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
• preparing for later lectures on temporal logic
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 aspects

• 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 aside

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

or: a single event dispatch method
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 file example
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();
```
first: all code in one aspect
\[ \Sigma = \text{File} \rightarrow \{\text{READ, WRITE}\} \]

- file$_1$ → 1
- file$_2$ → 2
- file$_3$ → 1
- file$_4$ → null
- ...  

null : closed

**Identity**

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

either: a method for each kind of event
```java
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);
    }
}
logic aside

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

or: a single event dispatch method
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 EngineE {
    void dispatch(Event event) { ... }
    ...
}
parameterized temporal operators

– Define two temporal operators:
  • \( \text{Response}(o.A,o.B) \): whenever method \( A \) is called on object \( o \), eventually method \( B \) will be called on \( o \).
  • \( \text{Request}(o.A,o.B) \): whenever method \( B \) is called on \( o \), in the past method \( A \) must have been called on \( o \).
file example again

**R1:** A file should eventually be closed once opened.

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

```java
File file1 = new File("data1", File.WRITE);
file1.open();
file1.write("monitor");
File file2 = new File("data1", File.WRITE);
file2.close();
terminate();
```

assume existence of a method: File.open()
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);
        }
    }
}
concrete Response and Request properties R1 and R2

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

java.lang.Exception: *** Matching opening method not found on object: data1
at aspectj2.temporal.Request.error(Request.aj:15)
at aspectj2.temporal.Request.ajc$before$aspectj2_temporal_Request$2$b839b92d(Request.aj:29)
at aspectj2.temporal.Test.main(Test.java:1)
java.lang.Exception: *** Matching closing method not found on object: data1
at aspectj2.temporal.Response.error(Response.aj:15)
at aspectj2.temporal.Response.ajc$before$aspectj2_temporal_Response$3$e56d9f7(Response.aj:34)
at aspectj2.temporal.Test.main(Test.java:1)
timing properties

• 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.

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

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.

program aspect + logic

\[ it_1 \rightarrow \text{some value} \]
\[ it_2 \rightarrow \text{some value} \]
\[ it_3 \rightarrow \text{some value} \]
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

An Iterator next() operation throws a:

ConcurrentModificationException

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

Enumerator does not!
use of enumerators

An enumeration should not be propagated after the underlying vector has been changed.

```java
Vector v = new Vector();
v.add(1);
v.add(2);
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);
}
```

produces:
1
3
4

error
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

\[
\begin{align*}
\text{DsState} & = Ds \rightarrow \text{State} \\
\text{EnumState} & = \text{Enum} \rightarrow \text{State} \\
\text{EnumDs} & = \text{Enum} \rightarrow Ds
\end{align*}
\]
...  
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

```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**
... 
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);
}

---

danger
aspect SafeEnum {
    private Map ds_state = new WeakIdentityHashMap();  // the checker
    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, ds_state.get(ds));
        }

    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());
    }
}
properties of Java library APIs

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));
```

produces: 
false

error
one map is needed

• recording what hashmaps a collection is stored in

Map = Collection \rightarrow \text{HashMap-set}

\text{coll}_1 \rightarrow \{\text{hm}_1, \text{hm}_2\}
\text{coll}_2 \rightarrow \{\text{hm}_2, \text{hm}_3, \text{hm}_4\}
\text{coll}_3 \rightarrow \{\text{hm}_4\}

don’t update collection when set is non-empty
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));
```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));
```

Example monitored run

C \rightarrow \{ S \}

add
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) returning (): 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) returning (): 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 (Iterator iter = sets.iterator(); iter.hasNext();)
                error("*** Modified collection "+c+" in " + iter.next());
    }
}
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 at most once
  - release as many times as acquired (order unimportant)
  - release in reverse order
an example

```java
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

\[
\begin{align*}
\text{Data} &= \text{Level} \rightarrow \text{LockHist} \\
\text{Data} &= \text{LockHist-stack} \\
\text{LockHist} &= \text{Set} \mid \text{Bag} \mid \text{Stack}
\end{align*}
\]

3 \rightarrow [L_1, L_2, L_3]
4 \rightarrow [L_6, L_7]
7 \rightarrow [L_5]

public interface LockHist {
    public void lock(Lock l);
    public boolean unlock(Lock l);
    public boolean isEmpty();
    public void clear();
}
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
aspect CflowDepth {
    pointcut anyfunc() : execution(* *(..));

    static ThreadLocal cflowdepth = new ThreadLocal() {... 0 ...}; // initialized to zero

    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() { ... emptystack ...};
    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();
    }
}

the checker
end