Großer Beleg

Towards the Adaptation of Generic Types and an Adaptation Aware Developer Guide

submitted by

Jan Lehmann

born 07.07.1982 in Görlitz

Technische Universität Dresden

Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

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

Submitted December 3, 2007

Contents

List of Figures

Chapter 1

Introduction

A software framework provides the basic architecture for a set of programs by providing the common functionality for this software [18]. Thus a framework eases the development of big and complex software. For a company frameworks can help to develop complete product lines of applications of the same domain. It is important to define a good API for the framework, allowing third party developers to extend the product's functionality by writing plug-ins and add-ons. An example for such a software product is the B2 ERP system currently developed by ComArch [1]. It will provide the possibility to be extended by third party plug-ins in order to fit the customer's needs.

At the same time, a framework needs to be maintained in order to

- meet new requirements
- respect changed requirements
- fix bugs and typographic errors
- correct design flaws
- extend functionality

These maintenance operations may change the framework API, too, and plug-ins may stop working with the new version of the API.

The manual adaptation of these changes is time consuming and error prone. Further difficulties are introduced when plug-ins are developed by third party developers: On the one hand, the framework developers cannot adapt the plug-ins themselves since they may not have access to the plug-in's source code. On the other hand, plug-in developers does not know the changes performed to the framework until they have familiarized with the new API.

In order to overcome these problems an adaptation approach was developed by the University of Technology Dresden in cooperation with ComArch. The result of this cooperation is a technology to generate an adapter layer giving the API designer more freedom in API changes by preserving the compatibility to existing clients. This approach is described in section 2.1.

One of the limitations of the adaptation approach is that it is not possible to adapt generic types and methods. Since generics are an important part of

an API due to their ability to enforce type safety, it is desirable to support adaptation of them.

Another problem is that there exist some changes that cannot be adapted or require more afford to be adapted. There are also design decisions that may invalidate the adaptation.

Two main contributions of this thesis are:

1. development of a generator for adapters for generic types and implementation of a prototype (GAG) (chapter 3)

2. creation of a guide helping developers to keep their frameworks adaptable (chapter 4)

Chapter 2

Background on change-based adaptation

In order to increase the freedom of framework developers in changing the API without breaking existing clients, an adaptation approach was developed. The project was called Plug-in Development Environment (PDE) and was funded by the SAB [7]. The following chapter gives an overview over the existing adapter techniques. Further information may be found in [21], [10] and [19].

2.1 The Plug-in Development Environment - An Overview

One of this project's intents was to extend the freedom of framework developers. The framework developer should be allowed to perform a wide range of changes to the framework as long as the functionality is not reduced. Since API changes will break existing clients in most cases, these changes cannot be performed without additional efforts. To offer a framework developer the possibility to change existing parts of the API, an adaptation approach has been chosen.

An adapter's intend is to "Convert the interface of a class into another interface clients expect" [14]. Following this intend the adapter implements the old API known by the client and redirects each call to this API to the new API of the framework. A schema is shown in Figure 2.1. The left side of the Figure shows the framework in version 1. A plug-in written for this version does not need additional adaptation. When the framework evolves to version 2, the API changes, as shown on the right hand side of the figure. Now the plug-in, which does not know the new API, is not able to work with the framework anymore. Therefor a description of the changes between the two versions of the framework is used to generate an adapter. This adapter allows the plug-in to work with the new framework.

The existing prototypes are implemented in C♯ [13] using the Microsoft .Net framework [4]. The .Net platform has been chosen because it is used by ComArch to develop the B2 system. Using .Net also enabled us to use the provided features like versioning and code signing.

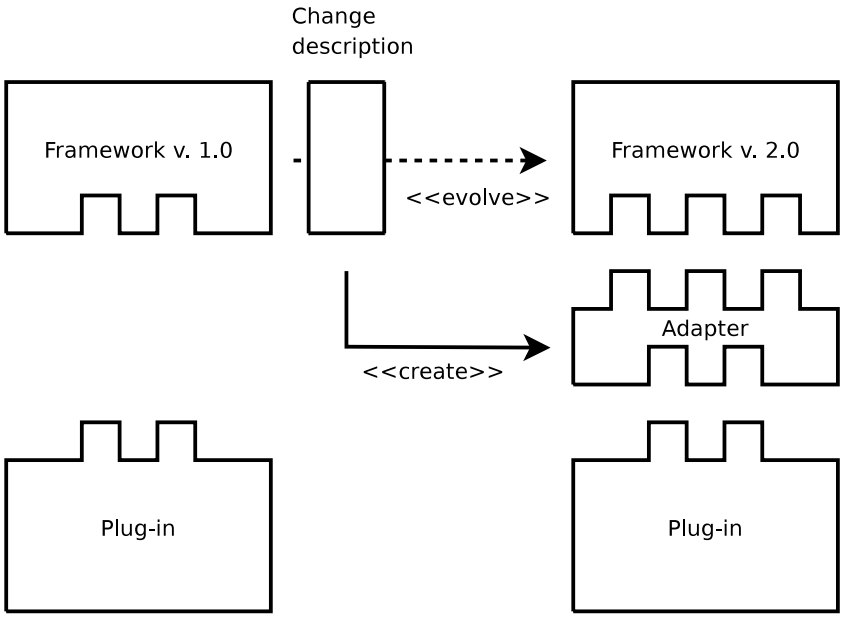


Figure 2.1: Adapter schema

2.2 The Plug-In Development Environment - In Detail

The adapter layer completely replaces the old framework API and co-exists with the new framework. Each API type is replaced by an adapter class, which keeps a reference to a new framework type, the adaptee, in a private field. In order to make the adaptee accessible, the adapter introduces a readable property and an additional constructor as depicted in Figure 2.2. Besides these additional members, each method, property and constructor of the old type are implemented, too.

When a client calls a method, the adapter not just calls the appropriate

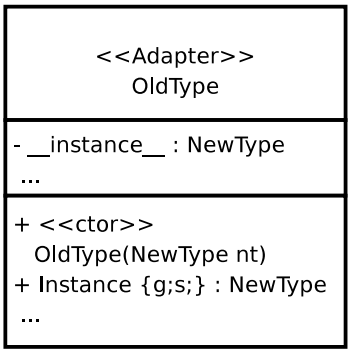


Figure 2.2: Schema of an adapter type

method of the adaptee, but performs the following steps:

1. Check each argument, if it is an adapter type.
2. If an argument is an adapter, fetch the adaptee with help of the additional property and replace the adapter by the adaptee.
3. Optionally reorder the arguments, leave arguments out or provide default argument values.
4. Call the adaptee's method.
5. Check if the returned value is a custom type and create a new adapter for this type using the adapter type's additional constructor.
6. Return the result.

Retrieving an adapter's adaptee or creating a new adapter are called unwrapping and wrapping [20]. These actions are necessary for the following reason. Assume there is an adapter A having a method m with an argument B. If m is called and B is a type provided by the .Net framework or a third party library, it can be passed directly to method m' of A's adaptee. However, if B is an adapter itself, the adaptee of A does not know about this type and m' would not accept the argument due to a type mismatch. Thus B has to be unwrapped in order to pass B' (the adaptee of adapter B) to m'. m' may return a framework type C' which is only known by the new framework version and thus has to be wrapped by the according adapter C which is eventually returned by m.

Having different versions of the API installed at the same time requires to keep the API assemblies within the .Net Global Assembly Cache (GAC). The GAC is a special folder maintaining different version of the same assembly, which is not possible otherwise, since both assemblies would have the same (file-)name and thus cannot be kept together within a normal folder. The .Net typeloading mechanism takes care of loading the correct version of a required type. If the adapter assemblies get another version number than the old framework assemblies (e.g. version 1.5.0.1 instead of 1.5.0.0), the type loader needs to be instructed to use the new version (the adapter) even if the old version is still installed. This can be done by installing a configuration file like shown in Figure 2.3.

The PDE project resulted in two different approaches. The main difference is of the supported changes:

(1) ComeBack! [2], the approach developed an the University of Technology Dresden, limits itself to refactorings (structure changing but semantic preserving changes) because they have clearly defined semantics [15]. The information about the refactorings is available within the used IDE (like Eclipse [3]) or can be semi-automatically extracted [12]. The advantage of ComeBack!'s approach is the fully automatic adaptation which requires not manual intervention. However only about 80% of the possible changes are supported.

(2) ADE [21], the Adapter Development Environmen developed at ComArch, introduces a custom, very simple language called ADL (Adapter Definition Language) to define the adapter itself. Therefor the adapter developer

```
<?xml version="1.0">
<configuration>
    <runtime>
        <assemblyBinding
            xmlns="urn:schemas-microsoft-com:asm.v1">
            <dependentAssembly>
                <assemblyIdentity name="AssemblyName"
                    publicKeyToken="..."
                    culture="neutral" />
                <bindingRedirect
                    oldVersion="1.5.0.0"
                    newVersion="1.5.0.1" />
            </dependentAssembly>
        </assemblyBinding>
    </runtime>
</configuration>
```

Figure 2.3: XML configuration file for assembly version redirection

needs to manually declare the steps to be performed by the adapter. The adapter definition process is supported by a Visual Studio plug-in which provides drag-and-drop capabilities and simple heuristics to semi-automatically define the adapter. Although the ADL is limited to method calls and assignments, it is possible to adapt virtually every possible type of change. However it requires manual coding of the adapter.

Both of the aforementioned approaches do not support the adaptation of generic types. Thus a tool supporting their adaptation was developed during this work and is described in the following chapter.

Chapter 3

Generic adaptation

Both Comeback! and ADE do not support the adaptation of generic types. However the support of generic types is important since the use of generics allows a developer to parametrize types with other types which leads to a much better type safety during framework instantiation. Thus, an active usage of generics by B2 developers is envisaged.

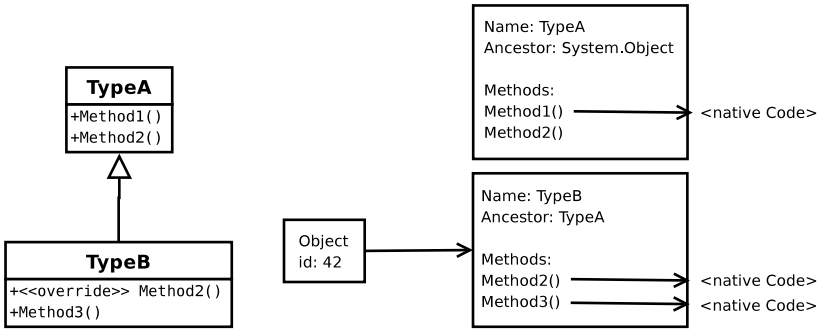
3.1 The .Net run-time system

As opposed to Java, generics in .Net are run-time features. The Common Language Runtime (CLR) keeps track about the generic information

The CLR maintains a meta-object for each type used by a program. This meta-object has the predefined type `System.Type` and may be used for introspection to get type members, type attributes or generic parameters. Static members are also located in the meta-object as well as pointers to the native code of members defined or overridden by this type, if they are already compiled by the Just-in-time-compiler. Members defined in a type's baseclass but not overriden by the type itself are maintained by the baseclass' meta-object. An example is shown in Figure 3.1. Figure 3.1(a) shows a simple type hierarchy where `TypeB` inherits from `TypeA` and overrides `Method2`. Figure 3.1(b) shows the situation if an object of `TypeB` was created. The meta-object for this type points to the native code of methods `Method2` and `Method3` whereas the native code for `Method1` is maintained by `TypeA`'s meta-object, since this method is defined in this type and not overridden by `TypeB`. A programmer may get a reference to a type's meta-object by using the `typeof()` operator or by calling and object's `GetType()` method.

Besides the information about types, the .Net Library provides meta-classes for type members. These meta-classes are derived from class `MemberInfo` and are called:

- `MethodInfo`
- `ConstructorInfo`
- `PropertyInfo`



(a) Example type hierarchy

(b) Reference of meta-objects

Figure 3.1: Example of meta-object handling

These types are needed in order to perform dynamic invocations when using reflection but also when generating code with help of the mechanisms provided in namespace `System.Reflection.Emit`. In order to generate code that calls a method or a constructor one needs the corresponding `MethodInfo` or `ConstructorInfo`. Properties are invoked by calling the respective getter or setter method depending on the action performed on the property.

3.2 Generics in .Net

Java bytecode does not contain information about genericity. The Java compiler erases the generic information during compilation and all occurrences of the type parameter are replaced by type `Object`. In addition, the return types of methods are automatically enriched with casts to the type the generic type was instantiated with. The .Net intermediate language preserves generic information. The JIT-compiler is responsible for generating specialized code for each instantiation of a generic type. In difference to C++, where templates are completely expanded at compile time, what means that for each instantiation of a template an own type is created in object code, the .Net JIT-compiler tries to share as much code as possible between type instances. The .Net typeloader checks whether a particular instantiation of a generic type is compatible with an existing one. The criteria for this is the memory representation of the given type parameter. Reference types are always compatible with each other since their memory representation is always a pointer which can be handled uniformly. If the types are incompatible, which holds for most value types, the JIT-compiler will generate a specialized version of the generic type. This allows the developer to instantiate a generic type with primitive types like `int` or `float`, which is not possible in Java.

A generic type may have two states: open and constructed. To be in open state means that the type parameters are not specified yet. It is possible to get the type information of an open generic type with the operator `typeof` and the name of the generic type without the type parameter. The open type of a generic list e.g. can be retrieved by `typeof(System.Collections.Generic.List<>)`. One can also use this type normally via reflection mechanisms but it is not

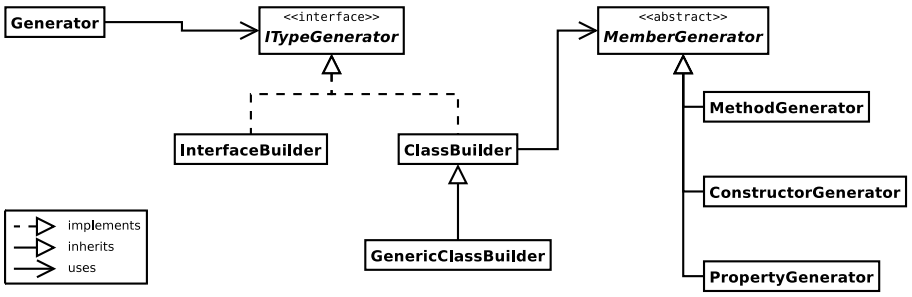


Figure 3.2: Structure of `GenericAdapterGenerator`

possible to define a variable having an open type. Variables can only be of constructed generic types, which are types with all type parameters specified. There is no possibility to create 'half-open' types by specifying only some of the type parameters. Although it is possible to get information about open generic types with help of reflection, these information may be incomplete.

Further information about the typesystem in .Net and about the CLR in general can be found in [17].

3.3 Generator for Adapters for Generic Types (GAG)

The intention of the developed *GAG* was to introduce the possibility of adapting generic types. Since there are two adaptation approaches, the adapter was held as generic as possible.

GAG consists of three main parts. The first part, steering the whole adapter generation process, is the `GenericAdapterGenerator`. The generator is responsible for creating the assembly and all types in that assembly. The retrieval of type information for types that are already generated by the adapter is not possible with the .Net type resolving mechanism, since the adapter's assembly does not exist at this point of time. However, such retrieval is needed to resolve references between adapter types or calls to members of different adapter types. For this reason the generator maintains an own lookup service for types located in the actually generated assembly. The structure of this part is depicted in Figure 3.2.

The second part of the adapter generator are classes forming a type structure. The existing .Net meta-types for representing types and type members require already existing types and type members. That's why the existing .Net meta-types cannot be used to define the adapter before the generation and a new hierarchy, similar to the one of .Net, was introduced (figures 3.3 and 3.4). Those classes represent the types to generate including their members. Each class that has to be generated is represented by an instance of type `Class` and contains a set of instances for each possible type of class member, namely `Method`, `Property`, `Constructor` and `Field`. Interfaces are represented as instances of type `Interface`. However, interfaces to not contain constructors or fields. Instead of `Method` or `Property` objects, they contain `MethodDeclaration`

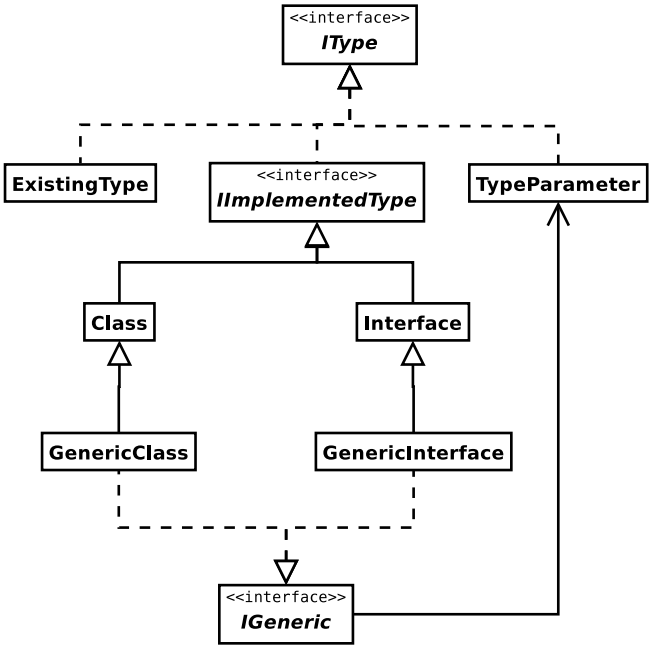


Figure 3.3: Adapter hierarchy for types

and `PropertyDeclaration` objects. Generic types are represented accordingly. The rather isolated type `Variable` is used to represent arguments and local variables within type members.

The third part of the adapter generator is the representation of the member's content. The content is defined by an object structure that is similar to an abstract syntax tree (Figure 3.5). This abstract representation is required in order to make *GAG* fit into both existing approaches, Comeback! as well as ADE.

With help of the introduced type hierarchy in connection with the content representation, one can define the adapter. The type hierarchy is used to define classes and interfaces with their respective members. That means that each class is represented by a `Class` object and each member of this class by a `Method`, `Property` or `Constructor` object, according to the member's type. For interfaces the same mechanism is used, however, interfaces do not contain constructors and the classes representing the members have the suffix `Declaration` since they do not offer the possibility for defining method bodies. The method bodies of class members are defined with the types shown in Figure 3.5. Each of these types is able to generate the intermediate language code necessary for the represented operation. The whole adapter generation is steered by the adapter generator types shown in Figure 3.2.

3.3.1 Workarounds

The adapter generator needs to use workarounds to cope with peculiarities of the .Net type system. The main problem is that the adapter generator cannot make

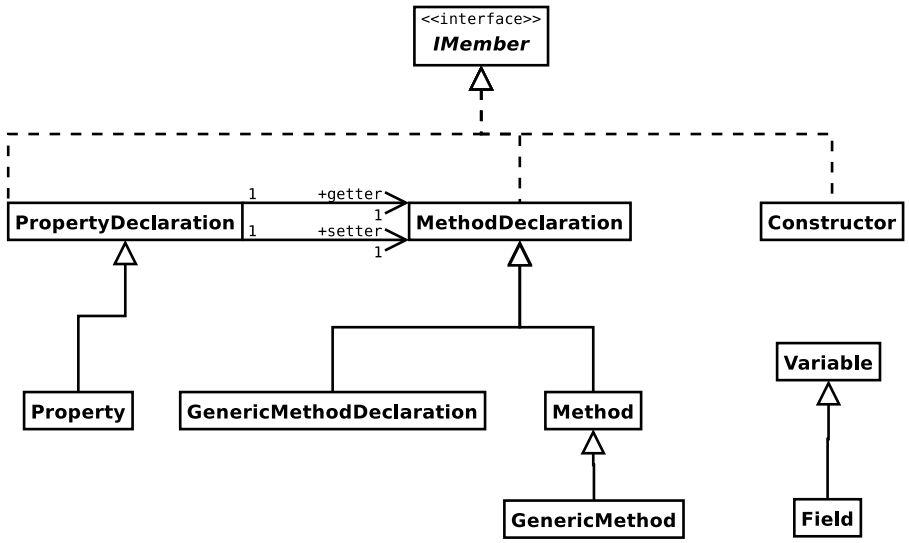


Figure 3.4: Adapter hierarchy for type members

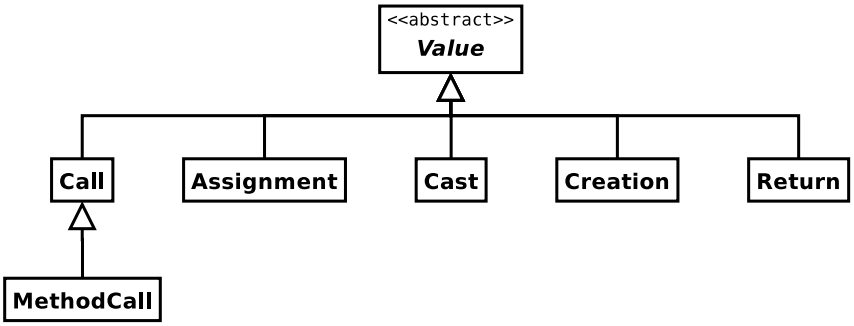


Figure 3.5: Type hierarchy for member content

```
List<> openList;
List<int> constructedList = new List<int>();

openList = constructedList;
```

Figure 3.6: Illegal assignment

```
PropertyInfo p1 = typeof(List<>).GetProperty("Count");
PropertyInfo p2 = typeof(List<int>).GetProperty("Count");
```

Figure 3.7: Different `PropertyInfo` objects

assumptions about the type the adapter will be instantiated with. That implies that the adapter itself has to perform work that should normally be performed by the generator, such as generating the correct method calls or instantiating the correct adapted type.

Adaptee type

For a non-generic adapter the type of the adaptee is known at the time of adapter generation. This is not the case for a generic adapter, for which the only information known is the adaptee's open generic type. However the open type is not sufficient since it lacks all information about the type parameter. That makes the open type incompatible with the constructed type. It also prevents assignments as depicted in Figure 3.6. Due to this incompatibility the reference to the generic adaptee has to be of type `Object` to be able to adapt any possible incarnation of the generic type.

Member invocation

When a member of the adapter is invoked, the adapter calls the appropriate member of the adaptee. In general, this call is done directly by emitting the respective command in Intermediate Language. In order to generate the correct method call, one has to have information which method to call. This includes the exact type of the class or interface defining the method and the corresponding `MemberInfo` object describing the method. However, *GAG* does not know the adaptee's specific type and thus cannot get the retrieve information about the correct member to call since this requires to know the adaptee's constructed type.

The situation is depicted in Figure 3.7. Although both instances ot type `PropertyInfo` describe the property `Count` as defined in the .Net framework class `System.Collections.Generic.List<T>` they to not represent the same property. The reason is the following: `p1` was retrieved from the open type of the class as it would be possible in the adapter generator whereas `p2` was retrieved from a concrete instance of the generic list as it appears to be at runtime. It is even not possible to invoke the getter of `p1` on an instance of type `List<int>`.

This behavior makes it impossible to generate calls to members other than the ones defined in any non-generic basetype. In consequence any member

```
public static T DynamicCast<T> (Object o) {
    return (T)o;
}
```

Figure 3.8: Dynamic casting method

defined in a generic basetype or interface of the adaptee or the adaptee itself has to be called by dynamic invocation using .Net's reflection mechanism. At first the adapter has to get the correct type of the member to invoke. This is done by calling the respective `GetX()` method, where `X` is either `Method`, `Property` or `Constructor`. A disadvantage of this approach is the negative impact on performance as it can be seen in [16].

Casting

When invoking a method via reflection the return type is given as instance of type `Object`. If this return type should be adapted, too, the adapter can use a hypothetical helper method to perform the instantiation of this adapter. However, if this method can be overloaded or is otherwise dependent of the given object's type, the adapter will have to perform a cast. Since the return type of the invoked method may depend on a type parameter, it is not guaranteed that the adapter generator knows about the type to cast the object to. That is why the casting has to be performed at runtime.

The cast operator provided by the .Net Intermediate Language requires the target type of the cast to be specified at compile time. There is no mechanism in Intermediate Language that allows the specification of the cast's target type at runtime. Thus a helper method similar to the one depicted in Figure 3.8 is used in order to perform this task. With help of reflection the type parameter for this method can be defined at runtime.

3.4 Changes in generic types

Genericity defines a contract between the framework or API providing a generic type and a client using this type. The framework requires the client to specify the exact type it wants to use, while the client can expect that this type is used instead of some placeholder, which is the type parameter in this case. There are different ways of specifying a contract for a type. Section 3.4.1 describes the problems with contracts in common. Section 3.4.2 matches this explicitly to generics and tries to examine solutions for special changes that may be applied to a generic type.

3.4.1 Contracts

By publishing interfaces the framework developer offers a contract the clients have to fulfill. On the other hand, the framework is also required to fulfill this contract. In this section contracts are described on the most general example of non-generic methods.

```
public void Sort(List<Object> subject) {
    /* perform sorting */
}
```

Figure 3.9: Framework method requiring a specific class

```
public void Sort(IEnumerable<Object> subject) {
    /* perform sorting */
}
```

Figure 3.10: Framework method requiring an interface

One can distinguish between changing a method's precondition and changing a method's postcondition. Each condition may become weaker or stronger, which has different impact depending on the type of condition.

Preconditions

Preconditions are the requirements the client has to meet in order to conform the framework's contract. Only if the precondition becomes weaker, the adapter may simply pass the argument (after wrapping, if needed) given by the client. Due to the polymorphic nature of object-oriented languages the type conforming to the more restrictive requirements also fulfills the weaker contract. An example is presented in Figures 3.9 and 3.10. In Figure 3.9 the framework required a specific type to be provided by the client. A refactoring weakened this condition so that the method now only requires a type implementing `IEnumerable`. Since `List` implements this interface, an adapter could simply pass the old argument to the new method.

Making a precondition stronger breaks a client, since the framework requires the client to provide more than the latter does. If the client provided an argument of a type that is compatible with the new precondition, too, the client will continue to work. For example the method in Figure 3.10 was changed into the method in Figure 3.9. Each client that already passed a list instance is still compatible with the new version of the method. Each client that provided any other type implementing `IEnumerable<T>` will break.

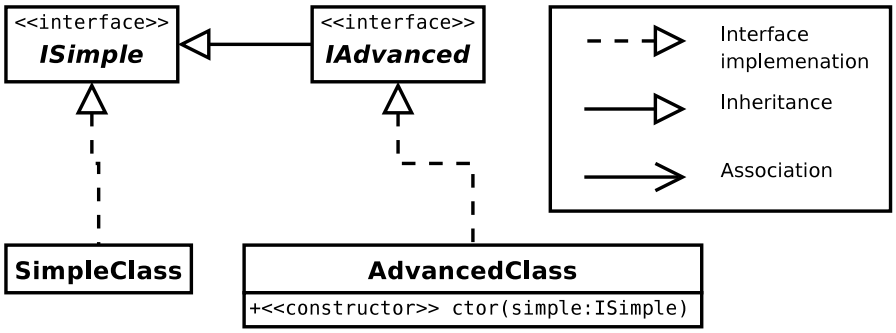
Another possibility is to convert the inappropriate types into types that are accepted by the framework. An example is shown in Figure 3.11 where method `SomeMethod` in the new framework now requires an argument of type `IAdvanced` (like the adaptee in Figure 3.11(b)) instead of `ISimple` (as the adapter in Figure 3.11(b)). If the adapter should be able to call the new version of `SomeMethod`, it needs the possibility to convert the type implementing `ISimple` into a type implementing `IAdvanced`. Therefor at least one type implementing the latter interface needs to be prepared. The type `AdvancedClass` in Figure 3.11(a) implements interface `IAdvanced` and therefor implicitly `ISimple`. The type also provides a constructor taking a type implementing `ISimple`. The adapter can use this constructor in order to convert types implementing only the simple interface into `AdvancedClass` and is therefor possible to call `SomeMethod` on the adaptee. Disadvantageous on this approach is that the instance passed to

the adapter gets copied and is not used directly. If the argument gets changed by the method these changes will not show up in the client. The adapter can prevent this by copying back all properties from the newly constructed type to the given argument for the price of a decreased performance.

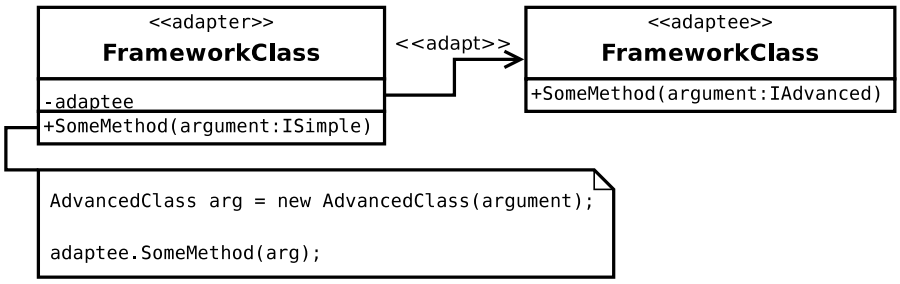
Postconditions

Postconditions are the parts of the contract the framework guarantees to comply to. They may also become either weaker or stronger. If they become stronger, the case is similar to weaker preconditions – the old contract is still fulfilled. If postconditions become weaker, we have similar problems to stronger preconditions, as the client expects more from the framework than the framework is now able to deliver.

When the postconditions are weakened, the result of the weaker framework method may be wrapped into a type compatible to the one the client expects, too. The only thing the developer has to make sure is that the framework does not keep an instance of the returned value in order to track changes done to values of this type. Converting the return type would invalidate the reference inside the framework since the client does not use the same object to perform operations.



(a) Hierarchy of convertible types



(b) Adapter using the conversion capabilities

Figure 3.11: Example of using convertible types

3.4.2 Generics

A well know example for the use of generics are collections as they can be found in package `java.lang.*` or the namespace `System.Collections.Generic.*` in Java and .Net respectively. Since type parameters are mostly used as arguments and/or return types of methods, changing the requirements of type parameters will also change the pre- and postconditions of the methods contained in the generic type.

Introduction of generics

The introduction of (unbounded) generics into the framework is relatively easy to adapt. Since it is known which parts of a type or method became generic, one may decide already at adapter generation time how to instantiate the type parameters.

When genericity is introduced to a type, the most obvious way is to use `Object` as type parameter to instantiate this type inside the corresponding adapter. Using `Object` should be sufficient because its a base type for each type in the .Net type system. Even value types may be used in this case since they will be automatically boxed, that means they will be converted into a reference type automatically.

When introducing genericity to a method, it should be possible to infer the types to instantiate the framework's method with. It is known which argument of the old method, offered by the adapter, matches to which argument in the new method, called by the adapter. Thus one knows which type to use for each type parameter. Figure 3.12 shows the introduction of genericity to a method called `SomeMethod`. The adapter may instantiate the adaptee's generic method with the types used by the method provided in the adapter. That means that `TReturn` is instantiated with type `Object` and `TArg` is instantiated by `String`. If a type parameter occurs more than once and the occurrences have different argument types, the least specialised basetype may be suitable as type parameter. Since these information are available at the time of adapter generation these actions can be performed by the adapter generator.

However, sometimes it may be needed to instantiate the method to call dynamically, for example if some argument was replaced by a generic parameter. This may be the case for factory methods like `Object Create(Type t)` if they are changed to `T Create<T>()`. In this case the argument given to the old method has to be evaluated by the adapter and the new method has to be instantiated accordingly at runtime.

Removal of generics

Although turning a generic type into a non generic one may be rare, it is presented here for completeness. Removing generic parameters drastically reduces the flexibility of types and methods. That implies that the new, now non-generic type has to be general enough to cope with all situations where the old, generic type was used. It also requires from the adapter to perform work, formerly done by the framework (especially type conversions and casting of return types needed).

Generally removal of generic arguments should be considered as change altering the semantics of a type (or method) and thus avoided.

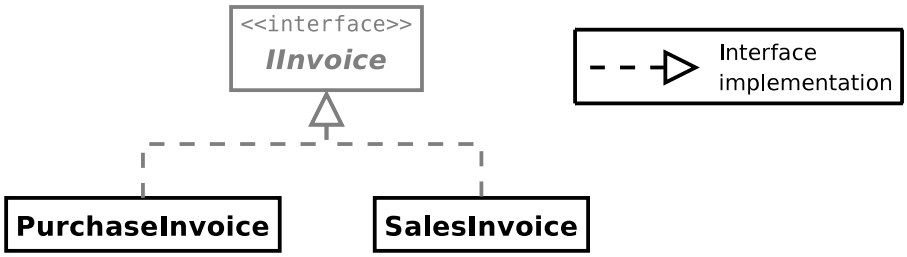
```
public Object SomeMethod(String arg0) {
    // perform work
}
```

(a) Old, non-generic method

```
public TReturn SomeMethod<TReturn, TArg> (TArg arg0) {
    // perform work
}
```

(b) Generic method

Figure 3.12: Introduction of genericity to a method



(a) Simple invoice inheritance structure

```
public T FactoryMethod<T>() where T : IInvoice {
    // object instantiation code
}
```

(b) Generic factory method

Figure 3.13: Usage of bounded polymorphism

Bounded polymorphism

Bounded polymorphism has to be used, if special features of types are to be used inside the generic type. With bounded polymorphism the framework developer can restrict the type parameters. These restrictions may be interfaces a given type has to implement or base classes it has to be derived from.

Since type restrictions may be considered as preconditions, weakening them does not affect the client. Moreover, this remains true for the return types since the client specifies the type it expects to be returned. Similar to ordinary preconditions, requiring a more specialized type as minimal type parameter will certainly break the client since it might not fulfill the new requirement.

Introducing bounded polymorphism is a special case of requiring a more specialized type as type parameter. If the type parameter given by the client is not compatible with the type restriction, the adapter cannot instantiate the adaptee. If the given type parameter is compatible with the type restriction the adaptee can be instantiated using this type. An example for introducing bounded polymorphism is the introduction of a common interface of some types. Figure 3.13(a) shows the introduction of interface `IInvoice` to the type hierarchy. The black parts show the situation before the introduction. Both classes where implicitly derived from `Object` having no common interface. The new

version introduces the grey parts, namely the interface and the implementation of the interface in both invoice classes. Figure 3.13(b) shows the corresponding generic factory method. The first version would look the same but without the type restriction shown in the figure. This method is responsible to create an instance of the type given as type parameter. Although the interface `IInvoice` is not known to the client, the adapter may use the respective types of the new framework version in order to instantiate the factory method.

However, the possibility to adapt generic types or methods, where bounded polymorphism has been introduced, strongly depends on the intended usage of the generic type. The more generic the type was before the introduction of the type restrictions, the less possible it might be to adapt the changes. Thus introducing bounded polymorphism should be avoided by framework developers to not invalidate the adaptation approach.

Chapter 4

Guidelines

Any adaptation technique is limited in the supported changes. Although the adaptation approach can seriously extend the freedom a framework developer has in applying changes to the API, the developer still needs some discipline when designing the API. The following chapter provides a guide for framework developers to ease the task of adapting framework changes.

The guide is twofold. Section 4.1 describes points to keep in mind during the development of a framework. Some of them may sound trivial since they are discussed as "good style design" in most books on object-oriented design. However, they are crucial for the adaptation so they are listed here again for completeness. Section 4.2 describes the limitations of changes the developer should take into account not to invalidate our adaptation approach.

4.1 Design restrictions

In general software design is done from a very high level abstraction to the final model, resulting in source code. Usually the UML is used in order to specify the models, which are stepwise refined until code may be generated. Ideally the whole public interface of an API should be defined in UML. The programmer should only need to fill in the empty method bodies. Modern programming languages provide a wide range of features which make some aspects of programming more elegant. These aspects are namely the usage of exceptions, anonymous classes (in Java), anonymous methods (C♯), delegates (C♯) or events (C♯) which may lead to more clarity and robustness but may also prevent the API from being adapted since they introduce code automatically generated by the compiler and thus difficult to reimplement (like delegates and events), or they break the normal control flow of the program (like exceptions).

The following section will describe some of the cases where adaptation may fail or will be complicated.

4.1.1 Public fields

Information hiding and encapsulation are principles commonly used and accepted in object-oriented programming. However, object oriented languages like Java or C♯ do not force a developer to encapsulate fields. A programmer

may be tempted to omit getter and setter methods in order to keep the type's interface simple.

This approach has some severe shortcomings for adaptation since it prevents the adapter from adapting calls to this field. If the field's type `F` changes, the adapter layer has to redirect calls to the old version of `F` to the new version of `F`. This also affects the type `T` containing the field. There must be an adapter for `T` that accepts the old version of `F` from the client and calls `T`'s new version with the new version of `F`. With the field being public this call forwarding is not possible. Any client expecting the field being of old type `F` will break. The main reason is the Adapter design pattern that is used. This pattern only supports adaptation of method calls but does not work at the level of fields.

The solution is obvious: a developer should follow the principle of encapsulation and provide getter and setter methods (or readable and writable properties for .Net) for each API field. An adapter will then be able to perform the adaptation by forwarding to these accessor methods and old clients will not break.

4.1.2 Constant values

Beside fields, constants are often publicly exposed with by fields declared as `public static` and `readonly` or an equivalent modifier. Such constant declarations can be found at several places in the Java API, for example in class `BorderLayout`. That uses fields declared as `public static final String` to determine the position of child widgets.

The problems occurring in this approach are inherently the same as described in section 4.1.1. In addition to the type (Sun could decide to use `int` values instead of strings to specify the layout constraints), also the value or name of the constants could change. For example instead of using `NORTH`, `EAST`, `SOUTH` and `WEST` values like `TOP`, `RIGHT`, `BOTTOM` and `LEFT` could be used.

One solution here is also to encapsulate constants into getter methods or readable properties simply returning the constant value as shown in Figure 4.1. This works for arbitrary constants. However constants are often used to provide flags or named constants for numeric values. The aforementioned Java `BorderLayout` example may be refactored to use enumeration types instead of constants as depicted in Figure 4.2. Enumeration types provide the possibility of adaptation since it is possible to generate an adapter for the enumeration type itself. The usage of enumeration also introduces a better type safety since a developer is not tempted to pass an arbitrary string where the enumeration is required.

```
// old constant:
public static readonly string CAPTION = "some caption";

// constant encapsulated within a property
public string CAPTION {
    get { return "some caption"; }
}
```

Figure 4.1: Encapsulation of a constant value.

```
public class BorderLayout
    implements LayoutManager2, Serializable {

    public enum Orientation {
        CENTER,
        NORTH,
        SOUTH,
        EAST,
        WEST
    }

    /* other implementations */
}
```

Figure 4.2: Definition of orientation with an enumeration.

4.1.3 Member names

Besides reproducing the complete interface of its adaptee type an adapter adds some additional members. At least a constructor to create the adapter from a given instance of the adaptee and a property to access the adaptee's instance are introduced.

Since the adapter is introduced into an existing API it is possible that the name of some members of the adapter's interface interfere with members of the replaced type's interface. The result depends on the decisions made during the adapter generation process. The first possibility is that the generated adapter is invalid due to the introduced members not being generated. Second, it could be possible that the adapter hides members of the replaced type by its own members. The third - and mostly the best, since most alarming - case is an exception at generation time.

The responsibility to avoid this problem lies on both sides, the API developers and the adapter generator's developers. The members introduced by the adapter have to be explicitly documented. Furthermore, the possibility of naming conflicts can be reduced by choosing unlikely names for the adapter members such as `__Instance__`. The API developers should read the adapter's documentation in order to know and avoid using such "reserved" names.

4.1.4 Reflection

In languages like C♯ or Java reflection is limited to introspection and is used mostly to:

1. gather information about the reflected types
2. create instances of reflected types
3. invoke members of these types

Information retrieval

The gathering of information about a class is often used prior to invoking members or creating types with reflection. With help of reflection it is possible to get information about existing classes in a .Net assembly or about members within a type.

Plug-In developers that do not like their plug-ins to be adapted to the new API may use reflection in order to find out if they are communicating with an original API class or an adapter. Since the adapter's layout differs from the original type's layout, it is easy to find out if adaptation is used, especially if the adapter's layout is well documented as proposed in section 4.1.3.

There is no single solution for preventing third party developers from using reflection. One possibility is to use obfuscation on the API. However, obfuscation can only make reflection more complicated and does not guarantee security [9].

Invoking members

Since the pure information retrieval is not that useful, the next step in reflection process is usually the invocation of found type members. .Net offers powerful reflection API's that can be used in order to find and invoke members as well as create types. Developers may become careless when using reflection, since methods like `GetMethods()` return the methods of a type "reliably" in the same order.

A developer using reflection may be tempted to rely on this feature, regardless of the warnings provided with the MSDN Library documentation. Figure 4.3 shows a situation where this might have unforeseen consequences in the context of adaptation. Class `FrameworkClass` (Figure 4.3(a)) contains three methods, one having a long parameter list. Since this method is an overloaded method one cannot retrieve it simply by its name but has to give the whole parameter list, resulting in much typing. Knowing that the aforementioned method `GetMethods()` delivers the whole list of methods in a fixed order one could use the index of the method inside the array in order to get the desired `MethodInfo` object (see Figure 4.3(c)). After this the class invokes the method. This example works as long as the .Net method is not changed or the plug-in does not work with an adapter.

However, as described earlier, the adapter changes the layout of the class like in Figure 4.3(b). Due to the layout changes, `GetMethods()` may return the methods in a different order than before. The plug-in will still get a valid `MethodInfo` object, but the invocation will fail with a runtime error due to wrong parameter types or, even worse, will abusively invoke another method. Assuming `GetMethods` returns the methods in the same order as defined in the class, the plug-in would now get the method taking only the `String` as argument instead of the long argument list. The invocation of this method would result in an exception since the argument list of the invoked method does not match the given number of arguments.

If a plug-in cannot be prevented from using reflection, as described in section 4.1.4, the importance of using a method like `Type.GetMethod(String name, Type[] parameters)` has to be emphasised. Such a method will always return the intended member, regardless of recompliations or adapter usage.

```
public class FrameworkClass {
    public void Method1() { /* ... */ }

    public void Method2(String arg1) { /* ... */ }

    public void Method2(String arg1,
        int arg2, /* ... */, Object arg10) {
        /* ... */
    }
}
```

(a) A framework class defining several methods

```
public class PlugInClass {
    public void ReadInformation() {
        Type fc = typeof(FrameworkClass);

        // always returns the same method
        MethodInfo mi = fc.GetMethods()[2];

        // invokes the method retrieved above
        mi.Invoke(/* ... */);
    }
}
```

(b) The adapter for `FrameworkClass` defining an additional method

```
public class FrameworkClass {
    // method returning the new version of the adapted
    // type
    public FrameworkClass GetDelegatee() { /* ... */ }

    public void Method1() { /* ... */ }

    public void Method2(String arg1) { /* ... */ }

    public void Method2(String arg1,
        int arg2, /* ... */, Object arg10) {
        /* ... */
    }
}
```

(c) The plug-in using reflection to invoke a member of `FrameworkClass`

Figure 4.3: Example of a dangerous usage of reflection

Instantiating types

Similar to the invocation of members, reflection can be used to create type instances using information about the constructor. Except for warnings in the MSDN library documentation, the responsible method `GetConstructors()` shows up the same behavior as `GetMethods()` and returns the constructors in a reliable order.

The problems, occurring when using reflection in the described way, are the same as described in section 4.1.4. Although it is unlikely to instantiate the wrong type, at least the wrong constructor may be invoked resulting in a runtime error.

As with the problems, the solution is similar to the one in the last section. If the plug-in developer needs information about the constructor to invoke, he should use `GetConstructor(Type[])` in order to declare the needed arguments. Another solution would be to use method `CreateInstance` in type `Activator`.

4.1.5 Serialization

Serialization is used mainly

a) when communicating via a network

b) for persisting objects

A plug-in can be responsible for synchronizing two distant instances of the framework (Figure 4.4). This synchronization requires sending over a network framework objects that need to be serialized. Another plug-in may store framework objects for archiving purposes. For some reasons the plug-in developer decides to use serialization to a file instead of storing the objects to a database.

Both scenarios have in common that they rely on the fact that objects stay the same. When using adaptation, this will not be true in any cases. In the synchronization scenario, one of the frameworks might have been upgraded to a new version, while the communication partner is still the old version. Assuming that the plug-in did not change, the plug-in for the first framework instance serializes the adapter while the plug-in in the second instance serializes the old framework types. Both kinds of objects are not compatible by default since their layout differ from each other. The plug-in working with the old version of the framework expects the old business entities possibly containing a large number of data while the plug-in working with the adapter serializes adapter types only containing few references to entities of the new version of the framework.

The same problems may occur in the second scenario with the archiving plug-in. Figure 4.5 shows a situation where the restore of an object previously stored by the plug-in may fail. As long as the framework stays the same, deserialization of stored objects will work. If the framework is now replaced by a newer version, the deserialization will fail, although the plug-in continues to work with the adapter that mimics the old API. Generally, objects serialized before the introduction of an adapter will fail to be deserialized after the adapter is introduced.

In order to overcome these drawbacks a framework developer should not rely on the standard serialization techniques provided by Java or .Net, but must always implement his own serialization mechanism. A class that has its own serialization mechanism in .Net looks like in Figure 4.6. The attribute `Serializable`

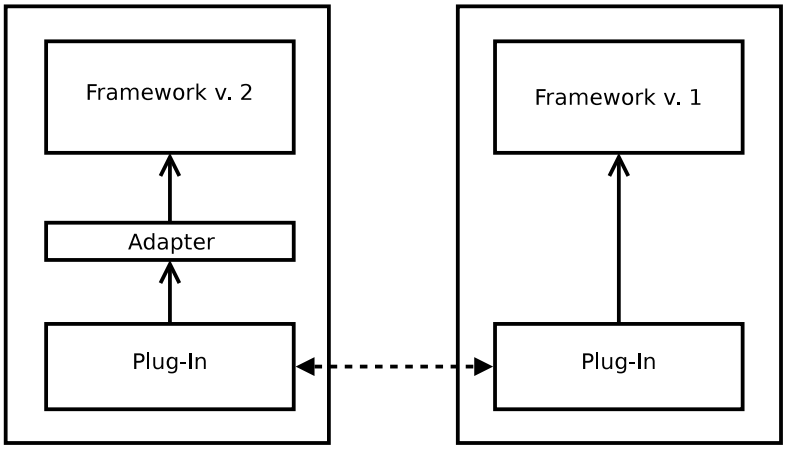


Figure 4.4: Plug-In communication scenario

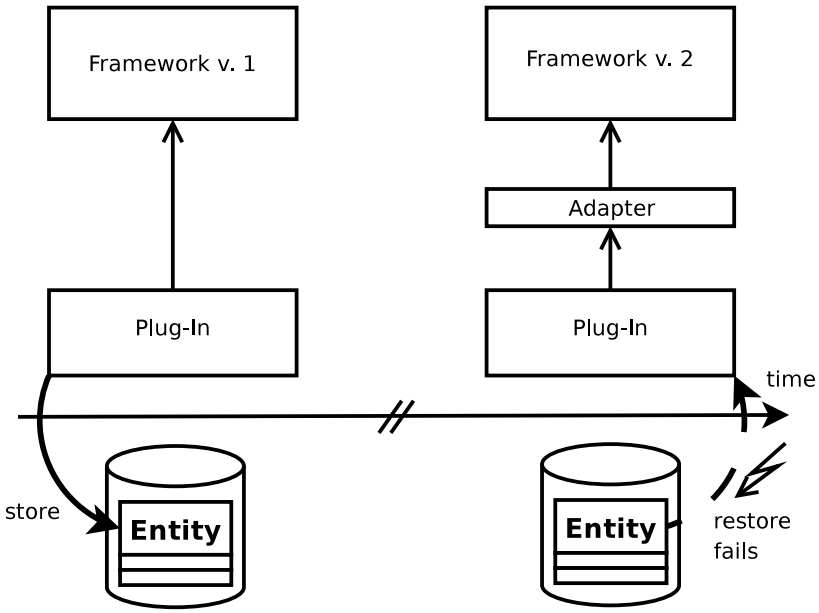


Figure 4.5: Storage of framework objects

```
[Serializable]
public class SerializingClass : ISerializable {
    private string _name;
    private int _id;

    public SerializingClass() { /* ... */ }

    // constructor to restore the serialized data
    public SerializingClass(SerializationInfo info,
        SerializationContext context) {
        _name = (string)info
            .GetValue("serialized_name", typeof(string));
        _id = (int)info
            .GetValue("serialized_id", typeof(int));
    }

    #region ISerializable members
    // serialize the fields to a datastructure
    public void GetObjectData(SerializationInfo info,
        SerializationContext context) {
        info.AddValue("serialized_name", _name);
        info.AddValue("serialized_id", _id);
    }
    #endregion

    // further methods and properties
}
```

Figure 4.6: .Net class having its own serialization mechanism

marks the class as serializable. The custom serialization is done by implementing the interface `ISerializable` which requires to implement method `GetObjectData`. The given `SerializationInfo` object will store each field to be serialized identified by a custom string. Deserialization of the class is done by a special constructor requiring the same arguments as method `GetObjectData`.

With help of these customized serialization the adapter could deserialize the old object, passing the values to the new, adapted one. The adapter can also serialize the adaptee in a way that is compatible with the old version of the framework, ensuring exchangeability of objects between different framework versions.

4.2 Change restrictions

The main goal of the adaptation approach was to enable the developer to apply changes to the API that would normally break any client using it. However, the adapters are not almighty so not every change is allowed to be performed, since they invalidate the adaptation approach. Sometimes those changes seem to be quite obvious to identify like semantic changes or the pure deletion of members,

other problems are not that obvious.

4.2.1 Deletion of members

Deletion of members from an API can happen in two ways, either by physically deleting it or by changing its access modifier (e.g. from `public` to `private`). An API-member in this case means type members like method, constructors or properties as well as types themselves. The deletion normally happens if a member becomes obsolete or if a functionality is not considered for public use anymore.

Since the functionality represented by the deleted API-member is not substituted by another member or a set of other members, clients, relying on the deleted operations or types, will stop working. Both existing approaches for generating the adapters are not able to cope with API-member deletion.

There are some possibilities how the adapter may compensate the deletion. These are neither implemented not tested yet because they have several shortcommings. In case of type members it may be possible to record the deletion and generate the adapter in a way that it does not call a new implementation but handles the request itself. However, the person generating the adapter has to guarantee that the "reimplemented" member does not have any side effects, especially that it does not change the type's state, since this may lead to inconsistencies. An example for side effects is shown in figures 4.7 and 4.8. In this example the API provides a type `Iterator`, which has two methods in the old version (Figure 4.7(a)). Method `GetCurrent` returns the current object in a collection and `GoToNext` proceeds to the next item. Thus a client will always get the same object from `GetCurrent` as long as `GoToNext` is not called, which lies in the responsibility of the client, too. The new version of the API introduces a more convenient behavior (Figure 4.7(b)). The `Iterator` now only contains method `GetCurrent`, which does both, returning the current object and proceeding to the next item in the collection. The adapter (Figure 4.8) now reimplements method `GoToNext` and this is where the unintended side effect occurs. If an existing client uses both methods together, the navigation will not go as intended, typically every second item will be skipped. This is surly not what was intended formerly.

The adapter can also provide a dummy implementation which simply does nothing. In the aforementioned example this may solve the problem with the missed items. However, this approach is also problematic, especially with non-void methods. Always returning some default value may not be sufficient as in the example. If `GoToNext`'s return value indicated if the switch to the new item was successfull, returning `true` would result in an infinite loop. On the other hand, the client would not proceed further then the first item, if the adapter always returns false.

As this simple example showed, there are several issues to be considered when trying to compensate the deletion of API-members. In the example the adapter could hide the deletion and even could reproduce the repeatable read behavior, but this would require more code then the original version of the type and requires manual work. In general, the API developer should avoid the pure removal of functionality or, at least, take additional precautions when providing default values for adapters.

```
public class Iterator {
    public object GetCurrent() {
        // returns the current item
    }

    public bool GoToNext() {
        // proceeds to the next item
    }
}
```

(a) Old version

```
public class Iterator {
    public object GetCurrent() {
        // returns the current item
        // and proceeds to the next
    }
}
```

(b) New version

Figure 4.7: Changed iteratorclass

4.2.2 Exceptions

Exceptions are special types to be returned from a method in error situations. That means if an unexpected situation appears in the framework, which cannot be handled directly inside the framework, the framework developer may throw an exception that should be caught by the client. The client can than decide how to react in the respective situation.

Since exceptions are normal types (usually derived from a special common subtype), it is possible that they change over the time. Changes that may be applied to exception types are the same as to other types, namely renaming, addition/deletion of members and so on. Special to exceptions is that they appear apart from the normal program flow. If an exceptions is thrown the normal program flow is aborted, at least, until the exception is caught, a behavior that is called forward exception strategy and is used in Java and .Net. That makes the adaptation of exceptions both important and difficult since both the exception types name and version may change, which leads to the situation that an altered exception will not be caught anymore.

Another way for changing exceptions is the kind and number of exceptions thrown by the framework, meaning reducing, increasing the number of thrown exceptions. Also exchanging two types of exceptions is possible. This causes further problems especially when using unchecked exceptions, which are exceptions that do not need to be declared and thus cannot be determined from a method's signature. The .Net runtime system uses unchecked exceptions exclusively. In Java the developer has to derive from `RuntimeException` if there should be no need to declare a custom exception in a method's signature. Otherwise it belongs to the method signature.

The following two sections will describe both of the aforementioned change types and present some workarounds and possible solutions for future versions of

```
// adapter
public class Iterator {
    // reference to the new iterator
    private Iterator _newInstance;

    public object GetCurrent() {
        return _newInstance.GetCurrent();
    }

    public bool GoToNext() {
        // reimplements old method and
        // proceeds to next item
    }
}
```

Figure 4.8: The erroneous adapter reproducing `GoToNext`-method

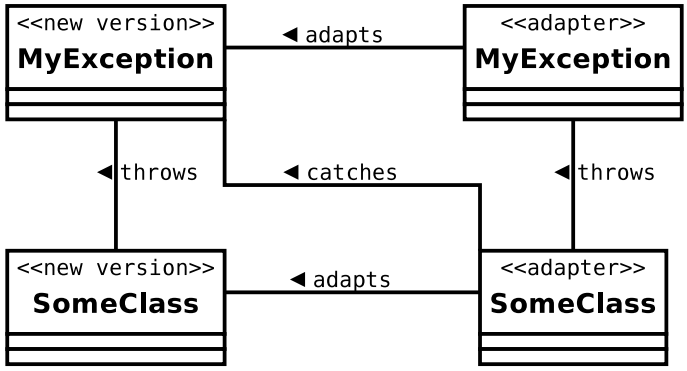


Figure 4.9: Adapting exceptions - the class structure

the adapter generator. The actual versions are not capable of handling changed exceptions.

Changing the exception type

Generating an adapter for an exception is just like generating an adapter for any other type. The difference to other types is that there is no defined point in the applications control flow where the adapter is instantiated. In particular in a black-box framework the client instantiates types by calling their constructor or by using some factory methods. Exceptions, however, are instantiated by the framework and the client can just catch them. Thus the API developer needs to generate an adapter for each type that throws the altered exception type.

Figure 4.9 shows a possible class structure for an exception adaptation scenario. The exception adapter is a normal adapter for the new version exception class. If one of the classes in the new version of the API throws the new version exception, this exception is caught by this class' adapter. The adapter than

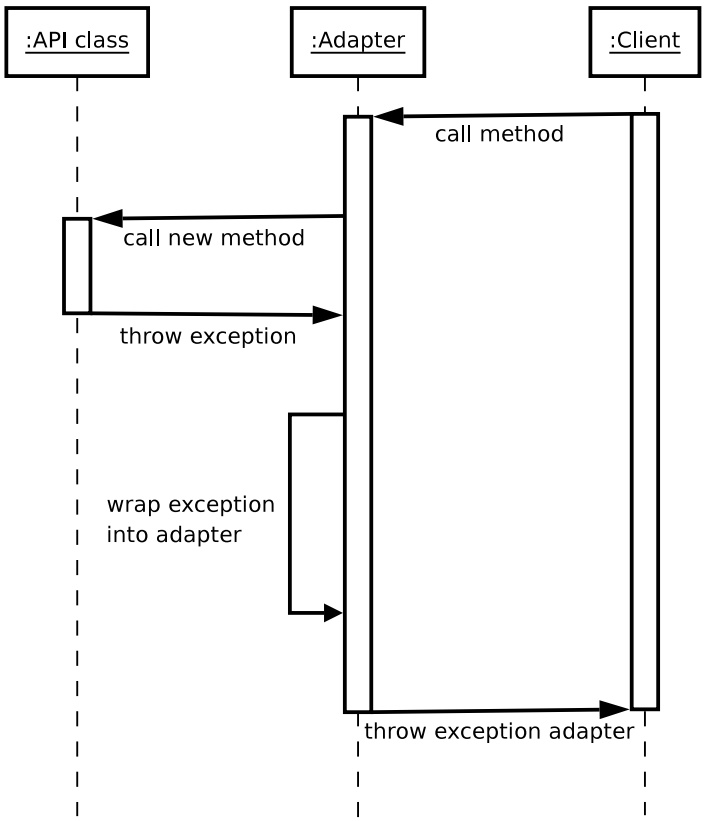


Figure 4.10: Adapting exceptions - control flow

wraps the new exception into the appropriate exception adapter and rethrows it. This workflow is presented in Figure 4.10.

The drawback of this scenario is the point where the API type adapter catches the new exception and rethrows the exception adapter. In Java, when using checked exceptions, one may generate a `catch` block for each exception that has to be adapted. Inside the `catch` block the right adapter for the caught exception is created. The knowledge of which exception adapter to create resides inside the adapter generator. This option is not available in .Net environments. The only way for an adapter to catch the desired exceptions is to catch all exceptions and than dynamically decide which exception adapter to create. This dynamic way of adapting exceptions implies the usage of some lookup mechanism in order to find the appropriate adapter for an exception. If no exception adapter was found, the exception handler would just rethrow the caught exception. The whole process is shown in Figure 4.11.

Changing the number of thrown exceptions

Changes in the exceptions thrown by a method is mostly relevant for systems with checked exceptions like Java since otherwise all exceptions would be caught.

The simplest scenario would be if one exception type is replaced by another.

```
// adapter method
public void SaveCustomer(Customer c) {
    try {
        // call the method that is adapted
        _adaptee.SaveCustomer(c);
    }
    catch (Exception e) {
        // check if we can adapt the exception
        if (AdaptationHelper.IsAdaptable(e)) {
            // create the adapter and throw it
            throw AdaptationHelper.Adapt(e);
        } else {
            // throw the exception that was caught
            rethrow;
        }
    }
}
```

Figure 4.11: Exception adaptation in .Net based adapters

This case would be the same as in the above example. The adapter would catch the new exception and rethrow the old one. Removing an exception from a method's signature will result in useless `catch` block at the client side but are otherwise harmless.

More difficult is the handling of added exceptions. The adapter designer has two options, ignoring the exception mimics the behavior of the adapter in the unchecked exception scenario. If the exception in question is thrown it will most likely break the client. The second option is to substitute the added exception by an exception already thrown by the old framework. However, it is difficult to choose the exception to throw instead of the new one. The wrong choice will result in inappropriate exception handling, having the same effect as an empty `catch` block – the application may misbehave due to the error but the user cannot determine why. Which way is chosen by the adapter designer depends on the type of the exception and the expected frequency of the exception.

4.2.3 Semantic Changes

After an API is published, the developer of this API usually gets feedback how the API proves itself under practical conditions. Sometimes flaws in the program flow will be found and the programmer will have to correct this misbehavior. These corrections do not necessarily change a type's or member's appearance but its behavior, that is, go beyond mere structural changes. However those changes are sometimes necessary and should be adapted up to a certain point.

One can mainly distinct between two kinds of semantic changes.

1. Transparent protocol changes: changes to the internal behavior of a member with respect to the actions performed during a client request. A good example is the `Iterator` class of Figure 4.7 in section 4.2.1. Whereas the old version required the client to proceed to the next item, the new version

```
// adapter
public class Iterator {
    // reference to the new iterator
    private Iterator _newInstance;

    // temporary store the current object
    private Object _current;

    public object GetCurrent() {
        if(_current == null) {
            _current = _newInstance.GetCurrent();
        }

        return _current;
    }

    public bool GoToNext() {
        _current = null;
    }
}
```

Figure 4.12: A protocol adapter

proceeds with the call to the method returning the actual item. Thus the client can repeatably read the current item in the old version of the type, whereas this is not possible in the new version.

The adapter is responsible for shielding the client from those protocol changes. In general it is easier for the adapter to reimplement additional responsibilities of the client since they are just additional method calls to be made by the adapter. In a case shown in Figure 4.7 it would require much more from the adapter since it has to implement quite complex logic. An adapter that implements the behavior of the `Iterator` in Figure 4.7(a) on top of the new version in Figure 4.7(b) may look like the one in Figure 4.12.

In case of protocol changes it is strongly recommended that tests for the old framework are run against the adapter in order to find out if the changes are really made transparent for the client. An API developer should also put in consideration that this protocol adaptation, although looking quite simple in the aforementioned example, puts an additional responsibility to the adapter. If now an error occurs, it maybe not only in the client or the framework, but also in the adapter since in addition to logic for converting types and calling members it contains also new business logic.

2. Intended for existing clients: changes that are not supposed to be transparent to the client, especially in cases where values changed that should be propagated to the client. An example is the raise of the VAT (value added tax) rate as done in Germany at the beginning of 2007 where the VAT rate raised from 16% to 19%. It is clearly wanted that the client is

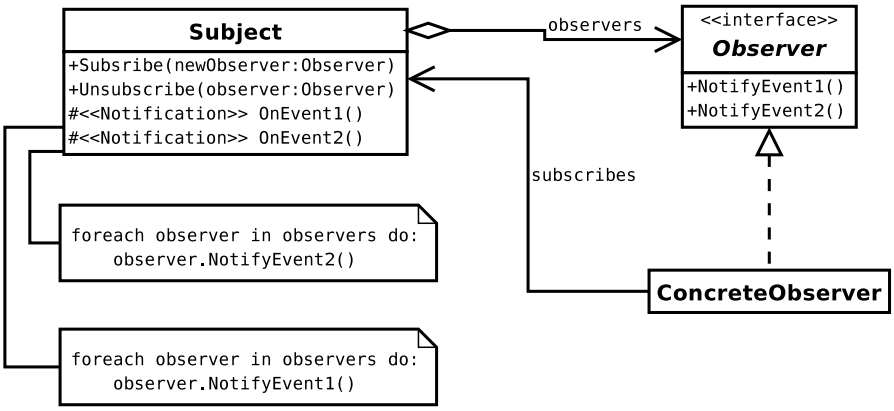


Figure 4.13: Structure of observer

aware of those changes since it should still work correctly.

In this case the adapter does not need to implement further adaptation logic, instead, it must be tested by tests written for the *new* version of the framework to check, whether it indeed offers the updated functionality.

4.2.4 Event handling

Event handling is used when components should be loosely coupled in order to keep the set of subscribers to an event extensible and the subscribers itself exchangeable. In general some variant of the observer pattern will be used in order to realize the event handling. The observer itself is defined by an interface which provides some notification method(s) that are called by the observer's subject if it changes its state. The actual logic of these methods is defined by the concrete observer classes which implements the observer's interface. The subject provides methods for subscribing and unsubscribing as observer and maintains a list of all observers that have to be notified. The structure of observer is presented in Figure 4.13. In order to keep it simple it leaves out the abstract subject, so it differs slightly from the one in [14].

Observer models can be divided into two models. One model is the so called push model. That means that the subject provides the altered data as argument to the notification method. The observer does not need to call the subject in order to get the actual state. The second model is called pull model, where the observer is just notified by the subject that something has changed. The observer must call the getter methods provided by the subject in order to get the new values.

There exist several possibilities to implement this pattern, depending on the features provided by the used programming language and the types provided by the underlying class library. Each of these possibilities have in common that they rely on defined interfaces for both the subject and the observer. That requires the adapter to perform several adaptations:

- The adapter has to adapt the notification method called by the subject. That requires the adapter to be an observer itself by implementing the

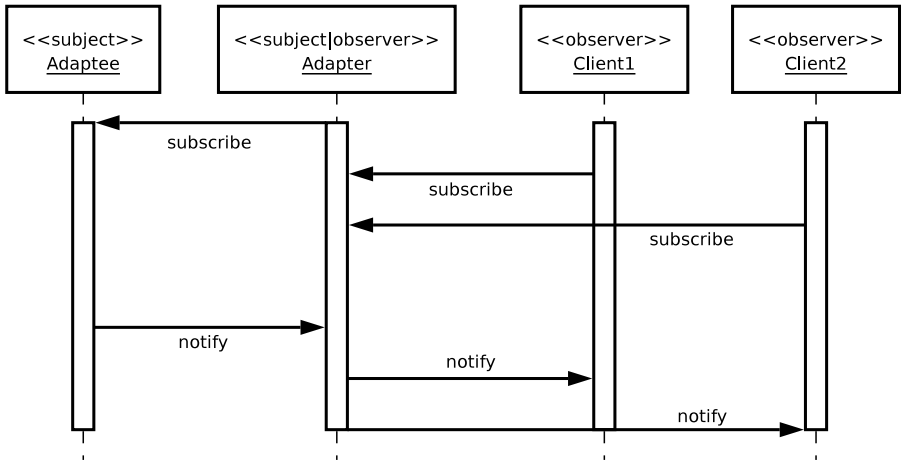


Figure 4.14: Event adaptation workflow

new version's observer interface.

- In order to allow old clients to subscribe to events, the adapter needs to act as subject for old clients which requires maintaining the lists of subscribers by the adapter. However the getter methods have to be forwarded to the real subject.

- In push model the adapter needs to create an adapter for the altered data.

If an event occurs in the framework, the adapter gets notified. The adapter then notifies all clients subscribed to this event. This workflow is shown in Figure 4.14. When using the push model, the adapter wraps the data sent with the notification before notifying the clients. When using the pull model, the clients query the adapter for the new state. The adapter forwards this call as any other call to the new framework.

The following sections describes possibilities to realize the observer pattern in a language independent way as well as special ways for .Net and Java.

Language independent observer implementation

An obvious way to use the observer mechanism is to directly implement the observer structure. The subject needs to provide methods for subscribing and unsubscribing as observer as well as getter methods when using the pull model of the observer pattern. The observer provides methods for being notified when an event occurs.

When this approach is used, a rather sophisticated adapter is required. It needs to implement each observer interface accepted by the new version of the type it represents and needs to act as subject for each observer accepted by the old type it replaces. The client will subscribe itself to the adapter and the adapter will subscribe to the API class. The adaptation process is the same as described above.

```
delegate void SomeDelegate(int arg1, String arg2);
```

Figure 4.15: Declaration of a delegate type

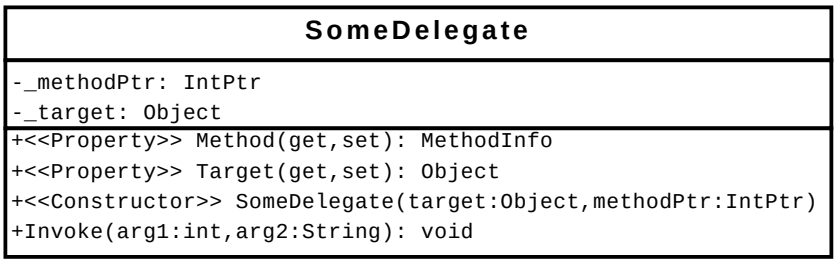


Figure 4.16: Resulting delegate class

Using the Java class library

The Java class library provides a built in observer mechanism. For observers the interface `java.util.Observer` is provided. Each type wanting to be an observer is required to implement this interfaces. The subject must be derived from class `java.util.Observable`. This class contains the basic logic for registering and maintaining observers. It also provides two overloads for a method called `notifyObservers`, one with an object describing the change and one without.

The adaptation of this mechanism is easier to realize than in the general approach due to the fact that the notification and subscription mechanisms are standardised. The adapter must do both deriving from `java.util.Observable` and implementing `java.util.Observer`. As in the general case, the clients subscribe to the adapter which itself is subscribed at the framework. The rest of the adaptation is similar to the one described above.

.Net Events

In .Net there is a syntactically more elegant solution for realizing the observer pattern. In .Net it is possible to use events which are based on delegates. Delegates can be best described as typesafe method pointers. Although delegates are defined in a similar way as a function pointer in C (Figure 4.15), the result of a delegate declaration is a class derived from type `MulticastDelegate`.

Figure 4.16 shows parts of the resulting type of the declaration of Figure 4.15. The class contains two private fields, an `IntPtr` pointing to the address of the method to call and an `Object` describing the object to invoke the method on (for static methods this is `null`). These fields are made accessible by public properties. In order to actually invoke the delegate the method `Invoke` is called. This method has the same arguments and return type as defined in the delegate's declaration. This description only shows the relevant subset of a delegate type. The type generated by the compiler contains some more members and is able to call more than one subscriber.

Since delegates have to be publicly accessible to allow clients for subscribing to it, each client that can subscribe can also call the delegate. It also give a

```
public event SomeDelegate MyEvent;
```

Figure 4.17: Event declaration

client the possibility to call the delegate. That means that a notification may be caused not only by the subject but by every type referencing this subject. Since event notifications should only be caused by the subject itself and not by an arbitrary class having a reference to the subject, the delegate concept was extended by the concept of events. The declaration of an event is similar to the declaration of a delegate. However, the single line in Figure 4.17 is translated into a private delegate variable and two methods, having the same visibility as the event, for subscribing and unsubscribing to the delegate.

In order to adapt events, again an adapter for the type offering the event is needed. As in the previous approaches this adapter needs to offer all events the replaced version offered and it needs to subscribe to the events of the adaptee. However, the adapter generator needs to regenerate the delegate types for all events except for the case the event is based on a standard .Net `EventHandler<T>` delegate. This delegate is defined in the .Net library and provides a common signature for event handler methods. The type parameter `T` is a type derived from class `EventArgs`.

If only the adaptation of a delegate is required, for example if this delegate is used as argument for some method, one needs to generate an adapter for the delegate class itself. This class needs to provide a method suitable for the new version of the delegate. This method would perform the adaptation and then call the `Invoke` method of the delegate itself.

Chapter 5

Related work

Preserving the binary compatibility of an API is not a new topic. There are several approaches trying to solve this problem. Each approach somehow restricts the developer in the kind of changes that can be done to the API. The first, described in section 5.1, tries to control changes by stating rules that have to be followed in API design. Another approach uses middleware in order to prevent changes from becoming visible to clients (section 5.2). The last approach, described in section 5.3, tries to recreate an environment known by an existing client.

5.1 Prohibition

When an API is published, developers expect this API to be stable, at least, for a certain time. It is also required that the compatibility of an API and a client is stable over API changes. Without having adaptation the possibilities to achieve binary compatibility are limited. A good introduction about which changes are possible without breaking the binary compatibility in Java is done in [11], where tables describing the compatibility of a change are used. Table 5.1 shows a shortened table describing the compatibility of changes to API class members.

The addition of members to the API may be done without disturbing the functionality of clients. Thus it is always possible to extend a library's or framework's functionality. Since direct changes to existing API members will break clients but adding new members is mostly harmless, the so called "2-convention" may be used if changes cannot be avoided. When changes to a type should be applied, a new type with the same name is created and suffixed with a trailing "2". An interface named `ICustomerManager` will become `ICustomerManager2` due to this conventions, hence now there exist two types having the same intent but different interfaces. In addition the old type may be marked as `deprecated` or `obsolete`. The deprecated type will be kept some further versions of the API and then deleted. The deprecation phase will give third party developers some time to migrate their clients to the new API, so that the removal of the deprecated type will be safe for the majority of clients.

This approach is simple and compatible with all programming languages or platforms. However, it takes a long time to get rid of old types which is especially

Change body of method or constructor	Binary compatible
Change formal parameter name	Binary compatible
Change method name	Breaks compatibility
Add or delete formal parameter	Breaks compatibility
Change type of a formal parameter	Breaks compatibility
Change result type (including void)	Breaks compatibility
Add checked exceptions thrown	Breaks compatibility
Add unchecked exceptions thrown	Binary compatible
Delete checked exceptions thrown	Breaks compatibility
Delete unchecked exceptions thrown	Binary compatible
Re-order list of exceptions thrown	Binary compatible
Decrease access; that is, from protected access to default or private access; or from public access to protected, default, or private access	Breaks compatibility
Increase access; that is, from protected access to public access	Binary compatible
Change abstract to non-abstract	Binary compatible
Change non-abstract to abstract	Breaks compatibility
Change final to non-final	Binary compatible
Change static to non-static	Breaks compatibility
Change non-static to static	Breaks compatibility
Change native to non-native	Binary compatible
Change non-native to native	Binary compatible
Change synchronized to non-synchronized	Binary compatible
Change non-synchronized to synchronized	Binary compatible
Delete type parameter	Breaks compatibility
Re-order type parameters	Breaks compatibility
Rename type parameter	Binary compatible
Add, delete, or change type bounds of type parameter	Breaks compatibility
Change last parameter from array type T[] to variable arity T...	Binary compatible
Change last parameter from variable arity T... to array type T[]	Breaks compatibility

Table 5.1: Compatibility of changes to methods and constructors of API classes

```
@Remote
public interface ICustomerManager {
    public Customer[] GetCustomers();
    public Customer GetCustomer(int id);
}
```

Figure 5.1: Remote interface for an EJB

```
@Stateless
public class CustomerManager implements ICustomerManager {
    public Customer[] GetCustomers() {
        // get customers here
    }

    public Customer GetCustomer(int id) {
        // get special customer here
    }
}
```

Figure 5.2: Simple EJB

bad when introducing a new API. There is also a cosmetic drawback: The new versions will all have a suffix. More disturbing is the fact that typographic errors or broken naming conventions will survive longer if the respective type has an otherwise good design. For such minor errors introducing a completely new type is overkill. Most important, this approach is too limited.

5.2 Prevention

Especially when the focus is based on the communication of components, systems that are based on middleware are used, e.g. COM [5], CORBA [6] or Enterprise Java Beans [8]. Common to all of these approaches is that the functionality of a component is described by an interface. This interface may be described in a certain programming language as in case of EJB (where Java is used) or some kind of meta language as in case of CORBA (which uses a language independent Interface Definition Language (IDL)). If a client wants to use a component and therefor needs a reference to it, it does not create the desired component directly but asks the middleware for an existing instance. The middleware then uses some locator mechanism (like JDNI for EJB) to retrieve the desired component.

A simple definition of a component can be seen in Figures 5.1 to 5.3. The example code is for Enterprise Java Beans. First the interface `ICustomerManager` is defined, describing the capabilities of the component. The component itself is a stateless session bean as shown in Figure 5.2. In order to use the bean, a client needs a reference, which it retrieves with help of JNDI.

Using solely the interfaces of some component has the advantage that the component implementing the functionality may be exchanged as long as the

```
public static void callEJB() {
    InitialContext ic = new InitialContext();
    ICustomerManager manager = (ICustomerManager) ic
        .lookup(ICustomerManager.class.getName());

    /* further code */
}
```

Figure 5.3: Getting the EJB

```
@Remote
public interface ICustomerManager2 {
    public Customer[] GetCustomers(String query);
    public Customer GetCustomer(int id);
}
```

Figure 5.4: The changed `ICustomerManager` interface

interfaces are stable. Functionality of a component can be extended by just implementing further interfaces. If the API has to be changed this can be done by adding new interfaces which are named differently from the existing ones. Some technologies like COM or .Net allow to add version information to a type which even allows to keep the same name for a type. If the name has to be changed, the "2-convention" as mentioned in section 5.1 can be used. The actual component does not need to be duplicated, instead, an existing component can be reused, which then implements both the old and the new interface. As long as the old interface is still implemented the clients will not break (Figure 5.4).

If method `GetCustomers` of interface `ICustomerManager` should be changed in a way that it is possible to give a query restricting the set of customers retrieved, one may define a new interface as in Figure 5.4. Than one only needs to implement the new interface in the old bean as shown in Figure 5.5. The functionality of the old interface remains, so old clients will still continue to work but new clients may use the new, improved interface.

Restricting the client to using interfaces enables changes to the implementing classes, since they only need to implement further interfaces if anything should be changed. Furthermore the client is absolved from creating or locating a desired instance of a type since the latter is delivered by the middleware. However, framework and clients are restricted to communicate via the used middleware. At the level of the API addition is still the only action that is allowed to be performed.

5.3 Facilitation

In order to support changes beyond the addition of types and members, one needs further runtime support in order to adapt the changes to the client. That means that an adapter layer is used, which shields the client from seeing the changes. However, the clients are not restricted to use interfaces but may use

```
@Stateless
public class CustomerManager implements
    ICustomerManager, ICustomerManager2 {

    public Customer[] GetCustomers(String query) {
        // get selected customers here
    }

    public Customer[] GetCustomers() {
        // delegate to the new method
        return GetCustomers(null);
    }

    public Customer GetCustomer(int id) {
        // get special customer here
    }
}
```

Figure 5.5: The EJB implementing both interfaces

each kind of member available in the API. The adapter layer is responsible for providing the correct functionality.

One possible solution is described in [10]. The approach defines a set of refactorings that are allowed to be performed to the API. For each of these refactorings, which are considered as structure changing but semantic preserving changes, a so called comeback, a rule reproducing the state before the refactoring, is defined. Given an ordered set of refactorings performed between two versions of an API, one is able to automatically generate an adapter layer, that reproduces the old API for existing clients. Now every time the the client requests a type from the old API it gets an adapter which will then delegate the work to the correct parts of the new API.

The adaptation approach greatly improves the freedom of a developer, since virtually every change can be handled by the adapter. The downside is that the generation of the adapter requires some affords. Moreover, generated adapters may decrease performance.

Chapter 6

Conclusion and Future Work

With help of frameworks, companies are able to build product lines. A good framework API also allows third party developers to extend these applications by providing additional functions or extending existing functionality. Problems arise when frameworks and their APIs are maintained, since maintenance may change the API and thus invalidating existing clients. Maintaining the framework is required in order to meet new requirements and fix bugs inside the framework.

The B2-PDE project solves the maintenance problem by introducing an adapter layer shielding the client from the changes done to the API. Due to this adapter layer existing clients continue to work even if the API has change in a way that would normally break this client.

One of the contributions of this work is to extend the adapter generation by providing the ability of generating adapters for generic types as well as implementing this method in a prototype. Another contribution is the statement of design and change restrictions to be considered by the framework developer in order to keep the framework adaptable. In addition to listing the restrictions, possible solutions are provided in order to overcome these limitations.

Future versions of *GAG* need to be extended by the possibility to generate adapters for generic interfaces and generic interface members. Also support for bounded polymorphism still needs to be implemented. Further investigation may be needed in order to find out the impact of the dynamic invocation on the performance of the adapter.

The existing adapter generators have to be extended with the capabilities of generic adaptation. It may be also desirable to include some of the suggested solutions in the developer guide into the existing adapter generators in order to ease the work of the adapter developer.

Bibliography

[1] Comarch homepage. http://www.comarch.com.

[2] ComeBack! homepage. http://comeback.sf.net.

[3] Eclipse homepage. http://www.eclipse.org.

[4] Microsoft .NET homepage. http://www.microsoft.com/net.

[5] MSDN Library, Component Object Model. http://msdn2.microsoft.com/en-us/library/ms680573.aspx.

[6] Object Management Group homepage. http://www.omg.org/.

[7] SAB homepage. http://www.sab.sachsen.de.

[8] Sun JEE Homepage. http://java.sun.com/javaee/.

[9] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *Lecture Notes in Computer Science*, 2139, 2001.

[10] Ilie Şavga and Michael Rudolf. Refactoring-based support for binary compatibility in evolving frameworks. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 175–184, New York, NY, USA, 2007. ACM Press.

[11] Jim des Rivières. Evolving Java-based APIs. http://wiki.eclipse.org/Evolving_Java-based_APIs, 2007.

[12] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In *ECOOP'06: European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 404–428. Springer, 2006.

[13] ECMA. *ECMA-334: C# Language Specification*. 2002.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[15] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.

[16] Joel Pobar. Dodge Common Performance Pitfalls to Craft Speedy Applications. http://msdn.microsoft.com/msdnmag/issues/05/07/Reflection/.

[17] Jeffrey Richter. *Microsoft .Net Framework Programmierung in C#.* Microsoft Press, 2006.

[18] Dirk Riehle. *Framework Design: A Role Modeling Approach.* PhD thesis, Swiss Federal Institute of Technology Zurich, Universitaet Hamburg, 2000.

[19] Michael Rudolf. Δ-Guided Plugin Adaptation in .NET. Master's thesis, University of Technology Dresden, October 2006. Bachelor's thesis.

[20] Michael Rudolf. Anatomy of a Delegating Adapter Method for .NET and Java. 2007.

[21] Ilie Savga, Michael Rudolf, Jacek Sliwerski, Jan Lehmann, and Harald Wendel. API changes - how far would you go? In *CSMR'07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 329–330. IEEE Computer Society, March 2007.

Confirmation

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

Dresden, December 3, 2007