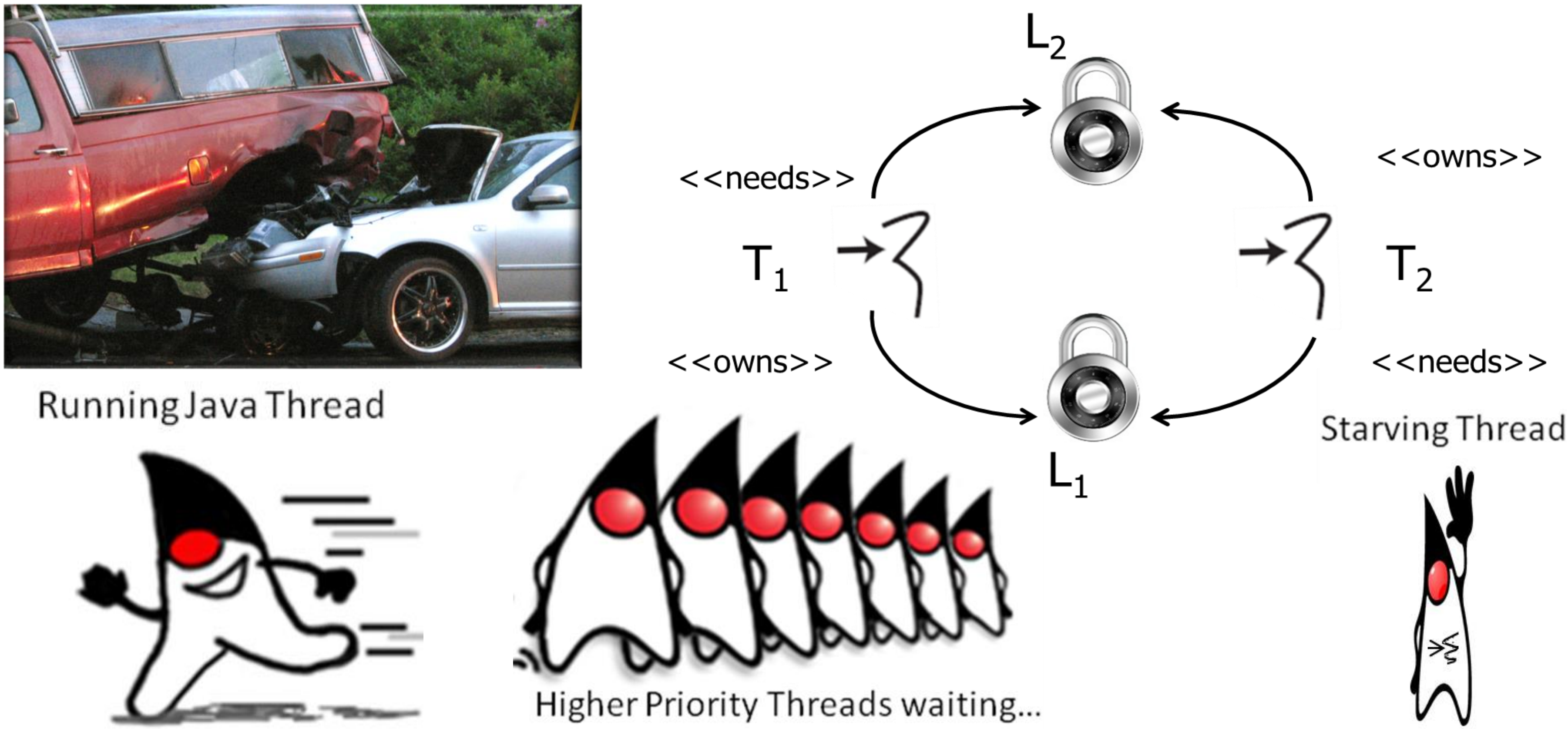
Java Monitor Objects: Evaluating the Motivating Example

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

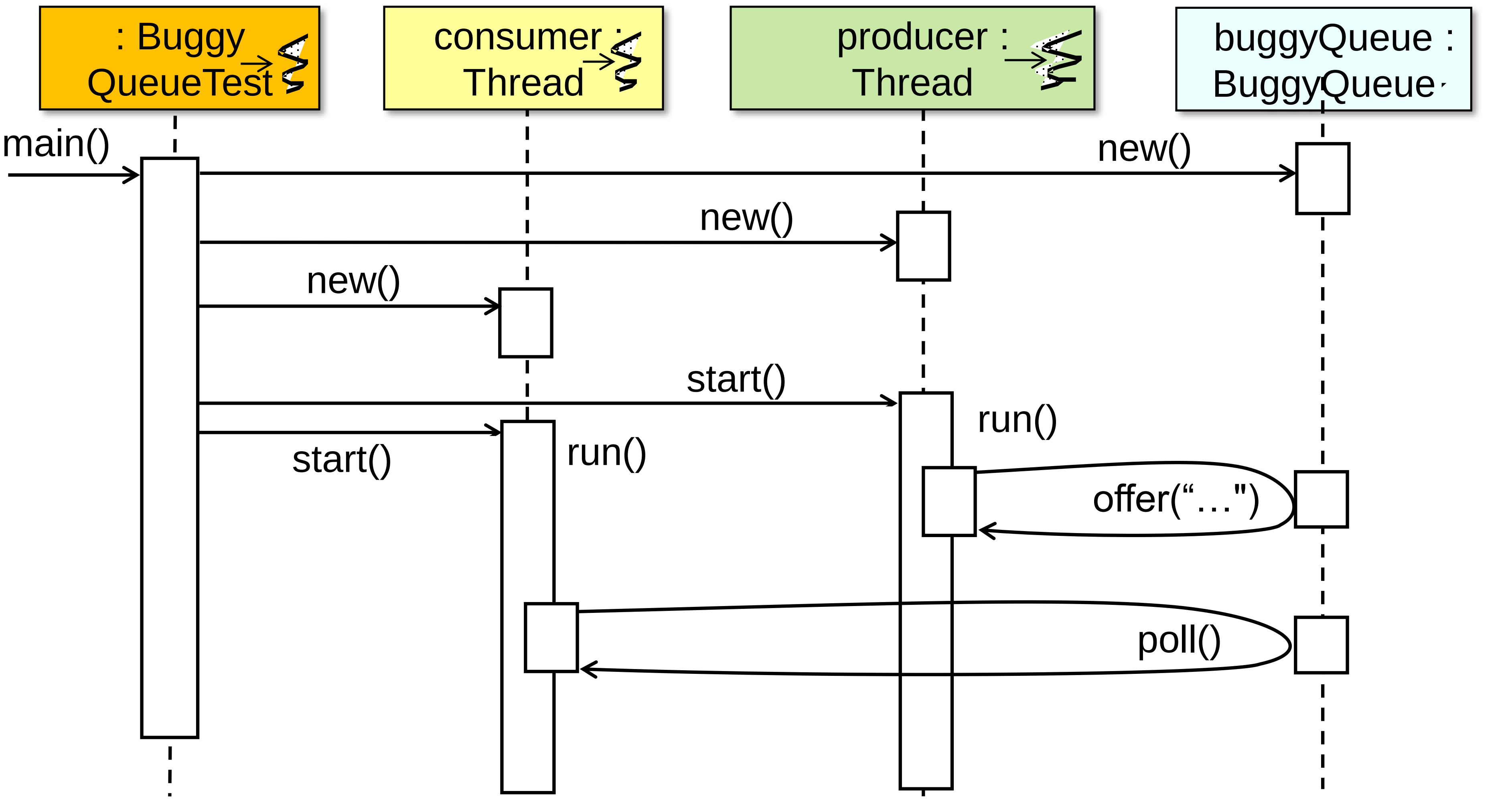
- Understand what monitors are & know how Java built-in monitor objects can ensure mutual exclusion & coordination between threads
- Recognize common synchronization problems in concurrent Java programs
- Be aware of common complexities in concurrent programs like BuggyQueue



Evaluating the Buggy Producer/Consumer

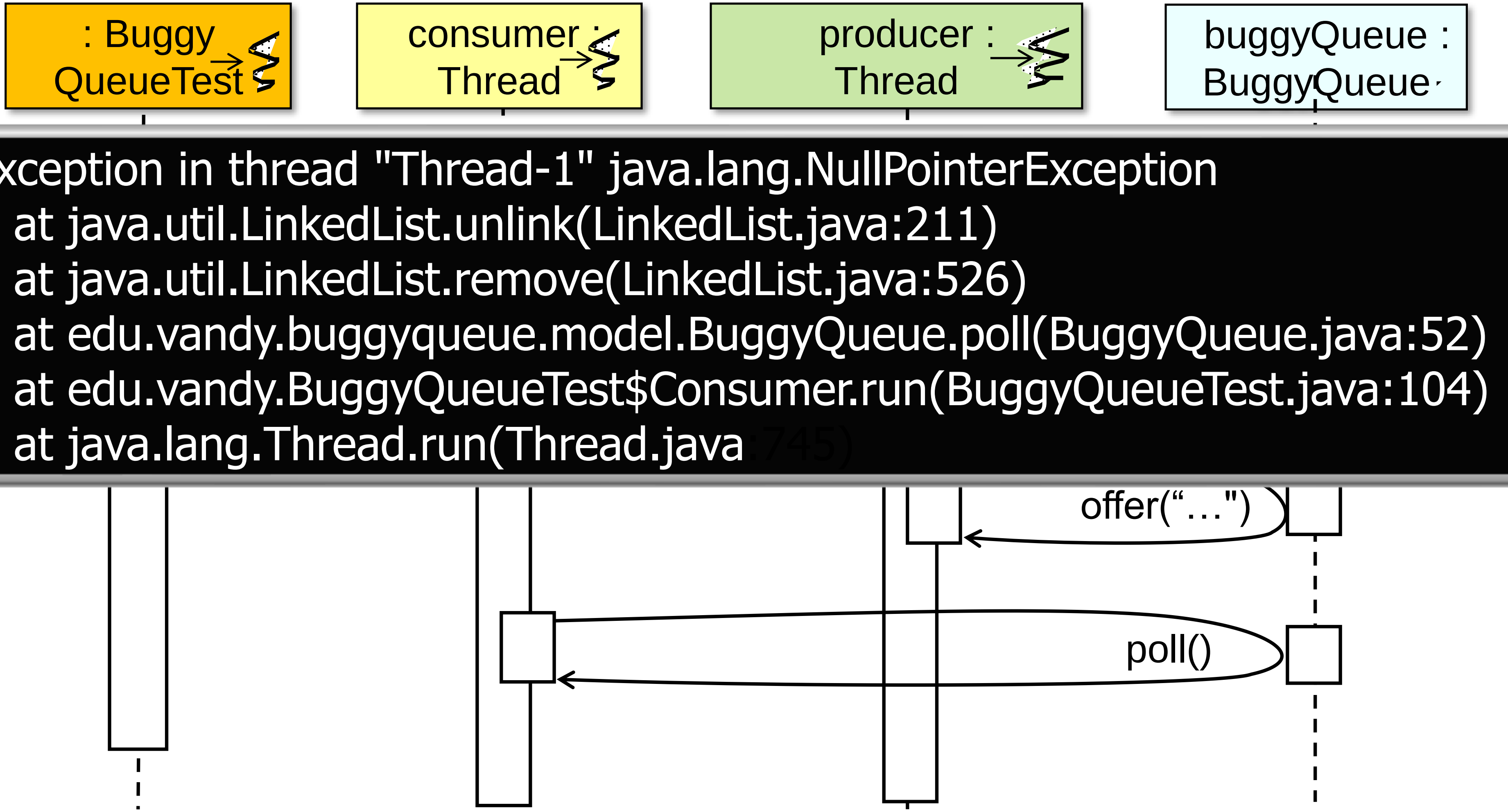
Evaluating the Buggy Producer/Consumer

- Key question: what's the output & why?



Evaluating the Buggy Producer/Consumer

- Key question: what's the output & why?




```
Exception in thread "Thread-1" java.lang.NullPointerException
   at java.util.LinkedList.unlink(LinkedList.java:211)
   at java.util.LinkedList.remove(LinkedList.java:526)
   at edu.vandy.buggyqueue.model.BuggyQueue.poll(BuggyQueue.java:52)
   at edu.vandy.BuggyQueueTest$Consumer.run(BuggyQueueTest.java:104)
   at java.lang.Thread.run(Thread.java:745)
```

Depending on the implementation of the BuggyQueue class & the underlying LinkedList the app & test program may simply "hang"

Evaluating the Buggy Producer/Consumer

- Key question: what's the output & why?

```
static class BuggyQueue<E> implements BoundedQueue<E> {
  private LinkedList<E> mList = new LinkedList<E>();

  public boolean offer(E e) {
    if (!isFull()) { mList.add(e); return true; }
    else return false;
  }

  public E poll() {
    if (!isEmpty()) return mList.remove(0); else return false; }
    ...
  }
```

There's no protection against critical sections being run by multiple threads concurrently

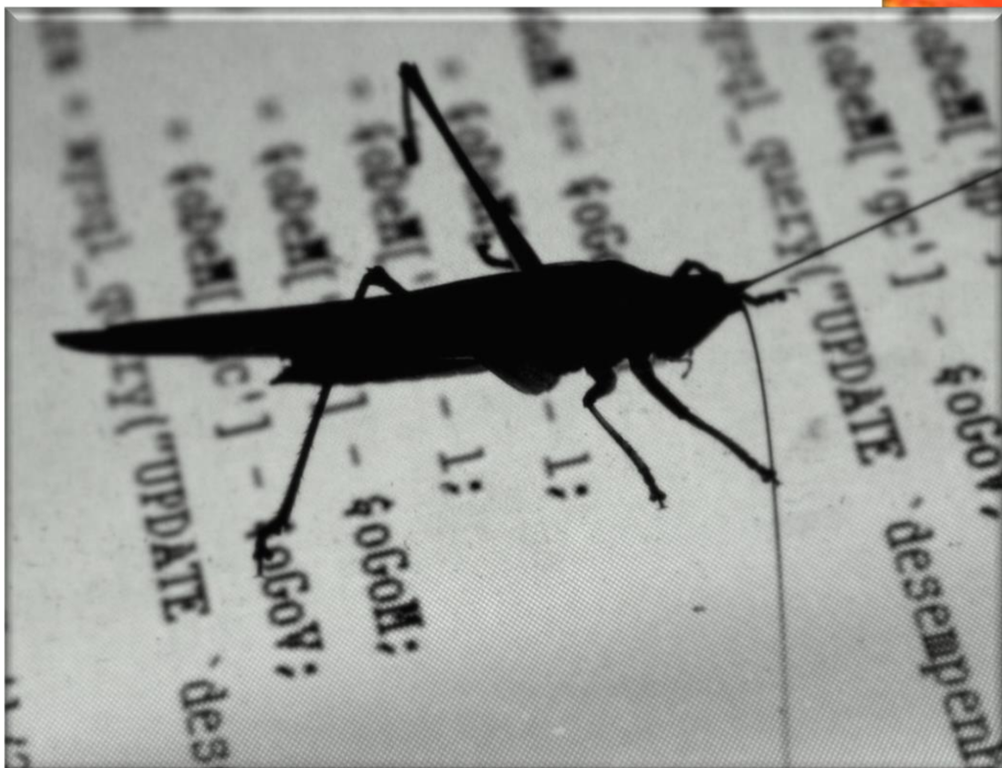
Note that this implementation is not synchronized. If multiple threads access a linked list concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements; merely setting the value of an element is not a structural modification.)

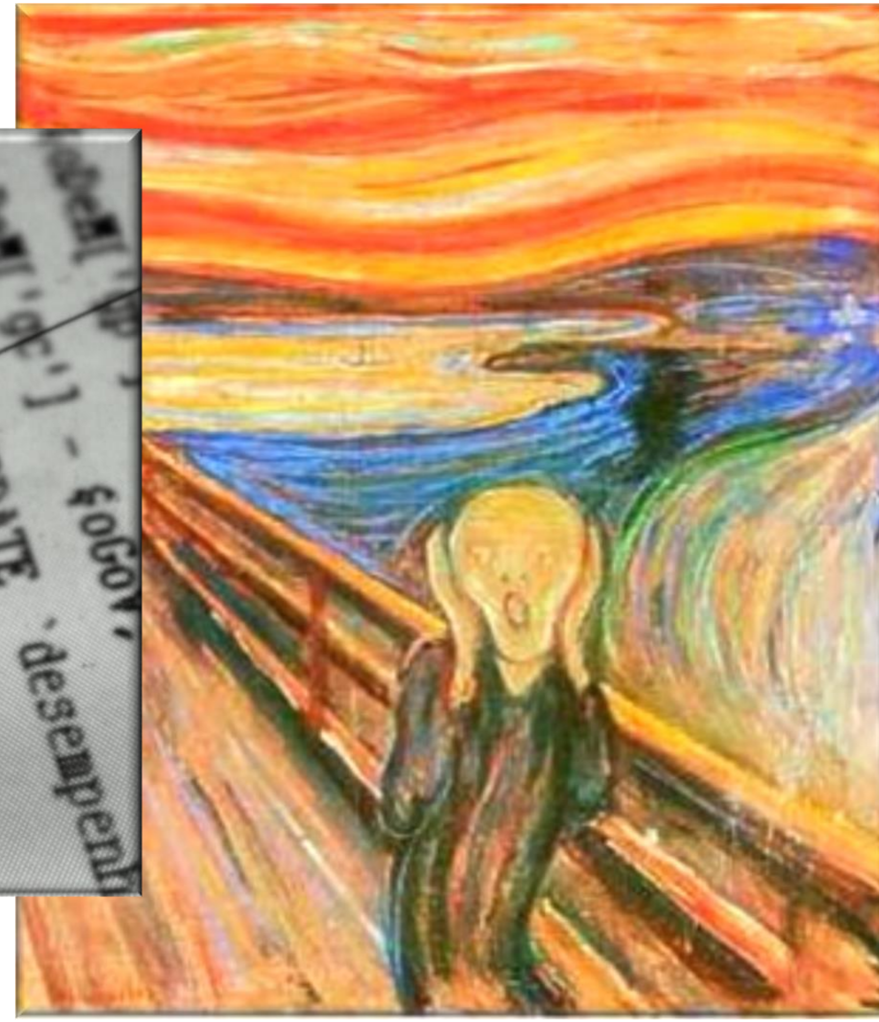
See docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

Common Complexities in Concurrent Programs

Common Complexities in Concurrent Programs

- Concurrent programs are hard to develop & debug, due to various inherent & accidental complexities

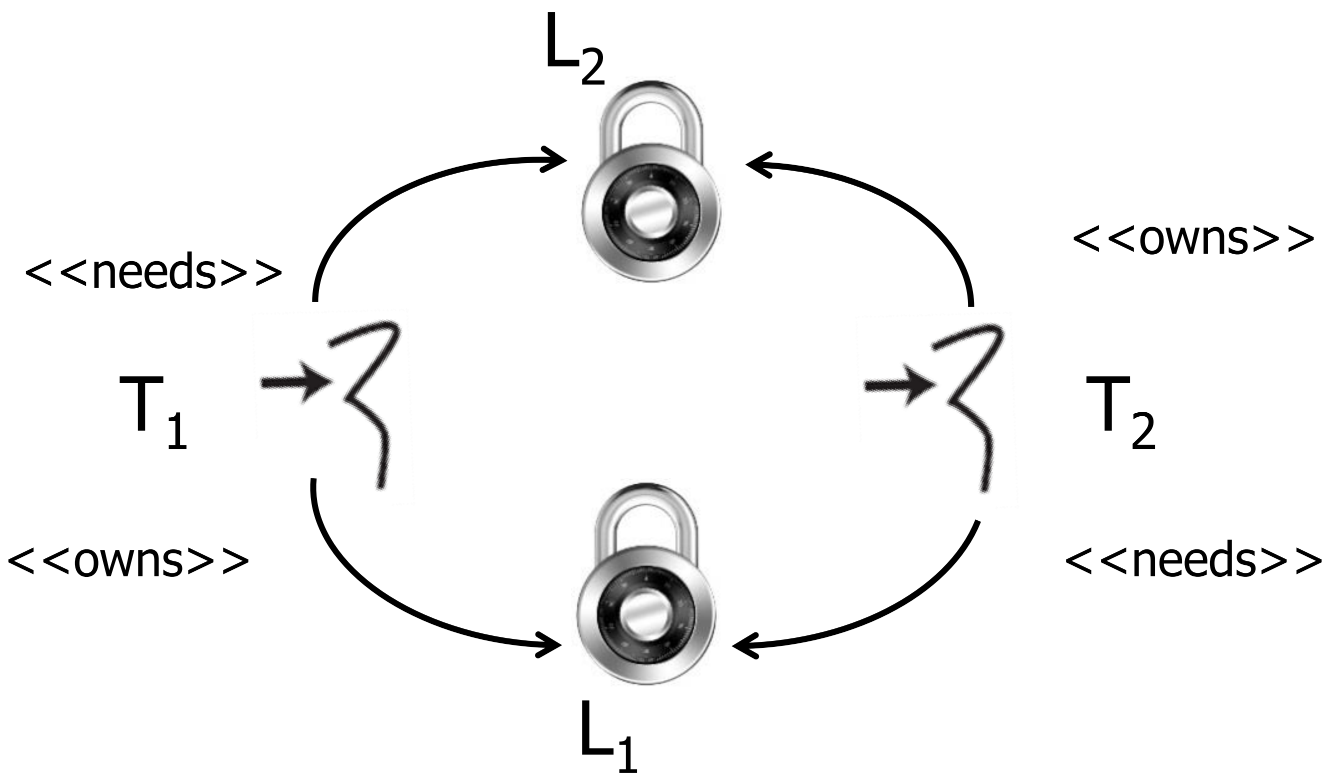




See stackoverflow.com/questions/499634/how-to-detect-and-debug-multi-threading-problems

Common Complexities in Concurrent Programs

- Concurrent programs are hard to develop & debug, due to various inherent & accidental complexities, e.g.
  - Deadlock
    - *Occurs when two or more competing actions are each waiting for the other to finish, & thus none ever do*

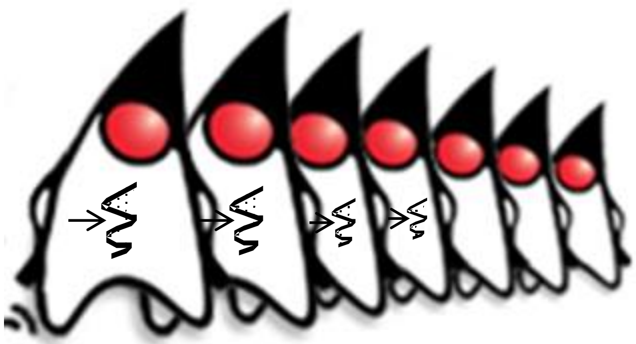


See en.wikipedia.org/wiki/Deadlock

Common Complexities in Concurrent Programs

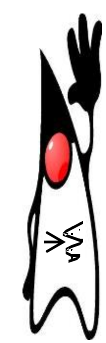
- Concurrent programs are hard to develop & debug, due to various inherent & accidental complexities, e.g.
  - Deadlock
  - Starvation
    - *A thread is perpetually denied necessary resources to process its work*

Running Java Thread



Higher Priority Threads waiting...

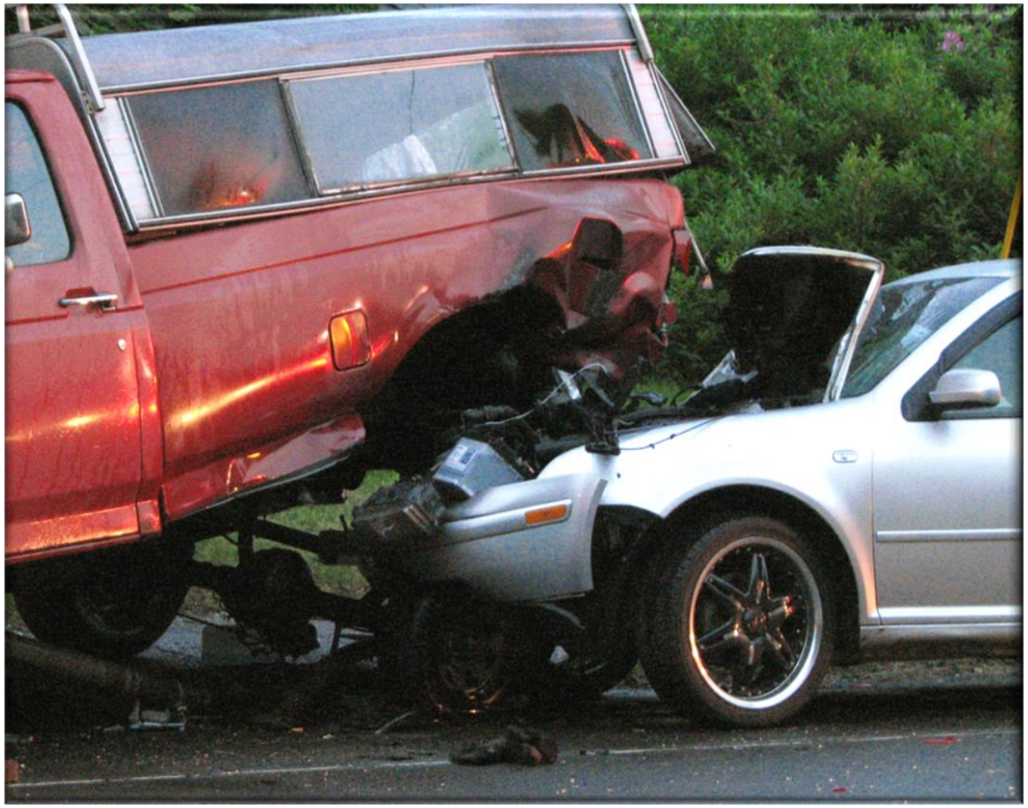
Starving Thread



See en.wikipedia.org/wiki/Starvation_(computer_science)

Common Complexities in Concurrent Programs

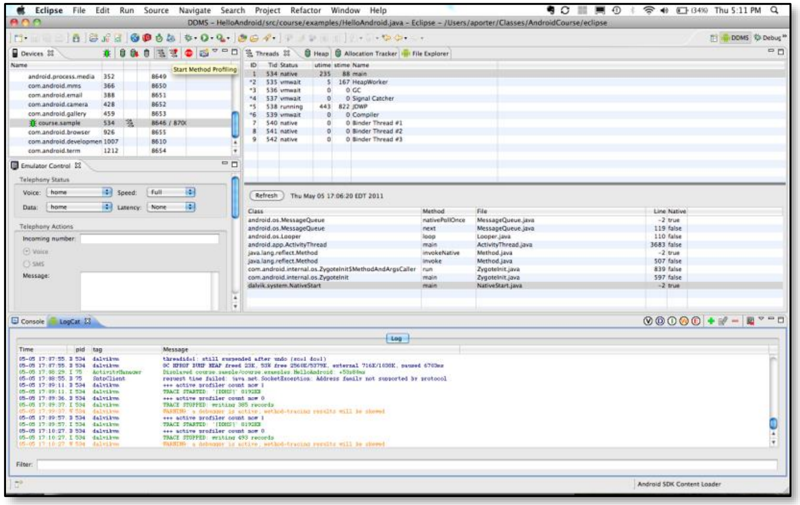
- Concurrent programs are hard to develop & debug, due to various inherent & accidental complexities, e.g.
  - Deadlock
  - Starvation
  - Race conditions
    - *Arise when an application depends on the sequence or timing of threads for it to operate properly*

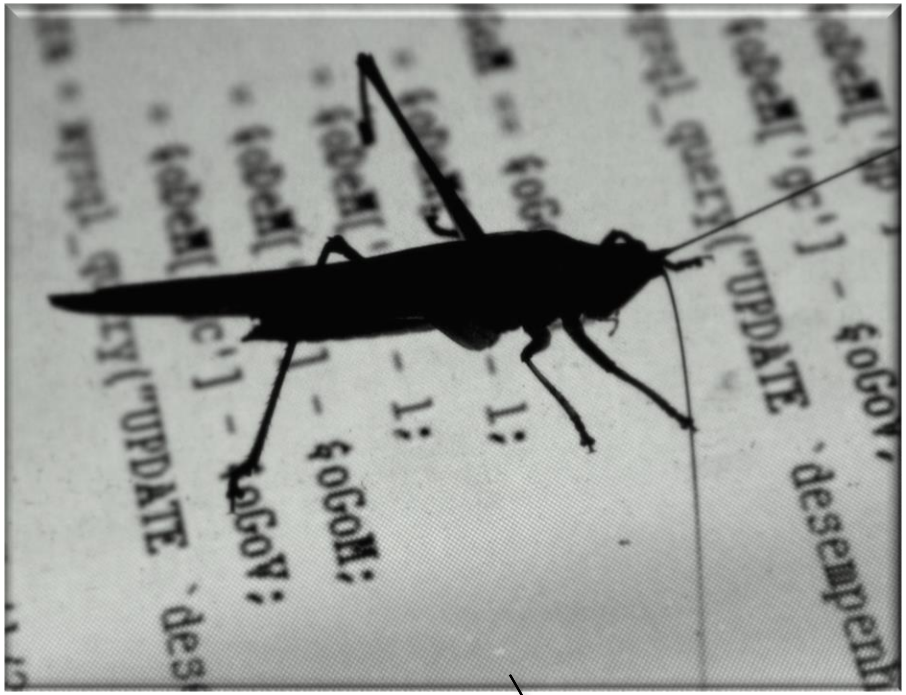


See en.wikipedia.org/wiki/Race_condition

Common Complexities in Concurrent Programs

- Concurrent programs are hard to develop & debug, due to various inherent & accidental complexities, e.g.
  - Deadlock
  - Starvation
  - Race conditions
  - Tool limitations
    - e.g., behavior in the debugger doesn’t reflect actual behavior



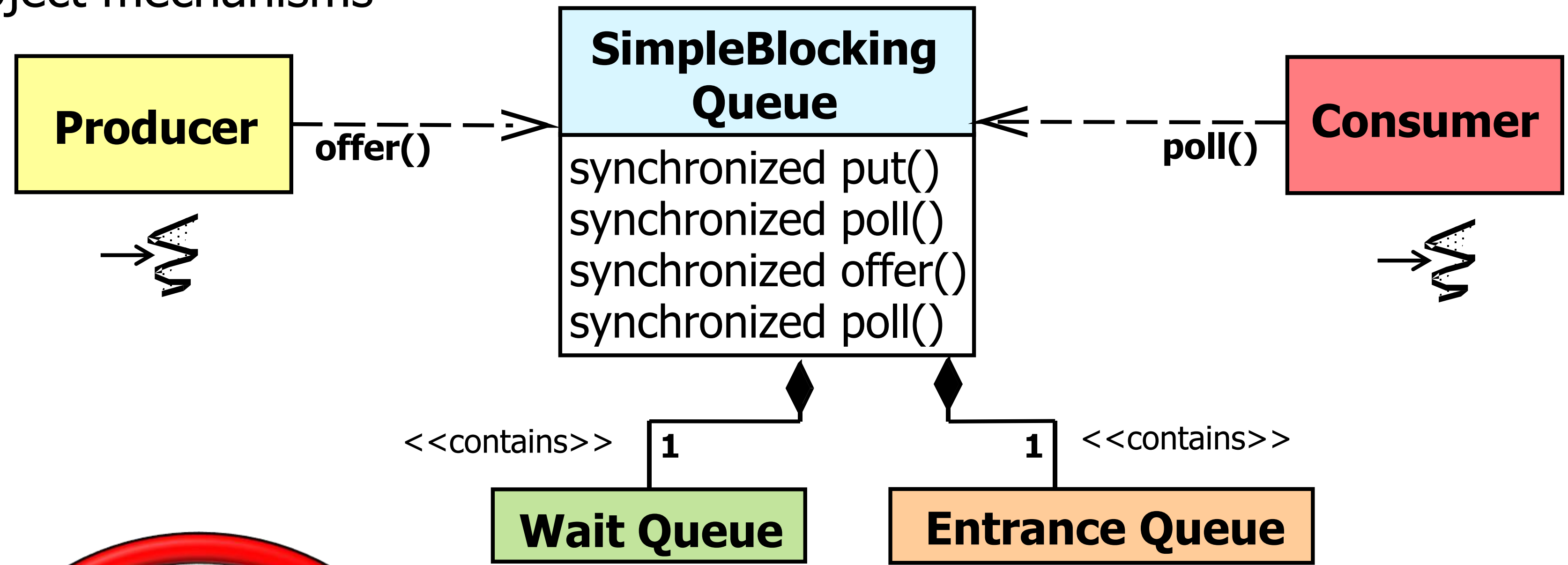


The act of observing a system can alter its state

See en.wikipedia.org/wiki/Heisenbug

Common Complexities in Concurrent Programs

- Some concurrency complexities can be fixed by applying Java built-in monitor object mechanisms



Common Complexities in Concurrent Programs

- There are also helpful techniques for debugging concurrent software

C/C++

Tweet 0 Like 5 Share Permalink

Multithreaded Debugging Techniques

By Shameem Akhter and Jason Roberts, April 23, 2007

Post a Comment

Debugging multithreaded applications can be a challenging task.

Debugging multithreaded applications can be a challenging task. The increased complexity of multithreaded programs results in a large number of possible states that the program may be in at any given time. Determining the state of the program at the time of failure can be difficult; understanding why a particular state is troublesome can be even more difficult. Multithreaded programs often fail in unexpected ways, and often in a nondeterministic fashion. Bugs may manifest themselves in a sporadic fashion, frustrating developers who are accustomed to troubleshooting issues that are consistently reproducible and predictable. Finally, multithreaded applications can fail in a drastic fashion-deadlocks cause an application or worse yet, the entire system, to hang. Users tend to find these types of failures to be unacceptable.

General Debug Techniques

Regardless of which library or platform that you are developing on, several general principles can be applied to debugging multithreaded software applications.

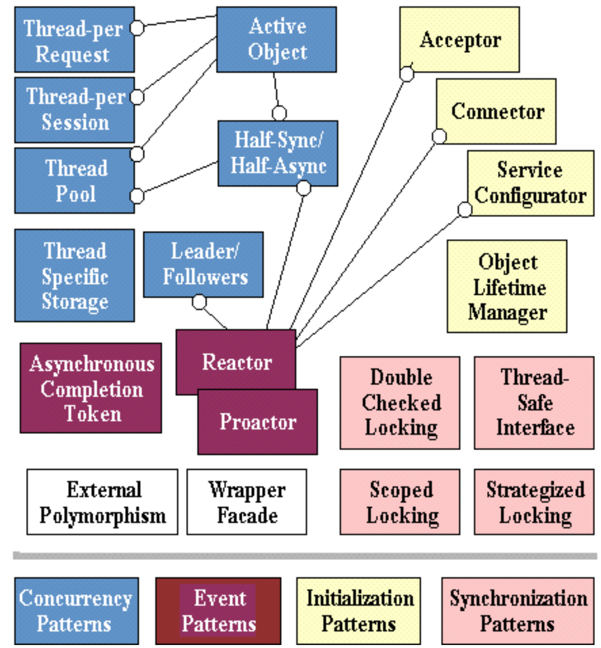
The first technique for eliminating bugs in multithreaded code is to avoid introducing the bug in the first place. Many software defects can be prevented by using proper software development practices. The later a problem is found in the product development lifecycle, the more expensive it is to fix. Given the complexity of multithreaded programs, it is critical that multithreaded applications are properly designed up front.

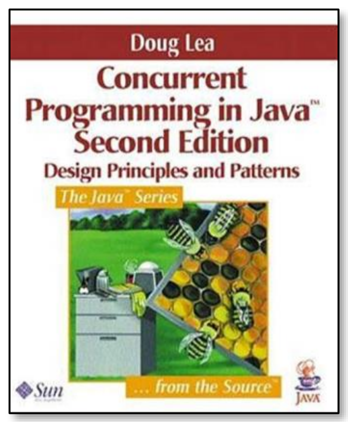
How often have you, as a software developer, experienced the following situation? Someone on the team that you're working on gets a great idea for a new product or feature. A quick prototype that illustrates the idea is implemented and a quick demo, using a trivial use-case, is presented to management. Management loves the idea and immediately informs sales and marketing of the new product or feature. Marketing then informs the customer of the feature, and in order to make a sale, promises the customer the feature in the next release. Meanwhile, the engineering team, whose original intent of presenting the idea was to get resources to properly implement the product or feature sometime in the future, is now faced with the task of delivering on a customer commitment immediately. As a result of time constraints, it is often the case that the only option is to take the prototype, and try to turn it into production code.

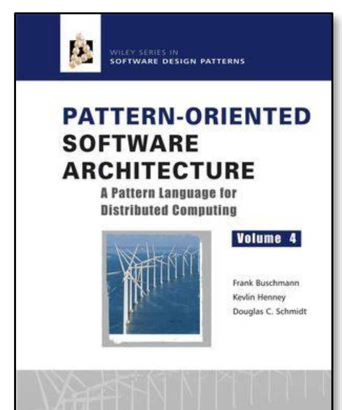
See www.drdobbs.com/cpp/multithreaded-debugging-techniques/199200938

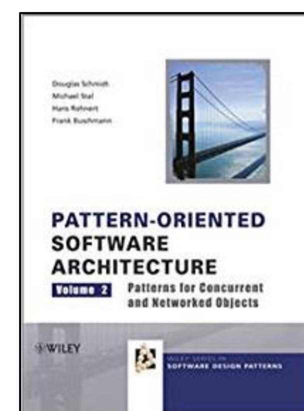
Common Complexities in Concurrent Programs

- There are also helpful techniques for debugging concurrent software, e.g.
  - Use well-established concurrency & synchronization patterns





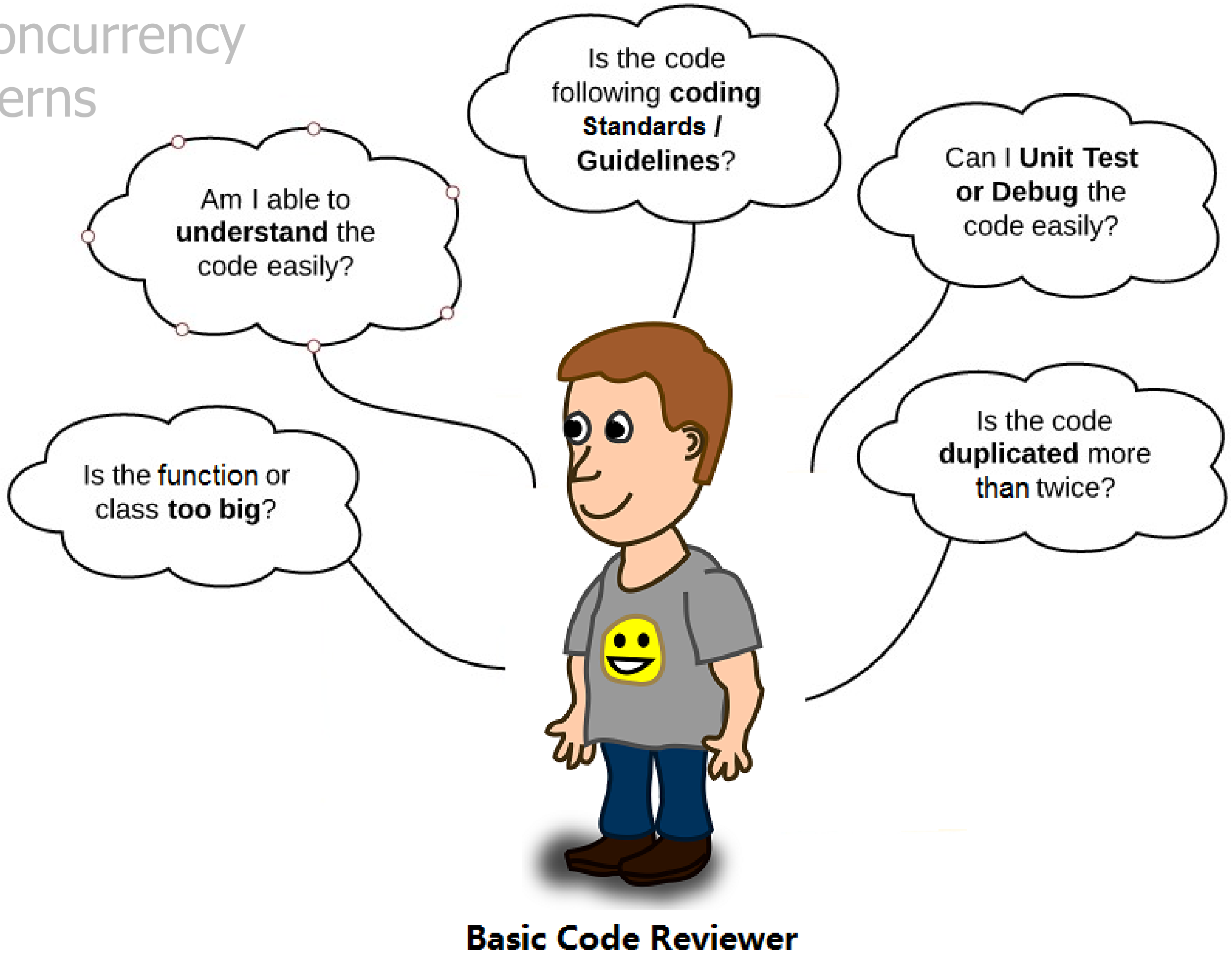




See en.wikipedia.org/wiki/Concurrency_pattern

Common Complexities in Concurrent Programs

- There are also helpful techniques for debugging concurrent software, e.g.
  - Use well-established concurrency & synchronization patterns
  - Conduct code reviews



See en.wikipedia.org/wiki/Code_review

Common Complexities in Concurrent Programs

- There are also helpful techniques for debugging concurrent software, e.g.
  - Use well-established concurrency & synchronization patterns
  - Conduct code reviews
  - Apply analysis tools

Static Analysis Tools for Concurrency

- FindBugs – works on Java. In the list of bugs detected all of the "Multithreaded correctness" bugs are relevant to concurrency. Command-line interface or eclipse plugin (eclipse plugin update site:http://findbugs.cs.umd.edu/eclipse/)
- Lint – a UNIX tool for C
- JLint – a Java version of Lint that is available as stand alone or eclipse plugin (eclipse plugin update site:http://www.jutils.com/eclipse-update)
- Parasoft JTest – commercial tool that combines static analysis and testing. Has capability to check for thread safety in multithreaded Java programs.
- Coverity Static Analysis and Static Analysis Custom Checkers – commercial tool that can be used to create custom static analyzers to find concurrency bugs in C/C++ programs.
- GrammaTech's CodeSonar – commercial tool that can detect a special case race condition and locking issues in C/C++ (see datasheet for list of all bugs detected).
- Chord – static and dynamic analysis tool for Java (listed above as well).
- JSure for Concurrency – a commercial tool from SureLogic that is currently available in early release.
- ESC/Java 2 – can detect race conditions and deadlocks – requires annotation (more…)
- Relay – static race detection
- RacerX – uses flow-sensitive static analysis tool for detection race conditions and deadlocks in C [paper] [slides]
- SyncChecker – a tool developed by F. Otto and T. Moschny for finding race conditions and deadlocks in Java. Reduce false positives by combining static analysis with points-to and may-happen-in-parallel (MHP) information.
- Warlock – race detection tool for C – requires annotation.

See www.sqrlab.ca/blog/2012/03/02/static-analysis-tools-for-concurrency

Common Complexities in Concurrent Programs

- There are also helpful techniques for debugging concurrent software, e.g.
  - Use well-established concurrency & synchronization patterns
  - Conduct code reviews
  - Apply analysis tools
  - Instrument code with logging & tracing statements

C:\WINDOWS\system32\cmd.exe

```
 --> main
   --> base_class::base_class
     counter=[0]
     param=[setec astronomy]
   <-- base_class::base_class in 0s
   --> class_a
   <-- class_a in 0.015s
   --> base_class::foobar
     --> base_class::foo
     <-- base_class::foo in 0s
     --> class_a::bar
       --> class_a::some_helper
         x=[1]
         y=[10]
         z=[a]
       <-- class_a::some_helper in 0s
     <-- class_a::bar in 0.016s
     counter=[109]
   <-- base_class::foobar in 0.016s
   --> base_class::base_class
     counter=[0]
     param=[default b param]
   <-- base_class::base_class in 0s
   --> class_b
   <-- class_b in 0s
   --> base_class::foobar
     --> base_class::foo
     <-- base_class::foo in 0s
     --> class_b::bar
       will_fail=[0]
       --> base_class::bar
       <-- base_class::bar in 0.015s
     <-- class_b::bar in 0.015s
     counter=[2]
   <-- base_class::foobar in 0.015s
   all methods called, compiling results
   collection(integer_list, 2 items)
   {
      [0]=[109]
      [1]=[2]
   }
   --> base_class::base_class
     counter=[0]
     param=[default b param]
   <-- base_class::base_class in 0s
   --> class_b
   <-- class_b in 0.016s
   --> base_class::foobar
     --> base_class::foo
     <-- base_class::foo in 0s
     --> class_b::bar
       will_fail=[1]
       throwing an exception as I was configured to fail
*    <-- class_b::bar in 0s
*  <-- base_class::foobar in 0s
bound to happend
 <-- main in 0.093s
Press any key to continue . . . _
```

See www.dre.vanderbilt.edu/~schmidt/PDF/DSIS_Chapter_Waddington.pdf