ANATOMY OF A DELEGATING ADAPTER METHOD FOR .NET AND JAVA

MICHAEL RUDOLF

VERSION 0.3, 2008–07–03

This document is intended to shed some light on the inner workings of delegation methods used in the ComeBack! tool. As the static adapter structure and the corresponding generation process has already been described in various publications, runtime and implementation-level issues are explained hereinafter. It concludes with an overview of the challenges that remain to be investigated in detail.

Version History

Version	Date	Changes
0.1	2007–11–18	Initial version
0.2	2008–06–23	Changed formatting
0.3	2008–07–03	Added section *Avoiding Multiple Adaptation*

Introduction

Versions and Adapters

The static aspects of adapters and their generation as employed in the ComeBack! project [1] have already been described in several publications. Figure 1 depicts the necessity of an adapter to compensate for an *ExtractSubclass* refactoring that has been applied to a framework in the course of its evolution. Without these adapters clients will break, they hide away the changes by reifying a prior framework version required by the plugins. The actual implementation technique used in the automatically generated adapter types is *delegation*: each method provided there will simply forward to the framework. This document

[1] ComeBack! project homepage. `http://comeback.sourceforge.net/`. Last visited November, 2007.

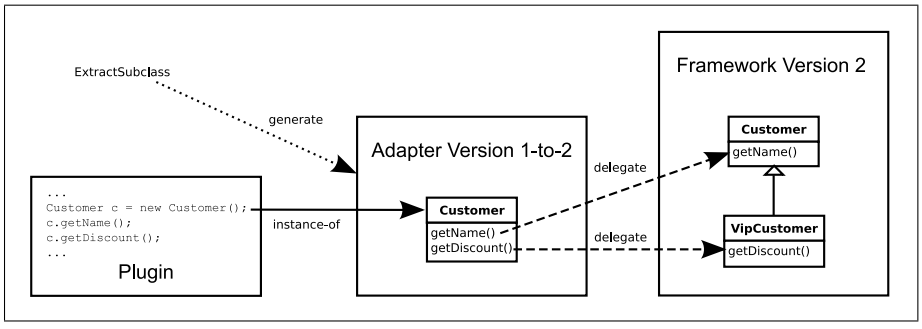

Figure 1: Adapter bridging an *ExtractSubclass* refactoring

details the issues with implementing such delegation methods and their runtime behavior.

Object Domain Segregation

The main purpose of a delegating adapter method is, besides forwarding the call, the *segregation* of objects into separate *domains*. This is illustrated in Fig. 2: plugin and framework objects form a distinct group each, and only if an object from one domain needs to be passed into the other it is wrapped inside an adapter instance. This behavior is basically the extension of the static adapter concept described above to the runtime. It serves as the driving rationale behind the design of delegation methods explained in the next section.

Object domain segregation is necessary for making sure, that the framework and its plugins are only passed instances of those types they know about. However, as communication between the two parties via sending messages is one of the key elements in object-oriented programming, the adapter layer needs to account for objects crossing domain boundaries. This is done by *wrapping* objects inside instances of their corresponding adapter types. These wrappers implement that logic, so that whole graphs of interconnected objects can migrate from one domain to another. Conversely, wrapped objects being passed back into their domain of origin need to be *unwrapped*. In addition to that, adapters can also compensate for refactorings and other changes, as already mentioned in the section *Versions and Adapters*.

For communication of objects between the two domains in Fig. 2 to begin, at least one object needs to be present in the communication partner's domain, a so-called *handle*. However, for object migration to take place as described above, an already existing adapter has to perform the wrapping, implying a recursive problem. Nevertheless, this is no instance of the prominent hen-or-egg problem, as the framework types (or their adapters) are known to plugins at compile time. As a consequence, instances of them can be created directly or obtained by calling static methods. Another solution for communication initialization is the use of *dependency injection* or *inversion of control* (IoC), where plugins are given a communication handle by the framework that instantiates them.

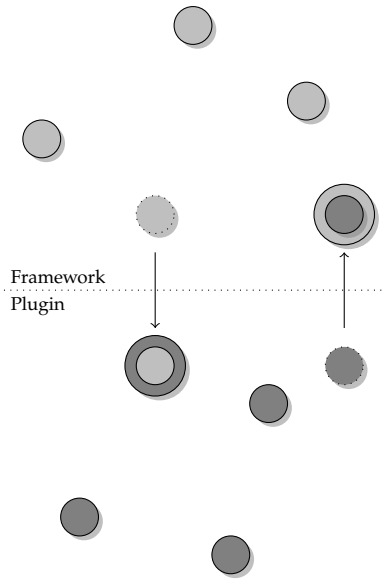

Figure 2: Segregating objects into domains

Dissection

This listing shows the pseudo code of a delegating adapter method in its entirety:

```
<AccessModifier> <ReturnType> Name(Parameters) {
  //Eventually (un-)wrap the arguments
  foreach (argument in Parameters) {
    if (argument != null) {
      if (isAdapter(argument)) {
        unwrap(argument);
      } else if (isUserDefined(argument)) {
        wrap(value);
      }
    }
  }

  Object value;
  try {
    value = ((TargetType)delegatee).Name(arguments);
  } catch(Throwable t) {
    //Eventually (un-)wrap the exception object
    if (isUserDefined(t)) {
      AdaptedThrowable at = wrap(t);
      throw at;
    } else if (isAdapter(t)) {
      t = unwrap(t);
      throw t;
    } else {
      throw t;
    }
  }

  //Eventually (un-)wrap the return value
  if (value != null) {
    if (isAdapter(value)) {
      value = unwrap(value);
    } else if (isUserDefined(value)) {
      value = wrap(value);
    }
  }
  return value;
}
```

The following subsections are dedicated to the specific parts that together produce the full functionality of a delegating adapter method.

Arguments and Return Values

```
  //Eventually (un-)wrap the arguments
  foreach (argument in Parameters) {
    if (argument != null) {
      if (isAdapter(argument)) {
        unwrap(argument);
      } else if (isUserDefined(argument)) {
        wrap(value);
      }
    }
  }
```

The very first part of a delegation method takes care of preparing the arguments before passing them on to the delegatee. The pseudo

code shows that process as a loop iterating over every argument. However, as the number of method parameters is known at adapter generation time,[2] that loop would in practice get unrolled before code emitting to improve performance. This implies that methods taking no parameter will not suffer from any overhead before the delegation call is placed. As both the Java [3] and the .NET [4] runtime environments have a stack architecture, the call arguments can be directly replaced on the stack without additional overhead. Every non-null argument is examined and treated in one of the seven ways:

a) it is an adapter type and has to be replaced by its corresponding user-defined type,

b) it is a user-defined type and has to be replaced by its corresponding adapter type,

c) it is a collection and its elements have to be examined and handled similarly to these seven cases,

d) it is an array and its elements have to be examined and handled similarly to these seven cases,

e) it is an adapter and has to be unwrapped before passing on,

f) it is an instance of a user-defined type and has to be wrapped before passing on, or

g) it is an instance of a standard library type and does not need to be modified.

The first two items in the above list make sure that objects do not come in touch with types from outside of their domain involuntarily. As types are legal method arguments and return values, callers might pass in types from their domain, while the callee expects only types from its domain. As an example, consider a data conversion method offered by the framework taking the target user-defined type as an argument in addition to the data that should be converted. As the caller only knows about the adapter types corresponding to the possible target types of the conversion, the framework will not be able to serve the client's request, because it cannot handle adapter types as conversion targets. Therefore, adapter type objects need to be replaced by their corresponding user-defined type and vice versa.

The items c and d are not depicted in the pseudo code above, as they merely contribute to preventing object domain leakage, which is described later on in the section *Preventing Object Domain Leakage*. Array conversion involves creating a new array of the same length as the given argument and copying each element from the source to the destination thereby eventually wrapping or unwrapping it. However, this is only necessary in those cases, where the array can potentially contain adapters or user-defined types. For instance, arrays with component type `String`[5] can safely be ignored while `Object` arrays need to be adapted. The given argument is not altered

[2] We deliberately do not elaborate on varargs here, as they are most often implemented using arrays.

[3] Sun Java homepage. `http://java.sun.com/`. Last visited November, 2007.

[4] Microsoft .NET homepage. `http://www.microsoft.com/net`. Last visited November, 2007.

[5] In the remainder of this document we omit the package name `java.lang` from class names.

in-place, as the caller could then possibly see the changes. Note, that this implies that the callee will not be able to see any changes made by the caller during or after the call. Although it is bad object-oriented practice, sharing arrays is sometimes used for performance reasons and will be broken by this approach. The same technique used for array conversion is also applied to collections.

From the explanation in the section *Object Domain Segregation* it becomes obvious that the delegation method constitutes the boundary of object domains and therefore has to implement proper object migration. Item e) in the above list takes care of objects being passed back into their domain of origin: the wrapping adapter is removed and the actual object will be passed on to the callee. Item f) performs the symmetric operation in which plain objects are wrapped inside an instance of the corresponding adapter type before they are handed on.

```
29  //Eventually (un-)wrap the return value
30  if (value != null) {
31    if (isAdapter(value)) {
32      value = unwrap(value);
33    } else if (isUserDefined(value)) {
34      value = wrap(value);
35    }
36  }
37  return value;
```

The handling of method return values is exactly the same as for the arguments.

Note, that a delegation method does not need to be synchronized, even if it is used concurrently by a multi-threaded client. In that case the adapted object originally provided for it by serializing access to its internal state accordingly. And as the only state an adapter maintains is the immutable delegation field, no synchronization is needed for performing the delegation calls. However, that implies, that removing multi-threading capabilities from an object will not be compensated for by the generated adapters (unless the removal of synchronization can be modeled as a refactoring).

Exception Handling

```
13    Object value;
14    try {
15      value = ((TargetType)delegatee).Name(arguments);
16    } catch(Throwable t) {
17      //Eventually (un-)wrap the exception object
18      if (isUserDefined(t)) {
19        AdaptedThrowable at = wrap(t);
20        throw at;
21      } else if (isAdapter(t)) {
22        t = unwrap(t);
23        throw t;
24      } else {
25        throw t;
26      }
27    }
```

The central part of the method implements the actual delegation call inside a try-catch block. This is necessary, because the callee could throw user-defined exceptions, which we need to adapt as well. In the catch block we therefore have to check, whether the exception is part of the standard .NET or Java library or whether it is a user-defined type and thus reveals some aspect of the callee. If so, we just wrap it inside the corresponding adapter. Either way, we re-throw the exception, so that the caller notices the failure and can react appropriately. It is important to note, that we have to catch all exceptions, be them checked or unchecked in the case of Java, as we cannot assume anything about the exception propagation behavior of the called method and we do not want exceptions to be accidentally propagated to other object domains.

Note, that this approach cannot compensate for changes in the exception semtanics of a method, e.g., when adding a new exception to the list of possibly thrown ones. In that case protocol adaptation is needed to introduce exception handling code into the generated adapters. However, when changing the exception semantics such that subtypes of previously thrown exceptions are thrown, no adaptation is needed at all, as the existing exception handling code will account for that. Consider, for example, a method throwing exceptions of type `java.io.IOException` being changed to indicate failures using instances of `java.io.FileNotFoundException` – no adaptation is required due to the subtype relationship of the two exception types.

As an aside, the delegation call does not make use of *dynamic dispatch*, because the target method is known at generation-time and must not be made subject to overriding in potential subclasses of the target type. This is indicated in the pseudo code using an explicit cast operation. In .NET this can be implemented by using the non-virtual IL opcode `call` instead of `callvirt`. Unfortunately Java does not offer non-virtual method calls [6], so that care must be taken when overriding methods in non-abstract classes.

[6] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, April 1999.

Helper Methods

The pseudo code in the previous subsections contains some highlighted calls to helper methods for brevity. Here is what they are supposed to do:

- **`isAdapter`**`(Object)` checks whether the given object is an adapter. This should be done by tagging adapters with a special *annotation*[7] at generation time and testing for it at runtime. In Java you can simply call `Class.isAnnotationPresent(Class)` to accomplish that.
- **`isUserDefined`**`(Class)` checks whether the given class is part of the framework. If not, it has to be part of the language environment. In Java this could be done using class name comparison, e.g., by testing whether the class name starts with "java.", "javax.", etc., as these packages are *sealed* and cannot be extended by applications. However, a special handling is required for third-party libraries, which are not subject to adaptation. .Net does not seem to provide a package sealing mechanism, so a different approach would need to be taken there.
- **`wrap`**`(Object)` replaces the given object with an instance of its corresponding adapter.
- **`unwrap`**`(Adapter)` is just a placeholder for an access (either via a method call or via direct field access) to the adapter's delegation variable. Due to the stack architecture of both Java and .NET this can be efficiently implemented.

[7] In .NET these are called *attributes*.

Generation-Time Optimizations

In many cases the handling of method arguments and return types can already be optimized at generation-time. Although this does not affect correctness, it will dramatically improve runtime performance. Often in the adapter generation process there is enough information available to judge whether a method argument or return type can ever be an array, a user-defined type, or an adapter. Exploiting that knowledge, the generator can safely remove checks and conversion code thereby speeding up method execution. Consider, for example, a Java method taking an argument of type `java.lang.String`. Because that class is marked as *final* and therefore cannot be subclassed, there will never be a user-defined type deriving from it. Therefore, the code illustrated using the helper methods **`isAdapter`** and **`isUserDefined`** can be omitted. Furthermore, arguments will never be arrays, so the corresponding testing and conversion code can also be left out.

In Java there also is potential for optimization in the catch block used to handle exception objects thrown by the delegation call. By exploiting the delegatee's meta-information on checked exceptions, the single catch block can be split up into several specific ones and a

generic one like sketched above. The former are used for hardcoding the wrapping or unwrapping of the exception object thereby avoiding runtime lookup, and the latter handles the rest.

The general means for deriving generation-time optimizations is based on assignment validity: if it is legal for a parameter or return type to be assigned instances of an adapter type, a user-defined type, or an array type, the code for **`isAdapter`**, **`isUserDefined`**, or array conversion, respectively, has to be emitted.

Challenges

Fast Runtime Adapter Type Lookup

Whenever a method argument or return value is determined to be an instance of a user-defined type, it has to be wrapped inside an instance of the corresponding adapter type. However, as the type is only known at runtime, the choice of an adapter type to instantiate can also only be made at runtime. Therefor a fast type lookup mechanism is needed that, given a user-defined type, returns the corresponding adapter type. Possible implementations of that mapping could use a hashtable or a binary tree and have to be deployed together with the adapter types. In order to cope with dynamically generated types (e.g., dynamic proxies in Java) this lookup has to recurse up the inheritance hierarchy until a user-defined super type is found for which an adapter type exists. As the number of user-defined types as well as the number of potential super types is finite, this recursion terminates eventually. The following listing sketches a non-recursive implementation in Java using a hash table as the backing store.

```
1  public static String lookup(Object obj) {
2    Class<?> clazz = obj.getClass();
3    String adapterType = null;
4    do {
5      adapterType = hashtable.get(clazz.getName());
6      clazz = clazz.getSuperclass();
7    } while (adapterType == null && clazz != Object.class);
8    return adapterType;
9  }
```

In case there is more than one corresponding adapter for a user-defined type (e.g., when adapting a type merge), a special decision logic has to be inserted at generation-time for choosing the right adapter type to instantiate. This can be implemented by examining the attributes of the object in question at runtime and by basing the decision for the adapter on the results.

Avoiding Multiple Adaptation

Once it has been determined that an instance of a user-defined type needs to be wrapped inside in adapter, the **`wrap`** function has to

make sure that there is no other adapter wrapping the same object. More formally, for two instances of a user-defined type u_1 and u_2 and their corresponding adapters a_1 and a_2 the following must hold:

$$u_1 == u_2 \iff a_1 == a_2.$$

In order to achieve this, the adapter layer must store the input and output of every wrapping process, because a single object might pass the object domain boundary several times or on multiple paths. If that is the case, no new adapter must be created but the existing adapter has to be used instead. However, this requirement incurs considerable overhead and needs to be implemented in a way that is not affecting the performance of wrapping objects too negatively. As the lifetime of every adapter object is bounded by the lifetime of its corresponding user-defined object, we can make use of garbage collection available in Java and .NET in order to manage the object pairs.

A possible solution would be to use a collection of weak references to user-defined objects mapping to weak references to the corresponding adapter objects. Weak references permit the referenced object to be garbage-collected, when there are no other hard references to it. As the user-defined objects and the adapters are cleared out of the map by the garbage collector, no manual memory management has to be implemented. The performance overhead of such a map is $\mathcal{O}(1)$, it only depends on the number of buckets. Note that the data structure must be thread-safe, as wrapping objects can be performed concurrently from multiple threads. The downside of such an implementation is its dependency on the proper implementation of object equality, because the map makes use of it for hashing (i.e., the `hashCode` and `equals` methods in Java and similarly named methods in .NET).

Preventing Object Domain Leakage

Presumably the most challenging task is preventing objects from leaking into other domains. This is the case, when instances of user-defined types are part of an object graph with a standard library object at its root. The most prominent examples are collections: when adding a user-defined object to an instance of a collection type provided as part of the standard library (e.g., `java.util.List`) and passing that collection to a delegation method thereby crossing the boundary of two object domains, the user-defined object will not be wrapped or unwrapped and thus leak into the target domain. However, collections are not the only source of object leakage. Another feature being dangerous in this regard is exception chaining, where an exception object can point to another one indicating what was the original cause. Basically this is just another name for wrapping exceptions inside another. The following listing shows a Java implementation accounting for that.

```
} catch (Throwable t) {
  List<Throwable> chain = new ArrayList<Throwable>(3);
  Throwable temp = t;
  while (temp != null) {
    chain.add(temp);
    temp = temp.getCause();
  }
  boolean changed = false;
  for (int i = chain.size() - 1; i >= 0; i--) {
    temp = chain.get(i);
    if (isAdapter(temp)) {
      temp = unwrap(temp);
      changed = true;
    } else if (isUserDefined(temp)) {
      temp = wrap(temp);
      changed = true;
    } else if (changed) {
      Throwable newTemp = temp.getClass().newInstance();
      newTemp.setStackTrace(temp.getStackTrace());
      newTemp.initCause(chain.get(i + 1));
      temp = newTemp;
    }
    chain.set(i, temp);
  }
  throw chain.get(0);
}
```

For being able to more precisely determine when object domain leakage might occur and when to wrap and unwrap objects, formal definitions of the described concepts need to be introduced. In addition to the two object domains depicted in Fig. 2, there is yet another domain the two *inherit* from. That *superdomain* comprises all other objects that both parties know about but that have their origin outside of the two *subdomains*. In practice this domain will contain all instances of standard library types, such as `String`. As domain inheritance is a superset-subset relationship, no domain segregation and consequently no wrapping or unwrapping is required.

This concept calls for a definition of what is meant by *domain of origin* of an object. In contrast to the intuitive understanding, an object's domain is not the same as the domain of the instantiating object – if a framework object instantiates another object, the two objects must not necessarily belong to the same domain. Instead, the domain of an object is that of the most specific public class or interface in the inheritance hierarchy of its type. This stems from the assumption, that an object can only be communicated with across domain boundaries via a known (and thus public) communication interface. Let $\mathcal{O}$ be the set of all objects at an arbitrary, but fixed point in runtime, $\mathcal{C}$ the set of all classes and interfaces, and $\mathcal{D}$ the set of all domains, then the domain of origin of an object $o \in \mathcal{O}$ is defined as follows:

$$domain : \mathcal{O} \rightarrow \mathcal{D},$$

$$domain(o) := classDomain(classOf(o)),$$

where $classOf : \mathcal{O} \rightarrow \mathcal{C}$ denotes the runtime type of o and *classDomain* returns the domain of the most specific public class or interface in the inheritance hierarchy:

$$classDomain : \mathcal{C} \rightarrow \mathcal{D},$$

$$classDomain(c) := \begin{cases} loader(i) & \text{if } \exists i \in interfaces(c) : i \text{ is public,} \\ classDomain(super(c)) & \text{if } c \text{ is not public,} \\ loader(c) & \text{otherwise,} \end{cases}$$

where $loader : \mathcal{C} \rightarrow \mathcal{D}$ returns a language-specific domain definition (e.g., the `ClassLoader` in Java), $interfaces : \mathcal{C} \rightarrow \mathcal{P}(\mathcal{C})$ returns the set of all declared interfaces, and $super : \mathcal{C} \rightarrow \mathcal{C}$ returns the superclass of the given class.

As an example consider a framework providing a non-public subclass of `java.io.InputStream` for accessing binary data. Although the actual type of such a stream is defined by the framework, its domain of origin is the superdomain, because the most specific public class that stream can be accessed with is a standard library type. As a consequence, no wrapping of that stream inside an adapter is necessary, because its communication interface cannot evolve and its semantics is fixed. In many cases this increases the feasibility of the approach and reduces adaptation overhead. However, this also permits objects to leak into other domains and undermines object domain segregation.

Thus, the actual challenge consists of finding all standard library types that might cloak instances of user-defined types and implement a specific handling for them in delegation methods. For the collection example this would entail iterating over all elements, examining each and reacting according to the principles outlined in the section *Arguments and Return Values*. In .NET, where generic type parameter information is preserved during compilation, generation-time optimizations can be applied for handling generic collections, as the latter can only contain elements conforming to the specified (upper and/or lower) bounds.

Acknowledgements

I would like to thank the members of the ComeBack! team for providing me with helpful feedback in the process of compiling this description. Special acknowledgement deserves Edmundo David Trigos for pointing out several important problems not discussed in previous versions of this document.

Colophon

This document was typeset with the free, cross-platform LaTeX typesetting system using the `tufte-handout` package,[8] version 2.00,

[8] `http://code.google.com/p/tufte-latex/`

which simulates the layout style espoused by visual design expert Edward Rolf Tufte.