Patterns, frameworks & Lejos:

part 1: Behavior control
How robots are like insects

• Insects are highly successful in the world, managing to perform the necessary functions of life despite the fact that they’re almost brainless
  – Insects have no memory, so they can’t learn
  – Instead, they employ a variety of instinctual behaviors to react to stimuli

• MIT scientist Rodney Brooks employed the insect model to apply to his work with robotics
  – He developed his ideas under the title “subsumption architecture”
  – The model is more commonly known today as the *behavior control pattern*
Behavior structures

• Sensors: input devices
  – For insects, these might be eyes or antennae
  – For robots: light, touch, and rotation sensors

• Actuators: akin to listeners
  – Determine how a bug (or robot) react to stimuli from sensors
  – Condition/action pairs are called behaviors
Behaviors

• Can be simple:
  – run away if a light comes on (cockroach)
  – back up if you run into something (robot)
• Can be complex:
  – find your way back to the hive (bee)
  – map a maze (robot)
• An individual behavior is a program that a robot (or bug) follows for some period of time
• Behavior control involves switching between modes of behavior in response to a variety of stimuli
Sensors & behaviors

- Sensors can be part of reacting to stimuli, not just reporting on them
- For example, a mapping behavior would require the active participation of a sensor or sensors to measure distances
- Behaviors can be started or stopped for reasons other than direct response to stimuli, as well
- For example, certain behaviors might be programmed to occur automatically at timed intervals
Behaviors and priority

• More than one behavior may be stimulated to kick in at any given time
  – For example, a mosquito might be stimulated to feed when it lands on your arm, but must at the same time be alert to fly away if you move to slap it
  – A robot may need to return to its home base and dump its memory even if it hasn’t finished mapping its course

• We can see that one behavior (staying alive, not overwriting memory) may take precedence and interrupt another behavior (eating, mapping)

• We say that a higher-level behavior may suppress a lower-level behavior
The Lejos Behavior API, located in the josx.robotics.* package, consists of one interface and one class.

The Behavior interface specifies three methods: takeControl, action, and suppress.

The Arbitrator class takes an array of Behavior objects as the argument to its constructor, and provides the higher-level control for the various behaviors.
Behavior interface methods

• boolean takeControl(): returns value indicating whether the behavior object it belongs to should become active; for example, if the touch sensor reports that the robot is blocked, the method should return true

• void action(): initiates activity when the behavior object becomes active

• void suppress(): terminates action method
Behavior types

• Behavior actions come in two varieties:
  – Discrete actions:
    • execute and finish quickly
    • often do not require a suppress() call
  – Continuous actions:
    • run for indefinite periods
    • won’t stop until they are suppressed
    • may run in a separate Thread
Arbitrator class

- **public Arbitrator(Behavior [] behaviors)**
  - creates Arbitrator object that regulates the set of behaviors in the array
  - the array index is used to indicate behavior priority; least important behaviors should be toward the front of the array (0…), most important toward the back (…n)

- **public void start():** starts the Arbitrator object

- The Arbitrator object decides which behavior should be active, and calls the takeControl method of the chosen behavior
Example

• The code samples that follow are intended to code behavior for a robot with two motors and a bumper connected to at least one touch sensor

• The intention is to illustrate how the Behavior Control pattern can be applied to a robot, and in particular how new behaviors can easily be incorporated without affecting behaviors already defined

• All of this code (and the ideas discussed) are from Brian Bagnall’s book *Core Lego Mindstorms Programming*
Behavior 1: robot goes forward

import josx.robotics.*;
import josx.platform.rcx.*;

public class DriveForward
    implements Behavior {
    public boolean takeControl() {
        return true;
    }
    public void suppress () {
        Motor.A.stop();
        Motor.C.stop();
    }
    public void action () {
        Motor.A.forward();
        Motor.C.forward();
    }
}

This class describes the default behavior of the robot (go forward unless something gets in the way), so takeControl always returns true; higher-level behaviors can suppress this if necessary.
Behavior 2: Robot backs up and turns around when it encounters an obstacle

```java
import josx.robotics.*;
import josx.platform.rcx.*;

public class HitWall implements Behavior {
    public boolean takeControl() {
        return Sensor.S2.readBooleanValue();
    }
    public void suppress() {
        Motor.A.stop();
        Motor.C.stop();
    }
}
```

In this case, the `takeControl()` method is based on input from the touch sensor(s) – if the bumper hits something, this behavior takes control.
Driver class: instantiates Behaviors & Arbitrator

```java
import josx.robotics.*;

public class BumperCar {
    public static void main (String [] args) {
        Behavior b1 = new DriveForward();
        Behavior b2 = new HitWall();
        Behavior[] bArray = {b1, b2};
        Arbitrator arby = new Arbitrator(bArray);
        arby.start();
    }
}
```

Note that the HitWall behavior is given a higher priority than the default (DriveForward) behavior, ensuring that the robot won’t just keep trying to run into the wall.
public void action() {

    // back up
    Motor.A.backward();
    Motor.C.backward();

    try {
        Thread.sleep(1000);
    } catch (Exception e) {
    }

    // turn around
    Motor.A.stop();
    try {
        Thread.sleep(300);
    } catch (Exception e) {
    }

    Motor.C.stop();

}
Adding behaviors

• Making the robot’s overall behavior more sophisticated is simply a matter of adding to its repertoire of behaviors; the procedure is:
  – define a class that implements the Behavior interface (and methods takeControl, suppress and action)
  – instantiate the class in the driver class, and add the instance to the array passed to the Arbitrator constructor; add the behavior so that its priority level is appropriate given the existing behavior priorities

• The next set of slides illustrate adding a behavior
New behavior: deal with drained batteries

import josx.robotics.*;
import josx.platform.rcx.*;

public class BatteryLow implements Behavior {
    private float LOW_LEVEL;
    private static final short [] note =
        {2349,115, 0,5, 1760,165, 0,35, 1760,28, 0,13, 1976,23,
         0,18, 1760,18, 0,23, 1568,15, 0,25, 1480,103, 0,18,
         1175,180, 0,20, 1760,18, 0,23, 1976,20, 0,20, 1760,15,
         0,25, 1568,15, 0,25, 2217,98, 0,23, 1760,88, 0,33, 1760,75,
         0,5, 1760,20, 0,20, 1760,20, 0,20, 1976,18, 0,23,
         1760,18, 0,23, 2217,225, 0,15, 2217,218};
    public BatteryLow(float volts) {
        LOW_LEVEL=volts;
    }
}
Example continued

```java
public boolean takeControl() {
    float voltLevel = (Battery.getVoltage());
    int displayNum = (int)(voltLevel * 100);
    LCD.setNumber(0x301f, displayNum, 0x3004);
    LCD.refresh();
    return voltLevel < LOW_LEVEL;
}
```

```java
double suppress () {} 
public void action() {
    play();
    try {Thread.sleep(3000);} catch(Exception e){} 
    System.exit(0); 
}
```
public static void play() {
    for(int x=0; x<note.length; x+=2) {
        final short w = note[x+1];
        Sound.playTone(note[x], w);
        try {
            Thread.sleep(w*10);
        } catch (InterruptedException e) {
        }
    } // ends loop
} // ends method
} // ends class
import josx.robotics.*;

public class BumperCar {
    public static void main (String [] args) {
        Behavior b1 = new DriveForward();
        Behavior b2 = new BatteryLow(6.5f);
        Behavior b3 = new HitWall();
        Behavior[] bArray = {b1, b2, b3};
        Arbitrator arby = new Arbitrator(bArray);
        arby.start();
    }
}

Complexity & conflicts

- As behaviors get added, the possibility of conflicting demands for resources becomes more likely:
  - Threads may be difficult to halt
  - Transient events (such as a touch sensor hit or change in light) may go undetected
- More complex conditions demand more sophisticated code
- One solution to the transient event problem is to incorporate a SensorListener into a Behavior, as demonstrated in the next example
import josx.robotics.*;
import josx.platform.rcx.*;

public class HitWall2 implements Behavior, SensorListener {
    boolean hasCollided;
    public HitWall2() {
        hasCollided = false;
        Sensor.S2.addSensorListener(this);
    }
    public void stateChanged(Sensor bumper, int oldValue, int newValue) {
        if(bumper.readBooleanValue() == true)
            hasCollided = true;
    }
}
public boolean takeControl() {
    if(hasCollided) {
        hasCollided = false;
        return true;
    }
    else
        return false;
}

public void suppress() {
    Motor.A.stop();
    Motor.C.stop();
}

public void action() {
    Motor.A.backward();
    Motor.C.backward();
    try {Thread.sleep(1000);} catch (Exception e) {} 
    Motor.A.stop();
    try {Thread.sleep(300);} catch (Exception e) {} 
    Motor.C.stop();
}
A closer look at arbitration

- Arbitrator object cycles through its array of Behavior objects, checking each takeControl() method to see if the corresponding action() method should be called.
- When it encounters a true value from a takeControl(), it executes suppress() on the Behavior currently in action (provided the new Behavior does not have a lower priority), and runs the new action() method.
- When action() returns, the Arbitrator starts its loop again:
  - If takeControl() from the current Behavior still says true, the Arbitrator does not run the same action again.
  - This prevents a single behavior from constantly running and suppressing itself.
  - Instead, the Arbitrator moves on to another Behavior, and when that one completes its action the Arbitrator calls action() on the lower level Behavior again.