Bachelor's Thesis

Δ-Guided Plugin Adaptation in .NET

submitted by

Michael Rudolf

born 11.05.1983 in Berlin

Technische Universität Dresden

Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Supervisor: M.Sc. Ilie Savga
Professor: Dr. rer. nat. habil. Uwe Aßmann

Submitted September 27, 2006

Contents

List of Figures

Chapter 1

Introduction

When a framework evolves, changes to its API may occur. A new framework release may then invalidate existing plugins – modules, which used a previous framework version. These plugins have to be manually adapted or, at least, recompiled to work with the new release. To avoid this effect, we provide a new technique for API binary compatibility, which permits existing plugins to link and run with a new framework release without recompiling. This is achieved by carefully defining the changes that can be applied to an API, recording these changes upon their occurrence and generating adaptation code out of change specifications. The adaptation is performed at the framework site and is thus transparent for plugins.

1.1 Motivation

This thesis has been written in the course of a collaboration project of the Technical University of Dresden and Comarch [Com], an international IT business solution company from Cracow, Poland. Comarch is developing an Enterprise Resource Planning (ERP) System called B2[1]. It will integrate and automate many of the business practices associated with the operations or production aspects of a company, such as sales and delivery, billing and production, inventory and human resource management. The framework will be developed for the .NET platform, and it should be extended by plugins, which were provided by third-parties. However, there would be regular releases of new framework versions, which would have meant that each time a new version was installed at a customer site, all third-party plugins had to be updated to work with the new framework version. In order to avoid the expenses and the machinery necessary on every version update, a technique was needed that could ensure the *backward-compatibility* of the new framework version with the old plugins. Effectively, the framework should be allowed to evolve, while all plugins developed against older versions of the framework should remain usable without the need of changing or even recompiling them. Thus, newer framework versions should be compatible with older plugins.

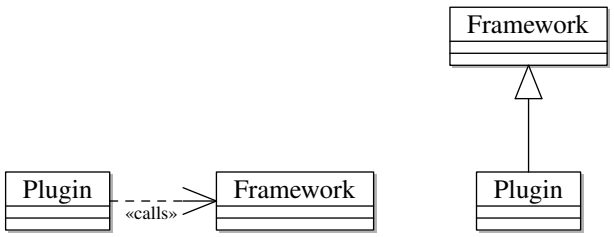
[1] It is a pivot name, the system will get the actual name upon its first release.

1.2 Libraries, Frameworks, and Plugins

Frameworks are building blocks for applications and other frameworks. They model a specific domain or an important aspect thereof [Rie00]. Object-oriented frameworks are designed and implemented along the principles of object orientation, they promote encapsulation and reusability. The interaction with its clients takes place at well-defined boundaries and serves as a means of classification of frameworks. In [JF88] frameworks, whose classes can be used as-is using object instantiation and delegation, are referred to as *blackbox frameworks*. If a framework contains abstract classes that have to be subclassed in order to be used, it is called *whitebox framework*. However, the majority of real-world frameworks combine both ways of usage and are therefore called *graybox* frameworks. The parts of a framework, that can be used and/or extended by clients, are called *extension points*.

Respectively, framework clients, which are either frameworks themselves or applications, are either called use-clients or extension clients, depending on the type of interaction with the framework. These clients instantiate the framework and usually consist of several classes that are organized in modules, referred to as *plugins*. Figure 1.1 shows ways of interaction between plugins and frameworks.

Software libraries are, similar to frameworks, also often-used software artifacts. However, there is an important difference between the two. Libraries merely export a specific functionality to be used by applications, they do not provide a scaffolding for them. Therefore, they are typically a lot less complex than frameworks. In contrast to frameworks, libraries only represent a very limited part of a domain and can rarely be extended.



(a) Blackbox Framework (b) Whitebox Framework

Figure 1.1: Ways of Interaction of a Plugin with a Framework (UML diagram)

1.3 Refactoring and Plugin Adaptation

As software evolves, that is, flaws are corrected and functionality is modified, one can often observe a decay in code quality [Leh96]. This is due to the fact that the applied changes have clearly not been part of the software design and they might have been introduced only to accomplish short-term goals, sometimes even without full understanding of the code design. According to Beck, *refactoring* is the opposite of this practice [FBB+99]. The term was originally coined in the Smalltalk community and describes a change made to the internal structure of software to make it easier to understand and cheaper to modify

without changing its observable behavior. Thus, refactoring software means to improve on legibility of code in a controlled manner. By means of small steps that are accompanied by tests to ensure that the behavior has not changed, the code is cleaned up.

This process is essential for the evolution of frameworks, because due to their complexity they are often developed jointly by many people, who have to understand each other's code. Furthermore, the larger a system is, the more flaws it will contain [Tan01, p. 617]. And, finally, no human-created software is perfect from the first instant on.

In order to be backward-compatible to plugins designed for an older development stage of a framework, the latter has not only to expose a consistent behavior to its clients, but it also needs to maintain its *Application Programming Interface* (API) and, consequently, its *Application Binary Interface* (ABI). The API is a design- and compile-time view on all the types and methods a framework provides to its clients. The ABI, on the other hand, refers to the runtime environment. In the original sense, an ABI consists of programming conventions that applications have to follow to run under an operating system. Therefore, it includes a set of system calls and the technique to invoke the system calls, as well as rules about the usage of memory addresses and machine registers [Lev99]. However, in the context of frameworks ABI refers to the framework types that can be openly used by plugins. In order for a plugin to be correctly loaded, all framework types it depends on need to be available at runtime.

However, these restrictions rather impede the fruitful evolution of a framework. Therefore, this thesis presents a technique enabling the maturing of a framework API through a set of refactorings, while preserving backwards-compatibility to its plugins.

1.4 The .NET Environment

Microsoft .NET is a platform consisting of a runtime environment, called the Common Language Runtime (CLR), and a large class library with extensive support for diverse standards. The CLR specification itself, called Common Language Infrastructure (CLI), has been submitted as a standard to the *European association for standardizing information and communication systems* (Ecma International, formerly known as ECMA – European Computer Manufacturers Association) [ECM02]. It takes the same place in the system's architecture as does the Java Virtual Machine in a Java environment. However, the CLR was not designed with ubiquitous platform independence in mind, its purpose is merely to provide memory management and exception handling, guarantee security, and execute a .NET program on the underlying Windows operating system and hardware. Nevertheless, there are efforts to provide the same functionality on other operating systems as well.

In contrast to Java, which was in its first incarnation designed to be a runtime-interpreted language, .NET does not offer anything similar. Instead, every .NET program is just-in-time-compiled to machine code, which will then be executed by the underlying hardware. The CLR supports a wide variety of programming languages, although not every single concept of each of them might be implemented [MG00]. This is done by compiling the source code into an assembler-like intermediate representation called Common Intermediate

Language (CIL). Therefore, the smallest entities .NET software is made of are called assemblies and take the form of .dll or .exe files. Similar to Java bytecode CIL contains every information necessary to run the program, but in addition to that an assembly provides information about the types defined in the assembly and the types and their containing assemblies required by the assembly together with versioning information.

Java has the notion of native code that can be called from within Java code. A very similar approach is taken by .NET: the code that is compiled into IL code, put into assemblies, and executed by the CLR is called *managed code*, while platform-specific code is referred to as being *unmanaged*. The latter will directly be run by the operating system without being able to make use of the services provided by .NET, such as automatic memory management through the help of garbage collection, security protocols enforcement, exception handling, etc. Microsoft has even created derivative programming languages from well-known languages directly incorporating these features, like C#, VB.NET, and ASP.NET.

1.5 Outline

This thesis presents a technique for the adaptation of plugins in object-oriented frameworks in order to compensate for refactoring changes. This first chapter provided an introduction to the terminology of frameworks, refactorings, and the .NET environment. Chapter 2 supplies an overview of the changes applicable to object-oriented frameworks and manageable using adaptation, while chapter 3 outlines a means of specifying them for computerized processing. The next chapter then elaborates on the different approaches for plugin adaptation that were investigated, and chapter 5 explains another approach in detail: class and interface adapters. The last chapter discusses possible improvements and further research needs.

Chapter 2

Categorization of Refactorings

2.1 Classification of Changes

Plugin adaptation for object-oriented frameworks can only be carried out successfully, if the evolution process of the framework is constrained to a controllable set of changes. Therefore, this chapter is dedicated to defining basic criteria fundamental to plugin adaptation and a set of changes adhering to the former.

Dig and Johnson [DJ05] classify changes by their impact on a client program's ability to run with the changed framework into non-breaking and breaking changes. Non-breaking means, that existing clients will work as before without recompilation. By "work as before" we mean that the set of interesting observable outputs for a given input is not changed.[1] Non-breaking changes are not further regarded, because they do not require plugins to be adapted. A change "breaks" a client, if the latter cannot continue to work. The client fails to recompile, link or run with a new framework version. By "fail to run" we mean that the set of interesting observable outputs change. Breaking changes can be further subdivided into semantic-preserving and semantic-modifying changes. The latter group is not investigated further here, because of the complexity and amount of information necessary to cope with them in plugin adaptation. Figure 2.1 shows this classification of changes.

2.2 Refactorings in Frameworks

When classifying framework changes in the context of plugin adaptation, only those changes were of interest to us, that have an impact on framework parts available to plugins. Changes to the internal framework classes do not require plugin adaptation (unless they modify the framework's behavior, however, we will not investigate this type of change further in this thesis). Changes to a

[1] We do not consider the type of input, though (e.g. values, representing time or memory allocation).

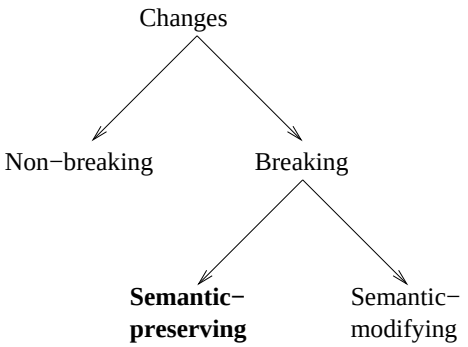


Figure 2.1: Classification of Framework Changes

framework extension point, in contrast, might alter entities seen and used by framework clients. Thus, they are changing the framework's API.

There are many types of changes that can be applied to a framework, amongst them are bug fixes, the addition of new features[2], and refactorings. Bug fixes are most often limited to statements contained in a method, they do not change the framework's API. The addition of new features per se does not do so either, it only enhances the existing API. Refactorings, however, were basically invented for source code restructuring and thus can have a major impact on a framework's API. Moreover, according to Dig and Johnson [DJ05], who investigated the evolution of four big frameworks, more than 85% of client-breaking changes were refactorings. Therefore, they are the driving force behind plugin adaptation.

Unfortunately, there is no uniform definition of refactoring throughout literature. Fowler defines a refactoring as the change of a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure [FBB+99]. In his PhD thesis, Opdyke is a little more detailed by stating that, given the same set of inputs, the original and the refactored program produce the same output values, if the refactoring's preconditions are met [Opd92]. Roberts extends this work by providing a mathematical foundation for the notion of semantic-preservation. He argues that a refactoring is a source code transformation that depends on specific preconditions and guarantees a number of postconditions to be met [Rob99]. This definition is abstract enough to be applied to many different use contexts, e.g. real-time applications with time constraints as pre- and postconditions. Therefore, this thesis will concentrate only on semantic-preserving changes, that can be specified in the manner mentioned above. However, the actual definition of pre- and postconditions for specific refactorings is at the moment not of interest for the plugin adaption approach presented here.

Consequently, this thesis deals with breaking semantic-preserving changes to object-oriented frameworks. However, this restriction does not render the work presented hereinafter unrealistic; instead, for the sake of interoperability framework development has to respect the golden rule "never to change a published interface." A *published* interface is not only public, it is also deployed and thus

[2]The addition of a new feature is sometimes performed using refactorings as well, e.g. *AddMethod*.

Change class	Example
Name Changes	Rename Method
Signature Changes	Add Parameter, Remove Parameter, Change Return Type
Location Changes	Move Field, Move Method, Pull Up Field, Pull Up Method, Push Down Field, Push Down Method
Split	Extract Class
Merge	Inline Class

Table 2.1: Classes of refactorings

potentially used by clients. Plugin adaptation can leverage this restrain insofar, as that only the framework's functionality is to be preserved but the API is permitted to change.

Another confinement of framework development is that information must not be lost by the application of refactorings. However, that means that the set of refactorings applicable to a framework is limited. In order to define this set it is adjuvant to classify available refactorings.

The majority of refactorings as proposed in [FBB+99] can be grouped into five classes: name changes, signature changes, location changes, splits, and merges. Table 2.1 gives examples for the five classes. Again, only API-modifying refactorings are of interest for plugin adaptation; refactorings like *Decompose Conditional* will not break plugins and therefore do not require adaptation.

Actually, for the classification of refactorings it would suffice to differentiate between two atomic change operations, namely *add* and *delete*. Out of these two, every other refactoring can be constructed. These are essentially the operations performed on the source code. However, these two basic operations lack the semantics connected with each refactoring shown in table 2.1. A complex source code modification composed of the two change types cannot be thoroughly understood to be reverted by the adapter generator. Consider for example the *Rename Method* refactoring. It consist of more than just the removal of the old method and the addition of the same method with a different name, it provides a semantic link between the two, namely, that both are semantically the "same" method. This information must be retained for the adapter generation process to function.

Many complex refactorings can be built up using very basic changes. For the sake of simplicity we suggest, that plugin adaptation should work on the atomic refactorings in lieu of the complex one represented by the basic refactorings. If, however, the complex change carries more semantics than "the sum of its parts" does, the complex refactoring needs to be used, lest information about the change is lost.

When concentrating on a specific subset of changes, namely semantic-preserving breaking ones, this also has an impact on how the framework can evolve. In order for plugin adaptation to completely cover all refactorings applied to the framework in the course of its development, the latter has to be restricted to employ only those changes supported by the adaptation technique. This means, that framework developers will need to have a list of the types of changes

they are allowed to use. Furthermore, the development respectively refactoring tool employed should be customized – if possible – to refuse the application of changes known to potentially break adaptation.

Chapter 3

Specification of Changesets

3.1 Required Refactoring Information

The information about the refactorings that have been applied to a framework in the course of its evolution is crucial for the plugin adaptation process. That is the reason why we call the latter Δ-guided[1]: only the changes made to a framework need to be known in order to perform adaptation. Other approaches rely on a complete specification of the extension point state a plugin expects. However, this is error-prone and requires more maintenance effort than using only the change information. When recording refactoring information, specifically three aspects are important:

1. The type of refactoring that has been applied,
2. The entity that has been changed,
3. The order in which the refactorings were made.

The first point has already been mentioned in chapter 2 and arises from the need to understand the semantics behind a change. Without that information no compensation for a certain refactoring is possible. In addition to that, the location of the change must be specified, so that adaptation can be applied to the very entity referenced. Due to the atomicity of refactorings, often several of them are performed on a single entity in order to achieve architectural improvements or increase source code legibility. In that case, the order in which the changes are applied is important. However, also changes made to different entities are sometimes interconnected. Consider, for example, two extension point types (types, that are part of an extension point), that are going to be renamed. The first type `Address` will be renamed to `AddressFactory`, because it contains the logic to create addresses rather than represent them. The second type `AddressData`, that is used to store postal code, place, etc., will be renamed to `Address` – the same name the first type carried before. However, refactorings transforming the `Address` type now refer to a different entity than they would have before the renamings. Therefore, the order of refactorings has to be accounted for in the change specification and consequently in the adaptation process.

[1] Δ is the symbol for the Greek letter Delta.

If the framework development is carried out using a software configuration management system with version control support, such as CVS, SVN, etc., the refactoring information required can be obtained, at least partly, from that system. However, in order to satisfy the requirement of no information loss it might still be necessary that the developer, who applied the refactoring, inserts information manually.

3.2 Change Definition Language

The refactoring information has to be processed during plugin adaptation, which leads to the requirement of machine-readability. Therefore, it has to be represented appropriately. In the following we will call the language used to accomplish this *Change Definition Language*. If there is no tool support that hides away the details of the language, it should also be human-readable, because framework developers might need to make modifications to the information obtained from a version control system, or they might have to insert additional information to compensate for complex refactoring semantics not detectable automatically. The best of both worlds is provided by the eXtensible Markup Language (XML) [BPSM+04], which is in addition extensively supported by .NET.

XML provides two means to instantiate a domain-specific data markup language from it, Document Type Definitions (DTD) and XML Schemas [Fal04]. They are used to model the structure of the data to be described in an XML document. Besides that, there are a couple of auxiliary technologies, such as XPath for navigating [CD99], XQuery for querying [BCF+06], and XSLT for transforming [Cla99] XML documents. The .NET Class Library provides implementations for all these standards.

In figure 3.1 an exemplary document type definition for changeset specifications is shown. This DTD is also being used in the prototype implementation presented in chapter 5. An XML document adhering to the DTD contains a single `changeset` element with the mandatory attributes `extension`, `fromVersion`, and `toVersion`. The first attribute specifies the extension point the changeset document refers to. The other two attributes contain the version numbers that make up the refactoring interval. Thus, all changes that have been applied to the specified framework extension point between the two versions will be contained in the changeset document. The `changeset` element can have any number of `change` elements as children. They describe single refactorings with the help of their attributes. The `type` attribute is required and contains the refactoring type. The refactoring's target type name can be found in the `targetType` attribute, which is obligatory as well. It indicates the type context, in which the change has taken place. All other attributes are not necessary for every type of change and are therefore not obligatory. The `targetMember` attribute refers to the type member the change has been applied against, while `newValue` contains the change subject's new state in a textual representation (i.e., a type or method declaration). The `accessHint` attribute should be used to provide additional information to the adapter generation process. For example, this becomes necessary, if a new parameter has been added to a method in an extension point type. In that case the generated adapter needs to know which value to provide for the new parameter.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE changeset [
  <!ELEMENT changeset (change*)>
  <!ATTLIST changeset
    extension CDATA #REQUIRED
    fromVersion CDATA #REQUIRED
    toVersion CDATA #REQUIRED
  >

  <!ELEMENT change EMPTY>
  <!ATTLIST change
    type (Rename | Move | Add | Split | Merge | AddParameter |
             RemoveParameter | ChangeReturnType) #REQUIRED
    targetType CDATA #REQUIRED
    targetMember CDATA #IMPLIED
    newValue CDATA #IMPLIED
    accessHint CDATA #IMPLIED
  >
]>
```

Figure 3.1: Example XML Document Type Definition

Figure 3.2 shows an example of a changeset specification using XML. The changeset describes changes applied to extension point `extPtA` between versions 1 and 2. The first change contained is a name change of the type `OrigType`. In the course of framework evolution it has been renamed to `RenamedType`. The second change in the changeset document depicts a refactoring, in which a method parameter has been removed. The method `MyMethod` defined in the type `RenamedType` expected one integer parameter, which has been obsoleted during framework development and was removed.

```
<?xml version="1.0" encoding="UTF-8"?>
<changeset extension="extPtA" fromVersion="1" toVersion="2">
  <change targetType="OrigType" newValue="RenamedType" type="rename" />
  <change targetType="RenamedType" targetMember="MyMethod(int)"
    newValue="MyMethod()" type="removeParameter" />
  ...
</changeset>
```

Figure 3.2: Example XML Changeset Specification

Chapter 4

Approaches for Plugin Adaptation

This chapter introduces and evaluates several different techniques that were investigated for the use in plugin adaptation.

4.1 Remote Procedure Calls and the Dynamic Proxy Pattern

4.1.1 Concept

Remote Procedure Calls

Remote Procedure Calls (RPC) were originally invented to allow programs to call procedures located on other CPUs without the necessity of message passing or I/O [BN84]. The called procedure looks to the caller exactly as if it were located on the same computer. The calling procedure is referred to as *client*, while the remote procedure is called *server*. In order for that location transparency to work, a library in the client's address space and another one in the server's address space is required. These libraries implement the remoting of the procedure call and the transmission of parameters (henceforth referred to as *marshalling* and *unmarshalling*). However, RPC does not require the called procedure to reside on a different computer than the calling one, in fact, they can be provided by different programs on the same computer or even be part of the same application.

Figure 4.1 shows the schematics for a remote procedure call. Communication between client and server is routed through stubs on both sides. The server functionality is defined in an interface, against which the client has been compiled. The client stub implements this interface to be able to mimic the server. It accepts the client's call, creates a message for it, marshalls the call parameters and adds them to the message, and sends the message over the communication line. The server stub receives the message, unmarshalls the parameters, and calls the server method described in the message. Return values and error messages are handled equally.

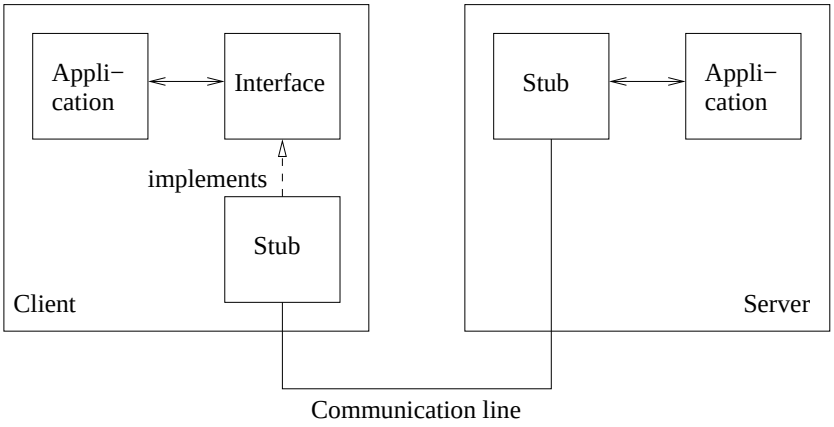


Figure 4.1: Communication Structure of a Remote Procedure Call

Many distributed computer systems were built upon that technique, amongst them are the Common Object Request Broker Architecture (CORBA) [Vin97] and Microsoft's Distributed Component Object Model (DCOM) [BK98]. The CORBA platform even introduced a new language for specifying service interfaces, the Interface Definition Language (IDL). It has enjoyed wide popularity ever since, even far beyond CORBA. IDL is used for defining service interfaces in a language-independent fashion; an IDL compiler can then create language-specific counterparts (i.e., a Java interface), upon which the service and its clients are implemented. In addition to that, the IDL compiler also generates client and server stubs (the latter being called *skeletons*), which represent the communication partners for the client and the server program respectively. The location transparency is achieved by means of a skeleton caching and lookup mechanism, called Object Request Broker (ORB). CORBA furthermore takes care of converting endianness, which enables programming language and operating system independence.

In CORBA or DCOM plugin adaptation could be accomplished by placing the adaptation logic into the client stubs. Plugins would then communicate with the framework through these stubs. They essentially present a view of the framework to the plugins, but this view does not necessarily have to equal the current framework version. Plugin adaptation would be achieved, if the client stub could present a view of the framework for the specific version required by the plugin.

Dynamic Proxies

A slightly different technique is proxying. Here, both the client stub and the server stub resp. skeleton are the same runtime object called *proxy*. Its task is not the remoting of method calls but to provide a level of indirection between the callee and the caller. This intermediate object can then be used to perform logging or access control thereby implementing a certain security policy. Proxy objects are generated dynamically at runtime. Everytime a client requests an instance of the server object (e.g., by calling a factory method), it is not returned the actual server instance but a proxy delegating to it. As a consequence, the

proxy has to provide the same interface to the callee as the server object. Figure 4.2 shows a UML diagram for the dynamic proxy pattern.

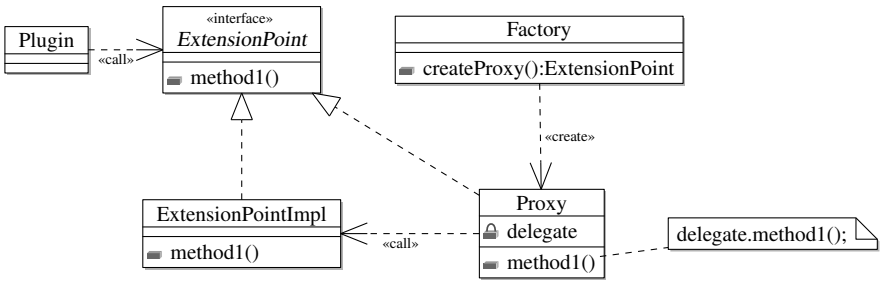


Figure 4.2: The Dynamic Proxy Pattern (UML diagram)

The .NET platform targeted by this thesis builds on this technique to provide Inter-Process Communication (IPC) and Remote Procedure Calls. The so-called .NET Remoting infrastructure abstracts away the details of communication and thereby offers great flexibility to its users. Therefore, it introduces the notion of Application Domains (AppDomains), that serve as logical containers for applications or parts thereof. They isolate the parts they contain in order to increase program stability and security. Communication between different AppDomains can be established, and uses dynamic proxies internally. To make use of this infrastructure, the framework as well as each plugin would need to run in its own AppDomain. As a side effect, this architecture pattern also provides location transparency, so that plugins would no longer need to reside on the same computer as the framework.

To implement plugin adaptation using dynamic proxies, the proxy objects would need to be modified in order to perform the adaptation. Instead of just redirecting a method call to the server, a proxy would have to transform the call in order to compensate for the framework changes.

4.1.2 Evaluation

When using third-party middleware, such as CORBA, DCOM, or .NET Remoting, as the foundation for plugin adaptation, an architecture and runtime dependency on that infrastructure is introduced. This dependency furthermore carries limitations intrinsic to the technique used. For example, in CORBA and DCOM communication can only take place through well-defined interfaces specified in IDL. Consequently framework use would be restricted to placing method calls against these interfaces, without supporting inheritance. This limits development flexibility and conjures up the need for implementing inheritance-based design patterns using workarounds.

.NET Remoting also places a burden on the framework and plugin developers. In order for program parts residing in different AppDomains to communicate with each other through the passing of objects, the latter must be specifically designed for that purpose. In detail that means, that the objects need to make their serializing logic explicit to be marshalled by value, or extend a certain base class to be marshalled by reference.

Although no middleware dependency will arise, the same restrictions apply

when using dynamic proxies. Moreover, in comparison to using RPC-like communication infrastructure as provided in CORBA and DCOM, dynamic proxies suffer a major disadvantage: they are generated dynamically at runtime. Client stubs and server skeletons, in contrast, are created once from the IDL specification and are compiled to binary form. The adaptation logic can be hardcoded into the stubs, while dynamic proxies would need to provide that at runtime and would thus inflict runtime overhead on the system.

Another issue that cannot be handled easily for dynamic proxies are name collisions. In order for older plugins to be loaded correctly, they need a copy of the extension point interfaces they were compiled against. If the names of the interfaces delivered with the framework have not changed since the plugin was compiled, but these interfaces have been modified in another way, the system will try to load both and fail. This is due to the fact, that a type's name works as a unique identifier for the type. In .NET, additional information (such as version number, culture information, etc.) can be used to form a unique type identifier. However, this remedy can only be used in conjunction with assembly signing, so considerable effort is required to work around this problem.

4.2 Aspect-Oriented Programming

4.2.1 Concept

Aspect-oriented software development describes the activity of programming with multiple crosscutting concerns or *aspects* [FECA04]. Often-quoted examples of aspects are security, logging, distribution, and transactionality. System developers express the behavior for each concern in its own module which is then *woven* together with the other modules into a working system. Thus, each concern can be developed separately, while the specification of interaction between concerns is formalized independently using so-called *join points*. Without aspect-orientation crosscutting concerns would have to be implemented over and over again in different places. This becomes obvious when imagining that the exemplary aspects mentioned above would be required by several methods. In that case, every method would need to know about how to obtain a reference to a logger, how to validate the current action against a security policy, etc. Code that is distributed in such a fashion is called tangled. If the way of logging or the security enforcement should be changed, many classes would need to be touched in order to adapt every occurrence of that aspect. Aspect-oriented programming (AOP) eases this problem by separating these aspects from the domain functionality provided by the software. Figure 4.3 shows the basic modus operandi of an aspect-oriented system.

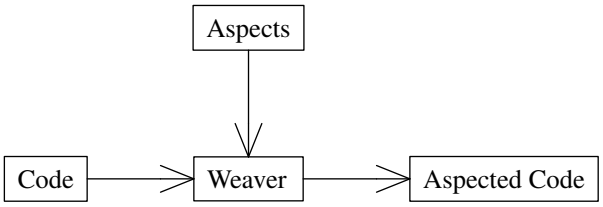


Figure 4.3: Aspect-Oriented Programming Workflow

Aspect-oriented programming is a technique orthogonal to classic programming techniques like functional programming, object-oriented programming, etc. That means, that aspect-orientation can be used in conjunction with all of these techniques in order to take care of crosscutting concerns. However, as we evaluated it in the context of plugin adaptation for object-oriented frameworks in .NET, this section will concentrate on object-oriented implementations of this technique for the .NET platform.

All implementations of the aspect-oriented programming paradigm can be distinguished into two classes: compile-time and runtime AOP. When performing compile-time aspect-orientation, the aspects' code and the program's functional code are woven together before the compilation takes place, that is, the weaver is a kind of precompiler. With runtime AOP, however, aspects and functional code can be compiled separately and the weaving is done either statically when the program is loaded or dynamically during program execution. There are different approaches on how this can be implemented in .NET, such as rewriting the compiled code at load-time to contain calls to hook methods that check whether aspect code should be executed or not, see for example [Gil] and [Ver]. Another way of achieving runtime aspect-orientation is to subscribe to the CLR as a debugger that gets notified whenever a method is called and can then act appropriately, see [FGA04].

4.2.2 Evaluation

When reconsidering the setting described in the first chapter it becomes clear that compile-time aspect-orientation cannot be applied in order to achieve plugin adaptation. This is due to the fact that each customer installing a new version of the Comarch framework would need to recompile both the framework and the third-party plugins. First of all, this would only be doable, if the framework's and the plugins' source code were available, and, secondly, it cannot be expected from a customer to perform compilation on her own, because this would mean to install a compiler. Furthermore, this would prohibit the installation of additional third-party plugins later on without having to recompile the whole system.

Implementations permitting runtime weaving of aspects do not suffer these disadvantages. However, as of this writing the .NET platform is relatively young and in early development compared to Java and so there are only a few available.

Besides that, the main challenge when using aspect-oriented programming for plugin adaptation is how to specify reusability or backward-compatibility as an aspect. Aspects usually encapsulate non-functional software requirements – as is reusability. In spite of that, a more sophisticated implementation technique would be necessary as when implementing logging or security policy enforcement. Instead, one could use the weaving functionality under the hood to weave adaptations into the program code. However, this would revoke many advantages gained from using aspect-oriented software development, because no modeling or development of aspects independent from the rest of the program takes place.

Chapter 5

Class and Interface Adapters

5.1 General Adaptation Architecture

The plugin adapter generation approach presented in this thesis builds up on class and interface adapters, that serve as an intermediate layer between the plugins and the framework. The adaptation process is guided by change information specified using a Change Definition Language. This process, as shown in figure 5.1, consists of three phases: refactoring detection, change representation and actual adaptation.

- **Detection.** We combine capturing the refactorings in the IDE (like in CatchUp! [HD05]) with semi-automated specifications in a version control system (e.g., CVS [CVS]). On the one hand, we want to reuse and extend the refactoring facilities of an IDE (such as Visual Studio) in a CatchUp-like manner. For complex refactorings, which may not be directly supported in the IDE, we imply the use of .NET annotations in combination with CVS and semi-automated detection of refactorings for marking changes.
- **Representation.** The core of the technology is the Change Definition Language (CDL), in which all the relevant information about the software change is represented. Such information includes, for each framework version, which software entities were changed and by which refactorings. We call this information change specification. For further details, see chapter 3.
- **Adaptation.** Based on the change specification, the actual adaptation is performed. The change information is used to reconstruct a certain framework state that is then serialized into class and interface adapters.

When recreating the framework state at a given version, not the complete framework is rebuilt, only the extension points, that is the parts visible to plugins. The class and interface adapters directly communicate with the actual framework, so that there is no adapter chaining possibly incurring performance degradation. The generated adapters are stored in their own assembly, that

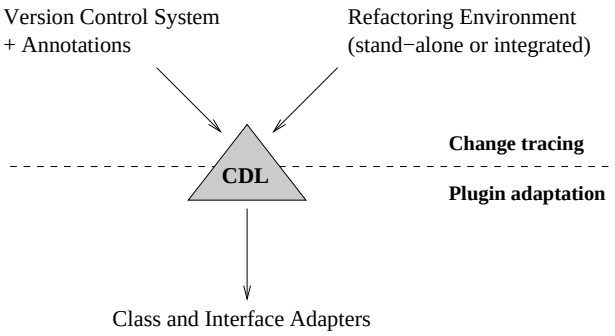


Figure 5.1: Plugin Adaptation Workflow

takes the place of the framework in the version represented by the adapters. These adapters reify the framework's extension points in their binary form (that is, their ABI), which is why the plugins will transparently link and run against them. Application binary compatibility is thereby recovered.

Figure 5.2 depicts the role of adapters during the evolution of a framework throughout three consecutive versions. The first picture shows the framework's initial version together with a plugin compiled against it. No adapters are needed. In the second picture, the framework has evolved to version 2, and a new plugin has been developed for it. The plugin for the first version requires adaptation to be performed in order to run with the new framework version. Therefore, the adapter generator uses the change specification in order to recreate the framework extension point expected by the plugin. The third picture shows another step in the framework's evolution. Here, both plugins one and two need adapters to be placed in between them and the framework. These adapters are not chained, they directly forward to the actual framework. Again, a new plugin is developed for the current framework version.

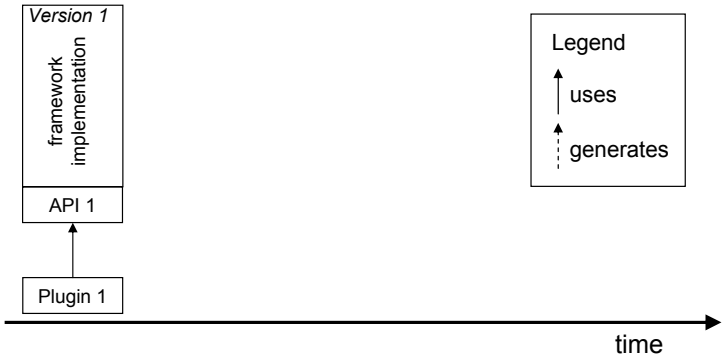
For further details on the general adaptation architecture we refer to [SRB06].

5.2 Adapter Concept

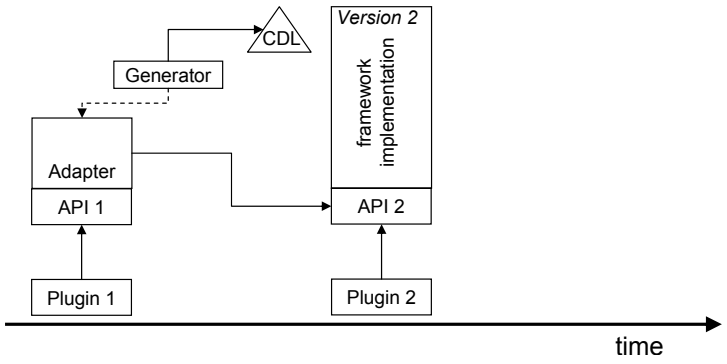
A class adapter is a design pattern that converts the interface of a class into another interface clients expect [GHJV95]. *Interface* here refers to the entirety of methods a class provides to its clients[1]. The adapter design pattern – sometimes also referred to as *wrapper* – can be described by four roles, that are played by the participating runtime objects:

- **Target.** Defines the interface the client uses.
- **Client.** Collaborates with objects conforming to the target interface.
- **Adaptee.** Defines an existing interface that needs to be adapted.
- **Adapter.** Adapts the interface of the adaptee to the target interface.

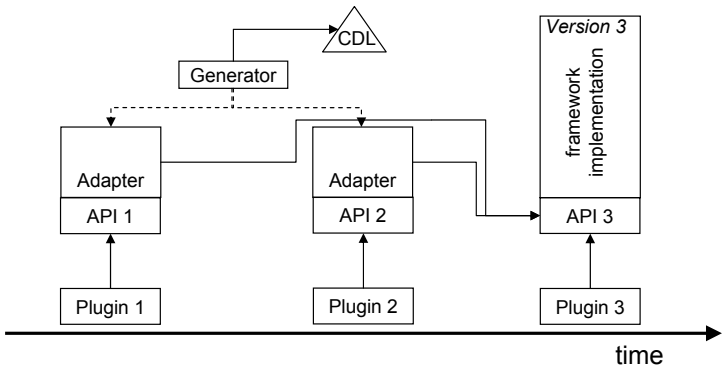
[1] Technically speaking, a class' *interface* also contains all public fields declared by that class; however, as this violates the elementary principle of data encapsulation respectively information hiding in object-oriented software development, we will ignore them here.



(a) Version 1



(b) Version 2



(c) Version 3

Figure 5.2: Runtime System Architecture showing the Evolution of the Framework

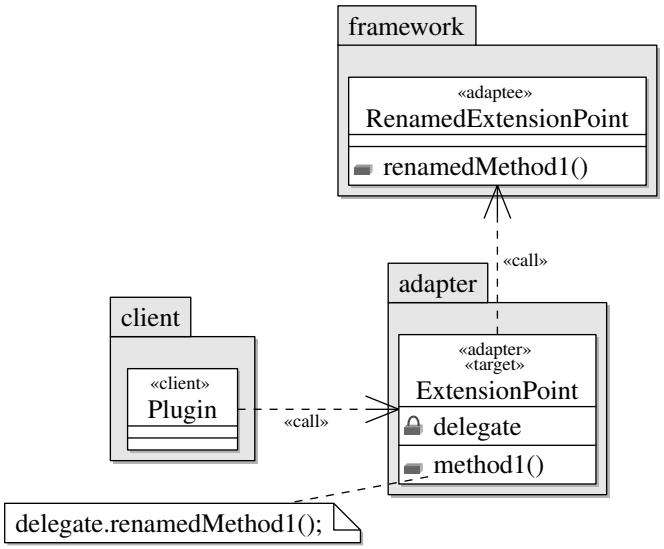


Figure 5.3: Class Adapter annotated with Roles (UML diagram)

In this approach, the adapter and adaptee classes are not different concerning their semantics or domain, they just represent two different states of the same extension point class with respect to framework evolution. The generated class adapter plays both the adapter and the target role. Adaptation is implemented using *delegation*: each method defined in the adapter forwards to the corresponding method in the adaptee. This is also the place where compensation for framework refactorings is performed. The methods forwarded do not only comprise those defined in the immediate adaptee class, in fact all methods in the inheritance hierarchy have to be delegated. Figure 5.3 shows a UML class diagram for a class adapter.

Interfaces have to be treated differently. They cannot redirect method calls to delegate objects because they only constitute a contract an implementor agrees to obey. Nevertheless, they have to be regenerated, if the entire framework extension point state is to be recreated by the adaptation process. In order to perform adaptation for interfaces, two additional adapter classes have to be generated. Each of them implements one interface version and redirects to an instance of the other version similar to a class adapter. This is necessary whenever an interface implementation is to pass through the adaptation layer.

Figure 5.4 shows the forward interface adapter that is handed to the plugin, whenever it needs to access an interface implementation provided by the framework. The framework's implementation is "wrapped up" inside the interface adapter and passed on to the plugin. Due to the interface adapter implementing the generated old interface version, the implementation provided to the plugin looks like an instance of the old interface.

The backward interface adapter shown in figure 5.5 becomes necessary, when the plugin has to implement an extension point interface and present an instance of that implementation to the framework. This is essentially the concept of a *callback* – the framework can then use that instance and notify the plugin by calling methods on the instance. The backward interface adapter takes the

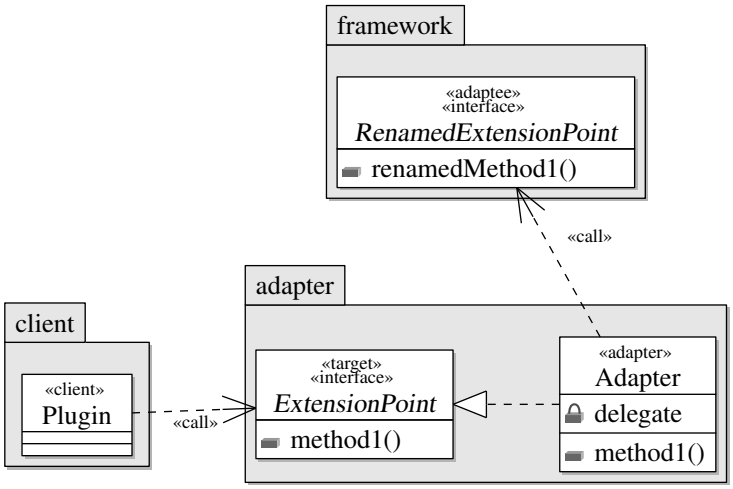


Figure 5.4: Forward Interface Adapter annotated with Roles (UML diagram)

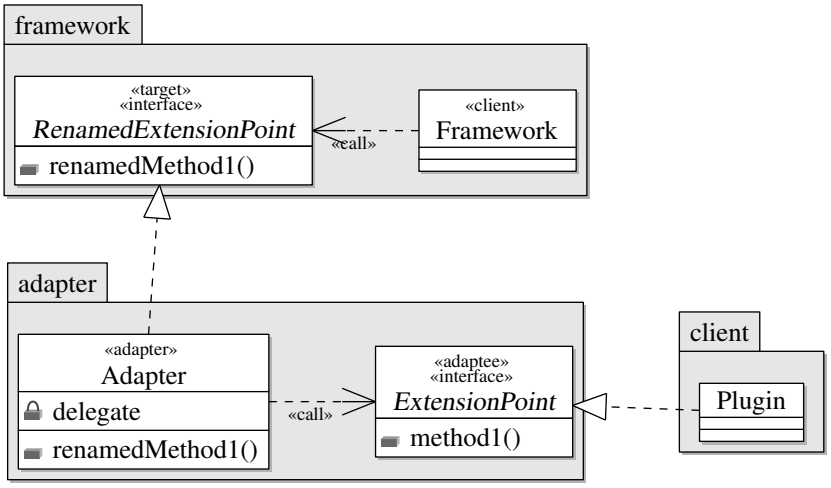


Figure 5.5: Backward Interface Adapter annotated with Roles (UML diagram)

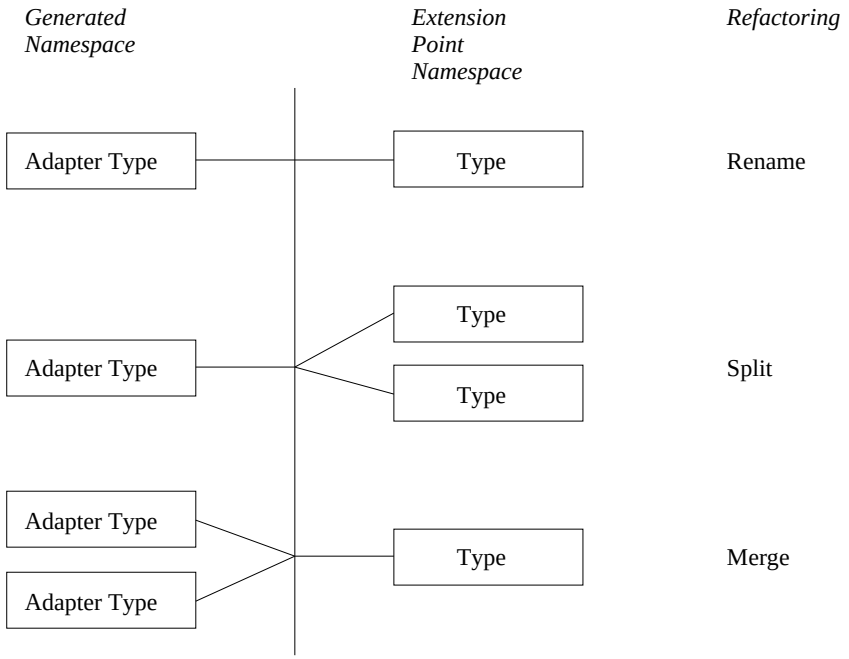


Figure 5.6: Type Mappings for Refactoring Classes

plugin's implementation of the old interface and presents it to the framework as an implementation of the actual interface.

The aforementioned adaptation technique works for very basic change types, such as the renaming of a method or a class. These changes are simple 1:1 mappings, that is, the application of the change transforms a single source entity into a single target entity. However, more complex changes also require more effort in compensating them. Type splits and merges are typical examples for 1:n, respectively n:1 mappings. Figure 5.6 shows the three classes of type mappings: the first column displays the generated class and interface adapters mapping to the extension point types shown in the second one, while the third column names a refactoring exemplary for the depicted mapping. In order to cope with these changes, class and interface adapters have to be extended to maintain several delegate objects instead of a single one.

A side-effect of this extension is that framework developers need to supply information about the relations of the delegate objects in addition to the actual split or merge change. This becomes obvious in the following example: consider an extension point type `OldType` that has been split in two (`NewTypeA` and `NewTypeB`). The class or interface adapter that reifies `OldType` thus contains two delegate fields, one for `NewTypeA` and the other for `NewTypeB`. If a method of a framework extension point type returns one of the two new types, the adapter is wrapped around it, meaning that one of its delegate fields contains the method's return value. However, with only one delegate field initialized, the adapter only represents half of the state and functionality of the original entity `OldType`. It therefore needs to know how to obtain the second delegate object. This additional information has to be specified by the framework developers in the form of an *access hint*, as shown in figure 3.1 on page 17.

5.3 Prototype

In order to provide a proof-of-concept for the plugin adaptation approach presented in this thesis, a prototype has been written. It uses the XML Change Definition Language format described in chapter 3. As the prototype has been architected in a modular fashion, the XML format can easily be exchanged with a more sophisticated language. However, it is limited to just a few change types and supports only the core language features. These are, nevertheless, sufficient to build even complex applications. Chapter 6 discusses the future work associated with extending the prototype and implementing support for further language elements.

The process of adapter assembly generation is deterministic. The only inputs required are the framework's extension point assembly that is to be adapted and the change information describing its evolution. Likewise, the sole output created is the adapter assembly containing the adaptation logic. Therefore, the prototype does not offer any graphical user interface. Instead, it is deployed as a command-line executable. Figure 5.7 shows the input/output architecture of the prototype.

The prototypal adapter generator has been implemented in C#. This choice has several reasons, the most striking one being that the framework requiring plugin adaptation and thereby motivating this thesis is written in C#. Thus, the adapter generator was to generate an adaptation layer for the .NET platform. Luckily, the .NET class library already offers broad support for the tasks needed to be carried out. The `System.Reflection.Emit` namespace abstracts away many hassles connected with generating CIL code, while `System.Xml` types implement a variety of ways to deal with XML documents. The use of these two building blocks is shown in figure 5.8. Besides that, implementing the prototype in C# helped a lot to further the understanding of certain language features the adapter generator had to cope with.

Algorithm

The algorithm the adapter generator prototype implements is exemplarily sketched in figure 5.9 for an extension point in version 3. The first step is the loading of inputs – both the extension point to adapt and the corresponding change specifications. The extension point is then introspected using the types provided in the `System.Reflection` namespace, and a metadata structure containing all information about the extension point types is built up. For each previous version an adapter assembly redirecting from that version to the current one

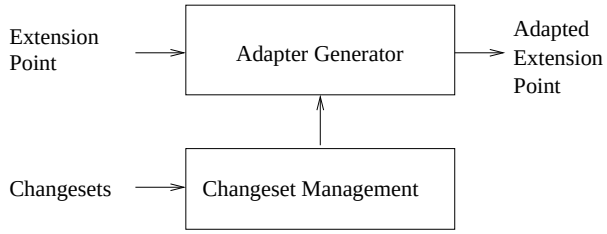


Figure 5.7: Input/Output Architecture of the Prototype

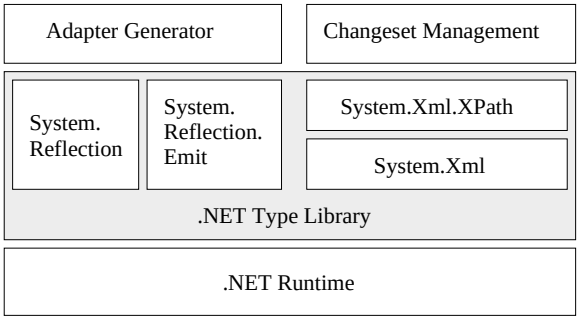


Figure 5.8: Structural Architecture of the Prototype

is created. Thus, the prototype invocation depicted in the example would produce two adapter assemblies, the first reifying extension point version 2 and the second bridging the gap between version 1 and 3.

```
Load changesets and extension point;
version := 3
Create Type Metadata Containers from extension point
For each (version -= 1) > 0 do
  Obtain changeset for version difference
  For each change in changeset do
    Apply change to Type Metadata Container
  Create adapter types
  Create adapter assembly for version
```

Figure 5.9: Adapter Generation Algorithm in Pseudocode

A single adapter assembly is constructed as following. First, a changeset for the version difference between the metadata containers and the version requested has to be obtained. Second, while iterating over it in reverse order, each change contained in the changeset is applied to the metadata container. The transformation taking place effectively reverts the changes made to the extension point in the course of framework evolution. After all changes contained in the changeset have been worked through, the transformed metadata containers are serialized to types using the `System.Reflection.Emit` namespace. Finally, the types are placed within a .NET assembly that is written out into a file. In case another adapter assembly is to be constructed for the same extension point, the metadata containers are reused and simply further transformed until they reflect the right extension point state. The metadata transformation for three changes is shown in figure 5.10.

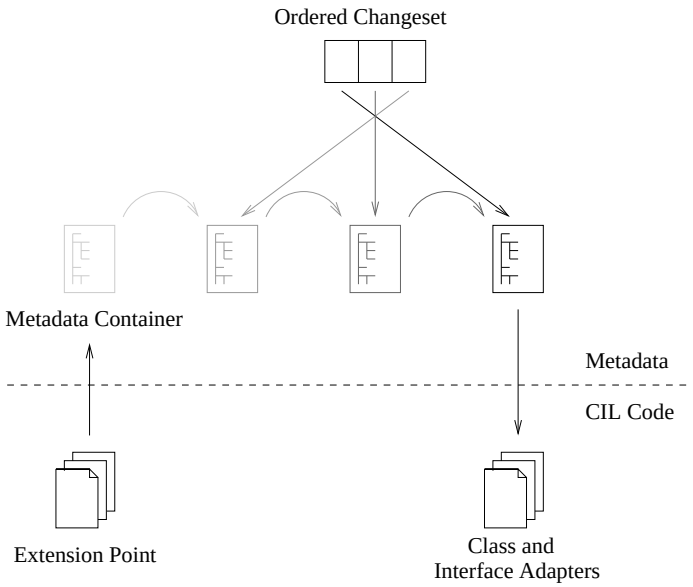


Figure 5.10: Backward Metadata Transformation using an Ordered Changeset

5.4 Testing the Prototype

Requirements of the Testing Environment

Testing is an elementary part of software development, it is crucial to software quality and vital to the success of the product [Som01]. This is even more important in the context of the plugin adapter generation process, due to the complexity of the tasks connected to it. Type and method generation, even though carried out by means of the `System.Reflection.Emit` namespace's helper classes, is a complicated matter. For that reason, testing becomes a necessity. However, the generator does not contain a lot of business logic, that could be tested using unit tests. Instead, the correctness of the generated CIL code has to be validated and the interaction of the generated code with a plugin and a framework needs to be tested. This, unfortunately, cannot easily be achieved using simple unit tests, a different approach is needed.

When designing a testing environment for the plugin adapter generator prototype, a few requirements had to be satisfied. The environment needed to be:

- **Separated.** The testing setup should not hinder feature additions to the prototype. This means, that extending the adapter generator with new feature support should not require changing the testing environment, otherwise maintenance effort would double.

- **Extensible.** New features added to the prototype should be easily testable by simply adding a new test case to the testing setup. As a consequence, the testing environment must be designed in an extensible fashion, so that adding a new test does not require more work than was necessary for adding the feature support to the prototype.

- **Generative.** In order to keep maintenance costs small, the testing environment should reuse as much code and also generate as much as possible.

Necessary Artifacts

A test run consists of two phases. At the beginning, the adapter generator prototype is run against a framework extension point by supplying a changeset document. The adapter assembly produced by the generator then has to be checked by placing it in between a plugin and the actual extension point. Thus, the following artifacts are required for running a test case:

1. Adapter generator run
   - **Extension point binary, version 2.** Will be delegated to by the generated adapter assembly, is part of the input to the generator.
   - **Changeset.** Specifies the evolution of the extension point, is part of the input to the generator.
2. Adapter assembly test
   - **Plugin.** Calls the extension point through the generated adapter assembly.
   - **Extension point source, version 1.** Needed for compiling the plugin and for deriving version 2.[2]

When designing the prototype, the B2 framework by Comarch wasn't available yet. Therefore, we had to mimic a framework extension point and create a plugin for it in order to be able to test the adapter generator during development.

Testing Solution

The most effective testing approach would be to just use the extension point source in version 1 and apply the changes specified in the changeset in order to obtain the version 2. This way, test developers would not need to maintain two versions of the same extension point. In order to perform this forward transformation of the extension point source, some logic is needed for matching each change to a source and target location in the source files and applying it then.

Unfortunately, the `System.CodeDom` namespace intended for parsing source code and representing it in an object-oriented tree-like model cannot be used for this task, because it only provides a limited subset of C# language constructs and expressions. Furthermore, the CodeDom implementation delivered by .NET does not contain a C# parser. Even utilizing regular expressions in order to find places in the source code that should be modified will not suffice. This is due to the fact, that the Change Definition Language has intentionally been designed to contain only changes made to the extension point API as seen by the plugin. A refactoring, however, comprises more than just API changes, it needs to ensure

[2]For the compilation of the plugin it would actually suffice to have the binary of the extension point in version 1, because compiling source code to .NET CIL does not require any header or source files to be available.

that other source code locations in the extension point referring to the changed API entity are modified as well. This is achieved by performing use- and call-graph analysis and by applying graph transformation and rewriting. In fact, a full-fledged refactoring engine is necessary to accomplish the transformation. However, this is not feasible for a prototypal implementation.

The actual testing environment used during the development of the prototype is less optimal than described above, because it is not generative. Two extension point versions have to be maintained separately in source code and kept in sync with the change specification. Nevertheless, it was easy to set up and does not depend on third-party products. However, when further developing the prototype, some effort should be spent on optimizing the testing setup, in order to decrease the amount of work necessary for adding another test case.

5.5 Implications

The class and interface adapter generation approach presented in this thesis offers several advantages over other techniques described in chapter 4. The most notable one is the absence of any middleware dependencies in comparison to CORBA or DCOM. This also means, that there are no special development constraints to be taken into account, such as the use of IDL or the implementation or extension of special interfaces or classes. Furthermore, the few limitations imposed by the adapter generation approach, that are discussed in the next paragraph, only apply to the public API seen by plugins, the framework development process is not affected. Another advantage can be phrased with the term "100% .NET" – in contrast to AOP this approach does not require special compilers or aspect weavers, it works with pure .NET. The only maintenance effort required is limited to creating changesets describing the evolution of the framework. The consequence of this straightforward technique is a gained freedom with respect to the location of adapater generation; it does not matter, if the adapter generator is run at the framework development site or at the customer site, this just depends on whether the changeset information should be deployed with the framework or not. Even if adaptation is carried out at the customer site, she will not notice anything, because the adapter generator can be called automatically during the installation process.

As already mentioned, the use of adapter generation also has a few limitations, although they do not really limit framework developers when properly employing object-oriented software development techniques. The main downside is, simply put, the fact that class and interface adapters constitute no *silver bullet* for plugin adaptation. Nevertheless it achieves a high efficiency when a few things are born in mind: The golden rule "Never change a published interface" as formulated by Fowler [FBB+99] can be weakened to "Never change a published behavior nor remove information." The first part of the revised rule refers to the types of changes applied to a framework in the course of its evolution – only behavior-preserving refactorings are supported by the adaptation approach presented in this thesis. The second part relates to an imperative resulting from the need to completely regenerate a previous extension point state; in order for that to work, no information about that previous state (i.e., in the form of changesets or deprecated methods) must be lost.

Some more concrete restrictions result from the feature set implemented by

the adapter generator prototype. At the moment, there is no support for structs or enums, and although it might be straightforward to add such support it has not been evaluated. A slightly different complication is introduced by the use of static members fields, as they can only be supported to a limited degree, namely constants. Static fields can only be adapted, if they hold unmodifiable primitive values, not because of the lack of implementation in the prototype but for reasons intrinsic to class and interface adapters. Another advise, that also holds true in general regardless of adaptation, is to never let plugins feel the need to use the rich reflection functionality provided in .NET in order to analyze the extension point. Otherwise plugins might discover that the extension point view provided by the class and interface adapters is different to the one they were originally compiled against (i.e., in terms of more constructors and additional types).

Chapter 6

Future Work

6.1 Support for more Language Features

Due to the limited time frame available for this thesis, the adapter generator prototype was not developed for functional completeness. As of this writing, it only supports a limited number of changes. Nevertheless, it has been designed with extensibility in mind using common design patterns. Therefore, adding new changes should be an easy task.

The prototype supports only a subset of IL language features. Yet, the supported set of features suffices to create complex programs. A future task would be the addition of support for the following language features:

- **Attributes.** In IL one can associate additional meta information with API entities using attributes. This metadata does not provide any functionality, but it should be duplicated in the generated adapter assembly.

- **Delegates.** Function pointers as known from C are called delegates in C#. They are represented by special types deriving from `System.Delegate`, whose methods' bodies are created automatically at runtime by the Virtual Execution System. Adaptation support for delegates could be implemented by providing custom delegate adapter types.

- **Enums.** These special types inheriting from `System.Enum` provide enumerations of named constants. Thereby they define a value space that cannot be circumvented, because it will be checked by the compiler. It might be possible to support enums in plugin adaptation by marshalling their members to and unmarshalling them from their underlying primitive type.

- **Events.** They are special type members and provide a language-level implementation of the observer pattern. Delegates can be added to and removed from events. Subscribed delegates will be notified when the event's fire method is invoked.

In order to support these features, the set of changes applicable to them has to be investigated and custom adaptation techniques have to be developed.

6.2 Optimizing the Prototype

Another task for further investigation is the optimization of the prototypal adapter generator. In parts, the generation process could be parallelized for better performance. Many pieces of the generated adapter types are independent of each other and could therefore be generated independently.

In many places the generated adapter assemblies use object instantiation for creating new adapters. Instead of creating a new object everytime, cloning could be used. This is due to the fact that the only state an adapter object possesses consists of the delegate fields, which would need to be set after cloning an existing adapter object.

Care must also be taken when extension point interface implementations are used excessively, in order to avoid multiple adaptation. Whenever an adapter has to handle an interface type either as a parameter or return type, the adapted method does not check whether it is already passed an interface adapter, instead a new adapter is instantiated and wrapped around the object. To address this issue, a runtime check would need to be introduced to all locations where interface adapters are created.

Further improvements comprise error handling and adapter assembly signing for the use in trusted environments.

6.3 Automatic Changeset Generation

A crucial point in the concept of plugin adaptation as presented in this thesis is the specification of framework changes in a Change Definition Language. However, the more complex the framework under refactoring is, the more changes will potentially be applied to it. Requiring the framework developers to manually track each change and record it in a changeset document is error-prone and will considerably slow down the process of refactoring. Therefore, the information about framework changes should be generated automatically as far as possible. Because of its complexity, the probability of a framework's source code to be administered by means of a source code management and versioning system is very high. The differences between two framework versions could thus be extracted from such a system and stored in a changeset document. The framework developers would then only need to fill out the gaps resulting from incomplete refactoring detection or provide hints to the adapter generation process for complex refactorings like *Add Parameter*. Consequently, a future task would be the integration of the adapter generation process with an automatic changeset generation process, compare [Wem06].

6.4 Validating Plugin Conformance

Many extension point types are *stateful*, that is, they define a communication protocol that can be represented by a finite state machine. Unfortunately CIL does not provide language features for ensuring a plugin's adherence to that communication protocol. Consider, for example, an extension point interface representing a connection to a framework resource. A connection can be in different states (i.e. open, closed, error, etc.). The connection interface might provide methods to change the connection's state and other methods to perform

tasks specific to a certain state. However, there is no possibility to ensure that a plugin only calls those methods available in the state the connection currently is in, other than directly implementing the checking logic inside these methods. From a software designer's point of view this is a dissatisfactory solution, because it does not provide a separation of concerns: the validation code is cluttered with the actual connection logic.

The class and interface adapters' role as a communication tunnel between the framework and its plugins can be taken advantage of in order to remedy this situation. By providing the adapter assembly generator with a protocol description in lieu of or in addition to a change specification, the produced adapters can implement the checking logic for the extension point. That way, the two concerns are properly separated and the extension point only contains actual business logic, thereby increasing source code legibility and simplifying development. However, in order to accomplish this functionality in the generated adapters, a powerful protocol description language is needed and the generator prototype would need to be extended to be able to process information provided in such a language. A language already available for this task is the Object Constraint Language (OCL), defined in the Unified Modeling Language (UML) specification [OMG01]. It serves for defining constraints in UML models, that are not expressable graphically. A UML modeling tool used for designing the framework's extension point could then extract the constraint information given in OCL and pass them in to the adapter generation process. As this is a broad research field, this thesis can only sketch some of the opportunities arising from the plugin adaptation approach discussed.

Abbreviations

ABI	Application Binary Interface
AOP	Aspect-Oriented Programming
API	Application Programming Interface
CDL	Change Definition Language
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLR	Common Language Runtine
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
DTD	Document Type Definition
IDE	Integrated Development Environment
IDL	Interface Definition Language
IPC	Inter-Process Communication
OCL	Object Constraint Language
ORB	Object Request Broker
RPC	Remote Procedure Call
UML	Unified Modeling Language
XML	eXtensible Markup Language

Bibliography

[BCF+06] Scott Boag, Don Chamberlin, Mary F. Fernandéz, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. W3C Candidate Recommendation `http://www.w3.org/TR/xquery/`, June 2006.

[BK98] Nat Brown and Charlie Kindel. Distributed Component Object Model Protocol – DCOM/1.0. Microsoft, January 1998.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls, February 1984.

[BPSM+04] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition). Technical report, W3C, Cambridge, MA, February 2004.

[CD99] James Clark and Steve DeRose. XML Path Language (XPath) Version 1.0. W3C Recommendation, `http://www.w3.org/TR/xpath`, November 1999.

[Cla99] James Clark. XSL Transformations (XSLT) Version 1.0. W3C Recommendation `http://www.w3.org/TR/xslt`, November 1999.

[Com] Comarch. `http://www.comarch.com`.

[CVS] Concurrent Versions System. `http://cvs.nongnu.org/`.

[DJ05] Danny Dig and Ralph Johnson. The Role of Refactorings in API Evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[ECM02] ECMA. Standard ECMA-335: Common Language Infrastructure. ECMA International, December 2002. `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[Fal04] David C. Fallside. XML Schema Part 0: Primer Second Edition. W3C Recommendation `http://www.w3.org/TR/xmlschema-0/`, October 2004.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, June 1999.

[FECA04] Robert E. Filman, Tzilla Elrad, Siobhan Clarke, and Mehmet Aksit. *Aspect-Oriented Software Development*. Addison-Wesley Professional, October 2004.

[FGA04] Andreas Frei, Patrick Grawehr, and Gustavo Alonso. A dynamic AOP-engine for .NET. Technical Report 445, ETH Zürich, May 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gil] Thomas Gil. AspectDNG. `http://aspectdng.tigris.org/`.

[HD05] Johannes Henkel and Amer Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 274–283, New York, NY, USA, 2005. ACM Press.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

[Leh96] Meir Manny Lehman. Laws of software evolution revisited. In *EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, London, UK, 1996. Springer-Verlag.

[Lev99] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[MG00] Erik Meijer and John Gough. Technical Overview of the Common Language Runtime. Microsoft, 2000.

[OMG01] OMG. Unified Modeling Language, version 1.4.2. `http://www.omg.org/cgi-bin/doc?formal/05-04-01`, June 2001.

[Opd92] William Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Rie00] Dirk Riehle. *Framework Design, A Role Modeling Approach*. PhD thesis, ETH Zürich, 2000.

[Rob99] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana Champaign, 1999.

[Som01] Ian Sommerville. *Software Engineering*. Addison Wesley Longman Publishing Co., Inc., Boston, MA, USA, 6th edition, 2001.

[SRB06] Ilie Savga, Michael Rudolf, and Andreas Bartho. A Refactoring-centered Approach for API Binary Compatibility. Submitted to the IASTED Conference, 2006.

[Tan01] Andrew Stuart Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2 edition, 2001.

[Ver] Hamilton Verissimo. AspectSharp. `http://www.castleproject.org/index.php/AspectSharp`.

[Vin97] Steve Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments, February 1997.

[Wem06] Ulf Wemmie. Using Version Control System for Tracking Adaptation-Relevant Software Changes. Bachelor Thesis, 2006.

Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, September 27, 2006