Using Java's Generics Mechanism to Improve Type Safety in the Command Pattern*

Karl Ridgeway
Rosetta Stone, Ltd.
Arlington, VA
karl.ridgeway@gmail.com

David Bernstein†
James Madison University
Dept. of Computer Science
Harrisonburg, VA
bernstdh@jmu.edu

John Magnotti
Auburn University
Dept. of Psychology
Auburn, AL
john.magnotti@auburn.edu

ABSTRACT

This paper discusses ways in which Java's generics mechanism can be used to improve a Remote Procedure Call implementation using the Command and Proxy patterns. Specifically, it shows how generics can be used to specify return types and receiver types.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Design—*methodologies*

General Terms

Design

Keywords

Command Pattern, Proxy Pattern, RPC

1. INTRODUCTION

We recently designed a system named *Moxaic*, a Java-based platform for presenting ultra-high resolution graphics on a collection of "commodity" displays (each with its own processor). As the effort progressed, we became aware of the fact that we were making extensive use of the Command pattern and that, by doing so, we were introducing type safety problems into the design. The purpose of this paper is to provide an overview of the kinds of type safety problems that can arise when using the Command pattern and the ways in which these problems can be remedied.

This work is closely related to a chapter found in [1], which discusses a generic `Functor` implemented using C++ templates. Alexandrescu discusses ways of using type parameters to improve the design of a C++ `Functor`. However, the example of the Command pattern in [1], like [4], is centered around its usage in user interface toolkits and therefore has limited applicability in an RPC context.

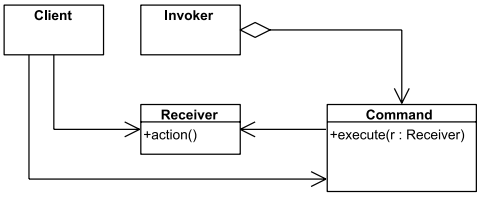

Figure 1: The Command Pattern (in UML)

Since the introduction of Generic Java [2], and, ultimately, its addition to the Java Language [5], much work has been done to better understand and incorporate Java generics into current solutions. However, little work has been done to improve the classic patterns found in [4] by implementing them with generic types. In [8], the authors examined some techniques for improving the Interpreter/Composite pattern through the use of generics.

It is the intent of this paper to similarly improve the Command and Proxy patterns, as used in the context of RPC, using Java generics.

1.1 The Design Patterns of Interest

The Command pattern is typically defined in UML as in Figure 1 and is traditionally described as a way to "encapsulate a request [or method call] as an object"[4]. As mentioned in [4], the Command pattern breaks a method invocation down into four fundamental constituent components. The `Receiver` is responsible for performing the operations necessary to complete a request. It contains the methods that the command will invoke (e.g. `action()`). The `Command` encapsulates an operation on a receiver. The `Invoker` instructs the command to execute its request on a receiver. Finally, the `Client` constructs a command, assigns it a receiver, and passes it to an invoker. A typical command invocation is illustrated in Figure 2:

In order to create a powerful remote procedure call (RPC) framework using the Command pattern, there should be a reusable component to provide remote access to any `Receiver`. A natural solution to this problem (see, for example, [4]) is to employ a remote proxy using the Proxy pattern.

As discussed in [7], the Proxy pattern is fundamentally an implementation of the Decorator pattern in which the proxy controls all access to its decorated resource. By implementing the same interface as its decorated resource, a

*Use with permission only.

†Corresponding author.

ACM-SE '08, March 28–29, 2008, Auburn, AL, USA.

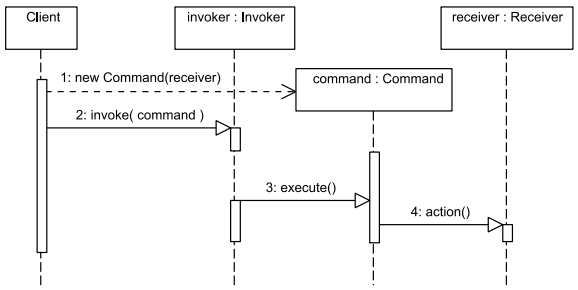

Figure 2: UML Sequence Diagram of a Command Invocation

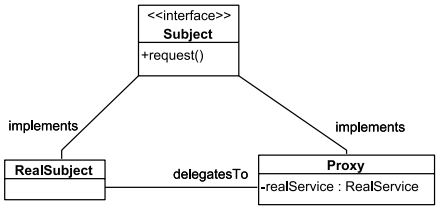

Figure 3: The Proxy Pattern (in UML)

`Proxy` can transparently handle all messages passed to the resource without necessitating changes to client code. This is illustrated in Figure 3.

1.2 The Setting: Moxaic

In the examples that follow, we will consider a type of command receiver found in *Moxaic*: a `Renderer`. The job of a `Renderer` is to present graphical content to the user, much like the Java AWT's built-in `Graphics` class. We were motivated to use the Command pattern in the context of `Renderer` for three reasons. First, it made remote message passing more convenient by using Java's object serialization mechanism to send commands. Second, it enabled us to queue and batch rendering commands. Finally, it enabled us to implement logging of rendering commands for application debugging. In the examples that follow, the `Renderer` acts as a receiver of commands. For each action that can be performed on a `Renderer`, there exists a command that implements that action.

Initially, the `Renderer` interface had the following specification, based on the Command pattern implementation found in [4]:

```
public interface Renderer
{
 /**
  * Returns the Graphics object that allows an
  * application to draw onto the device that this
  * Renderer represents
  */
  Graphics getGraphics();

  // ...
}
```

Each command type associated with a `Renderer` implemented the following interface:

```
public interface RendererCommand
{
  void execute(Renderer renderer);
}
```

A naive implementation of this pattern in the context of the Command pattern would involve creating a proxy for the receiver that delegates each method invocation. Because the `Renderer` plays the role of receiver in our example, this would involve creating a `RendererProxy` class:

```
public class RendererProxy implements Renderer
{
  private Renderer delegate;

  public Graphics getGraphics()
  {
   /**
    * Any logic pertaining to proxying the delegate
    * would go here
    */
    return delegate.getGraphics();
  }
}
```

Unfortunately, this approach makes the original intent of implementing the Proxy pattern in our `RendererCommand` example impossible. The Command pattern enables the serialization and transmission of messages by serializing the command object itself. In the above example, we essentially created a proxy for the `Renderer`, which plays the role of `Receiver` in the Command pattern. However, since the `Receiver` does not actually deal with the `Command` directly, it is not the correct participant to be proxied in this situation.

In general, the participant in the Command pattern that calls the `execute()` method on the `Command` is called the `Invoker`. To take advantage of the opportunity to serialize only the `Command` object, the invocation of the `execute()` method must be proxied.

In our `Renderer` example, the client code always played the role of invoker. In order to create a proxy for the invoker, we must encapsulate it in its own type:

```
public interface RendererService
{
  void invoke( RendererCommand command );
}

public class      RendererServiceImpl
       implements RendererService
{
  private Renderer renderer;

  public void invoke( RendererCommand command )
  {
    command.execute( renderer );
  }
}
```

With the role of invoker encapsulated, it is possible to create a `RendererServiceProxy`:

```
public class      RendererServiceProxy
       implements RendererService
```

```
{
  private RendererService delegate;

  public void invoke( RendererCommand command )
  {
    delegate.invoke( command );
  }
}
```

In practice, the receiver can also play the role of invoker to reduce the number of classes in the implementation:

```
public class      RendererImpl
       implements RendererService, Renderer
{
  public void invoke( RendererCommand command)
  {
    command.execute( this );
  }
}
```

1.3 The Terminology of Generics

Before continuing, it is important that we resolve some terminological issues that arise when discussing generics. Throughout this paper the phrase "generic type" refers to a class with type parameters. The phrase "parameterized type" refers to a type created from a generic type by providing an actual type argument ([6]).

We use the term "bind" to refer to the action of writing a parameterized type that refers to a specific class. For example, a parameterized type `List<String>` is a generic type `List<T>` bound to the `String` type. To avoid confusion, when we refer to the relationship between a generic type's type parameter (`<T>`) and its instantiated value (`String`), we say that the type parameter `<T>` is "realized" as the concrete type `String`. For more information on generic types and Java Generics, please refer to [6].

1.4 Organization of this Paper

Our objective in this paper is to discuss several type safety issues that arise when using the Command and Proxy patterns, particularly in the context of RPC. It is organized as follows.

We begin with a discussion of how to add type safety to the values returned by commands. We discuss several obvious techniques and their shortcomings and then consider the advantages of generics.

Next, we turn to a discussion of adding type safety to the receivers of commands. As we will explain, this problem is relatively straightforward until a proxy is introduced.

Hence, we next consider the issues that arise when combining the Command and Proxy patterns and show how Java's generics mechanism can be used to improve a Remote Procedure Call implementation. Specifically, we show how generics can be used to specify return types and receiver types.

Finally, we conclude with a brief discussion of future research.

2. TYPE SAFETY OF RETURNED VALUES

The first type safety problem we consider can be understood by comparing the Command pattern to the "ordinary" interpretation of method calls. Following [3], we define a method signature to include the method name, parameters (including types), the return type, the receiver (enclosing class), and, optionally, the thrown exceptions, type parameters, and other characteristics.[1] Command objects, as defined in [4], have a name (which is the name of the class), parameters (which are attributes of the command), and a receiver (which is specified by the parameter of the `execute()` method).

One problem with the classic implementation of command (shown above as `RendererCommand`) is that the return type of the `execute()` method is `void`. Therefore, `RendererCommand` objects by default cannot return a value. We will now consider techniques to enable `RendererCommands` to support return values.

2.1 Using an "Outbound" Parameter

An obvious way to add a return type to the Command pattern is to add an "outbound" parameter. In Java, this involves passing a `Container` object to the `execute()` method. The `Container` exists to encapsulate a reference to an `Object`. This reference should be set as the return value during the command's execution. When the command has finished executing, the `Client` may extract and read the result value from the `Container`.

For example, given a `Container` class like the following:

```
public class Container
{
  private Object value;

  public Object getValue()
  {
    return value;
  }

  public Object setValue( Object newValue )
  {
    this.value = newValue;
  }
}
```

the `RendererCommand` interface can be modified as follows:

```
public interface RendererCommand
{
  void execute( Renderer  renderer,
                Container returnValue );
}
```

Then, a concrete implementation of `RendererCommand`, defined as follows:

```
public class GetSize implements RendererCommand
{
 public void execute( Renderer rec,
                      Container returnValue )
 {
  returnValue.setValue(
   rec.getGraphics().getClipBounds().getSize());
 }
}
```

[1]For sake of simplicity, we will not consider issues that arise when methods are overloaded and/or overridden.

could be used in the following way:

```
Dimension result;
Container container = new Container();
Renderer renderer   = new Renderer();
GetSize getSize     = new GetSize();
command.execute( renderer, container);
result = (Dimension) container.getValue();
```

Unfortunately, this solution can cause problems when serializing (or 'marshalling') commands to be executed remotely, as in *Moxaic*. Most remote procedure call (RPC) mechanisms do not ensure that changes made to an `Object` after transmission to another address space are communicated back and applied to the originating address space. That is, most RPC systems essentially pass all parameters "by value". In this case, when a `Container` is transmitted to a foreign computer as an outbound parameter, changes to its `value` would not be reflected in the application running on the original computer. Therefore, communicating a return value by modifying a parameter as a side-effect would not be effective in a situation in which a receiver is remote.

2.2 Using a Wide Return Type

Another possible solution is to have the `execute()` method in the `RendererCommand` interface return an `Object` and narrow the return type in concrete implementations (using the covariant return type feature available in Java version 5.0):

```
public interface RendererCommand
{
  Object execute( Renderer renderer );
}
```

Then, given the following revised implementation of a `RendererCommand`:

```
public class GetSize implements RendererCommand
{
 /**
 * Note that this method narrows the return
 * type specified by the interface.  This
 * is made possible by the addition of
 * covariant return types in Java 5.0
 */
 public Dimension execute( Renderer rend )
 {
  return
   rend.getGraphics().getClipBounds().getSize();
 }
}
```

these classes could be used as follows:

```
  Dimension result;
  Renderer renderer = ...;
  GetSize getSize = new GetSize();
  result = getSize.execute(renderer);
```

This implementation is convenient because it is simple, type-safe, and can be employed in a situation where a `Renderer` is only remotely accessible (as long as the communications mechanism sends the return value back to the caller correctly).

2.3 Parameterizing the Return Type

Unfortunately, the solution above suffers from type safety problems when the Proxy pattern is used to enable remote accessibility.

Recall that we need to encapsulate the invoker in order to implement RPC using the Proxy pattern. To do so, the return value on the `RendererService` object's `invoke()` method must match the return value of any `RendererCommand` object's `execute()` method. This makes `Object` the natural choice. For example:

```
public class      RendererImpl
       implements RendererService, Renderer
{
  public Object invoke( RendererCommand command)
  {
    return command.execute( this );
  }
}
```

The problem with this approach is that the `RendererService` object will always need to widen the returned value. Fortunately, we can restore the desired type safety using the Java generics mechanism.

A straightforward solution to bind a command to its return type is to make command a generic type with its return value as a type parameter. In our `Renderer` example, we would modify the `RendererCommand` interface as follows: [2]

```
/**
 * @param <T> The type of the result
 */
public interface RendererCommand<T>
{
  T execute( Renderer renderer );
}
```

Additionally, an invoker must now have a corresponding method that is similarly parameterized in order to maintain type safety for return values. We would thus modify our Invoker (`RendererService`) as follows:

```
  /**
  * @param <T>     The type of the result
  * @param command Command to be executed.
  */
  <T> T invoke(RendererCommand<T> command);
```

The type parameter `<T>` on `RendererCommand` is realized by each implementing subclass of `RendererCommand`:

```
public class      GetSize
       implements RendererCommand<Dimension>
{
 public Dimension execute(Renderer rend)
 {
  return
    rend.getGraphics().getClipBounds().getSize();
 }
}
```

[2]This is similar to the approach taken in [1]. In that example, the `Functor` class is parameterized with its return type.

The type parameter `<T>` on `RendererService.execute()` is realized anew at every instance in client code where it is called. This solution guarantees generic type safety for command return values, eliminating the need for client side type checking code. This effectively restores type safety for return types when employing the Command pattern.

3. TYPE SAFETY OF RECEIVERS

The second type safety problem we consider is best understood by comparing the Command pattern to the "message passing" interpretation of method calls. In this interpretation, a "message" is sent to a receiving object that then behaves accordingly. The receiving object must be of appropriate type (i.e., the type that defined the "message") but has no other restrictions placed upon it.

When using the Command pattern, on the other hand, an actual command object is sent to the invoker and that command object must contain a typed receiver. Hence, our generic command implementation suffers from the fact that no command may be defined that operates on a type that does not inherit from receiver. In other words,no implementing subclass of receiver may define additional command-enabled methods that are not found in the receiver base class. This is a result of a restriction on overridden method parameters defined in the Java language[5] – a subclass of command cannot override the parameter of the `execute()` method to specify a different receiver type. To overcome these flexibility issues, concrete command classes must cast their receiver to the appropriate type to use the extended functionality. Unfortunately, this cast removes some of the static type checking afforded by the Java compiler.

A simple solution is to omit the generic command type. In that approach, each receiver/command pair would specify their own independent (and incompatible) base classes. However, this is not an ideal solution when generic command and receiver base classes are needed (e.g., in a Remote Procedure Call implementation).

If it is essential to declare a generic command type, as it is in our application, then the default implementation lacks the static type checking normally found in method invocations in Java. For example, let us consider two (albeit somewhat contrived) extensions of `Renderer`, a `Rasterizer` and a `Plotter`. A `Rasterizer` has the ability to render sampled content, whereas a `Plotter` has methods to manipulate its pen (`moveTo()` and `drawTo()`).

```
public interface Rasterizer extends Renderer
{
  void drawRaster( Image image );
}

public interface Plotter extends Renderer
{
  void moveTo( Point2D point );
  void drawTo( Point2D point );
}
```

One way to proceed is to have two different command interfaces as follows:

```
public interface RasterizerCommand<T>
{
  T execute( Rasterizer rasterizer );
}

public interface PlotterCommand<T>
{
  T execute( Plotter plotter );
}
```

Unfortunately, since `RasterizerCommand` and `PlotterCommand` do not share a parent interface, they cannot be treated generically.

Alternatively, we can create a single `RendererCommand` interface, eliminating the parallel `RasterizerCommand` and `PlotterCommand` interfaces:

```
public interface RendererCommand<T>
{
  T execute( Renderer renderer );
}
```

Unfortunately, each implementation of `RendererCommand` must then cast its `Renderer` to the appropriate type.

For example, an implementation of a command intended for a `Plotter` might look like:

```
public class      DrawLine
       implements RendererCommand<Boolean>
{
  private Point2D begin, end;
  //...

  public Boolean execute( Renderer renderer )
  {
    Plotter plotter = (Plotter)renderer;
    plotter.moveTo(begin);
    plotter.drawTo(end);
    return true;
  }
}
```

However, because all `RendererCommands` may be executed on any `Renderer`, there is no compile-time guarantee that a `RendererCommand` intended for a `Plotter` is not executed on a `Rasterizer` instead. Therefore, this cast eliminates much of the natural static type checking afforded by the Java compiler when making direct method calls (and therefore creates more opportunity for programmer error).

In other words, this approach alone it does not specify a binding between the command and its intended receiver. To accomplish this, we add another type parameter to the `RendererCommand` as follows:[3]

```
public interface
         RendererCommand<T, R extends Renderer>
{
  T execute( R renderer );
}
```

4. TYPE SAFETY OF INVOKER PROXIES

The approach above forces the developer to make a trade-off between code duplication and static type checking. Specifically, there is no way to create a generic

[3]This idiom is mentioned in [4] in the discussion of parameterizing a generic `SimpleCommand` class (which contains nothing more than a function pointer) with the type of its receiver.

`RendererServiceProxy` that would guarantee static type checking for all `RendererService` types that it can represent.

In general, a specific `InvokerProxy` type must be made for every invoker that needs to be remotely accessible. Additionally, since these invoker types cannot share a common parent class/interface, the option to write generic code for invoker types. is removed. The alternative to this duplication is to depend on the code itself to perform dynamic type checking on every command invocation and to handle error cases.

To obviate the need for specialized proxy classes for each receiver, we can parameterize the generic invoker interface with its receiver type. In our example, this means making `RendererService` a generic type with the following declaration:

```
public interface
         RendererService <R extends Renderer>
{
  <T> T invoke(RendererCommand<T, R> command);
}
```

Using this approach, we can specify a generic `invoke()` method that will guarantee type safety for all `RendererCommand` types, regardless of return type or intended `Renderer` subtype. As noted above, the `Renderer` and `RendererService` implementation can still be combined in one class. For example, our `Rasterizer` implementation may look like the following:

```
/**
 * Plays the role of Invoker and Receiver
 * in the command pattern.
 */
public class      Rasterizer
       implements Renderer,
                  RendererService<Rasterizer>
{
  public <T>
    T call(RendererCommand<T, Rasterizer> command)
  {
    return command.call(this);
  }
}
```

In the development of *Moxaic*, this practice proved to be immensely useful in creating generic components for the Command pattern (such as proxies) and still maintaining all the type safety guarantees afforded by the Java compiler when invoking methods directly on objects.

5. CONCLUSION

As mentioned in the introduction, our initial design of *Moxaic* made extensive use of the command and Proxy patterns for remote procedure calls. This paper has considered several issues that arise in this context and discussed several possible solutions.

In general, we have shown that several generics idioms can be used to maintain type safety in an RPC framework implemented with the Command and Proxy patterns. That is, we have shown that, using the generics mechanism, the method invocation represented by the command can be made as type safe as any common method call.

In particular, we have discussed several type safety issues of the Command pattern and how they can be resolved in various ways. First, we have shown that there are problems with using an outbound parameter to specify a return type when serializing commands to be executed remotely. Second, we have shown that there are problems with using a wide return type when the Proxy pattern is used to enable remote accessibility. Third, we considered how the covariant return type feature of Java can be used to narrow return types and how the command type can be parameterized with its return type. Fourth, we have shown that the command type can be parameterized with its receiver type to overcome the type safety problem introduced by having generic command and invoker types. We have also shown how this solution can be extended by parameterizing the invoker type (and the proxy representing it) by its receiver type. Finally, we have shown how, by creating a generic invoker and command type and using casts when appropriate, generic RPC framework code could be shared for all commands/receivers.

Although the generics mechanism can certainly complicate otherwise straightforward code, these benefits far outweighed the cost in the implementation of *Moxaic*, are limited to the framework code and are generally not exposed to the API used by a client application. In future papers, we will discuss the *Moxaic* architecture in more detail and the roles played by other patterns.

6. ACKNOWLEDGMENTS

The authors would like to acknowledge the advice and comments of Chris Fox, Ori Ratner and David Smith.

7. REFERENCES

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2001.

[2] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. SIGPLAN: ACM Special Interest Group on Programming Languages, 1998.

[3] C. Fox. *Introduction to Software Engineering Design: Processes, Principles and Patterns with UML2*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2006.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns:Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[5] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification*. Prentice Hall, Upper Saddle River, 2005.

[6] A. Langer. Java generics frequently asked questions. http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html, 2007.

[7] M. Norton. Composing software design patterns. Master's thesis, James Madison University, 2003.

[8] M. Torgersen. The expression problem revisited : Four new solutions using generics. ECOOP, 2004.