

The Java Monitor Object

Synchronized Method Example



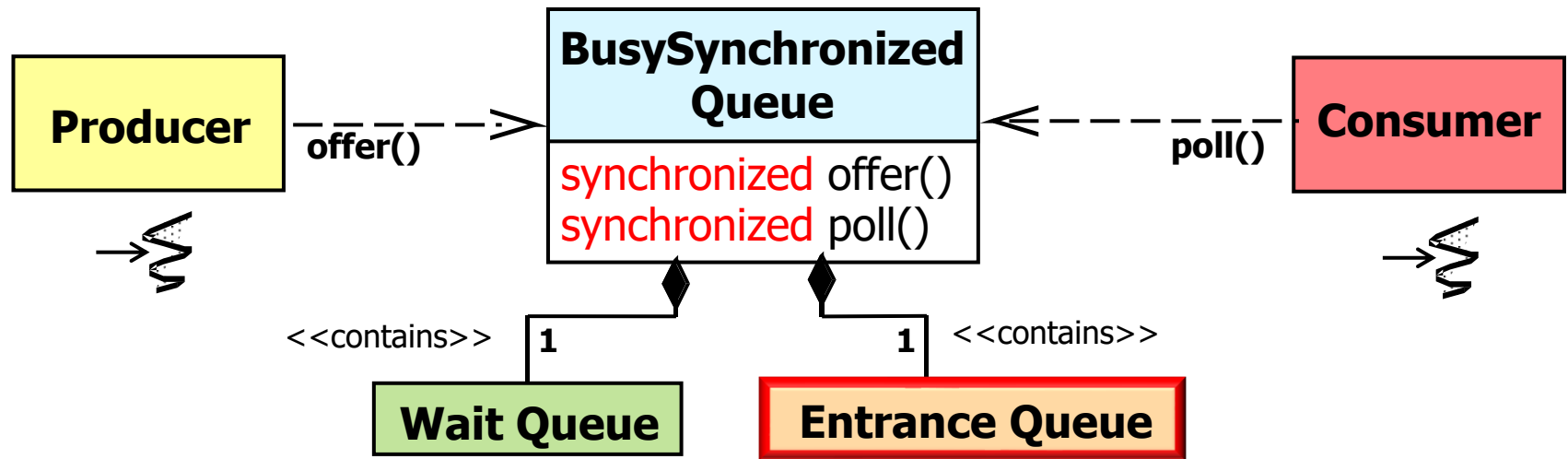
Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

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



Learning Objectives in this Part of the Lesson

- Recognize the synchronized methods/statements provided by Java build-in monitor objects to support *mutual exclusion*
- Understand how to fix race conditions in the buggy concurrent Java app by using synchronized methods



The use of synchronized methods only provides a partial solution, however...

Partial Solution Using Java Synchronized Methods

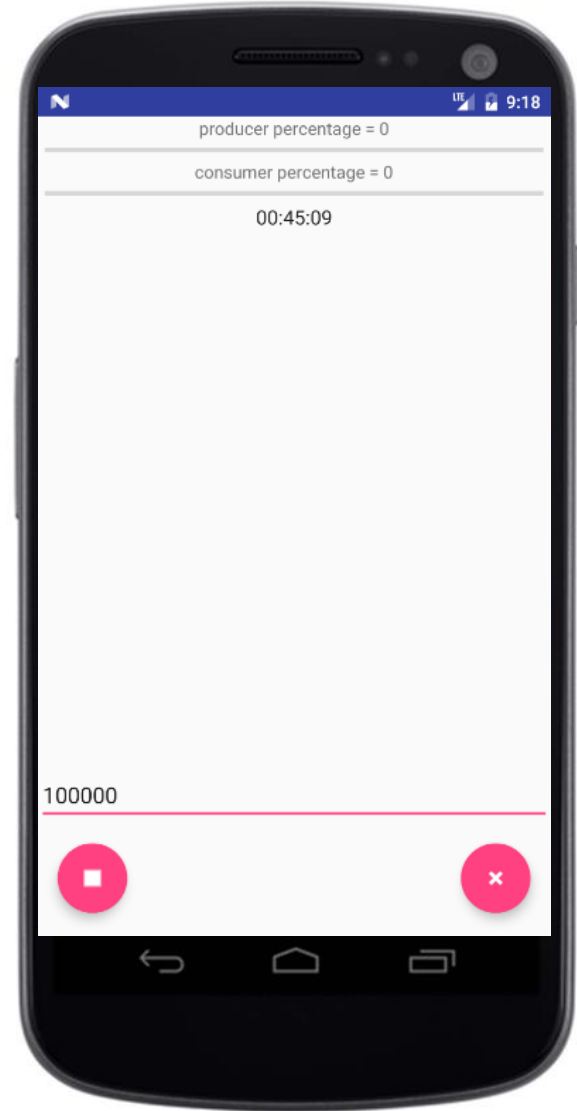
Partial Solution Using Java Synchronized Methods



See en.wikipedia.org/wiki/Crazy_Horse_Memorial

Partial Solution Using Java Synchronized Methods

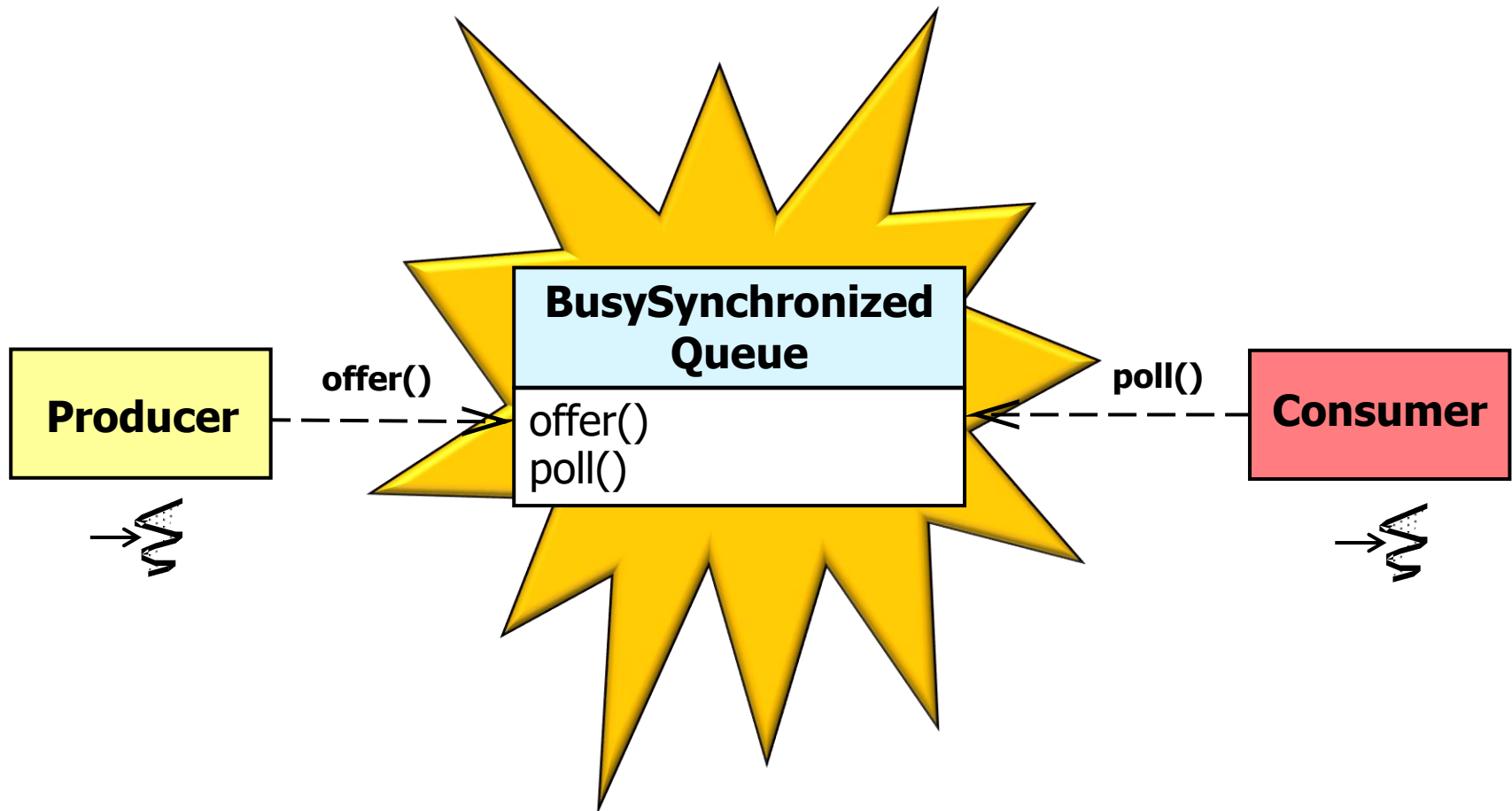
- A concurrent producer/consumer app that passes messages via the class "BusySynchronizedQueue"



See github.com/douglasraigschmidt/POSA/tree/master/ex/M3/Queues/BusySynchronizedQueue

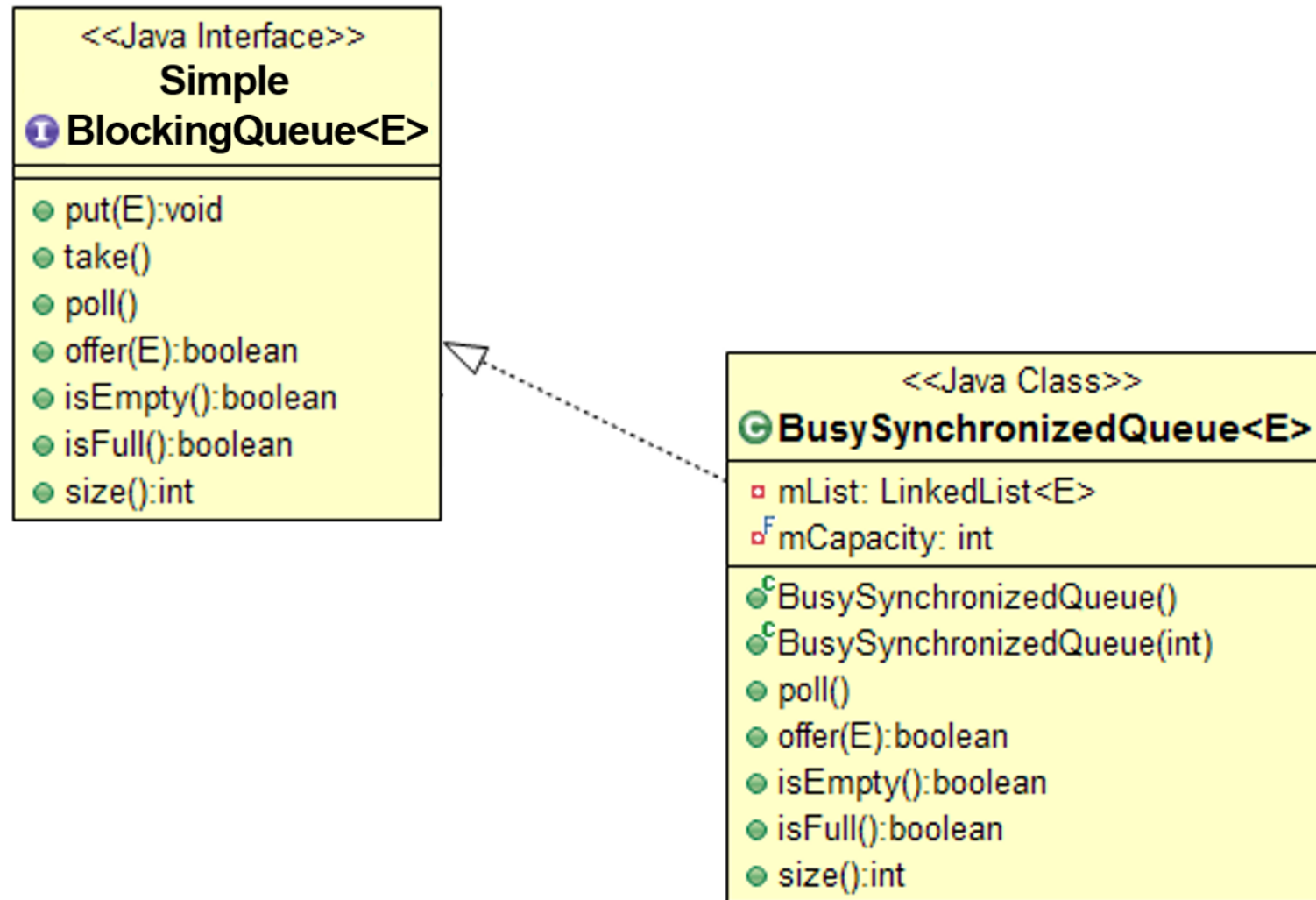
Partial Solution Using Java Synchronized Methods

- The BusySynchronizedQueue is modeled on the Java ArrayBlockingQueue



Partial Solution Using Java Synchronized Methods

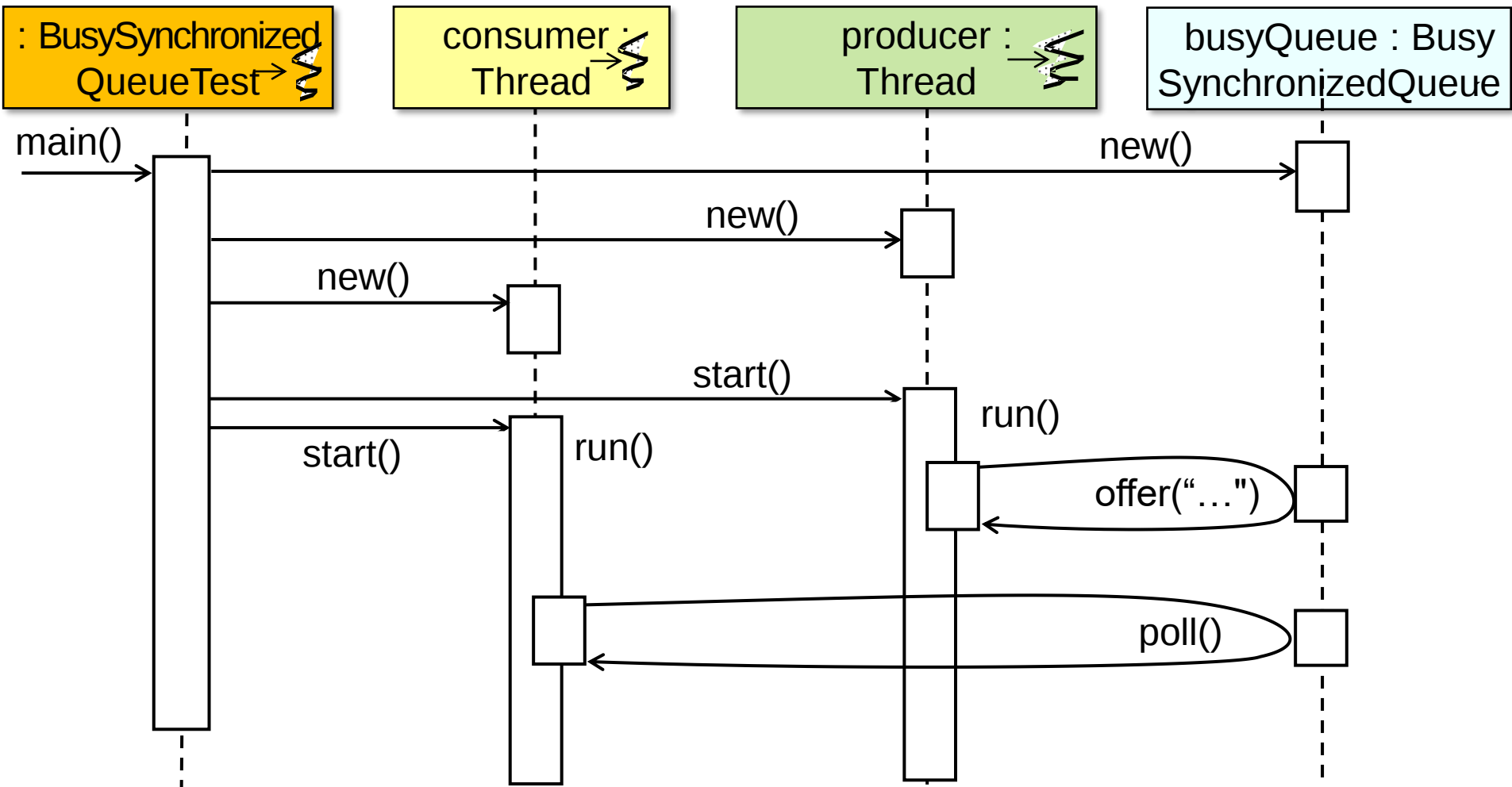
- UML class diagram showing the design of the BusySynchronizedQueue



See gM3/Queues/BusySynchronizedQueue/app/src/main/java/edu/vandy/busysynchronizedqueue/model

Partial Solution Using Java Synchronized Methods

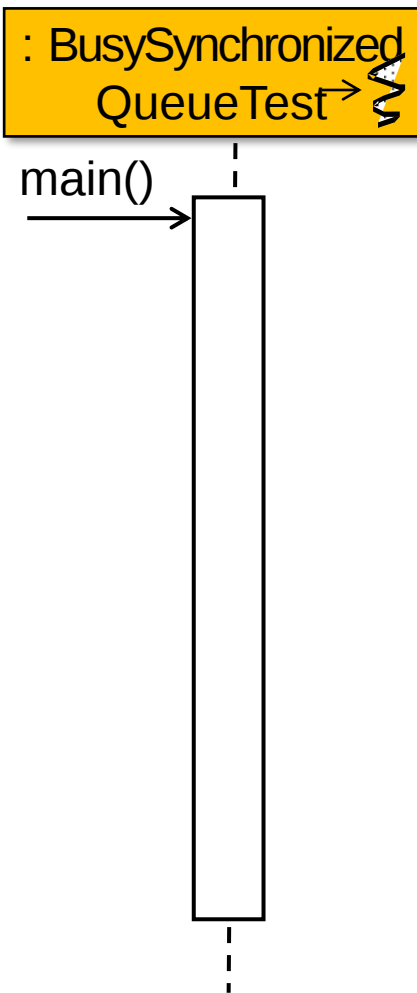
- UML sequence diagram of the BusySynchronizedQueue unit test



See github.com/douglasraigschmidt/POSA/tree/master/ex/M3/Queues/BusySynchronizedQueue/app/src/test/java/edu/vandy/busysynchronizedqueue

Partial Solution Using Java Synchronized Methods

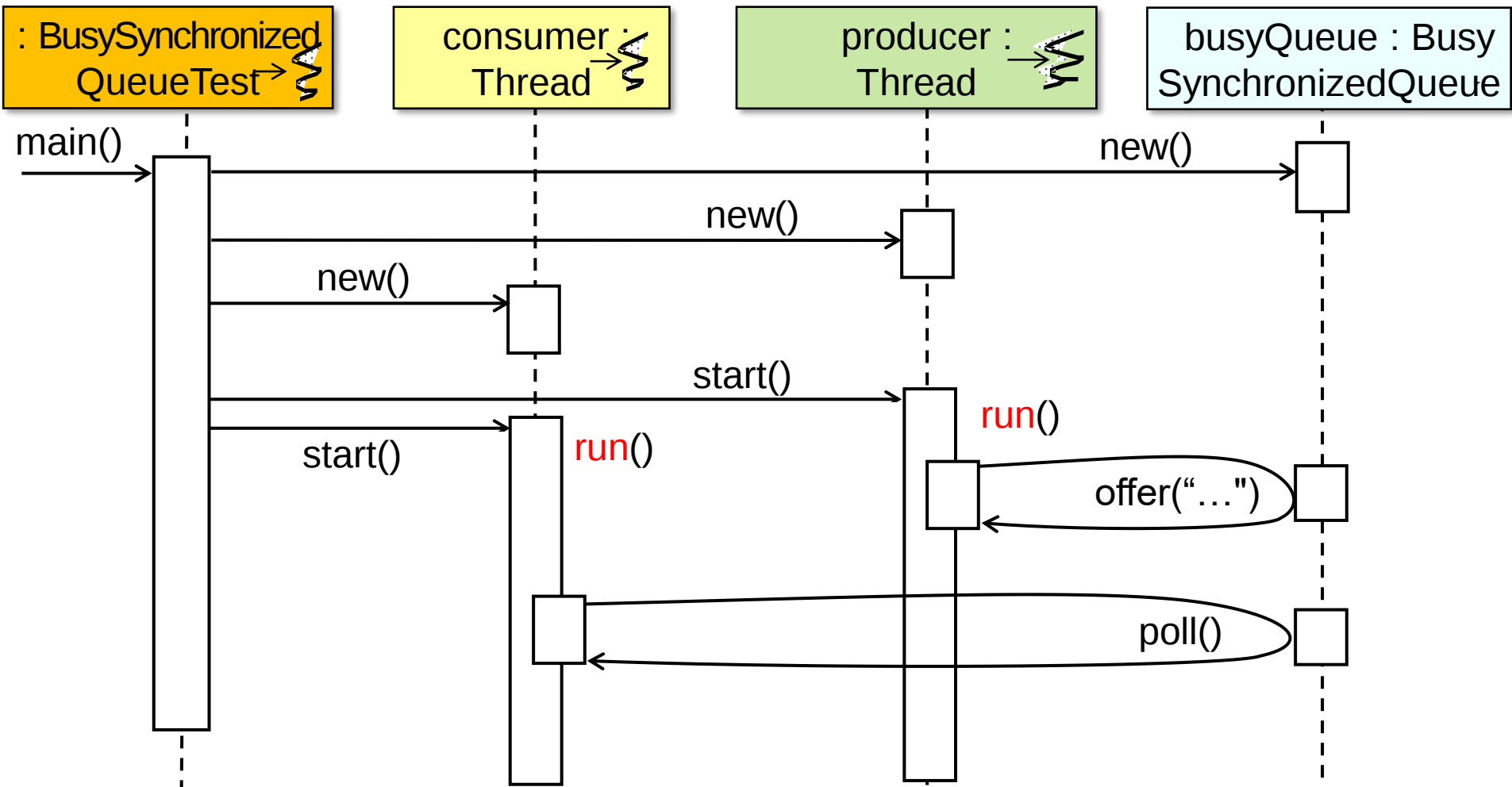
- UML sequence diagram of the BusySynchronizedQueue unit test



The main thread coordinates the other threads in the test

Partial Solution Using Java Synchronized Methods

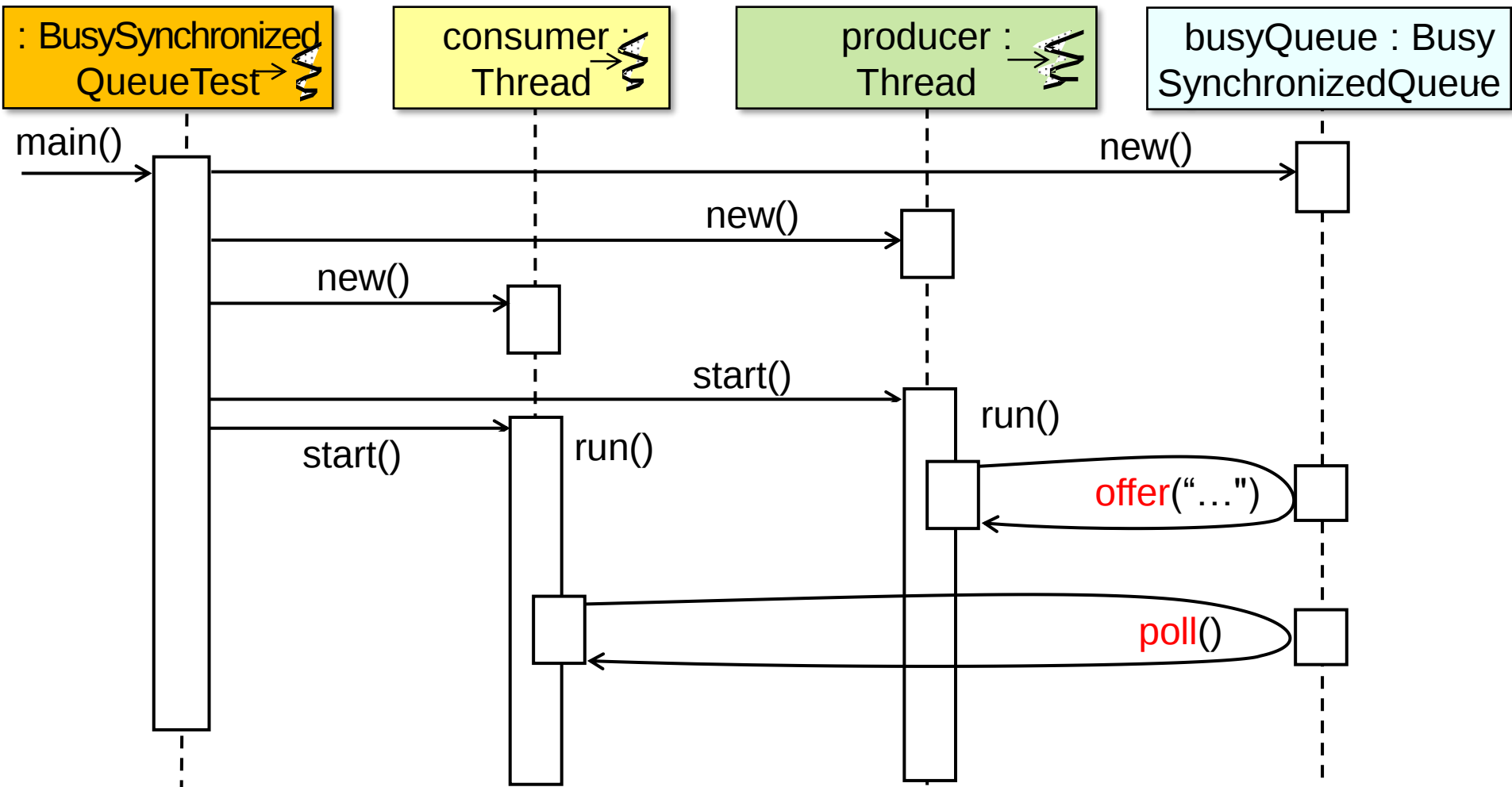
- UML sequence diagram of the BusySynchronizedQueue unit test



The consumer & producer threads generate & process messages sent via the BusySynchronizedQueue, respectively

Partial Solution Using Java Synchronized Methods

- UML sequence diagram of the BusySynchronizedQueue unit test



Although the test runs correctly (since `offer()` & `poll()` are synchronized) it is inefficient due to the "busy waiting"!!

Implementation of the BusySynchronizedQueue

Implementation of the BusySynchronizedQueue

- Java synchronized methods protects critical sections from concurrent access

```
class BusySynchronizedQueue<E>
    implements SimpleBlockingQueue<E> {
    private List<E> mList;
    private int mCapacity;

    public BusySynchronizedQueue(int capacity) {
        mCapacity = capacity; mList = new LinkedList<>();
    }

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

    public E synchronized poll() { return mList.poll(); }
    ...
}
```

See github.com/douglasraigschmidt/POSA/tree/master/ex/M3/Queues/BusySynchronizedQueue

Implementation of the BusySynchronizedQueue

- Java synchronized methods protects critical sections from concurrent access

```
class BusySynchronizedQueue<E>
```

```
    implements SimpleBlockingQueue<E> {
```

```
    private List<E> mList;
```

```
    private int mCapacity;
```

*Constructor initializes the fields
& requires no synchronization*

```
    public BusySynchronizedQueue(int capacity) {  
        mCapacity = capacity; mList = new LinkedList<>();  
    }
```

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

```
    public E synchronized poll() { return mList.poll(); }  
    ...
```



A constructor is only called once in one thread so there won't be race conditions

Implementation of the BusySynchronizedQueue

- Java synchronized methods protects critical sections from concurrent access

```
class BusySynchronizedQueue<E>
```

```
    implements SimpleBlockingQueue<E> {
```

```
    private List<E> mList;
```

```
    private int mCapacity;
```

```
    public BusySynchronizedQueue(int capacity) {  
        mCapacity = capacity; mList = new LinkedList  
    }
```

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

Only one synchronized method at a time can be active in any given object

```
    public E synchronized poll() { return mList.poll(); }
```

```
    ...
```



Implementation of the BusySynchronizedQueue

- Java synchronized methods protects critical sections from concurrent access

```
class BusySynchronizedQueue<E>
    implements SimpleBlockingQueue<E> {
    private List<E> mList;
    private int mCapacity;

    public BusySynchronizedQueue(int capacity) {
        mCapacity = capacity; mList = new LinkedList<>();
    }

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

    public E synchronized poll() { return mList.poll(); }
    ...
}
```



May be a liability for concurrently accessed objects, e.g., a double-ended queue implemented as a linked list

See cs.nyu.edu/courses/fall07/G22.2631-001/lists.slides2.pdf

Implementation of the BusySynchronizedQueue

- Adding the synchronized keyword has two effects

```
class BusySynchronizedQueue<E>
    implements SimpleBlockingQueue<E> {
    private List<E> mList;
    private int mCapacity;

    public BusySynchronizedQueue(int capacity,
        mCapacity = capacity; mList = new LinkedList<>();
    }

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

    public E synchronized poll() { return mList.poll(); }
    ...
}
```



See docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

Implementation of the BusySynchronizedQueue

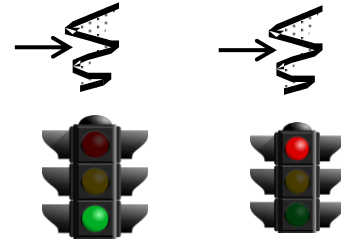
- Adding the synchronized keyword has two effects

```
class BusySynchronizedQueue<E>
    implements SimpleBlockingQueue<E> {
    private List<E> mList;
    private int mCapacity;

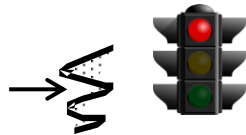
    public BusySynchronizedQueue(int capacity) {
        mCapacity = capacity; mList = new LinkedList<>();
    }

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

    public E synchronized poll() { return mList.poll(); }
    ...
}
```



Invocations of offer() & poll() on the same object can't interleave



i.e., each synchronized method is "atomic"

Implementation of the BusySynchronizedQueue

- Adding the synchronized keyword has two effects

```
class BusySynchronizedQueue<E>
    implements SimpleBlockingQueue<E> {
    private List<E> mList;
    private int mCapacity;

    public BusySynchronizedQueue(int capacity) {
        mCapacity = capacity; mList = new LinkedList<>();
    }

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

    public E synchronized poll() { return mList.poll(); }
    ...
}
```

Establishes a "happens-before" relation to ensure visibility of state changes to all threads

See en.wikipedia.org/wiki/Happened-before

End of the Java Monitor Object Synchronized Method Example