Lecture 7

- First mid-term: Feb 22
- Assignment on Construction, Publishing and Visibility is posted (and we will discuss momentarily)

Assignment: Construction, Visibility, and Publication

- Inspect real code for problems
- MARS MIPS simulator system
- Example

Example

```
class SimThread extends
   SwingWorker {
   private MIPSprogram p;
   private int pc, maxSteps;
   private int[] breakPoints;
   private boolean done;
   private ProcessingException
      pe;
   private volatile boolean
      stop = false;
   private volatile
      AbstractAction stopper;
   private AbstractAction
      starter;
   private int
      constructReturnReason;
```

At line 259 – in a SimThread

```
this.pe = new
   ProcessingException(el);
this.constructReturnReason =
   EXCEPTION;
this.done = true;
```

And at line 122 – in process that creates the SimThread

```
ProcessingException pe =
   simulatorThread.pe;
boolean done =
   simulatorThread.done;
```

Also, note the comments and code at lines 235-240. What's wrong with this?

Advice

- Some uses in this code are suspicious but you can't really tell whether they are wrong without looking at other modules. For things that catch your eye as suspicious write down your suspicions and the questions they raise about other modules.
- It's a lot of code. What language constructs do you need to focus on?
  -
  -
  -
- When doing the homework, notice what makes it hard to determine whether the code is right or wrong
  - Learn from example code – both good and bad

Two common problems

- Composing a thread-safe class from unsafe building blocks
- Composing a thread-safe class when the building blocks are already thread-safe

Confinement Technique (4.2) – thread-safe object built from unsafe objects

- Allow access to a thread-unsafe object only through another object that is thread-safe

```
public class PersonSet {
    private final Set<Person> mySet = new
  HashSet<Person>();
    public synchronized void add(Person p) {
       mySet.add(p); }
    public synchronized boolean contains(Person p) {
       return myset.contains(p); }
}
```

- hashSet is not ThreadSafe, PersonSet is
- Idea of ownership: PersonSet owns mySet but probably not the Persons contained in it

Monitors (4.2.1) (aside)

- Idea: a syntactic construct that pairs lock acquisition and release around a block of code
  - synchronized(foo) { … }
  - compared to explicit lock acquisition and release, encourages structured programs and avoids bugs due to failure to release locks
- Along with a mechanism for releasing a lock, suspending, and reacquiring the lock
  - wait()
- And a mechanism to wake up suspended threads
  - notify(), notifyAll()
- Monitors were independently invented by Sir Anthony Hoare and Per Brinch-Hansen in the mid-1970's.

Danger in Confinement Technique

- Inadvertant publication of what is supposed to be private (confined) mutable state

```
public synchronized MutablePoint getLocation(String id) {
   MutablePoint loc = locations.get(id);
   return loc == null ? Null : new MutablePoint(loc);
}
Public synchronized setLocation(String id, int x, int y) {
   MutablePoint loc = locations.get(id);
   if (loc == null) { … exception …}
   loc.x = x; loc.y = y
}
```

- My preference would be to express this interface using ImmutablePoints.

Thread-safe objects built from thread-safe components – Delegating safety (4.3)

- Delegation: giving responsibility for thread safety to the object(s) containing this object's state
  - ConcurrentMap (TS) instead of Map (not TS)
  - Atomic<foo>
- If this object's state involves multiple other objects delegation may or may not work
  - If the sub-objects are independent, ok
  - If the sub-objects are related, this object must provide its own synchronization – even if all the sub-objects are themselves thread-safe

Example

```
class PongPaddle {
   private final AtomicInteger left = new AtomicInteger(0);
   private final AtomicInteger right = new AtomicInteger(1);
   public void move(int dx) { left.getAndAdd(dx);
  right.getAndAdd(dx); }
   public void changeWidth(int dw) { right.getAndAdd(dw); }
   public boolean hit(int pos) {
      return left.get()<=pos && pos <= right.get();
   }
}
```

- No visibility concerns
- What is the invariant that relates left and right?
- What should we do to fix it?

Reduce, Reuse, Recycle

- I am very *anti* cut-and-paste coding
  - Hard on the reader
  - Hard on the maintainer
  - Instead of 1 change, n changes
  - Instead of 1 bug, n bugs
- Design code so there only needs to be one copy (use parameterization, polymorphic parameterization)
- Even better, reuse existing code that does almost the right thing
- How does this interact with synchronization?

Adding functionality (4.4)

- Example: add putIfAbsent to a collection that already supports atomic contains() and add() methods
- Four approaches
  - Modify existing class
  - Extend existing class or
  - Wrap existing class – “client-side” locking
  - Composition -
- Add to existing class
  - Best way but
  - Assumes you have control over the existing class

Extend the existing class

```
public class BetterVector<E> extends Vector<E> {
   public synchronized boolean putIfAbsent(E x) {
      boolean absent = !contains(x);
      if (absent) { add(x); }
      return absent;
   }
}
```

Already in Vector.

- Note the benefit of re-entrant locks!
- Note the implicit assumption that we understand the way that Vector does synchronization – using intrinsic locks, in this case
- Note that we don't have any dependency on Vector's implementation
- Vector is thread-safe
- Vector provides enough primitive building blocks to allow construction of putIfAbsent

Client-side locking

- Assume v is a thread-safe list obtained from `Collections.synchronizedList(new ArrayList<E>());`
- Type of this object is List<E> -- not extendable
- Any code that wants to do putIfAbsent item x to such a list, v, can write

```
synchronized (v) {
   if (!v.contains(x)) v.add(x);
}
```

- Could be placed in a helper class – beware you have to synchronize on the *list* and not on the helper object
- Now spreading the synchronization far and wide
- Still depending on knowing the synch policy for the wrapped object

Composition

- Mimic the idea of Collections.synchronizedList
  - Provide all the synchronization in a new object that extends the functionality of an existing object instance (not class)
  - Delegates most operations to the existing object

```
Public class ImprovedList<T> implements List<T> {
   private final List<T> list;
   public ImprovedList(List<T> list) {
      this.list = list; }
   public synchronized boolean putIfAbsent(T x) …
   public synchronized boolean contains(T x) {
      return list.contains(x); }
   …
```

Intro to Chapter 5 – Building Blocks

- Chapter 4 was about low-level techniques
- This chapter is about libraries – embodiments of the techniques
- Section 5.1 Synchronized collections – read to see why you want to use Concurrent collections instead (we will not cover)
  - The idioms described are even more unsafe than asserted in the book because of visibility problems

Chapter 5 topics

- Concurrent collections (5.2)
- The ubiquitous producer-consumer pattern (5.3)
- Interruptable methods (5.4)
- Primitive synchronizers (5.5)