Program specialization and metaprogramming

Laboratory work No. 11

Advanced Aspect-oriented Programming with AspectJ

1. Aims

To get acquainted with the advanced concepts of Aspect Oriented programming.

2. Tasks

Learn how to do resource-pool management modularization using AspectJ. Learn how to do policy-enforcement modularization using AspectJ. Learn how to do characteristic-based implementation modularization using AspectJ. Learn how to do implement flexible access control using AspectJ.

3. Work

A. Resource-pool management modularization using AspectJ

One common way to optimize resource usage recycles previously created resources such as threads and database connections. Let's see how AOP and AspectJ help implement resource-pool management in a modularized fashion.

First, let's implement a simple TCP/IP service for converting requested strings to uppercase. The server creates a new thread each time a new connection request arrives. Once a thread completes serving a connection, it terminates naturally. The implementation is simple Java without any AspectJ constructs:

```
// UppercaseServer.java
import java.io.*;
import java.net.*;
public class UppercaseServer {
   public static void main(String[] args) throws Exception {
    if (args.length != 1) {
       System.out.println("Usage: java UppercaseServer <portNum>");
       System.exit(1);
    }
    int portNum = Integer.parseInt(args[0]);
    ServerSocket serverSocket = new ServerSocket(portNum);

     while(true) {
       Socket requestSocket = serverSocket.accept();
       Thread serverThread
          = new Thread(new UppercaseWorker(requestSocket));
       serverThread.start();
     }
   }
}
class UppercaseWorker implements Runnable {
   private Socket _requestSocket;
   public UppercaseWorker(Socket requestSocket) throws IOException {
     System.out.println("Creating new worker");
     _requestSocket = requestSocket;
   }
   public void run() {
```

```
      BufferedReader requestReader = null;
      Writer responseWriter = null;
      try {
         requestReader
            = new BufferedReader(
                new InputStreamReader(_requestSocket.getInputStream()));
         responseWriter
            = new OutputStreamWriter(_requestSocket.getOutputStream());
         while(true) {
            String requestString = requestReader.readLine();
            if (requestString == null) {
               break;
            }
            System.out.println("Got request: " + requestString);
            responseWriter.write(requestString.toUpperCase() + "\n");
            responseWriter.flush();
         }
      } catch(IOException ex) {
      } finally {
         try {
            if (responseWriter != null) {
               responseWriter.close();
            }
            if (requestReader != null) {
               requestReader.close();
            }
            _requestSocket.close();
         } catch (IOException ex2) {
         }
      }
      System.out.println("Ending the session");
   }
}
```

Next, let's see how AspectJ can add a thread-pooling crosscutting concern. First, write a simple `ThreadPool` class that acts as a stack for available threads. The `get()` method extracts a thread from the stack, whereas the `put()` method pushes one in. The `put()` method also contains the `DelegatingThread` class that delegates the `run()` method to the `_delegatee` worker object:

```
// ThreadPool.java
import java.util.*;
public class ThreadPool {
   List _waitingThread = new Vector();
   public void put(DelegatingThread thread) {
      System.out.println("Putting back: " + thread);
      _waitingThread.add(thread);
   }
   public DelegatingThread get() {
      if (_waitingThread.size() != 0) {
         DelegatingThread availableThread
            = (DelegatingThread)_waitingThread.remove(0);
         System.out.println("Providing for work: " + availableThread);
         return availableThread;
      }
      return null;
   }
   static class DelegatingThread extends Thread {
      private Runnable _delegate;
      public void setDelegatee(Runnable delegatee) {
         _delegatee = delegatee;
      }
```

```
    public void run() {
      _delegatee.run();
    }
  }
}
```

You now possess the classes needed to add thread pooling. Next, write an aspect that uses the `ThreadPool` class to add thread pooling to the server:

```
// ThreadPooling.java
public aspect ThreadPooling {
  ThreadPool pool = new ThreadPool();
  //==========================================================================
  // Thread creation
  //==========================================================================
  pointcut threadCreation(Runnable runnable)
    : call(Thread.new(Runnable)) && args(runnable);
  Thread around(Runnable runnable) : threadCreation(runnable) {
    ThreadPool.DelegatingThread availableThread = pool.get();
    if (availableThread == null) {
      availableThread = new ThreadPool.DelegatingThread();
    }
    availableThread.setDelegatee(runnable);
    return availableThread;
  }
  //==========================================================================
  // Session
  //==========================================================================
  pointcut session(ThreadPool.DelegatingThread thread)
    : execution(void ThreadPool.DelegatingThread.run()) && this(thread);
  void around(ThreadPool.DelegatingThread thread) : session(thread) {
    while(true) {
      proceed(thread);
      pool.put(thread);
      synchronized(thread) {
        try {
          thread.wait();
        } catch(InterruptedException ex) {
        }
      }
    }
  }
  //==========================================================================
  // Thread start
  //==========================================================================
  pointcut threadStart(ThreadPool.DelegatingThread thread)
    : call(void Thread.start()) && target(thread);
  void around(Thread thread) : threadStart(thread) {
    if (thread.isAlive()) {
      // wake it up
      synchronized(thread) {
        thread.notifyAll();
      }
    } else {
      proceed(thread);
    }
  }
}
```

Let's examine the implementation in detail:

- Pointcut threadCreation() captures joinpoints, thus creating a new thread object taking a Runnable object as the argument.
- Advise the threadCreation() pointcut to first check the thread pool for available threads. If no thread is available, create a new one. In either case, set the delegatee to the Runnable object passed in and return that object instead. Note, you do not call proceed() in this advice, so therefore you never execute the captured operation.
- Pointcut session() captures the run() method's execution of any ThreadPool.DelegatingThread objects.
- By putting session() inside a while(true) loop, you advise session() to never finish the servicing. That ensures a thread, once created, never dies. Once a request is processed, you put the thread back into thread pool and put the thread into waiting state.
- Pointcut threadStart() captures a call to the Thread.start() method. It uses isAlive() to check if the thread previously started. That would happen for a thread obtained from a pool and now in a waiting state. Wake up the thread by notifying it. If the thread had not started yet, as for a freshly created thread, proceed with starting the thread.

You can use the same technique for pooling database-connection objects. Simply capture joinpoints that create new connections and advise them to use one from a connection pool instead, if an appropriate connection is available. You also need to capture joinpoints that close connection objects and advise them to instead put those objects back in the resource pool.

You can run example start two or more command shell. In one command shell, start the server by issuing:

java UppercaseServer <port-number>

For example,

java UppercaseServer 10000

In other command shells, start clients by issuing

java UppercaseClient <servername> <port-number>

For example,

java UppercaseClient localhost 10000

To test pooling aspect, kill a client (by typing ^C), start a new one, and observe messages printed in command shell running the server.

B. Policy-enforcement modularization

In this work, you'll implement policy enforcement to ensure no duplicate listener objects are added to the models, and listeners do not loiter around when the view they represent becomes usable. Since implementing both concerns requires capturing joinpoints that add listeners to models, you share the code by creating a base abstract EventListenerManagement aspect. This aspect contains an addListenerCall() pointcut that captures calls to methods adding a listener. It uses the modelAndListenerTypeMatch() pointcut to let derived aspects restrict methods captured by addListenerCall():

```
// EventListenerManagement.java
import java.util.*;
public abstract aspect EventListenerManagement {
   pointcut addListenerCall(Object model, EventListener listener)
      : call(void *.add*Listener(EventListener+))
      && target(model) && args(listener) && modelAndListenerTypeMatch();
   abstract pointcut modelAndListenerTypeMatch();
}
```

The uniqueness concern, before adding any listener, checks whether that listener was previously added. If that listener is already present, the operation does not proceed; otherwise, it adds the listener. The EventListenerUniqueness aspect implements the concern's core functionality, thus ensuring no duplicate listener objects in a model. That abstract aspect declares EventListenerManagement as its parent. It advises the addListenerCall() pointcut to check for the listener's uniqueness by looking in a list obtained by invoking getCurrentListeners(). It proceeds with adding the listener only if the list doesn't include a listener:

```
// EventListenerUniqueness.java
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.table.*;
public abstract aspect EventListenerUniqueness
   extends EventListenerManagement {
   void around(Object model, EventListener listener)
      : addListenerCall(model, listener) {
      EventListener[] listeners = getCurrentListeners(model);
      if (!Utils.isInArray(listeners, listener)) {
         System.out.println("Accepting " + listener);
         proceed(model, listener);
      } else {
         System.out.println("Already listening " + listener);
      }
   }
   public abstract EventListener[] getCurrentListeners(Object model);
}
```

The concrete aspect TableModelListenerUniqueness extends EventListenerUniqueness to apply the aspect to TableModel and related classes. It provides an implementation for the modelAndListenerTypeMatch() pointcut to restrict the model type to AbstractTableModel and the listener type to TableModelListener. Another concrete aspect, ListDataListenerUniqueness, does the same for list-related classes. For ListDataListenerUniqueness's source code, see course Moodle page.

```
// TableModelListenerUniqueness.java
import java.util.EventListener;
import javax.swing.event.TableModelListener;
import javax.swing.table.*;
aspect TableModelListenerUniqueness extends EventListenerUniqueness {
   pointcut modelAndListenerTypeMatch()
      : target(AbstractTableModel) && args(TableModelListener);
   public EventListener[] getCurrentListeners(Object model) {
      return ((AbstractTableModel)model)
         .getListeners(TableModelListener.class);
   }
}
```

Each concrete aspect, TableModelListenerUniqueness, handles the listener associated with a model type. Compiling your source code along with these aspects will ensure no duplicate listener is added.

Next, you implement a no loitering-views concern. Instead of adding listeners to a model directly, you can wrap it as a referent in a WeakReference object and add it. Since the added object must be of the correct listener type, use the Decorator pattern in a class extending WeakReference and implementing the required listener interface. The decorator implements the listener interface by delegating each method to the referent object.

The EventListenerWeakening aspect implements the concern's core functionality, thus ensuring that no view loiters after its destruction. That abstract aspect also advises addListenerCall() to proceed with the listener obtained by calling the getWeakListener() method:

```
// EventListenerWeakening.java
import java.lang.ref.*;
import java.util.*;
import javax.swing.event.*;
public abstract aspect EventListenerWeakening
  extends EventListenerManagement dominates EventListenerUniqueness {
  void around(Object model, EventListener listener)
    : addListenerCall(model, listener) {
    proceed(model, getWeakListener(listener));
  }
  public abstract EventListener getWeakListener(EventListener listener);
}
```

The WeakEventListener decorates the EventListener in addition to extending WeakReference. The nested RemoveGarbageCollectedListeners aspect removes WeakEventListener from the model when it detects that the referent is garbage collected. Here you check the collected referent in an event notification method.

```
// EventListenerWeakening.java (contd...)
public abstract class WeakEventListener extends WeakReference
  implements EventListener {
  public WeakEventListener(EventListener delegatee) {
    super(delegatee);
  }
  public EventListener getDelegatee() {
    return (EventListener)get();
  }
  public boolean equals(Object other) {
    if (getClass() != other.getClass()) {
      return false;
    }
    return getDelegatee() == ((WeakEventListener)other).getDelegatee();
  }

  public String toString() {
    return "WeakReference(" + get() + ")";
  }

  abstract static aspect RemoveGarbageCollectedListeners {
    pointcut eventNotification(WeakEventListener weakListener,
                               EventObject event)
      : execution(void WeakEventListener+.*(EventObject+))
      && this(weakListener) && args(event)
      && lexicalScopeMatch();
    abstract pointcut lexicalScopeMatch();

    public abstract void removeListener(EventObject event,
                                        EventListener listener);
    void around(WeakEventListener weakListener, EventObject event)
```

```
        : eventNotification(weakListener, event) {
        if (weakListener.getDelegatee() != null) {
          proceed(weakListener, event);
        } else {
          System.out.println("Removing listener: " + weakListener);
          removeListener(event, weakListener);
        }
      }
    }
}
```

The abstract lexicalScopeMatch() pointcut ensures only methods from specific WeakListener types are captured. Without this pointcut, execution of all WeakListeners notification would be caught -- for example, table as well as lists. Since removeListener() would cast arguments to specific types, a ClassCastException would throw.

The EventListenerWeakening aspect dominates EventListenerUniqueness, which causes EventListenerWeakening's advice to addListenerCall() to be applied before that of EventListenerUniqueness. That way, you create the wrapped listener first, and the uniqueness check using WeakEventListener.equals() works correctly. Since listeners added in a model are WeakReference wrapped, comparing bare listeners would cause a comparison using equals() to not match.

The TableModelListenerWeakening aspect handles table-related listeners. It uses a specialized WeakEventListener that implements TableModelListener by delegating to the referent object. You'll find the code for the aspect that handles the list-related listener in ListModelListenerWeakening.java:

```
// TableModelListenerWeakening.java
import java.util.*;
import javax.swing.event.*;
import javax.swing.table.*;
public aspect TableModelListenerWeakening extends EventListenerWeakening {
  pointcut modelAndListenerTypeMatch()
    : target(AbstractTableModel) && args(TableModelListener);
  public EventListener getWeakListener(EventListener listener) {
    System.out.println("Weakening " + listener);
    return new WeakTableModelListener((TableModelListener)listener);
  }
}
public class WeakTableModelListener extends WeakEventListener
  implements TableModelListener {
  public WeakTableModelListener(TableModelListener delegatee) {
    super(delegatee);
  }

  public void tableChanged(TableModelEvent e) {
    TableModelListener listener = (TableModelListener)getDelegatee();
    listener.tableChanged(e);
  }

  static aspect TableRemoveGarbageCollectedListeners
    extends WeakEventListener.RemoveGarbageCollectedListeners {

    pointcut lexicalScopeMatch() : within(WeakTableModelListener);
    public void removeListener(EventObject event, EventListener listener) {
      ((TableModel)event.getSource())
        .removeTableModelListener((TableModelListener)listener);
    }
  }
}
```

With the aspects, you've created a crosscutting concern to avoid multiple notifications to the same model listener, as well as avoided loitering view objects. The simple Test.java tests the functionality. The aspects use the utility Utils.java class.

C. Characteristic-based implementation modularization

Operations with the same characteristics should typically implement common behaviors. For example, a wait cursor should be put before any slow method executes. Similarly, you may need to authenticate access to all security-critical data. Since such concerns possess a crosscutting nature, AOP and AspectJ offer mechanisms to modularize them. Because a method's name might not indicate its characteristics, you need a different mechanism to capture such methods. To create such a mechanism, declare the aspect adding characteristic-based crosscutting behavior as an abstract aspect. In that aspect, declare an abstract pointcut for methods with characteristics under consideration. Finally, write an advice performing the required implementation.

Consider SlowMethodAspect's implementation. It declares the abstract slowMethods() pointcut and advises it to first put a wait cursor, proceed with the original operation, and finally restore the original cursor. Here's the code:

```
// SlowMethodAspect.java
import java.util.*;
import java.awt.*;
import java.awt.event.*;
public abstract aspect SlowMethodAspect {
   abstract pointcut slowMethods(Component uiComp);
   void around(Component uiComp) : slowMethods(uiComp) {
      Cursor originalCursor = uiComp.getCursor();
      Cursor waitCursor = Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR);
      uiComp.setCursor(waitCursor);
      try {
         proceed(uiComp);
      } finally {
         uiComp.setCursor(originalCursor);
      }
   }
}
```

Two test components, GUIComp1 and GUIComp2, nest a concrete implementation of the aspect. For example, GUIComp1 contains following aspect:

```
public static aspect SlowMethodsParticipant extends SlowMethodAspect {
pointcut slowMethods(Component uiComp)
        : execution(void GUIComp1.performOperation1())
        && this(uiComp);
}
```

Whereas, GUIComp2 contains following aspect:

```
public static aspect SlowMethodsParticipant extends SlowMethodAspect {
      pointcut slowMethods(Component uiComp)
         : (execution(void GUIComp2.performOperation1())
           || execution(void GUIComp2.performOperation2()))
         && this(uiComp);
}
```

That usage pattern lets you provide aspectual class interfaces and lets you capture semantics- and characteristics-based pointcuts not otherwise possible by property-based pointcuts.

You'll find the complete code for both test components in GUIComp1.java and GUIComp2.java, and a simple test application in Test.java (see Moodle page).

D. Implement flexible access control

With AspectJ you can declare compile-time warnings and errors, a mechanism with which you can enforce static crosscutting concerns. For example, such a mechanism can enforce an access-control crosscutting concern. Java's access controls -- public, private, package, and protected -- do not offer enough control in many cases.

Consider, for example, a typical Factory design-pattern implementation. Oftentimes, you want only the factory to create the product objects. In other words, classes other than the factory should be prohibited from accessing constructors of any product class. While marking constructors with package access offers the best choice, that works only when the factory resides in the same package as the manufactured classes -- quite an inflexible solution.

The following code declares a simple Product class with a constructor and a configure() method. It also declares a nested FlagAccessViolation aspect, which in turn declares one pointcut to detect constructor calls from classes other than ProductFactory or its subclasses, and another pointcut to detect configure() method calls from classes other than ProductConfigurator or its subclasses. It finally declares either such violations as compile-time errors. ProductFactory, ProductConfigurator, or its subclasses could reside in any package, and both pointcuts could specify packages if necessary. You can further restrict access to, say, the createProduct() method, using the withincode() pointcut instead of within:

```
// Product.java
public class Product {
  public Product() {
    // constructor implementation
  }
  public void configure() {
    // configuration implementation
  }
  static aspect FlagAccessViolation {
    pointcut factoryAccessViolation()
      : call(Product.new(..)) && !within(ProductFactory+);

    pointcut configuratorAccessViolation()
      : call(* Product.configure(..)) && !within(ProductConfigurator+);
    declare error
      : factoryAccessViolation() || configuratorAccessViolation()
      : "Access control violation";
  }
}
```

The ProductFactory class calls the Product.configure() method, thus causing a compile-time error with a specified "Access control violation" message:

```
// ProductFactory.java
public class ProductFactory {
  public Product createProduct() {
    return new Product();
```

```
    }

    public void configureProduct(Product product) {
        product.configure();
    }
}
```

The AspectJ's get() pointcut lets you capture access to a field. You can advise such a pointcut to create an object if the field was null. With this, the logic of checking for null and creating an object if needed is modularized.

Note, that compiling example in "access-control" directory will result in an compile time error. This is the correct behavior illustrating how aspectj can be used for compile-time crosscutting behavior modification.

4. Report

Prepare a report of your work in this lab using a standard word processing application. The report is a single chapter of your final semester report. Final semester report will have to be uploaded to course Moodle page.

The content of report:

1. Title
2. Aims and tasks
3. Work done, described in steps and illustrated by screenshots and written or modified source code. Provide comments what you have done and what were the results (program outputs). Present answers to the questions given in the lab description.
4. Conclusions – what you have learned?