Essay Concepts of Programming Languages
Delegates

Christian Luijten (496505) & Paul van Tilburg (459098)

2nd April 2004

Contents

1 Introduction

The delegate is a language construction first introduced by Microsoft in their implementation of Java, Visual J++ 6.0. When it was introduced, it brought up many discussions whether or not the delegate is a good and clean construct in programming languages, especially in Java. We will not go into these discussions thoroughly and leave the political issues as they are.

This document will explain what the delegate construct exactly is in the next paragraph. After that some alternatives for delegates are presented, followed by a short discussion of the advantages and disadvantages. As in the past years the delegate came to appear in more languages, an overview will be presented afterwards.

2 What are delegates?

A common way to look at delegates is to see them as function pointers (see also Section 3.1), but they are actually much more than that. The delegate introduces a type-safe, object-oriented reference to methods. These two 'new' aspects are explained in the following two parapgrahs.

2.1 Type-safety

Delegates are type-safe. This means that the compiler and in this case also the virtual machine will check and enforce type-safety when delegates are used. Since the delegate refers to a method, this method must match the delegate in the following ways:

1. The method must have the same number and type of arguments,
2. the method must have the same return type, and
3. the method must throw the same exceptions.

2.2 Object-oriented reference

Because delegates are objects too, they fit nicely into the object oriented languages as Java and C#. Delegate declarations are compiled into classes and delegate instances are just as any other instance.

The delegate instances encapsulate methods and thus function as references which can be invoked. In some languages (C# for example, see Section 5.1) there are two types of delegates:

bound method reference means that the method pointer that is encapsulated is bound to an instance of the object.

unbound method reference means that the method pointer is a static one. For example a pointer to a (static) class method[1].

[1] This is more like normal function pointers.

2.3 Delegation pattern

The name delegate refers to the often used concept in object oriented programming: delegation. It means that a task is handled over from one object to another.

This is related to the *delegation pattern* [WKP-DLGPAT] which is an technique where an object outwardly expresses certain behaviour but actually delegates it and the responsibility to another object.

This pattern is often occurs in event-handling. Logically delegates are very easy to use here. Compontents can have a list of delegates, thus objects encapsulating method references which need to be called when an event occurs. This is in an example in the next section (see Subsection 2.4):

2.4 Examples

This section illustrates the use of delegates. Both examples are written in Microsoft Visual J++ 6.0 and are derived from the J++ tutorial [MSDN-DLGTUT]. The first example shows the use of a delegate in the classic *generic sorting* setting[2].

Listing 1: Examples of delegate use in Micosoft Visual J++

```
// Define Comparer class that extends Delegate class.
delegate int Comparer(Object a, Object b);

// Create function to match Comparer delegate signature.
int stringCompare(Object a, Object b) {
  String x = (String) a;
  String y = (String) b;
  return x.compareTo(y);
}

void test() {
  // Create & instantiate delegate with
  // defined stringCompare function.
  Comparer c = new Comparer(this.stringCompare);
  String[] stringArr = ...

  // Call sorter function with array and delegate
  // which will be called by c.invoke(elem1, elem2)
  // for elements in stringArr.
  Sorter.sort(stringArr, c);

  ...
}
```

The next example uses the delegate in a more straightforward way by just encapsulating the method reference to an *event handler*.

3 Delegate alternatives

Although the delegation pattern (Section 2.3) is common to programming, the delegate construction is quite new. Clearly because of this, older languages must

[2]Often constructed by a generate sorting routine which determines the order of the elements to be sorted by a seperate function.

Listing 2: Use of delegates for event handling

```
// Define MouseEventHandler class that extends Delegate class.
multicast delegate void MouseEventHandler(Object sender,
                                          MouseEvent e);

// Create a button on the form.
Button myButton = new Button();

// Function that can handle an mouse event.
void myButton_mouseMove(Object sender, MouseEvent e) {
  p("mouse moved to " + e.x + "," + e.y);
}

void initForm() {
  // Couple the myButton_mouseMove method as event
  // handler by creating delegate MouseEventHandler.
  MouseEventHandler me =
    new MouseEventHandler(this.myButton_mouseMove);

  // Add the delegate (thus method reference) as addOnMouseMove
  // event handler.
  myButton.addOnMouseMove(me);

  ...
}
```

have alternatives and some new language choose not to implement it this way (as discussed in Section 4). Some of the alternatives are discussed in the following subsections.

3.1 The function pointer

The most straightforward used to delegate functionality and control in older languages is the use of the *function pointer* or reference[3]. The function pointer, available in for example C, Pascal, Modula, etc., refers to a function at other position in the code. This binding is static and can be checked compile-time for errors. Although type-safety can be ensured, the pointer brings along all the disadvantages of pointers.

3.2 Java inner classes

In Java it is possible to define classes within classes, which are called *inner classes*. These inner classes can be used to define methods with functionality to which for example handling an event can be delegated. This is illustrated in the following listing, which is an alternative implementation of the example given in Listing 2 (page 4).

[3]From the perspective of the delegate the reference is just a smart pointer and does not ensure complete type-safety and object-orientation as the delegate does.

Listing 3: Event handling with inner class instead of delegate

```
// Create a button on the applet.
JButton myJButton = new JButton();

// Inner class implementing the MouseMotionListener interface
// to handle mouse motion events.
class MyMouseMotionListener implements MouseMotionListener {
  public void mouseMoved(MouseEvent e) {
    println("mouse moved to " + e.x + ",", + e.y);
  }
}

public void init() {
  // Create instance of the MyMouseMotionListener
        // class for handling motion.
  MyMouseMotionListener ml = new MyMouseMotionListener();

        // Add the MyMouseMotionListener object to the list
        // of object that should get and handle the event.
  JButton.addMouseMotionListener(m;);

  ...

}
```

3.3 Dynamic bound delegates

Scripting languages often provide means of delegation. Since most scripting languages aren't strongly typed and most of the objects are dynamically bound, the delegate alternatives are dynamically bound too.

Although there are numerous scripting languages and delegation constructions therein, two examples of the same sort will be presented: the lambda construction in Python and the Proc block/iterator in Ruby. Both examples will increment elements of an array with 1.

Python features a construction to create a generic function which can be passed as an argument. This generic function is defined by the lambda construction analogue to the use in lambda calculus[4].

Listing 4: Incrementing array elements using the lambda construction in Python

```
# Create example array of elements 10, 20, 30
ar  = [10, 20, 30]

# Define generic function which takes one argument and
# increments it.
inc = lambda x: x + 1

# Apply the 'inc' lambda-function to all elements of array ar.
sar = map(inc, ar)
```

[4]See also `http://en2.wikipedia.org/wiki/Lambda_calculus` for more information

Ruby allows the programmer to create an object of a block of code, which for example can be passed to an iterator function.

Listing 5: Increment array elements using Proc object in Ruby

```
# Create example array of elements 10, 20, 30
ar = [10, 20, 30]

# Create Proc object from a code block which takes one argument
# and returns it increment.
inc = Proc.new do |x| x + 1 end

# Pass Proc object code as code block to map iterator function.
sar = ar.map(&inc)
```

As in Ruby everything is an object, the code block (a Proc object) referenced by `inc` is an object too. When a Proc object is used, the arity is checked. This makes the Ruby Proc object come very close to a delegate construction except that here the code is encapsulated instead of the method reference.

4 Advantages & disadvantages of delegates

When Microsoft introduced delegates in Visual J++, Sun Microsystems objected to implementing them in Java.

Sun states that "bound method references are unnecessary and detrimental to the language" in their white paper About Microsoft's 'delegates'[SUN-DLG]. Microsoft responds in the article The Truth About Delegates[MSDN-DLG]: "WFC[5]'s delegate-based event model results in more concise, more readable, and conceptually simpler source code than the JavaBeans' interface-based event model.".

4.1 Disadvantages

Sun Microsystems decided already in 1996 not to include delegates in the Java language. They did this after careful consideration and with consultation of Borland, who used bound method references in the Delphi Object Pascal language.

Delegates are according to Sun:

- Unnecessary and a detriment to the language, because of the already existing *inner class* (see Section 3.2), about which Sun says they "provide equal or superior functionality".
- Harming the simplicity of the Java language; they cause irregularity in the syntax and scoping rules and affect the object-orientedness of the language.
- Adding complexity to the language. The type system must include an "entirely new type" as the Delegate class cannot be extended like any other class, nor can they implement an interface. "New language rules are required" to match the expressions.

All citations come from [SUN-DLG].

[5] Windows Foundation Class

4.2 Advantages

Microsoft agrees with Sun about the need for the support of pluggable API's. However, they of course don't agree on the arguments about delegates brought forth by Sun.

On Sun's critics, Microsoft responds:

- Inner classes can make programs unreadable, while delegates can avoid this and can be very readable.
- Delegates can easily be fit into existing languages and VMs. There are no new rules needed in the Java language, neither is there any need for a completely new type as all delegates are simply an instance of the class Delegate. The fact that the Delegate class cannot be extended is not unique; the classes Array and String also are non-extendible.

4.3 Evaluation

Sun is right in stating that delegates can be used to create totally unreadable and badly programmed code. However, examples Microsoft gives show that delegates can also be used in a well structured manner.
The fact that code can be unreadable under certain circumstances doesn't make a construction bad, but the ease with which the code is made unreadable *does*.

Delegates can be implemented both efficient and portable. There is no reason at all why a VM[6] could not properly support them. Breakage with older VMs should not be an argument for holding back innovations.

5 Delegates in other languages

More and more languages make use of the delegate. These languages include C# and Visual Basic .NET.

5.1 Delegates in C#

C# is a derivate language of C++. The delegate obsoletes most of the purpose of function pointers which are, of course, all but object oriented.

A function pointer is merely a memory address of a certain (callback-)function, it can not carry information about the interface, like the number of parameters or the types of these parameters. Delegates are objects, which are able to carry these informations.

Recall Listing 1 sorting an array of strings using a delegate function. Listing 6 is the equivalent written in C#.

5.2 Delegates in Visual Basic .NET

The newest version of Visual Basic, VB.Net supports delegates. Listing 7 demonstrates the usage of delegates in VB.Net. However, the readability of Visual Basic is so poor, it is quite a task to follow the program flow.

[6] Virtual Machine

Listing 6: Example of delegate use in Microsoft C#

```
// Define Comparer class that extends Delegate class.
public delegate void Comparer(Object a, Object b);

// Create function to match Comparer delegate signature.
int stringCompare(Object a, Object b) {
  String x = (String) a;
  String y = (String) b;
  return x.compare(y);
}

void test() {
  // Create & instantiate delegate with
  // defined stringCompare function.
  Comparer c = new Comparer(this.stringCompare);
  String[] stringArr = ...

  // Call sorter function with array and delegate
  // which will be called by c.invoke(elem1, elem2)
  // for elements in stringArr.
  Sorter.sort(stringArr, c);

  ...
}
```

Listing 7: Example of delegate use in Microsoft VB.Net

```
Public Class DelegateDemo
  'Creates a delegate object called onClick'
  Public Delegate Sub onClick(ByVal messageString As String)

  'Defines a delegated function clickDelegate'
  Public Sub clickDelegate(ByVal messageString As String)
    Console.WriteLine(messageString)
  End Sub

  'clickDel is now a delegate to clickDelegate sub'
  Dim clickDel As onClick = AddressOf clickDelegate

  'Call clickDel to use the delegate'
  Public Sub test()
    clickDel('Hi')
  End Sub
End Class
```

6 Conclusion

The delegate has some interesting properties but also brings along some disadvantages if the construction is added to a language. In practice the delegate is a convenient construction but is less elegant in a program languange sense.

As conclusion we will assess the readability, writability and reliability factors.

6.1 Readability

In fully object oriented the delegate is a rather strange construction compared to the rest. Mostly because it defines a subclass implicitly which has both a return type and argument types enforced on methods that are encapsulated. It features a more compact syntax than for example inner classes though.

6.2 Writability

Using delegates enhances expressivity and abstraction. It is very easy to construct a method and make it passable by using the delegate. Complete functionality can be abstracted from by a single delegate with the encapsulated method reference.

6.3 Reliability

The delegate is completely type-safe and so type-checking can be enforced. Because also the definedness can be checked compile-time, the delegate will rule out a lot of errors that one would normally have with function references/pointers.

References

[CSC-VB.NET] Pramod Singh – Delegates in VB.NET `http://www.c-sharpcorner.com/vbnet/delegatesinvb.asp`

[DPB] Dumky's programming blog – The dark side of C# Delegates `http://blog.monstuff.com/archives/000037.html`

[MSDN-DLG] MSDN – The Truth About Delegates `http://msdn.microsoft.com/vjsharp/productinfo/visualj/visualj6/technical/articles/general/truth/default.aspx`

[MSDN-DLGTUT] MSDN – Delegates in Visual J++ 6.0 `http://msdn.microsoft.com/library/en-us/dnjpp/html/msdn_delegates.asp?frame=true`

[MSDN-DOTNET] MSDN – .NET, an introduction to Delegates `http://msdn.microsoft.com/msdnmag/issues/01/04/net/default.aspx`

[PYTH-REF] Nikos Drakos, Ross Moore – The Python Reference Manual `http://www.python.org/doc/1.6/ref/lambda.html#lambda`

[RUBYBOOK] David Thomas & Andrew Hunt – Programming Ruby `http://www.rubycentral.com/book/tut_containers.html`

[SUN-DLG] The Java^{TM} Language Team – About Microsoft's "Delegates" `http://java.sun.com/docs/white/delegates.html`

[WKP-DLG] Wikipedia – Delegation `http://en.wikipedia.org/wiki/Delegation`

[WKP-DLGPAT] Wikipedia – The delegation pattern `http://en.wikipedia.org/wiki/Delegation_pattern`