monitoring with AspectJ

CS 119

examples illustrating how to write monitors
Overview

- this and previous lecture should enable you to write monitors in AspectJ
- we will see examples of AspectJ monitors
- how to structure monitors
- trace view of program executions
  - reference to the past
  - obligations for the future
  - state machines
- giving an idea of how it is done without the use of special specification notation
- motivate later subjects
recall trace view of execution

- a formal view of an execution is to consider it as a sequence $\sigma$ of program states: $\sigma = s_1 s_2 s_3 \ldots s_n$
- during program execution we are at any point in time in the present moment now where the past is known but the future is not known.
past time properties

- If $A(x)$ happens now then $B(y)$ **must** have happened in the past where $R(x,y)$
future time properties

- If \( A(x) \) happens now then \( B(y) \) must happen in the future where \( R(x,y) \)
state machines

- A(x) and B(x) should happen in an alternating manner
monitor keeps state $\Sigma$

$\Sigma = \text{past events} \times \text{future obligations}$
two monitoring architectures

• write checking logic inside aspect

• write checking logic in separate class
• separating logic into a separate module makes specification writing modular.

either: a method for each kind of event
logic in separate class

- separating logic into a separate module makes specification writing modular.

or: a single event dispatch method
a file example

class File {
    public static final int READ = 1;
    public static final int WRITE = 2;

    public File(String name, int mode){...

    public void write(String text){...
    public String read(){...
    public void close() {...
    public String getName() {...
}
}
a requirement

A file should be accessed according to its mode READ or WRITE. A file cannot be accessed or closed unless it has been opened.

```java
File file = new File("data", File.WRITE);
file.write("monitor");
file.write("this if you can");
file.write(file.read());
file.close();
file.close();
```

error  error
first: logic inside aspect
\[ \Sigma = \text{File} \rightarrow \{\text{READ, WRITE}\} \]

\[
\begin{align*}
\text{file}_1 & \rightarrow 1 \\
\text{file}_2 & \rightarrow 2 \\
\text{file}_3 & \rightarrow 1 \\
\ldots
\end{align*}
\]

![Identity](Identity.png)

IdentityHashMap<File, Integer> modes = new IdentityHashMap();
Identity Maps in Java

• a normal hashmap does not work:
  – uses `equals` method to determine equality
  – and file objects may change wrt. equals

```java
HashMap<File,Integer> modes = new HashMap();
```

• necessary to use identity hashmap:
  – uses `==` to determine file equality

```java
IdentityHashMap<File,Integer> modes = new IdentityHashMap();
```
public aspect FileMonitor {
    pointcut open(int mode) : call(File.new(String,int)) && args(..,mode);
    pointcut write(File file) : call(void File.write(..)) && target(file);
    pointcut read(File file) : call(String File.read(..)) && target(file);
    pointcut close(File file) : call(void File.close()) && target(file);
    ...

    IdentityHashMap<File,Integer> modes = new IdentityHashMap();

    after(int mode) returning (File file) : open(mode) {
        modes.put(file,mode);
    }

    before(File file) : write(file) {
        Integer mode = modes.get(file);
        if(mode == null || mode != File.WRITE)
            error("illegal write to file " + file.getName());
    }

    before(File file) : read(file) {
        Integer mode = modes.get(file);
        if(mode == null || mode != File.READ)
            error("illegal read from file " + file.getName());
    }

    before(File file) : close(file) {
        if (modes.get(file) == null)
            error("attempt to close closed file " + file.getName());
        modes.remove(file);
    }
}
a simple way of getting an informative error message

```java
void error(String str) {
    try {
        throw new Exception ("*** " + str);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```
error message in Eclipse

```java
package aspectj2.architecture;

public class Test {
    public static void main(String[] args) {
        File file = new File("data", File.WRITE);
        file.write("monitor");
        file.write("this if you can");
        file.write(file.read());
        file.close();
        file.close();
    }
}
```

```
java.lang.Exception: *** illegal read from file data
at aspectj2.architecture.FileMonitor1.error(FileMonitor1 aj:16)
at aspectj2.architecture.FileMonitor1.ajc$before$aspectj2.architecture.FileMonitor1$35e8703177(FileMonitor1 aj:4)
at aspectj2.architecture.Test.main(Test.java:8)
```

```
java.lang.Exception: *** attempt to close closed file data
at aspectj2.architecture.FileMonitor1.error(FileMonitor1 aj:16)
at aspectj2.architecture.FileMonitor1.ajc$before$aspectj2.architecture.FileMonitor1$545c8f8209(FileMonitor1 aj:4)
at aspectj2.architecture.Test.main(Test.java:10)
```
second: logic in separate class

- separating logic into a separate module makes specification writing modular.

a method for each kind of event
class EngineM {
    IdentityHashMap<File, Integer> modes = new IdentityHashMap();
    ...
    void open(File file, int mode) {
        modes.put(file, mode);
    }
    void write(File file) {
        Integer mode = modes.get(file);
        if (mode == null || mode != File.WRITE) {
            error("illegal write to file " + file.getName());
        }
    }
    void read(File file) {
        Integer mode = modes.get(file);
        if (mode == null || mode != File.READ) {
            error("illegal read from file " + file.getName());
        }
    }
    void close(File file) {
        if (modes.get(file) == null)
            error("attempt to close closed file " + file.getName());
        modes.remove(file);
    }
}
public aspect FileMonitor {
  pointcut open(int mode) : call(File.new(String,int)) && args(..,mode);
  pointcut write(File file): call(void File.write(..)) && target(file);
  pointcut read(File file): call(String File.read(..)) && target(file);
  pointcut close(File file): call(void File.close()) && target(file);

  EngineM engine = new EngineM();

  after(int mode) returning (File file) : open(mode) {
    engine.open(file,mode);
  }

  before(File file) : write(file) {
    engine.write(file);
  }

  before(File file) : read(file) {
    engine.read(file);
  }

  before(File file) : close(file) {
    engine.close(file);
  }
}
using a single event dispatcher

```
dispatch(Event e) → logic
```

Diagram:
- Program
- Aspect
- Logic

Logic section: Σ
aspect FileMonitor {
  pointcut open(int mode) : ...;
  pointcut write(File file) : ...;
  pointcut read(File file) : ...;
  pointcut close(File file) : ...;

  EngineE engine = new EngineE();

  after(int mode) returning (File file) : open(mode) {
    engine.dispatch(new OpenEvent(file, mode));
  }
}

class OpenEvent implements Event {
  public File file;
  public int mode;

  public OpenEvent(File file, int mode) {
    this.file = file;
    this.mode = mode;
  }
}

class EngineE {
  void dispatch(Event event) { ... }
  ...
}
parameterized temporal operators

- define two temporal operators:
  - \textbf{Response}(o.Q,o.R) \textbf{ :} whenever method Q is called on object o, eventually method R will be called on o.
  - \textbf{Request}(o.P,o.Q) \textbf{ :} whenever method Q is called on o, in the past method P must have been called on o.

\begin{center}
\textbf{Request}(P,Q) \quad \textbf{Response}(Q,R)
\end{center}
file example again

assume existence of a method: File.open()

R1: A file should eventually be closed once opened.

R2: A file cannot be closed unless it has been opened.

File file1 = new File("data1", File.WRITE);
file1.open();
file1.write("monitor this");
// missing close of file1
File file2 = new File("data2", File.WRITE);
// missing open of file2
file2.close();
terminate();

error
abstract aspect Response {
    abstract pointcut firstMethod(Object o);
    abstract pointcut secondMethod(Object o);
    abstract pointcut shutDown();

    IdentityHashSet obligations = new IdentityHashSet();

    before(Object o) : firstMethod(o) {
        obligations.add(o);
    }

    before(Object o) : secondMethod(o) {
        obligations.remove(o);
    }

    before() : shutDown() {
        Iterator it = obligations.iterator();
        while(it.hasNext()) {
            error("Matching closing method not found on object: " + it.next());
        }
    }
}
abstract aspect Request {
    abstract pointcut firstMethod(Object o);
    abstract pointcut secondMethod(Object o);

    IdentityHashSet history = new IdentityHashSet();

    before(Object o) : firstMethod(o) {
        history.add(o);
    }

    before(Object o) : secondMethod(o) {
        if (!history.contains(o)) {
            error("Matching opening method not found on object " + o);
        } else {
            history.remove(o);
        }
    }
}
aspect R1 extends Response {
    pointcut firstMethod(Object o) :
        call(void File.open()) && target(o);
    pointcut secondMethod(Object o) :
        call(void File.close()) && target(o);
    pointcut shutDown() :
        call(void Test.terminate());
}

aspect R2 extends Request {
    pointcut firstMethod(Object o) :
        call(void File.open()) && target(o);
    pointcut secondMethod(Object o) :
        call(void File.close()) && target(o);
    pointcut shutDown() :
        call(void Test.terminate());
}
package aspectj2.temporal;

class Test {
    static void terminate() {
        public static void main(String[] args) {
            File file1 = new File("data1", File.WRITE);
            file1.open();
            file1.write("monitor this");
            // missing close of file1
            File file2 = new File("data2", File.WRITE);
            // missing open of file2
            file2.close();
            terminate();
        }
    }
}

java.lang.Exception: *** Matching opening method not found on object data2
    at aspectj2.temporal.Request.error(Request aj:15)
    at aspectj2.temporal.Request.$before$aspectj2_temporal_Request$2$839b92d(Request aj:29)
    at aspectj2.temporal.Test.main(Test java:13)

java.lang.Exception: *** Matching closing method not found on object: data1
    at aspectj2.temporal.Response.error(Response aj:15)
    at aspectj2.temporal.Response.$before$aspectj2_temporal_Response$3$e564b9f7(Response aj:34)
    at aspectj2.temporal.Test.main(Test java:13)
timing properties

• inside monitor: record milliseconds used with
  – System.currentTimeMillis() : “the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.”
  – requires a new event to occur
• use a “real” Timer object that spawns a thread which times out by itself. This does not require a new event to trigger. For example javax.swing.Timer.
• timestamp events in some other way in case they come from “outside” the application.
abstract
timed Response property

abstract aspect TimedResponse {
    abstract pointcut firstMethod(Object o);
    abstract pointcut secondMethod(Object o);
    abstract pointcut shutDown();
    abstract boolean timeok(long time);

    IdentityHashMap obligations = new IdentityHashMap();

    before(Object o) : firstMethod(o) {
        obligations.put(o, System.currentTimeMillis());
    }

    before(Object o) : secondMethod(o) {
        long secondTime = System.currentTimeMillis();
        Long firstTime = (Long) obligations.get(o);
        if (firstTime != null) {
            long time = secondTime - firstTime;
            if (!timeok(time))
                error("time " + time + " violates time constraint for " + o);
            obligations.remove(o);
        }
    }

    before() : shutDown() { … check for emptiness as before …}
}
back to the file example

TR1: After a file has been opened it should be closed within 5 seconds.

\( (o.\texttt{firstMethod} \rightarrow \Diamond^5 o.\texttt{secondMethod}) \)

aspect TR1 extends TimedResponse {
    pointcut firstMethod(Object o) :
        call(void File.open()) && target(o);
    pointcut secondMethod(Object o) :
        call(void File.close()) && target(o);
    pointcut shutDown() :
        call(void Test.terminate());

    boolean timeOk(long time) {
        return time <= 5000;
    }
}
monitoring
best programming practices

• generic properties we would want to hold for any program
• often concerns the use of various data types
• such properties are no different than domain specific properties
• some can be detected with static analysis
• however a dynamic analysis is relatively easy to program, whereas a static analysis either would not be possible or would require a substantial amount of work to implement
R1: There should be no two calls to `next()` without a call to `hasNext()` in between, on the same iterator.
use of iterators

There should be no two calls to Iterator.next() without a call to Iterator.hasNext() in between, on same iterator

```java
Vector<String> words = new Vector();
readWords(words);
Iterator it = words.iterator();
while(it.hasNext()) {
    String w1 = (String)it.next();
    String w2 = (String)it.next();
    storeCorrespondence(w1,w2);
    if (it.hasNext())
        System.out.println("there is more!");
}
```
slightly stronger property expressed as a state machine
we need a state machine per iterator

\[
\begin{align*}
\text{it}_1 & \rightarrow \\
\text{it}_2 & \rightarrow \\
\text{it}_3 & \rightarrow \\
\end{align*}
\]
the state design pattern

- allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- an object-oriented state machine

Gamma, Erich; Richard Helm, Ralph Johnson, John M. Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley
the state design pattern spelled out

• define a "context" class to present a single interface to the outside world.
• define a State abstract base class.
• represent the different "states" of the state machine as derived classes of the State base class.
• define state-specific behavior in the appropriate State derived classes.
• maintain a pointer to the current "state" in the "context" class.
• to change the state of the state machine, change the current "state" pointer.
class Machine {
    State state = State.doHasNext;
    void hasNext() {
        state = state.hasNext();
    }
    void next() {
        state = state.next();
    }
}

class State {
    static final State doNext = new DoNext();
    static final State doHasNext = new DoHasNext();

    State hasNext(){
        System.out.println("*** warning: hasNext called unnecessarily");
        return this;
    }

    State next(){
        System.out.println("*** error: next called illegally");
        return this;
    }
}

class DoHasNext extends State {
    State hasNext() {
        return doNext;
    }
}

class DoNext extends State {
    State next() {
        return doHasNext;
    }
}
aspect HasNextPolicy {
    WeakIdentityHashMap monitors = new WeakIdentityHashMap();

    pointcut createIter():
        call(* java.util.Collection+.iterator());
    pointcut hasNext(Iterator it):
        call(* java.util.Iterator+.hasNext()) && target(it);
    pointcut next(Iterator it):
        call(* java.util.Iterator+.next()) && target(it);

    after() returning (Iterator it): createIter() {
        monitors.put(it, new Machine());
    }

    before(Iterator it): hasNext(it) {
        ((Machine)monitors.get(it)).hasNext();
    }

    before(Iterator it): next(it) {
        ((Machine)monitors.get(it)).next();
    }
}
weak identity hash maps

- hashmap still an identity hashmap (using `==`)
- a normal (identity) hashmap keeps a mapping until it is explicitly deleted with `Map.remove()`.
- this becomes a problem since the monitor will then accumulate bindings between iterators and state machines. The garbage collector cannot collect the iterators when they are no longer used by the monitored application.
- a weak collection releases an object to the garbage collector when it is no longer used by any other part of the program.
properties of Java library APIs

R2: An enumeration should not be propagated after the underlying vector has been changed.
enumerators faster than iterators but less safe

"I'd tried using Iterator and Enumeration to compare their performance on a Vector object containing 100 000 Strings. Enumeration was consistently about 50% faster". - web blog

But: an Iterator next() operation throws a:

ConcurrentModificationException

if it detects that the underlying collection has been modified while iteration is underway.

Problem: Enumerator does not!
use of enumerators

An enumeration should not be propagated after the underlying vector has been changed.
three maps are needed

- **DsState**: recording when a data structure was last updated (maps to unique object)
- **EnumState**: recording the state of the data structure of an enumeration at creation time
- **EnumDs**: recording what data structure corresponds to what enumeration

\[
\begin{align*}
\text{DsState} & = \text{Ds} \rightarrow \text{State} \\
\text{EnumState} & = \text{Enum} \rightarrow \text{State} \\
\text{EnumDs} & = \text{Enum} \rightarrow \text{Ds}
\end{align*}
\]
... 
 v.add(3);
 Enumeration en = v.elements();
 while(en.hasMoreElements()) {
     Integer i = (Integer)en.nextElement();
     if (i == 2)
         v.add(4);
     else
         System.out.println(i);
 }

example
monitored
run
... v.add(3);
Enumeration en = v.elements();
while (en.hasMoreElements()) {
    Integer i = (Integer)en.nextElement();
    if (i == 2)
        v.add(4);
    else
        System.out.println(i);
}

example monitored run

ds \rightarrow state_1
example monitored run

```java
... v.add(3);
Enumeration en = v.elements();
while (en.hasMoreElements()) {
    Integer i = (Integer)en.nextElement();
    if (i == 2)
        v.add(4);
    else
        System.out.println(i);
}
```
...v.add(3);
Enumeration en = v.elements();
while (en.hasMoreElements()) {
    Integer i = (Integer)en.nextElement();
    if (i == 2)
        v.add(4);
    else
        System.out.println(i);
}

example
monitored
run
```java
... 
v.add(3);
Enumeration en = v.elements();
while(en.hasMoreElements()) {
    Integer i = (Integer)en.nextElement();
    if (i == 2)
        v.add(4);
    else
        System.out.println(i);
}
```
aspect SafeEnum {
    private Map ds_state = new WeakIdentityHashMap();
    private Map enum_state = new WeakIdentityHashMap();
    private Map enum_ds = new WeakIdentityHashMap();

    private static class StateId {}  

    pointcut vector_update() :
        call(* Vector.add*(..)) ||
        call(* Vector.clear()) ||
        call(* Vector.insertElementAt(..)) ||
        call(* Vector.remove*(..)) ||
        call(* Vector.retainAll(..)) ||
        call(* Vector.set*(..)) && scope();

    after(Vector ds) returning(Enumeration e) :
        call(Enumeration Vector.elements()) && target(ds) {
            enum_ds.put(e,ds);
            Object s = ds_state.get(ds);
            if (s != null) enum_state.put(e,s);
        }

    before(Enumeration e):
        call(Object Enumeration.nextElement()) && target(e) {
            if (ds_state.get(enum_ds.get(e)) != enum_state.get(e))
                error("nextElement called on enumerator after update");
        }

    after(Vector ds) : vector_update() && target(ds) {
        ds_state.put(ds,new StateId());
    }
}
R₃: An collection should not be modified while it is a member of a hashset (don’t change the hashcode).
A collection should not be modified while it is member of a hash set. In other words: don’t change the collection’s hashcode during this period.

```java
Set<Collection<String>> s = new HashSet();
Collection<String> c = new ArrayList();
c.add("this is ok");
s.add(c);
c.add("don't do this");
System.out.println(s.contains(c));
```
one map is needed

- recording what sets a collection is stored in

\[
\text{Map} = \text{Collection} \rightarrow \text{HashSet-set}
\]

\[
\begin{align*}
\text{coll}_1 & \rightarrow \{\text{hs}_1, \text{hs}_2\} \\
\text{coll}_2 & \rightarrow \{\text{hs}_2, \text{hs}_3, \text{hs}_4\} \\
\text{coll}_3 & \rightarrow \{\text{hs}_4\}
\end{align*}
\]
Set<Collection<String>> s = new HashSet();
Collection<String> c = new ArrayList();
c.add("this is ok");
s.add(c);
c.add("don't do this");
System.out.println(s.contains(c));
Set<Collection<String>> s = new HashSet();
Collection<String> c = new ArrayList();
c.add("this is ok");
s.add(c);
c.add("don't do this");
System.out.println(s.contains(c));
Set<Collection<String>> s = new HashSet();
Collection<String> c = new ArrayList();
c.add("this is ok");
s.add(c);
c.add("don't do this");
System.out.println(s.contains(c));
aspect HashSetPolicy {
  pointcut addCollection(HashSet s, Collection c):
    call(* Collection.add(Object)) && target(s) && args(c);
  pointcut removeCollection(HashSet s, Collection c):
    call(* Collection.remove(Object)) && target(s) && args(c);
  pointcut modifyCollection(Collection c):
    (call(* Collection.add(..)) || ...) && target(c);

WeakIdentityHashMap hashSets = new WeakIdentityHashMap();

after(HashSet s, Collection c) : addCollection(s,c) {
  if(hashSets.get(c)==null)
    hashSets.put(c,new WeakIdentityHashSet());
  WeakIdentityHashSet sets = (WeakIdentityHashSet) hashSets.get(c);
  sets.add(s);
}

after(HashSet s, Collection c) : removeCollection(s,c) {
  WeakIdentityHashSet sets = (WeakIdentityHashSet)hashSets.get(c);
  if(sets!=null) sets.remove(s);
}

before(Collection c): modifyCollection(c) {
  WeakIdentityHashSet sets = (WeakIdentityHashSet) hashSets.get(c);
  if(sets!=null)
    for (Object s : sets)
      error("*** Modified collection " + c + " in " + s);
}
}

the checker
lock release policies

Within one method invocation, locks should be acquired and released correctly, meaning …

• release locks in same invocation they are taken
• and either one of the following policies:
  – release exactly once

  \[ L(x) \ L(x) \ U(x) \ U(x) \]

  set

  – release as many times as acquired (order unimportant)

  \[ L(x) \ L(x) \ U(x) \ U(x) \]

  bag

  – release in reverse order

  \[ L(x) \ L(y) \ U(x) \ U(y) \]

  stack
an example

```
class Lock {
    String name;

    Lock(String name) {
        this.name = name;
    }

    synchronized void lock() {...}
    synchronized void unlock() {...}

    public String toString() {
        return name;
    }
}
```

```
void start() {
    a();
}

void a() {
    l1.lock();
    l2.lock();
    l1.unlock();
    l2.unlock();
    l3.lock();
    b();
}

void b() {
    l3.unlock();
}
```
stack of lock histories

Data = Level \rightarrow \text{LockHist}

Data = \text{LockHist-stack}

LockHist = \text{Set} \mid \text{Bag} \mid \text{Stack}

assuming reverse order discipline (stack)

3 \rightarrow L_1 \ L_2 \ L_3

4 \rightarrow L_6 \ L_7

7 \rightarrow L_5

\begin{verbatim}
public interface LockHist {
    public void lock(Lock l);
    public boolean unlock(Lock l);
    public boolean isEmpty();
    public void clear();
}
\end{verbatim}
class StackHist implements LockHist {
    Stack stack = new Stack();

    public void lock(Lock l) {
        stack.push(l);
    }

    public boolean unlock(Lock l) {
        boolean success = !stack.isEmpty() && stack.peek() == l;
        if (success)
            stack.pop();
        return success;
    }

    public boolean isEmpty() {
        return stack.isEmpty();
    }

    ...
}
computing method invocation level

ThreadLocal: this java.lang class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable.

```java
static ThreadLocal cflowdepth = new ThreadLocal() {
    protected synchronized Object initialValue() {
        return new Integer(0);
    }
};
```

```java
aspect CflowDepth {
    pointcut anyfunc() : execution(* *(..));

    static ThreadLocal cflowdepth = new ThreadLocal() {... 0 ...};

    before() : anyfunc() {
        Integer prev = (Integer) cflowdepth.get();
        cflowdepth.set(new Integer(prev.intValue() + 1));
    }

    after() : anyfunc() {
        Integer prev = (Integer) cflowdepth.get();
        cflowdepth.set(new Integer(prev.intValue() - 1));
    }
}
```
aspect LockChecker {
    public static ThreadLocal depthMap = new ThreadLocal();
    pointcut locking(Lock l) : call(* Lock.lock()) && target(l);
    pointcut unlocking(Lock l) : call(* Lock.unlock()) && target(l);

    before(Lock l) : locking(l) {
        HashMap map = (HashMap) depthMap.get();
        Integer depth = (Integer)CflowDepth.cflowdepth.get();
        LockHist lockhist = (LockHist) map.get(depth);
        if (lockhist == null) {
            lockhist = new StackHist();
            map.put(depth, lockhist);
        }
        lockhist.lock(l);
    }

    before(Lock l) : unlocking(l) {
        HashMap map = (HashMap) depthMap.get();
        Integer depth = (Integer)CflowDepth.cflowdepth.get();
        LockHist lockhist = (LockHist) map.get(depth);
        if (lockhist == null || !lockhist.unlock(l))
            error("unlock op. not preceeded by lock op." + l);
    }

    after() : CflowDepth.anyfunc() {
        HashMap map = (HashMap) depthMap.get();
        Integer depth = (Integer)CflowDepth.cflowdepth.get();
        LockHist lockHist = (LockHist) map.get(depth);
        if (lockHist != null && !lockHist.isEmpty())
            error("locks have not been released " + depth + lockHist);
        if (lockHist != null) lockHist.clear();
    }
}

declare precedence : CflowDepth, LockChecker;
aspect precedence

• if advice A occurs before advice B in same aspect then A has higher precedence than B

• if the following declaration is given:

  declare precedence : A, B;

  then A has higher priority than B
aspect precedence

before

highest

lowest

Join point

after

Join point

lowest

highest

around

highest

lowest

Join point
end