It takes about 20 % of the effort to build a prototype that has 80 % of the functionality or works in 80 % of all cases. Finishing the remaining 20 % to make the component "production ready" takes 80 % of the effort, which is often underestimated. This is especially true when dealing with .NET reflection.

Another lesson learned is that big companies such as Microsoft, are no guarantee for well-designed and well-engineered code as I previously conceived.

A last remark: 2 to 3 proof readers are well enough for a thesis. More are not worth the effort, as they are likely to start correcting each other.

List of Abbreviations

AOP aspect-oriented programming. 1, 10–13, 16, 18, 61

API application programming interface. 3, 4, 7, 8, 11, 12, 14, 16, 18, 19, 21–24, 26, 29–31, 34–38, 42–45, 48, 54, 60, 61

ASP.NET Active Server Pages .NET. 7

AST abstract syntax tree. 2, 23, 63

BCL Base Class Library. 6, 7, 23, 41, 43, 44, 59

CLI Common Language Infrastructure. 8, 12, 30, 33, 47, 51

CLR Common Language Runtime. 5, 7–9, 18, 22, 24, 34, 55

CodeDOM Code Document Object Model. 5–7, 23, 25, 62

CSV comma-separated values. 6

CTP Community Technology Preview. 16

DDD domain-driven design. 13, 14

DLR Dynamic Language Runtime. 43, 44

ET expression tree. 23, 24, 26, 33, 34, 36–39, 41–45, 47, 49, 53, 54, 59, 60, 62

GUID globally unique identifier. 58

HTML HyperText Markup Language. 6

IL intermediate language. 5, 7, 8, 12, 23, 24, 44, 45, 47, 55, 58–60, 62, 63

IoC inversion of control. 18, 19

JSP Java Server Pages. 4, 7

JVM Java virtual machine. 5

LGPL GNU Lesser General Public License. 13, 15

LINQ Language Integrated Query. 7, 23, 38

LSP Liskov substitution principle. 9

MDE model-driven engineering. 1

O/R mapper object-relational mapping tool. 10, 11, 13, 18, 48

PE Portable Executable. 16

Ref.Emit Reflection.Emit. 2, 7, 8, 13, 14, 16, 19, 22–27, 32, 41–43, 45–47, 50, 51, 55, 56, 58, 60, 62

SQL Structured Query Language. 7

T4 Text Template Transformation Toolkit. 5–7

VB.NET Visual Basic .NET. 6, 7, 16, 38, 51

VS Visual Studio. 6, 7, 16, 23, 63

WSDL Web Services Description Language. 7

XSLT Extensible Stylesheet Language Transformation. 4

Bibliography

[AM09] Arking, Jon and Scott Millett: *Professional Enterprise .NET*. Wrox Press Ltd., Birmingham, UK, 2009.

[ASU86] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Inc., Boston, MA, USA, 1986.

[BF09] Barnett, Mike and Manuel Fahndrich: *Common Compiler Infrastructure*, 2009. http://ccimetadata.codeplex.com.

[OH12] Öberg, Rickard and Niclas Hedhman: *The Qi4j Project*, February 2012. http://www.qi4j.org.

[CP] *The Castle Project*. http://www.castleproject.org.

[Chi11] Chiles, Bill: *Expression Trees v2 Specification*. Microsoft, June 2011.

[Coo11] Cooper, Simon: *Subterranean IL: Explicit overrides*, 2011. http://www.simple-talk.com/blogs/2011/12/12/subterranean-il-explicit-overrides.

[CT09] Chiles, Bill and Alex Turner: *Dynamic Language Runtime*. Microsoft, 2009.

[ECMA] ECMA International: *Standard ECMA-335 – Common Language Infrastructure (CLI)*, 5th edition, December 2010.

[MC] Evain, Jean Baptiste: *Mono Cecil*. http://mono-project.com/Cecil.

[Eva03] Evans, Eric: *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Professional, Boston, MA, USA, 2003.

[MCU] Evain, Jean Baptiste: *List of Mono Cecil Users*, May 2012. http://github.com/jbevain/cecil/wiki/Users.

[Fow02] Fowler, Martin: *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Inc., Boston, MA, USA, 2002.

[GDD09] Gasevic, Dragan, Dragan Djuric, and Vladan Devedzic: *Model Driven Engineering and Ontology Development*. Springer Publishing, 2nd edition, 2009.

[GG01] Gough, John John and K. John Gough: *Compiling for the .NET Common Language Runtime*. Prentice Hall, Upper Saddle River, NJ, USA, 2001.

[GHJV94] Gamma, Erich, Richard Helm, Ralph Johnson, and John M. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1st edition, 1994.

[HWR08] Hevery, Misko, Jonathan Wolter, and Russ Ruffer: *A Guide for Writing Testable Code*, 2008. http://misko.hevery.com/code-reviewers-guide.

[JN05] Jacobson, Ivar and Pan Wei Ng: *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley object technology series. Addison-Wesley Professional, 2005.

[Kay08] Kay, Michael: *XSLT 2.0 and XPath 2.0 Programmer's Reference*. Wrox Press Ltd., Birmingham, UK, 4th edition, 2008.

[KBHK09] Kuaté, P.H., C. Bauer, T. Harris, and G. King: *NHibernate in Action*. Manning Pubs Co Series. Manning, 2009.

[KFS06] Kühn, E., G. Fessl, and F. Schmied: *Aspect-Oriented Programming with Runtime-Generated Subclass Proxies and .NET Dynamic Methods*. Journal of .NET Technologies, 4:1801–1808, 2006.

[Lau08] Laureano, Philip: *Introducing the LinFu Framework, Part I – VI*, 2008. http://code.google.com/p/linfu.

[Lid06] Lidin, Serge: *Expert .NET 2.0 IL Assembler*. Apress, Berkely, CA, USA, 2006.

[Lin06] The Linux Information Project: *Machine Code Definition*, 2006. http://www.linfo.org/machine_code.html.

[LW94] Liskov, Barbara H. and Jeannette M. Wing: *A behavioral notion of subtyping*. ACM Trans. Program. Lang. Syst., 16:1811–1841, 1994.

[LY12] Lindholm, Tim and Frank Yellin: *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley Longman Publishing Inc., Boston, MA, USA, 2012.

[Mar00] Martin, Robert C.: *Design Principles and Design Patterns*. Technical report, Object Mentor, 2000.

[MSCD] Microsoft Developer Network (MSDN): *Dynamic Source Code Generation and Compilation*, 2009. http://msdn.microsoft.com/en-us/library/650ax5cx.aspx.

[MST4] Microsoft Developer Network (MSDN): *Code Generation and T4 Text Templates*, March 2011. http://msdn.microsoft.com/en-us/library/bb126445.

[Mono] *The Mono Project*. http://mono-project.com.

[Nyh02] Nyholm, Caroline: *Product Line Development – An Overview*. Building Reliable Component-Based Software Systems, Artech House Publishers:44–58, 2002.

[PS] *PostSharp*, 2009–2012. http://www.sharpcrafters.com.

[REM] *re-motion*, 2006-2012. http://www.re-motion.org.

[MSR] *Microsoft Roslyn CTP*, June 2012. http://msdn.com/roslyn.

[RPL03] Roth, Mark and Eduardo Pelegrí-Llopart: *Java Server Pages Specification Version 2.0*. Oracle Corporation, 2003.

[PSS] SharpCrafters s.r.o.: *PostSharp SDK Documentation*, November 2011. http://doc.sharpcrafters.com/postsharp-2.1/Default.aspx#PostSharp-2.1.chm/html/N_PostSharp_Sdk.htm.

[Sin11] Sinclair, Gavin: *MixIn Programming*. Cunningham & Cunningham, Inc., November 2011. http://c2.com/cgi/wiki?MixIn.

[Syc07] Sych, Oleg: *The Text Template Transformation Toolkit*, 2007. http://www.olegsych.com/2007/12/text-template-transformation-toolkit.

[TOHS99] Tarr, Peri, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr.: *N degrees of separation: multi-dimensional separation of concerns*. In ICSE '99, pages 107–119, New York, NY, USA, 1999. ACM.

[Ver04] Verissimo, Hamilton: *Castle's DynamicProxy for .NET*, 2004. http://www.codeproject.com/Articles/9055/Castle-s-DynamicProxy-for-NET.