Robotics and Autonomous Systems

Lecture 11: Behaviour-based robotics

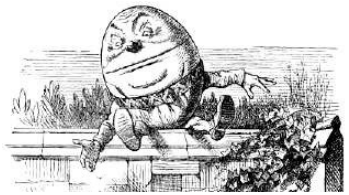
Richard Williams

Department of Computer Science
University of Liverpool

UNIVERSITY OF LIVERPOOL

Today

- One topic, two views

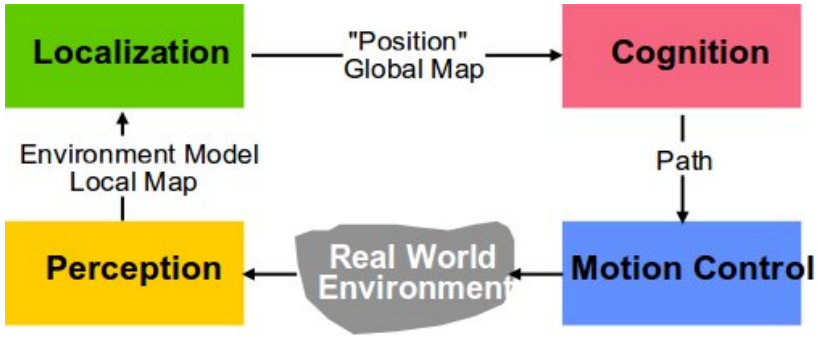


- Behavior-based approaches are an important area of research in robotics.
- They also provide a convenient way to program your robot.
- They are somewhat supported in LeJOS.

Acknowledgements

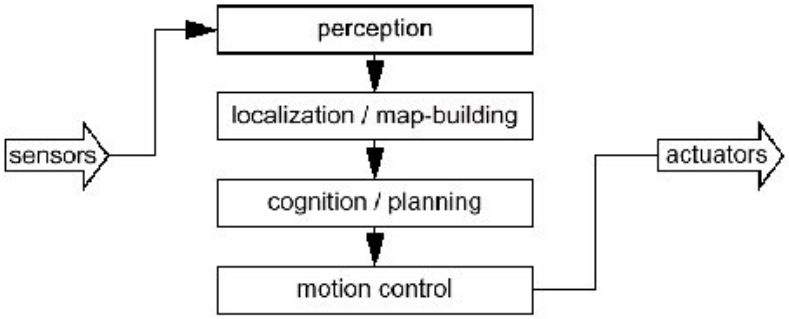
- Some sections based on Maja Matarić's "Introduction to Robotics".
- Code based on examples from `www.lejos.org`

Behaviors



- Our control loop diagram implies an ordering over the operations.

Behaviors

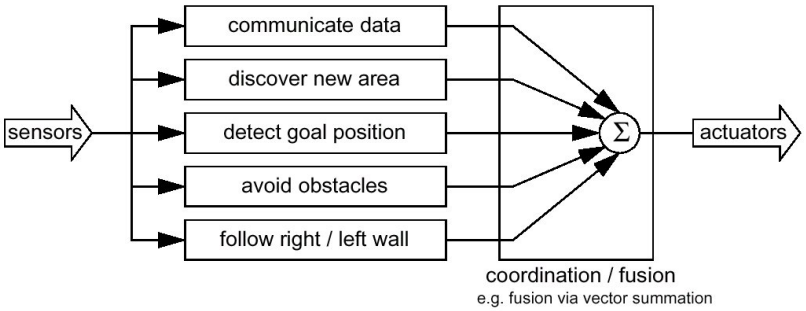


- Classic “Sense/Plan/Act” approach breaks it down serially like this.

Behaviors

- Behavior-based control sees things differently.

Behaviors

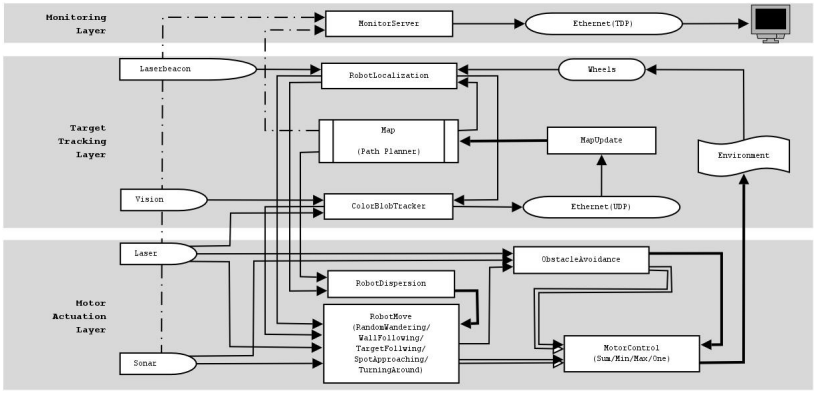


- Behavioral chunks of control each connecting sensors to motors.
- Implicitly parallel.

Behaviors

- Range of ways of combining behaviors.
- Some examples:
  - Pick the “best”
  - Sum the outputs
  - Use a weighted sum
- Flakey redux used a fuzzy combination which produced a nice integration of outputs.

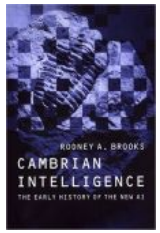
Complex sets of behaviors



(Boyoon Jung, USC)

Subsumption architecture

- A subsumption architecture is a hierarchy of task-accomplishing behaviours.
- Each behaviour is a rather simple rule-like structure.
- Each behaviour “competes” with others to exercise control over the agent.
- Lower layers represent more primitive kinds of behaviour, (such as avoiding obstacles), and have precedence over layers further up the hierarchy.

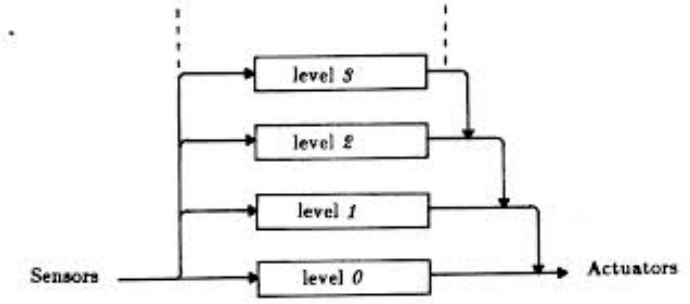


Subsumption architecture

- Originally implemented on Genghis.

Subsumption architecture

- It is the piling up of layers that gives the approach its power.



- Complex behavior builds from simple components.
  (R. Brooks, A Robust, Layered Control System for Robots, 1980)

Subsumption architecture

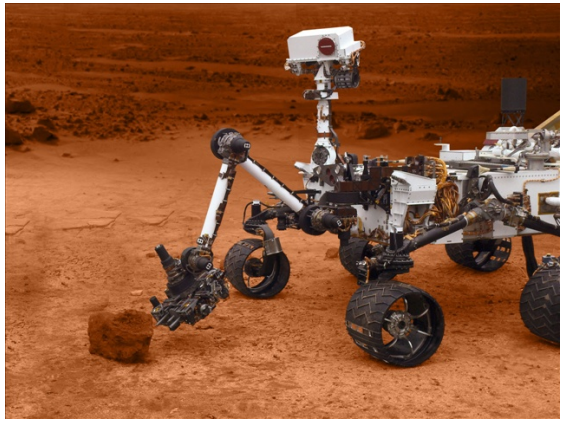
- Advantages in system building.
- Since each layer is independent, can:
  - Code
  - Test
  - Debug

  them independently.
- Then assemble into a complete system.

Subsumption architecture

- The resulting systems are, in terms of the amount of computation they do, extremely simple.
- However, some of the robots achieve quite impressive tasks.
- Steels' Mars explorer system, using the subsumption architecture, achieves near-optimal cooperative performance in simulated 'rock gathering on Mars' domain.

Mars explorer



The objective is to explore a distant planet, and in particular, to collect sample of a precious rock. The location of the samples is not known in advance, but it is known that they tend to be clustered.

Mars explorer

- For individual (non-cooperative) agents, the lowest-level behavior,
  - hence the behavior with the highest "priority"

  is obstacle avoidance:

  if detect an obstacle then change direction.
- Any samples carried by agents are dropped back at the mother-ship:

  if carrying samples and at the base then drop samples

  if carrying samples and not at the base then travel up gradient.
- The "gradient" in this case refers to a virtual "hill" that slopes up to the mother ship/base.

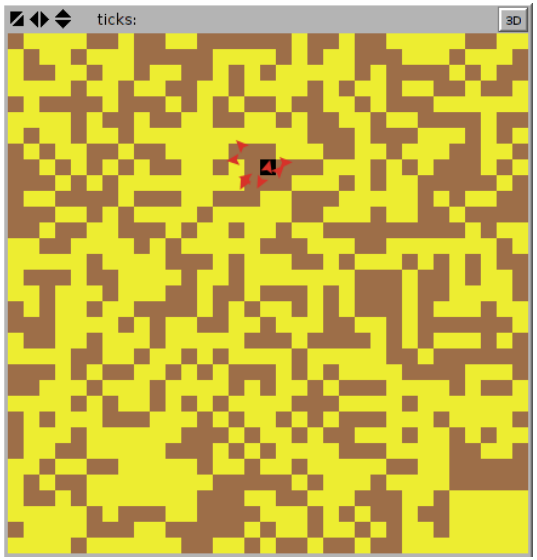
Mars explorer

- Agents will collect samples they find:
  if detect a sample then pick sample up.
- An agent with “nothing better to do” will explore randomly:
  if true then move randomly.

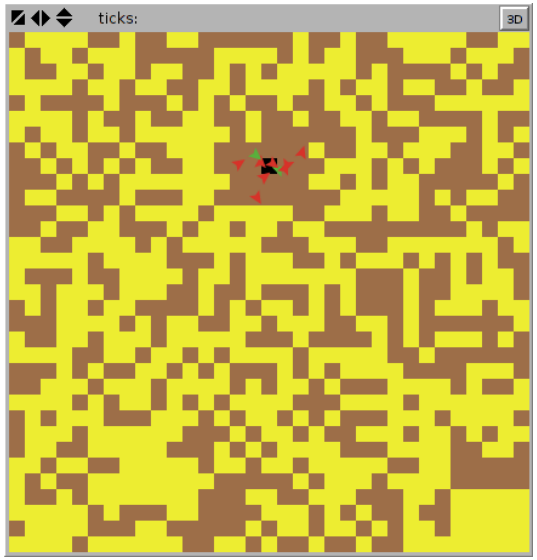
Mars explorer

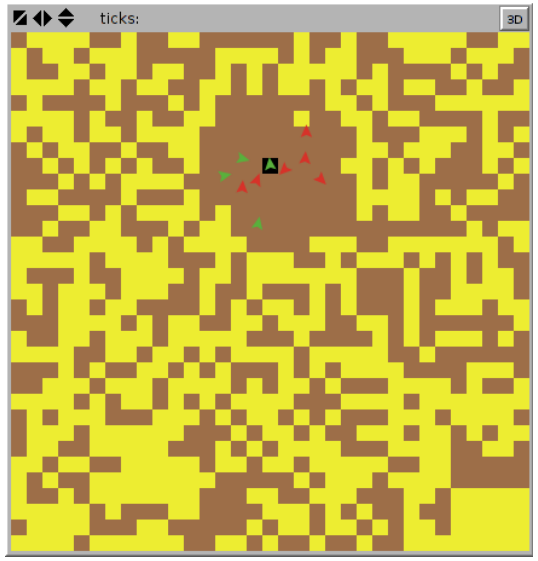
- Programming a number of robots in this way produces a cooperative swarm.

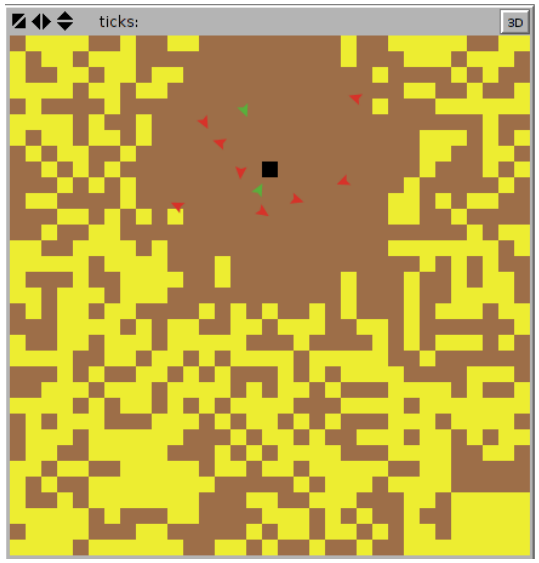
Mars explorer

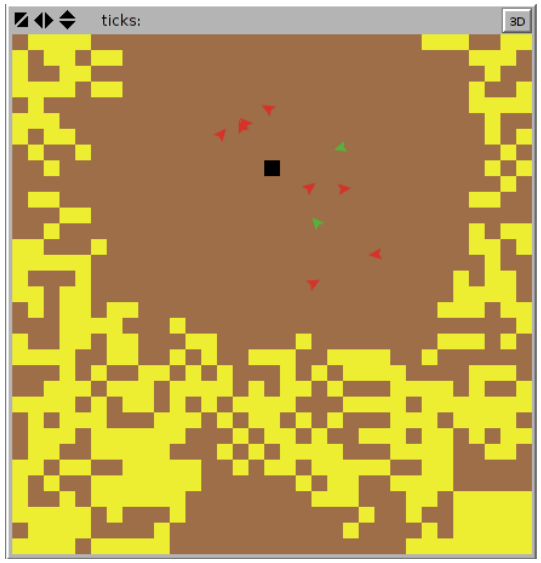


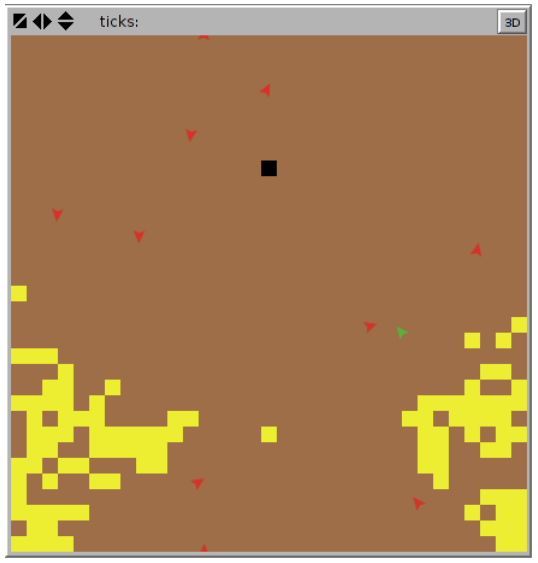
Mars explorer




ticks:
3D


ticks:
3D

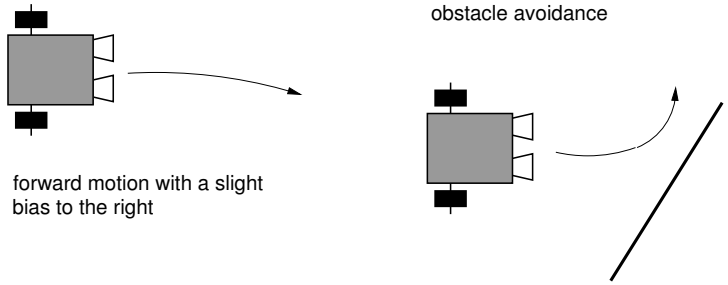

ticks:
3D


ticks:
3D

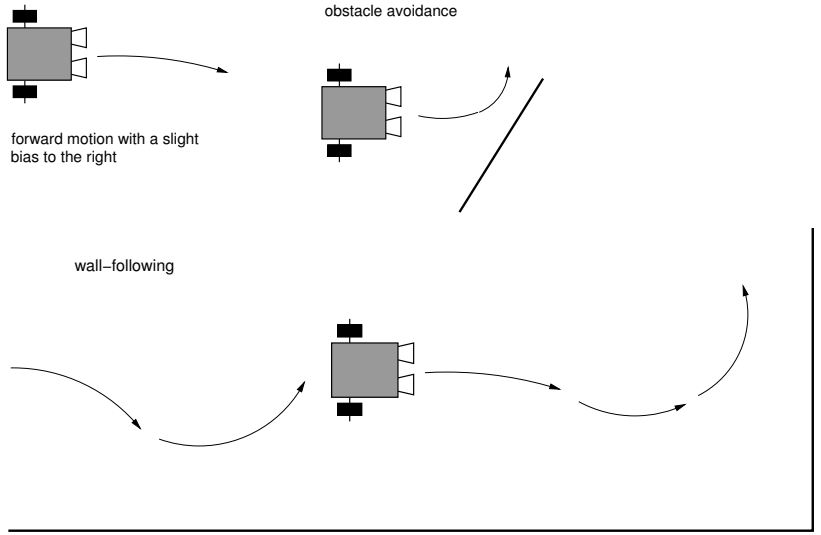
Not unlike systems in nature

Emergent behavior

- Putting simple behaviors together leads to synergies.

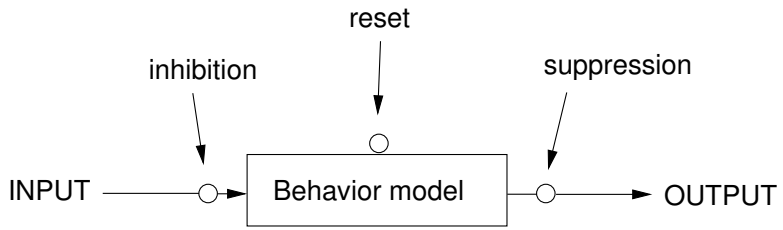


Emergent behavior



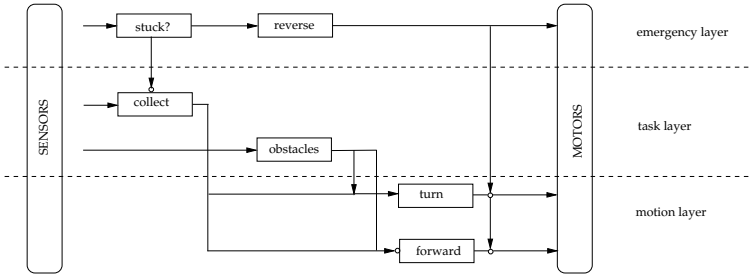
Abstract view of subsumption architecture

- Layered approach based on levels of competence
- Augmented finite state machine:



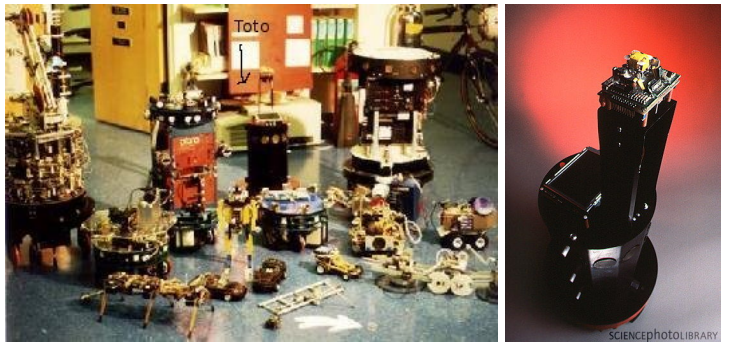
Abstract view of subsumption architecture

- A subsumption architecture machine:



Toto

- Can build sophisticated machines this way.
- Matarić's Toto was able to map spaces and execute plans all without a symbolic representation.



Toto

- Each feature/landmark is a set of sensor readings
  - Signature
- Recorded in a behavior as a triple:
  - Landmark type
  - Compass heading
  - Approximate length/size
- Distributed topological map.

Toto

- Whenever Toto visited a particular landmark, its associated map behavior would become activated
- If no behavior was activated, then the landmark was new, so a new behavior was created
- If an existing behavior was activated, it inhibited all other behaviors
- Localization was based on which behavior was active.
- No map object, but the set of behaviors clearly included map functionality.

Behaviors in LeJOS

- LeJOS has the `Behavior` class which provides support for implementing behaviour-based systems.
- Not quite a subsumption architecture, but clearly inspired by it.
- We'll look at a simple use of it to create a controller for the NXT.
- Another use of `StandardRobot`

Behaviors in LeJOS

- At any time, only one behavior can be active and in control of the robot
- Each behavior has a fixed priority
- Each behavior can determine whether it should take control
- The active behavior has higher priority then any other behavior that may take control

Behaviors in LeJOS

- The Behavior API consists of just one interface and one class:
  - `Behavior` interface—implemented by all behaviors
  - `Arbitrator` class—regulating priorities between behaviors
- This enables a very general and flexible approach to behaviors in LeJOS.
- Even though the implementation of each behavior vary, all behaviors are treated in the same way
- Both `Behavior` and `Arbitrator` are located in the `lejos.subsumption package`

`Behavior`

- Each behavior is implemented in its own class, which must implement the `Behavior` interface
- The `Behavior` interface requires a class to implement three methods:
  - `boolean takeControl()` —returns true if the behavior thinks it should take control.
  - `void action()` — the code executed when the behavior is in control.
  - `void suppress()` — called to terminate the code in the `action()` method.
- Unlike the full subsumption architecture there is no "inhibit".
- (I think `suppress` is closer in meaning to "inhibit" than "suppress" in the subsumption architecture.)

action()

- Typical example:

```
public void action(){
  suppressed = false;
  while(!suppressed){
  // do my thing
  }
}
```

- `suppressed` is a flag set by `suppress`
- Recommended design pattern: `action()` should quit quickly when `suppress` is called.
- Recommended design pattern: `action()` should leave the robot "clean". So, no motors running.

suppress()

- With `action()` as above, `suppress()` can be as simple as:

```
public void suppress(){
  suppressed = true;
}
```

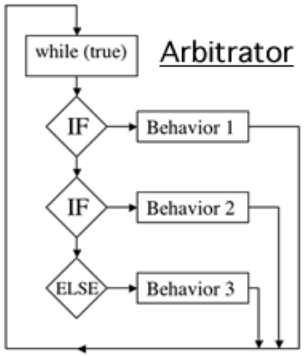
- Since this will immediately disable the loop in `action()`

Arbitrator

- The `Arbitrator` class allows us to select between competing behaviors
  - Behaviors that think they should be in control.
- `Arbitrator(Behavior[] behaviors, boolean returnWhenInactive)` with parameters:
  - Array of behaviors: lower index = lower priority
  - `returnWhenInactive`: if true the program exits when there is no behavior wanting to take control.
- Method: `public void start()`

Arbitrator

- List of behaviors is ordered.



- `Arbitrator` picks the first one that thinks it should be in control.
- (Despite this picture, which comes from the LeJOS website, I think that the higher indexed behaviors are considered before lower indexed ones).

Forward and avoid

- Forward: moves the robot forward
- Avoid: reacts to an obstacle
- The `Arbitrator` will let the Avoid method take over when an obstacle is detected
- Uses the `StandardRobot` and `RobotMonitor` classes we introduced in the previous lecture.
- All the code is on the course website.

Forward

```
import lejos.robotics.subsumption.*;

public class ForwardBehavior implements Behavior{
  public boolean suppressed;
  private StandardRobot robot;

  public ForwardBehavior(StandardRobot r){
    robot = r;
  }

  public boolean takeControl(){
    return true;
  }

  public void suppress(){
    suppressed = true;
  }
```


Forward

```
  public void action(){
    suppressed = false;
    robot.startMotors();
    while(!suppressed){
      Thread.yield();
    }
    robot.stopMotors();
    }
}
```

- `yield()` is being used here to allow other threads to run — effectively a `sleep()` that doesn't have a fixed duration.
- Not clear to me this is the best way to solve the problem here.

Avoid

```
import lejos.robotics.subsumption.*;

public class AvoidBehavior implements Behavior{
  public boolean suppressed;
  private StandardRobot robot;

  public AvoidBehavior(StandardRobot r){
    robot = r;
  }

  public boolean takeControl(){
    return (robot.isLeftBumpPressed()
      || robot.isRightBumpPressed());
  }

  public void suppress(){
    suppressed = true;
  }
```

Avoid

- The main bit of the behavior:

```
public void action(){
  robot.reverseMotors();
  try{
    Thread.sleep(2000);
  }
  catch(Exception e){
  }
  robot.turnMotors(true);
  try{
    Thread.sleep(2000);
  }
  catch(Exception e){
  }
  robot.stopMotors();
}
```

- This code doesn't allow the behavior to be suppressed.

How we combine them

```
import lejos.robotics.subsumption.*;

public class ForwardAvoid {
  public static void main(String[] args)
      throws Exception{
    StandardRobot me = new StandardRobot();
    RobotMonitor rm = new RobotMonitor(me, 300);
    rm.start();

    Behavior b1 = new ForwardBehavior(me);
    Behavior b2 = new AvoidBehavior(me);
    Behavior[] bArray = {b1, b2};
    Arbitrator arb = new Arbitrator(bArray, true);
    arb.start();
  }
}
```


Summary

- This lecture looked at behavior-based robotics
- We looked at:
  - Basic ideas,
  - Some history and examples; and
  - LeJOS support for programming this way.