Parallel Programming

Recitation Session 9

Thomas Weibel <weibelt@ethz.ch>

Laboratory for Software Technology,
Swiss Federal Institute of Technology Zürich

May 6, 2010

Executive Summary

- Lack of fairness when using monitors
  - Alternatives when we have to guarantee fairness
- JCSP: Classroom exercises
- MergeSort implementation using JCSP
- Swing and the Model-View-Controller pattern
- Conway's Game of Life

Outline

Monitors

```
public class Synchronizer {
  public synchronized void doSynchronized() {
    //do a lot of work which takes a long time
  }
}
```

No fairness!

- If more than one thread call the `doSynchronized()` method, some of them will be blocked until the first thread granted access has left the method
- If more than one thread are blocked waiting for access there is no guarantee about which thread is granted access next

wait()

- Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object
- A thread can also wake up without being notified, interrupted, or timing out: spurious wakeup
- While this will rarely occur in practice, applications must guard against it by testing for the condition that should have caused the thread to be awakened, and continuing to wait if the condition is not satisfied:

```
synchronized (obj) {
  while (<condition does not hold>) {
    obj.wait();
  }
  // continue
}
```

notify()

- Wakes up a single thread that is waiting on this object's monitor
- If any threads are waiting on this object, one of them is chosen to be awakened
- The choice is arbitrary and occurs at the discretion of the implementation
- The awakened thread will not be able to proceed until the current thread frees the lock on this object
- The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object
  - $\Rightarrow$ Enjoys no reliable privilege or disadvantage in being the next thread to lock this object

notifyAll()

- Wakes up all threads that are waiting on this object's monitor
- The awakened threads will not be able to proceed until the current thread frees the lock on this object
- The awakened threads will compete in the usual manner with any other threads that might be actively competing to synchronize on this object
  - $\Rightarrow$ Enjoys no reliable privilege or disadvantage in being the next thread to lock this object

Semaphore

- `java.util.concurrent.Semaphore`
  - `acquire()` instead of `P()`
  - `release()` instead of `V()`
- Constructors
  - `Semaphore(int permits)`
  - `Semaphore(int permits, boolean fair)`
    - `permits`: initial value
    - `fair`: if true then the semphore uses a FIFO to manage blocked threads
- When fairness is set `true`, the semaphore guarantees that threads invoking any of the `acquire` methods are selected to obtain permits in the order in which their invocation of those methods was processed (first-in-first-out; FIFO)

Lock

- `java.util.concurrent.locks.ReentrantLock`
- `ReentrantLock(boolean fair)`: if this lock should use a fair ordering policy
- When set true, under contention, locks favor granting access to the longest-waiting thread
- Programs using fair locks accessed by many threads may display lower overall throughput than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation
- Fairness of locks does not guarantee fairness of thread scheduling
- See `http://java.sun.com/javase/6/docs/api/`

Using Locks

```
class Foo {
  private final ReentrantLock lock =
    new ReentrantLock();

  public void bar() {
    // block until condition holds
    lock.lock();
    try {
      // method body
    } finally {
      lock.unlock()
    }
  }
}
```

Outline

Exercise 1

What is the output of the following JCSP program?

```
import org.jcsp.lang.*;

public class ClassroomExercise1 {
  public static void main(String[] args) {
    One2OneChannel chanA = Channel.one2one();
    One2OneChannel chanB = Channel.one2one();

    System.out.println("Start");
    new Parallel(
      new CSProcess[] {
        new Process1(chanA.out(), chanB.out()),
        new Process2(chanA.in(), chanB.in())
      }
    ).run();
    System.out.println("End");
  }
}
```

Exercise 1: Proccess 1

```
class Process1 implements CSProcess {
  private ChannelOutput out1;
  private ChannelOutput out2;

  public Process1(ChannelOutput out1,
                  ChannelOutput out2) {
    this.out1 = out1;
    this.out2 = out2;
  }

  public void run() {
    for (int i = 2; i <= 100; i = i + 2) {
      out1.write(new Integer(i));
      out2.write(new Integer(i + 1));
    }
  }
}
```

Exercise 1: Process 2

```
class Process2 implements CSProcess {
  private ChannelInput in1;
  private ChannelInput in2;

  public Process2(ChannelInput in1, ChannelInput in2) {
    this.in1 = in1;
    this.in2 = in2;
  }

  public void run() {
    while (true) {
      Integer d = (Integer) in2.read();
      System.out.print("Read: " + d);

      d = (Integer) in1.read();
      System.out.println(" " + d);
    }
  }
}
```

Exercise 1: Hints

- CSP provides synchronous communication
- Synchronous communication implies that sender and receiver must "meet" to exchange data
  - → if one of them does not reach the correct communication operations, the other one waits for ever

Exercise 1: Solution

- Process 1:
  1. Write on channel A
  2. Write on channel B
- Process 2:
  1. Read on channel B
  2. Read on channel A

→ Result: no progress!

Exercise 2

What is the output of the following JCSP program?

```
import org.jcsp.lang.*;

public class ClassroomExercise2 {
  public static void main(String[] args) {
    One2OneChannel chanA = Channel.one2one();
    One2OneChannel chanB = Channel.one2one();

    System.out.println("Start");
    new Parallel(
      new CSProcess[] {
        new ProcessA(chanA.out(), chanB.in()),
        new ProcessB(chanB.out(), chanA.in())
      }
    ).run();
    System.out.println("End");
  }
}
```

Exercise 2: Proccess A

```
class ProcessA implements CSProcess {
  private ChannelOutput out1;
  private ChannelInput in1;

  public ProcessA(ChannelOutput out, ChannelInput in) {
    this.out1 = out;
    this.in1 = in;
  }

  public void run() {
    for (int i = 1; i <= 5; i++) {
      out1.write(new Integer(i));
      System.out.println("Sent " + i);
    }
    for (int i = 1; i <= 5; i++) {
      Integer d = (Integer) in1.read();
      System.out.println("Read " + d.intValue());
    }
  }
}
```

Exercise 2: Process B

```
class ProcessB implements CSProcess {
  private ChannelInput in1;
  private ChannelOutput out1;

  public ProcessB(ChannelOutput out, ChannelInput in) {
    this.in1 = in;
    this.out1 = out;
  }

  public void run() {
    for (int i = 1; i <= 5; i++) {
      out1.write(new Integer(i));
      System.out.println("Sent " + i);
    }
    for (int i = 1; i <= 5; i++) {
      Integer d = (Integer) in1.read();
      System.out.println("Read " + d.intValue());
    }
  }
}
```

Exercise 2: Solution

- Process 1:
  1. Write on channel A
  2. Read on channel B
- Process 2:
  1. Write on channel B
  2. Read on channel A

→ Result: no progress!

Exercise 3

Rewrite the following program using channels with JCSP

```
public class SimpleSharedObject {
  public static void main(String[] args) {
    Buffer b = new Buffer();

    Thread t1 = new Thread(new Producer(b));
    Thread t2 = new Thread(new Consumer(b));
    t1.start();
    t2.start();
    try {
      t1.join();
      t2.join();
    } catch (InterruptedException E) {
    }

    System.out.println("Driver finished");
  }
}
```

Exercise 3: Buffer

```
class Buffer {
  Integer data; boolean full = false;

  synchronized void put(Integer i) {
    while (true) {
      try {
        if (full)
          wait();
        data = i; full = true; notifyAll(); break;
      } catch (InterruptedException e) {}
    }
  }
  synchronized Integer get() {
    Integer tmp;
    while (true) {
      try {
        if (!full)
          wait();
        tmp = this.data; full = false; notifyAll(); break;
      } catch (InterruptedException e) {}
    }
    return tmp;
  }
}
```

Exercise 3: Producer

```
class Producer implements Runnable {
  Buffer b;

  public Producer(Buffer b) {
    this.b = b;
  }

  public void run() {
    for (int i = 0; i < 10; i++) {
      System.out.println("Write: " + i);
      b.put(new Integer(i));
    }
    System.out.println("Producer finished");
  }
}
```

Exercise 3: Consumer

```
class Consumer implements Runnable {
  Buffer b;

  public Consumer(Buffer b) {
    this.b = b;
  }

  public void run() {
    for (int i = 0; i < 10; i++) {
      Integer d = b.get();
      System.out.println("Read: " + d.intValue());
    }
    System.out.println("Consumer finished");
  }
}
```

Exercise 3: Solution – Setup

```
import org.jcsp.lang.*;

public class SimpleJCSP {
  public static void main(String[] args) {
    One2OneChannel chan = Channel.one2one();

    new Parallel(
      new CSProcess[] {
        new CSPProducer(chan.out()),
        new CSPConsumer(chan.in())
      }
    ).run();
    System.out.println("Driver finished");
  }
}
```

Exercise 3: Solution – Producer

```
class CSPProducer implements CSProcess {
  private ChannelOutput out;

  public CSPProducer(ChannelOutput out) {
    this.out = out;
  }

  public void run() {
    for (int i = 0; i < 10; i++) {
      System.out.println("Write: " + i);
      out.write(new Integer(i));
    }
    System.out.println("Producer finished");
  }
}
```

Exercise 3: Solution – Consumer

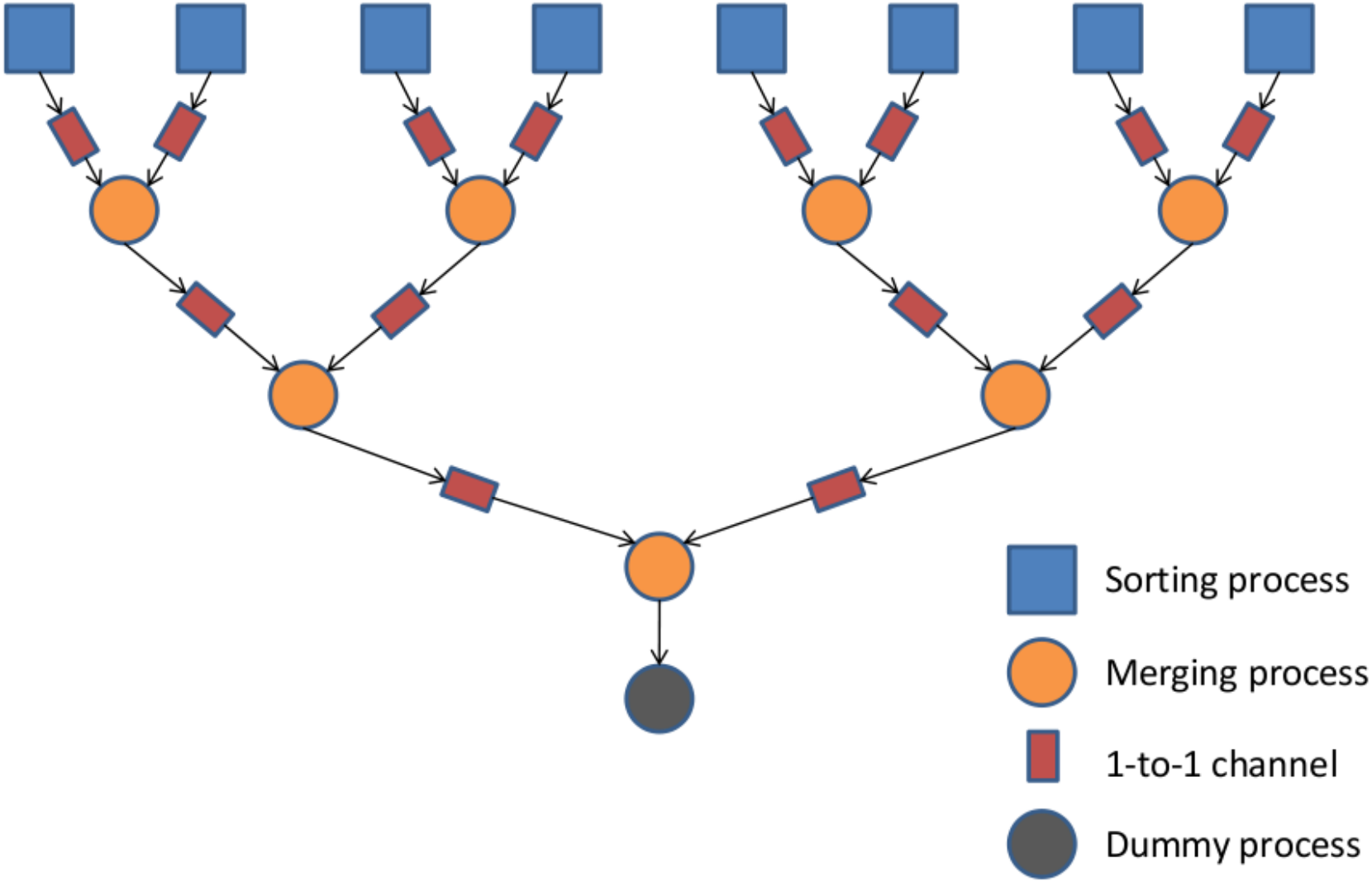
```
class CSPConsumer implements CSProcess {
  private ChannelInput in;

  public CSPConsumer(ChannelInput in) {
    this.in = in;
  }

  public void run() {
    for (int i = 0; i < 10; i++) {
      Integer d = (Integer) in.read();
      System.out.println("Read: " + d.intValue());
    }
    System.out.println("Consumer finished");
  }
}
```

Outline

Entities Layout



MergeSort

```
public class MergeSort {
  // ...

  public static void main(String[] args) {
    // create extra receiver process for the final merging process
    One2OneChannel collectingChannel = Channel.one2one();
    DummyReceiverProcess dummyReceiverProcess = new DummyReceiverProcess(
        collectingChannel.in());

    // create thread hierarchy, connecting it to the extra receiver
    divide(1000, 32, collectingChannel.out());

    // put all JCSP processes in an array
    CSProcess[] allProcesses = new CSProcess[sortIndex + mergeIndex + 1];

    // Create sorting and merging processes including the extra
    // receiver and put them into the array
    // ...

    Parallel parallel = new Parallel(allProcesses);
    parallel.run();
  }
}
```

divide()

```
// recursively create sorting/merging processes and the interconnecting
// channels
private static void divide(int size, int noThreads, ChannelOutput out) {
  if (noThreads == 0) {
    SortingProcess sortingProcess = new SortingProcess(
        size, out);
    sortingProcesses.add(sortingProcess);
    sortIndex++;
  } else if (size >= 1) {
    One2OneChannel channel0 = Channel.one2one();
    One2OneChannel channel1 = Channel.one2one();
    MergingProcess mergingProcess = new MergingProcess(
        channel0.in(), channel1.in(), out);
    mergingProcesses.add(mergingProcess);
    mergeIndex++;

    int noThreadsLeft = noThreads - 1;

    divide(size / 2, noThreadsLeft / 2, channel0.out());
    divide(size - size / 2, noThreadsLeft - noThreadsLeft / 2,
        channel1.out());
  }
}
```

Sorting Process

```
public class SortingProcess implements CSProcess {
  private int[] chunk;
  private int chunkLength;
  private ChannelOutput next;

  public SortingProcess(int chunkLength, ChannelOutput next) { /* ... */ }

  public void run() {
    chunk = new int[chunkLength];
    Random r = new Random();
    for (int i = 0; i < chunkLength; i++)
      chunk[i] = r.nextInt(100);

    // sort chunk
    Arrays.sort(chunk);

    // write chunk size on output channel
    next.write(new Integer(chunkLength));
    // write chunk data, one by one, into output channel
    for (int i = 0; i < chunkLength; i++)
      next.write(new Integer(chunk[i]));
  }
}
```

Merging Process

```
public class MergingProcess implements CSProcess {
  private int[] chunk0, chunk1, sortedChunk;
  private int index0 = -1, index1 = -1, value;
  private AltingChannelInput prev0, prev1;
  private ChannelOutput next;

  public MergingProcess(AltingChannelInput prev0,
      AltingChannelInput prev1, ChannelOutput next) { /* ... */ }

  public void run() {
    Guard[] guards = { prev0, prev1 };
    Alternative alt = new Alternative(guards);

    while (true) {
      switch (alt.priSelect()) {
      case 0: // read from first input channel
        value = (Integer) prev0.read();
        // this is the first integer read, i.e., the chunk size
        if (index0 == -1) // chunk size
          chunk0 = new int[value];
        else // actual data
          chunk0[index0] = value;
        index0++;
        break;
```

Merging Process

```
        case 1: // read from second input channel
          value = (Integer) prev1.read();
          if (index1 == -1)
            chunk1 = new int[value];
          else
            chunk1[index1] = value;
          index1++;
          break;
        }
        // polling is finished if both chunks exist and are filled completely
        if (index0 > -1 && index1 > -1 && index0 == chunk0.length
            && index1 == chunk1.length)
          break;
      }

      sortedChunk = new int[chunk0.length + chunk1.length];
      // merge the two sorted chunks ...

      // write sorted chunk on output channel
      next.write(sortedChunk.length);
      for (i = 0; i < sortedChunk.length; i++)
        next.write(new Integer(sortedChunk[i]));
    }
  }
```

Dummy Receiver Process

```
public class DummyReceiverProcess implements CSProcess {
  private ChannelInput prev;
  private int[] x;

  public DummyReceiverProcess(ChannelInput prev) { /* ... */ }

  public void run() {
    // read length of final, hopefully sorted, array
    int length = (Integer) prev.read();
    x = new int[length];
    // read final, hopefully sorted, array
    for (int i = 0; i < length; i++)
      x[i] = (Integer) prev.read();

    // check if the array is really sorted and print out result
    boolean isSorted = true;
    for (int i = 0; i < length - 1; i++)
      if (x[i] > x[i + 1]) {
        isSorted = false;
        break;
      }
    System.out.println("Succes = " + isSorted);
  }
}
```

Outline

Swing

- Swing is a platform-independent, Model-View-Controller GUI framework for Java
- Provides a native look and feel that emulates the look and feel of several platforms
- Supports a pluggable look and feel that allows applications to have a look and feel unrelated to the underlying platform

Example: SimpleButton (Initialization)

```
public class SimpleButton extends JPanel implements ActionListener {
  private final JButton computationButton, updateButton;
  private JTextField display;
  private final Counter counter = new Counter();

  private static void createAndShowGUI() {
    // Create and set up the window.
    JFrame frame = new JFrame("Simple Button with Action Listener");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Create and set up the content pane.
    SimpleButton newContentPane = new SimpleButton();
    newContentPane.setOpaque(true); // content panes must be opaque
    frame.setContentPane(newContentPane);

    // Display the window.
    frame.pack(); frame.setVisible(true);
  }

  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() { createAndShowGUI(); }
    });
  }
```

Example: SimpleButton (Constructor)

```
public SimpleButton() {
  computationButton = new JButton("Start computation");
  computationButton.addActionListener(this);
  computationButton.setActionCommand("compute");

  updateButton = new JButton("Update result");
  updateButton.addActionListener(this);
  updateButton.setActionCommand("update");

  display = new JTextField(20);
  display.setText(Integer.toString(0));
  display.setEditable(false);

  // Add Components to this container, using the default FlowLayout.
  add(computationButton);
  add(updateButton);
  add(display);
}
```

Example: SimpleButton (Action)

```
  /**
   * Starting the action triggers the computation
   */
  public void actionPerformed(ActionEvent e) {
    if ("compute".equals(e.getActionCommand())) {
      Thread[] t = new Thread[3];

      for (int i = 0; i < t.length; i++) {
        t[i] = new Thread(new ComputeTask(counter));
        t[i].start();
      }
    } else {
      // update command
      final int numActiveTasks = ComputeTask.getNumActiveTasks();
      if (numActiveTasks > 0) {
        display.setText(numActiveTasks
            + " are not finished, please wait");
      } else {
        display.setText(Integer.toString(counter.getValue()));
      }
    }
  }
}
```

Example: ComputeTask

```
public class ComputeTask implements Runnable {
  private final Counter counter;
  private static int numActiveTasks = 0;

  public ComputeTask(Counter c) {
    this.counter = c;
  }

  public void run() {
    incrementNumActiveTasks();
    // sleep for a random time
    try {
      Thread.sleep((long) (Math.random() * 5000));
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    counter.increment();
    decrementNumActiveTasks();
  }

  // synchronized methods incrementNumActiveTasks,
  // decrementNumActiveTasks, getNumActiveTasks
  // omitted...
}
```

Example: Counter

```
public class Counter {
  private int val = 0;

  synchronized void increment() {
    val++;
  }

  synchronized int getValue() {
    return val;
  }
}
```

Alternative to writing your own thread-safe wrapper classes:
AtomicInteger (http://java.sun.com/javase/6/docs/api/)

- incrementAndGet()
- decrementAndGet()
- get()

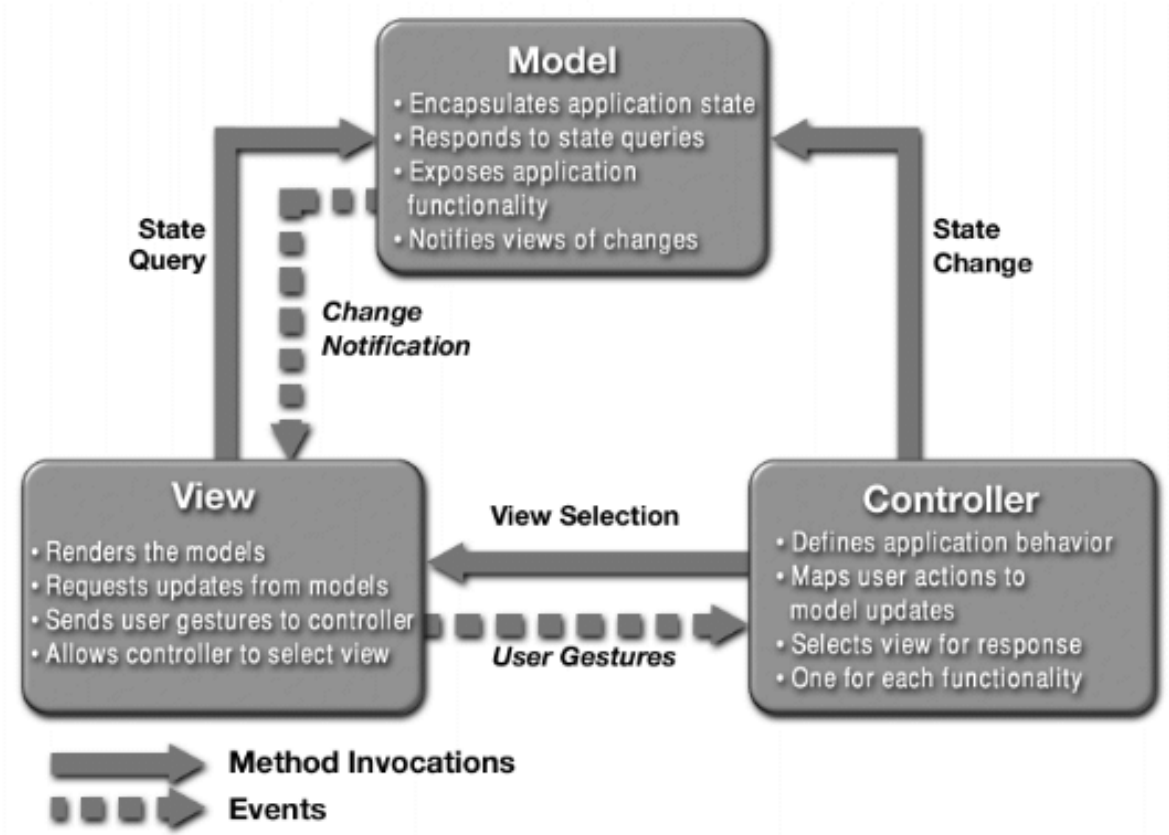
Model-View-Controller

Model: Domain-specific representation of the data upon which the application operates, when a model changes its state, it notifies its associated views so they can be refreshed

View: Renders the model into a form suitable for interaction, typically a user interface element, multiple views can exist for a single model for different purposes

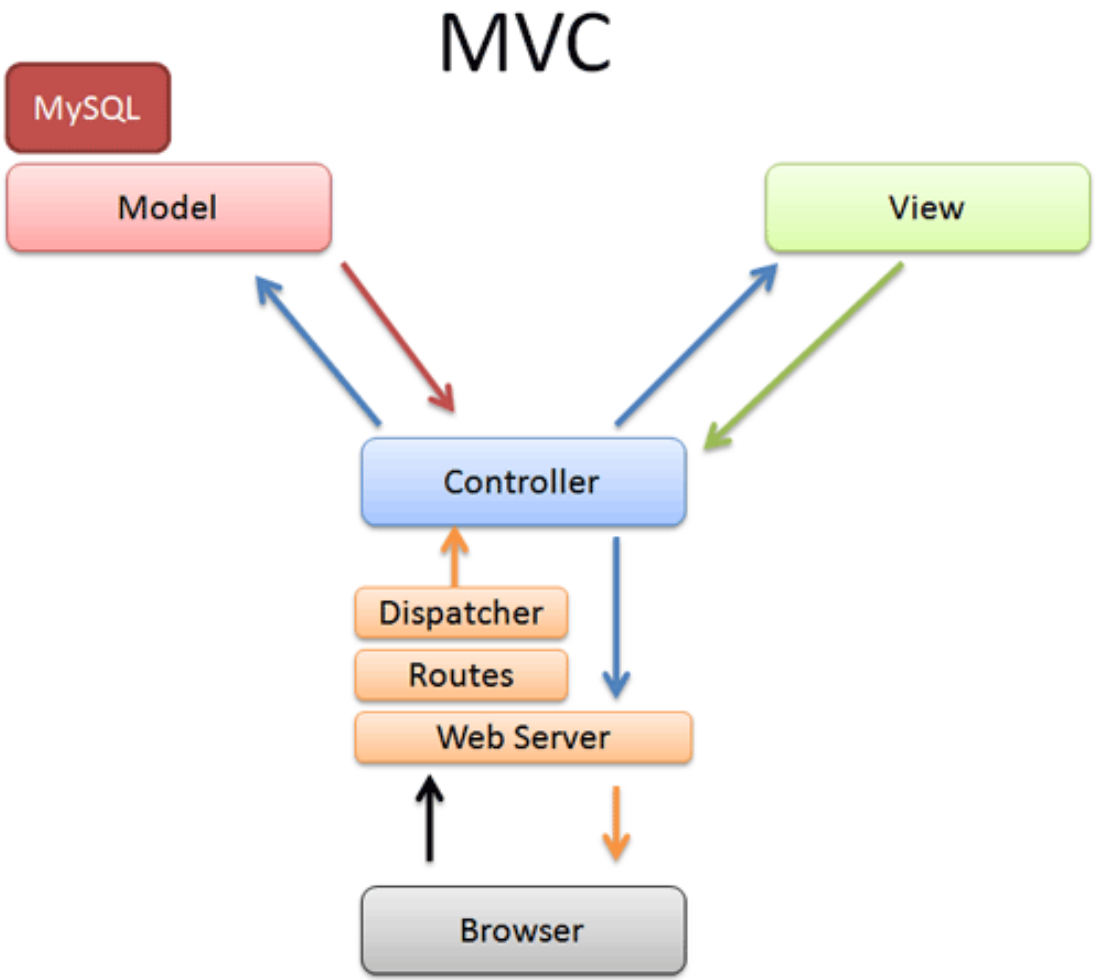
Controller: Receives input and initiates a response by making calls on model objects

Model-View-Controller: Sun



Source: `http://java.sun.com/blueprints/patterns/MVC-detailed.html`

Model-View-Controller: Ruby on Rails



Source: `http://betterexplained.com/articles/intermediate-rails`

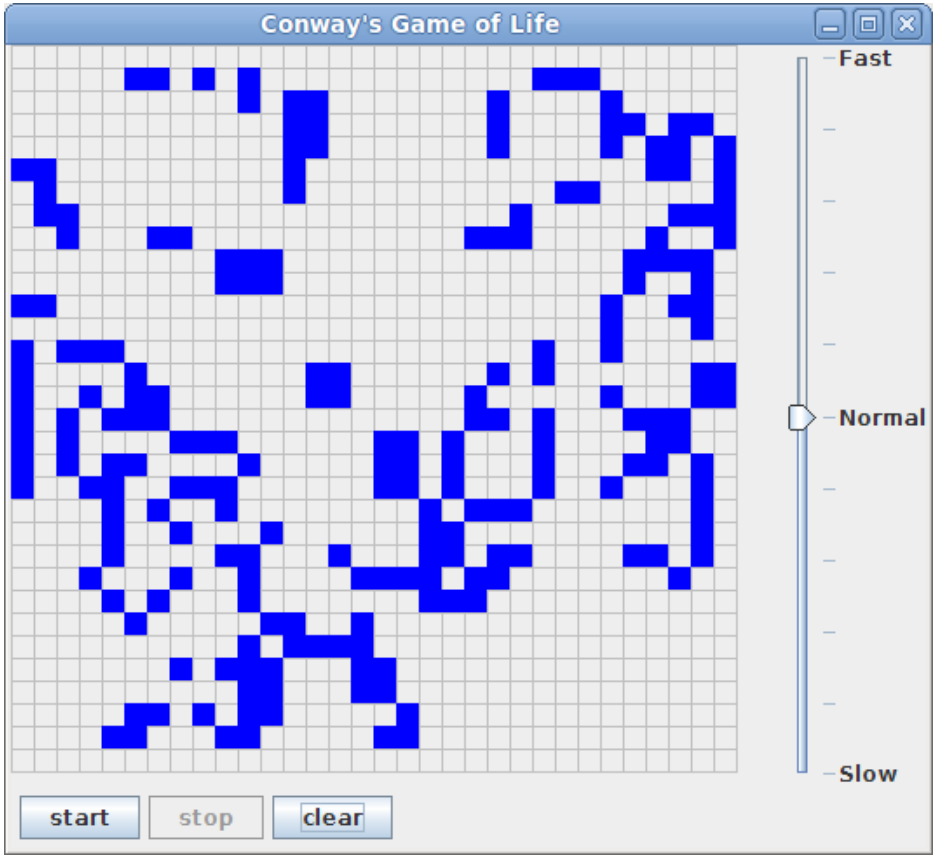
Tutorials

- Creating a GUI With JFC/Swing:
  `http://java.sun.com/docs/books/tutorial/uiswing/`
- Using Swing Components:
  `http://java.sun.com/docs/books/tutorial/uiswing/components/index.html`
- Ein (sehr) kleines Swing Tutorial:
  `http://www.mm.informatik.tu-darmstadt.de/courses/helpdesk/swing.html`

Outline

Conway's Game of Life

- Implement "Game of Life" with GUI based on Swing
- Model-View-Controller Pattern
- You can use the given code skeleton
- Let's see an example



See `http://en.wikipedia.org/wiki/Conway's_Game_of_Life`

Universe

- The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells
  - Our implementation uses a finite grid
- Each cell is in one of two possible states:
  - live
  - dead
- Every cell interacts with its eight neighbors, which are the cells that are directly
  - horizontally
  - vertically
  - diagonally

  adjacent

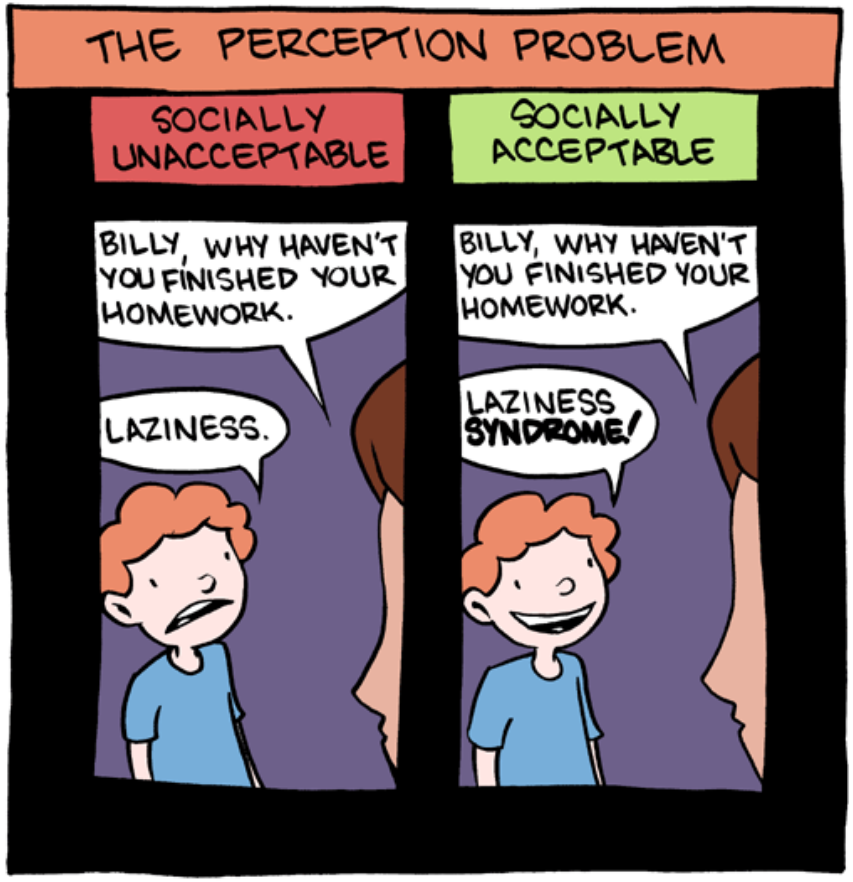
Rules

At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbours dies, as if caused by underpopulation
2. Any live cell with more than three live neighbours dies, as if by overcrowding
3. Any live cell with two or three live neighbours lives on to the next generation
4. Any dead cell with exactly three live neighbours becomes a live cell

Summary

- Monitors don't provide any fairness
- CSP provides synchronous communication
- MergeSort implementation using JCSP
- Try to separate Swing related code from your model
- Conway's Game of Life



Source: `http://www.smbc-comics.com/index.php?db=comics&id=1762`