An Implementation for Multiple Dispatch in Java using the ELIDE Framework

Peter Carbonetto
CPSC 539 Term Project
UBC Department of Computer Science
January 6, 2002

Introduction

Most object-oriented languages, such as Java and C++, provide dynamic message calling based on the type of the receiver object. This is called single or uni-receiver dispatch; the program makes a decision on which method to call based only on the type of the receiver object. This imposes restrictions on the programmer since there are situations where the program needs to select a method based on the type of several objects in a message. For most applications the ability to select behaviour based on a single argument is sufficient, but it happens occasionally that dispatching on two or more arguments is required. For example, in a graphical user interface events are passed to window objects in the application. When the application responds to a particular event, the action taken depends on both the type of the window and the type of event.

Ingalls gives an example where the program has a set of methods to display graphical objects (rectangle, oval, bitmap, etc.) on a set of display ports (monitor, printer, etc.). Each display port has a separate method to draw the different graphical objects. For example, the monitor class would have methods *displayRectangle*, *displayOval* and *displayBitmap*. With single dispatch, the programmer must design a general "display" method for the monitor class, and then create a dispatch mechanism to call the proper display method based on the type of the graphical object. His solution is to use relay methods [5]. This is a reasonable solution, except that it places a heavy burden on the programmer for figuring out the details of the method dispatch. Moreover, relaying methods increases in complexity exponentially, so the details of dispatch can become overwhelming.

If the programming language had a built-in mechanism to dispatch on multiple objects (i.e. multiple dispatch) then the programmer would not have to worry about how to call to the proper method. As long as the methods *display(Rectangle obj)*, *display(Oval obj)* and *display(Bitmap obj)* were defined for the monitor class, the appropriate method would be called based on the type of the argument.

A built-in mechanism for multiple dispatch allows the user to abstract the implementation details of method dispatching, as well as remove the asymmetry of class method ownership.

For this project, I implement a language construct for multiple dispatch in Java using the ELIDE package. The two principle objectives of the project are to create a mechanism for multiple dispatch using Java and to integrate it seamlessly into the language to allow people to apply the tool in an intuitive manner. In principle, the language extension should work under all circumstances, but there were a few obstacles posed by the ELIDE framework I was unable to overcome. These problems will be discussed later.

The ELIDE framework was designed at University of British Columbia for the purpose of adding high-level features by extending the vocabulary of the Java language. The modifier

```
dispatched<"name1, name2, ..., namen">
```

when used on a method call, designates the arguments on which to dispatch and changes the dispatching from static (i.e. overloading) to dynamic. Note that the method is already dispatched on the receiver object. Using the Ingalls example above, the programmer could then define a new class called *DisplayPortAndGraphicalObject* where all the multiple dispatch methods are located:

```
public class DisplayPortAndGraphicalObject
{
   public dispatched<> display (DisplayPort
                port, GraphicalObject obj);
   public display (Monitor port, Oval obj);
   public display (Printer port,
                  Rectangle obj);
   // etc.
}
```

What follows is a briefing on how to install and use the multiple dispatching package, a discussion of the issues in implementing multiple dispatch in Java, and finally the summary of the project.

Installing the Multiple Dispatch Package

This suite makes use of the ELIDE (Extension Language for Iterative Design Encoding) framework developed at the Software Practices Laboratory at University of British Columbia.[1] This project has been tested on Java Standard Edition version 1.2.

Once ELIDE is installed, create the class files by compiling the Dispatched.java source file. To do so, enter into the dispatch/src/ directory and open the Makefile for editing. Change the *elide* variable to point to the directory where the ELIDE software was installed. For example:

```
elide = /homes/grads2/pcarbo/project/elide/
```

Save the changes to the Makefile. Next, go into the elide directory (where the *elide.jar* file is located) and create a directory called *working* if it does not already exist. Now, go back to the dispatch/src/ directory and type "make" in the command line. This will build the Dispatched class file and place it in the elide/working/ directory. Now, this package can be used to generate Java source code files.

Generating Code Using the Multiple Dispatch Package

Assume for now that your Java source code file has been created and is located in a single directory. (Note: An explanation of the syntax and use of the *dispatched* keyword is given below.) To generate code for dispatching, follow the steps detailed here.

First, run Java using the *elide.jar* file on the Java source code. This will create a new set of source files, removing all instances of the *dispatched* keyword and replacing them with native Java code.

```
java -jar $(elide)elide.jar *.java -d
./output -P dispatch
```

where *$(elide)* is the directory specifying the location of the *elide.jar* file. This will create new source files and place them in the in the *output* subdirectory. The *-P dispatch* specifies the ELIDE transformation package, and should be left as is. Finally, compile the files into Java classes.

```
javac output/*.java
```

Now the java program located in the output/ subdirectory can be executed normally.

Using the *dispatched* Modifier

The *dispatched* keyword is a modifier for class method declarations, similar to such modifiers as *public*, *static* and *synchronized*. Like other modifiers, it is placed before the return type in a method declaration. The order in which the modifiers are written – including *dispatched* – does not matter. The method with the *dispatched* modifier acts as a template or prototype for the subordinate dispatch methods declared in the same class and its subclasses. What this means is that all methods of the same signature (i.e. same method name, same return type, and same non-polymorphic parameter types) will be dynamically dispatched on the specified parameter types. The template specifies the root class for each polymorphic parameter, so all methods must dispatch on non-strict subtypes of the template parameter types.

```
class Window {

   public dispatched<> boolean handleEvent
         (Event evt);

   public boolean handleEvent
         (KeyEvent evt) {
      // Body is omitted.
   }

   public boolean handleEvent
         (UpdateEvent evt) {
      // Body is omitted.
   }
}
```

Adding the *dispatched* modifier to more than one method of the same signature will cause a compile-time error since it could introduce ambiguities in dispatch behaviour. Therefore, the *dispatched* keyword cannot be used more than once for a particular method signature in a class hierarchy.

The *dispatched* keyword takes one optional argument. With no arguments, as in the above example, the method is dynamically dispatched based on the object type of all the method's parameters. The optional argument specifies which parameters are used in the dynamic selection of a

[1] Before using this software package, you must download and install ELIDE. See the ELIDE website at http://www.cs.ubc.ca/labs/spl/projects/elide/

method. This argument is comma-delimited string of the names of the polymorphic parameters, in no particular order. If we were to explicitly specify which parameters are used in the above example, the method declaration would look like this:

```
public dispatched<"evt">
Boolean handleEvent (Event evt);
```

The classes used for the polymorphic parameters must be declared in the same package and must be accessible by ELIDE. Therefore, the user must define all the classes at the same time, although not necessarily in the same source files. For future considerations, this limitation could be removed if this package also took advantage of the java.lang.reflect library to get information about previously compiled classes.

The body of the *dispatched* method must be left empty because it will be filled in with the dispatch mechanism (unless it is declared *abstract*).

Also note that the modifiers for the *dispatched* method cannot be altered within a single class hierarchy. For example, if the prototype method is declared *public* then the access cannot be changed to *protected* later on. This is a restriction imposed purely for ease of implementation, but it should not pose strong constraints on Java programs. Other restrictions on method modifiers, imposed to avoid inconsistent behaviour, include: 1) if the prototype *dispatched* method is declared *static*, then all the methods matching the prototype must also be *static*; 2) no methods other than the prototype may have the *abstract* keyword. The *synchronized* and *volatile* keywords should work, although this hasn't been verified. The access modifier for all methods will automatically be set to the prototype's access modifier.

If a method is not defined for a particular polytype (a polytype is defined as a set of class types), the method defined on a parent polytype is used. If there are no methods defined on the parent polytypes then the program throws an *UnsupportedOperationException*. On the other hand, if methods are defined for more than one parent polytype then the method definition is ambiguous and the program reports a runtime exception. More precisely, we can define the conditions that will result in an ambiguity on one or more polytypes. Given method *i* defined on polytype P_I, method *j* defined on polytype P_j, and the set of all polytypes $P = \{P_1, P_2, \ldots, P_n\}$, an ambiguity will not occur if and only if one of the following conditions hold:

1. P_i is a parent polytype of P_j
2. P_j is a parent polytype of P_i
3. $P_u \in P \mid$ (P_u is a sub-polytype of P_i) and (P_u is a sub-polytype of P_j) and ($P_v \in P \mid P_v$ is a sub-polytype of P_i) and (P_v is a sub-polytype of P_j) (P_v is a sub-polytype of P_u))

It is possible to declare a *dispatched* method as *abstract*, in which case the methods for dispatching can be defined in a subclass of the abstract class. Note that one must not explicitly override the *dispatched* method definition since, as mentioned previously, it should only be defined once in a single class hierarchy. To understand why this is, subclasses of the abstract class will automatically create a dispatch method to override the abstract *dispatched* method, assuming at least one method is defined in the subclass with the same signature. At this point in time, the *dispatched* keyword is not compatible with interfaces since I had some problems incorporating interfaces into the class hierarchy using ELIDE.

Discussion of Implementation

ELIDE was a very useful tool. It provided functionality for most of what I needed for implementing multiple dispatch. There were some features I felt were lacking and at times I was frustrated with the tool, but I was able to overcome most obstacles in a roundabout way. First, I will point out a few limitations to ELIDE in hopes that it will motivate future improvement.

Indentation was a small problem which hopefully could be fixed in a future release of the ELIDE package. All the code was nicely indented, with the exception the code in the method bodies. The source code for each method body was only one line long, so I was required to manually indent the generated code in order to properly read it.

A problem of greater severity, I found a peculiar bug in the ELIDE package: it removes all methods that do not have a return type. This is problematic for creating class constructors since they are not required to have return types; they are implicitly void. I solved this by adding the "void" return type to the class constructors.

When generating procedures, the body was occasionally removed from the method definition and placed in the body of the class. I have not

resolved this problem, so the user has to manually open up each generated source file and copy and paste the bodies of the methods back into their respective method calls. This is definitely an inconvenience, and should be fixed in the future.

For the purposes of the dispatching mechanism, I attempted to catch *ClassCastException* exceptions that arise due to the optimization of the dispatch mechanism, and re-throw them as exceptions of type *UnsupportedOperationException*. In this way, the user would get a consistent exception thrown for a failed dispatch. However, ELIDE has trouble dealing with *try/catch* control statements. Therefore, I gave up that attempt, but in the future I hope to implement better exception handling.

I encountered some problems in working with interfaces, due to the fact that I was unable to find a way to get a list of classes that implement an interface. Abstract classes do however function properly in my implementation. Also, I found that the *getSuperclasses()* function does not work properly in ELIDE. I got around this by using some other methods, but it is less efficient.

I relied on *if/else* statements to dispatch methods based on the dynamic types on arguments, where a single statement dispatched on the dynamic type of a single parameter. Thus, if the polytypes are of dimension *n*, there are *n* levels of *if/else* statements in the dispatch mechanism. All the declared subordinate methods are automatically declared as *private* so the user doesn't call them directly.

There is definitely room to optimize the dispatch mechanism in the future, since the current implementation may produce superfluous *if/else* statements. Since I ran out of time, I will leave optimization for future improvement.

Ambiguous method definitions are reported at runtime, but it would be useful to warn the user at compile time that certain parameter types may cause exceptions and report which method definitions could most efficiently satisfy the conditions for removal of ambiguous method calls. Again, I leave this for future improvement.

For practical considerations, I did not implement the relay methods as proposed by [5]. Relay methods are less complex and more efficient because they rely on Java's native dispatch mechanism. I did not implement multiple dispatch this way because ELIDE is limited in which classes it can modify – it can only alter those in source code, and not the classes that have already been compiled into class files. Relay methods require adding relay methods to the classes for which the dispatch mechanism depends on. For this implementation, there is no guarantee that the polymorphic parameter types can be modified by ELIDE. With the help of the java.lang.reflect library, relay methods could be implemented.

Conclusion

Implementing multiple dispatch proved to be a challenge due to the complexities of the Java programming language. Using the ELIDE framework, I extended the Java vocabulary with a method modifier, *dispatched*, that allows programmers to practically and intuitively implement multiple polymorphism without having to worry about the details of the dispatch mechanism. In detail, I described how to use the *dispatched* keyword. Finally, I discussed some major issues involved in implementing multiple dispatch in the Java programming language and offered suggestions for future improvement.

References

[1] Clifton, Curtis, Gary T. Leavens, Craig Chambers and Todd Millstein. *MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java.*

[2] Conway, Damian. *Multiple Dispatch and Subroutine Overloading in Perl.*

[3] Driesen, Karel. *Multiple Dispatch Techniques: a survey.* Student Writing Contest of the Society for Technical Communication, 1996.

[4] Holst, Wade. *The Tension between Expressive Power and Method-Dispatch Efficiency in Object-Oriented Languages.* Ph.D. Thesis, 2000.

[5] Ingalls, Daniel H. H. *A Simple Technique for Handling Multiple Polymorphism.* OOPSLA '86 Proceedings, September 1986, p. 347-9.