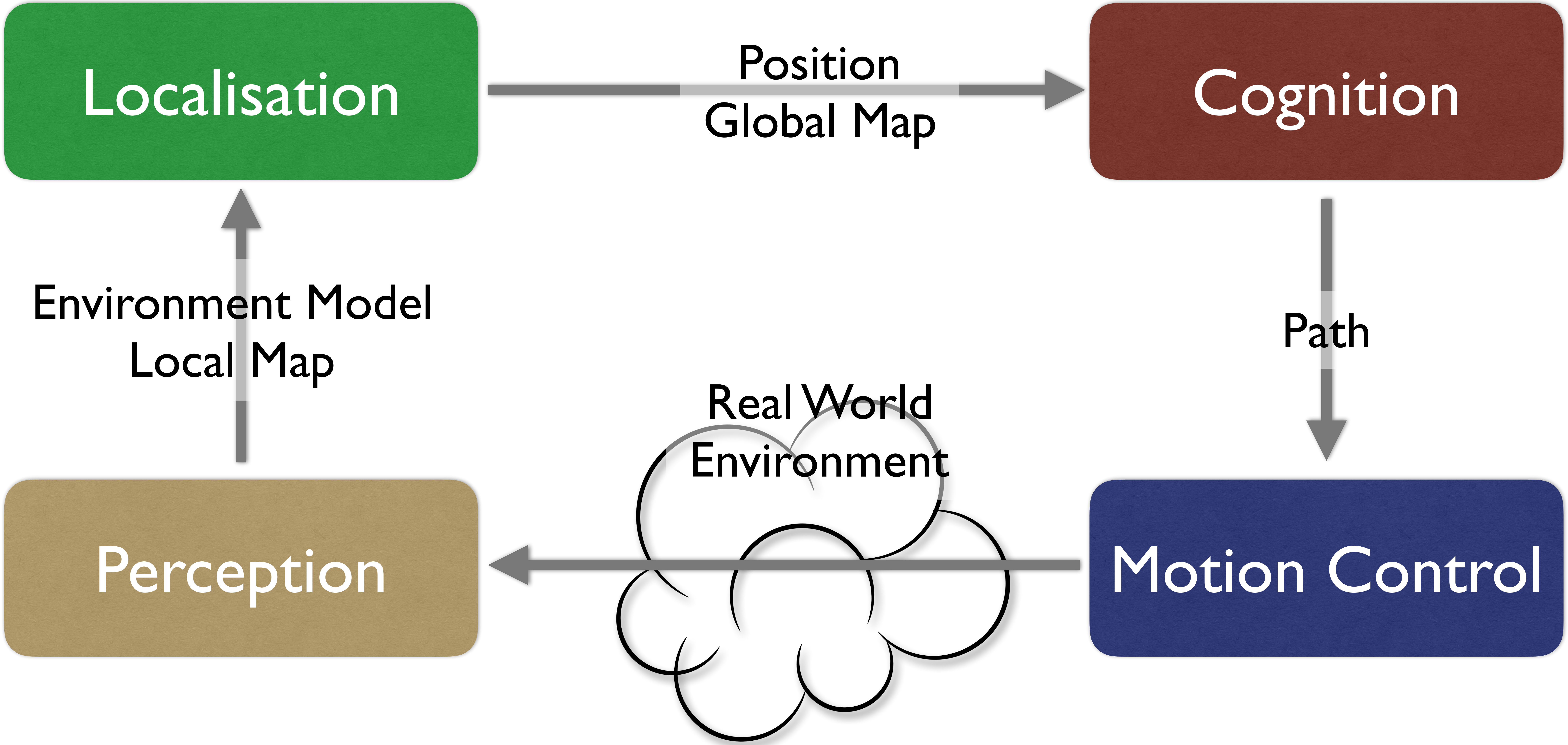
COMP329 Robotics and Autonomous Systems

Lecture 6: Behaviour Based Robots

Dr Terry R. Payne
Department of Computer Science

UNIVERSITY OF LIVERPOOL

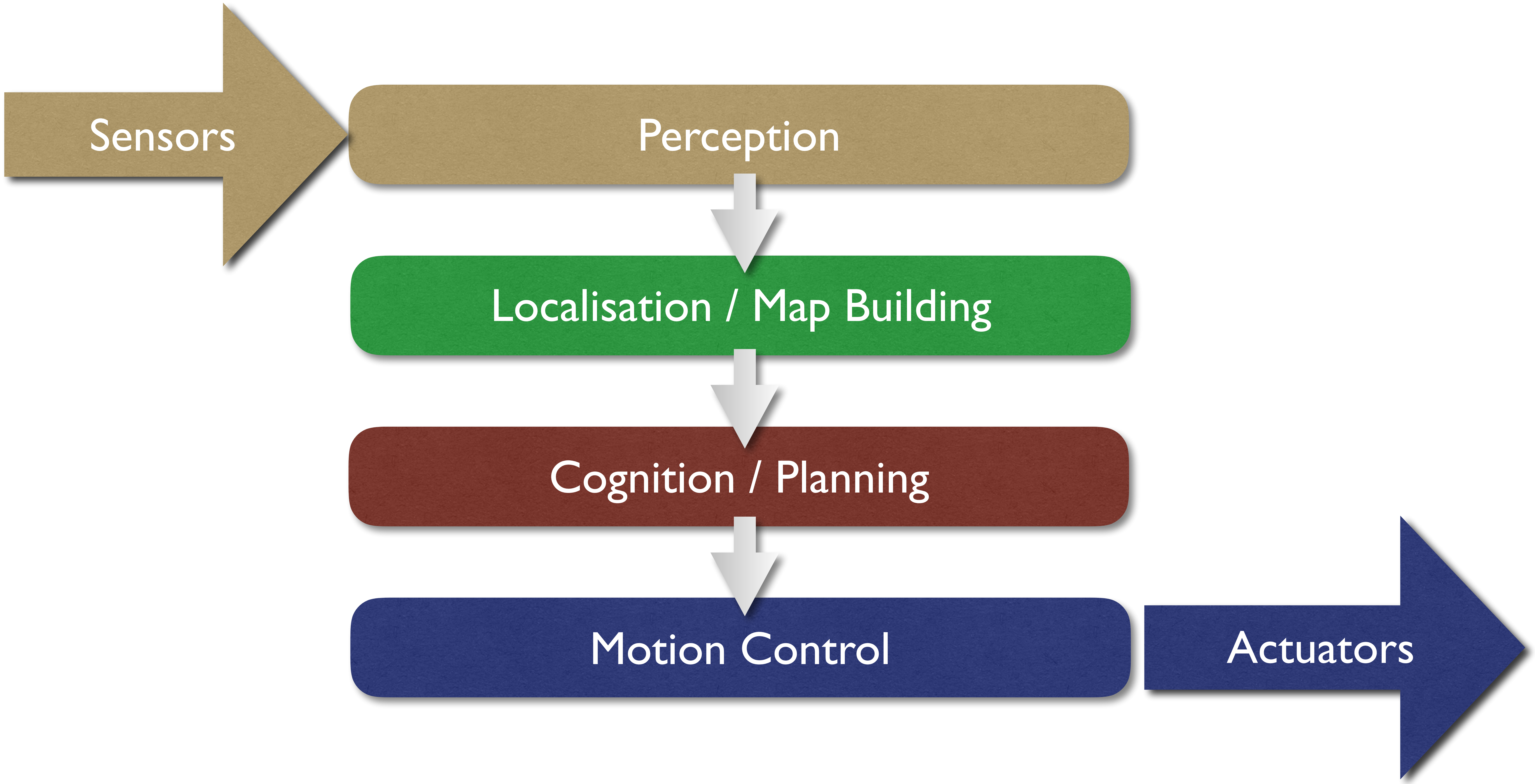
General control architecture



Our control loop diagram implies an ordering over the operations

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Behaviours



Sensors

Perception

Localisation / Map Building

Cognition / Planning

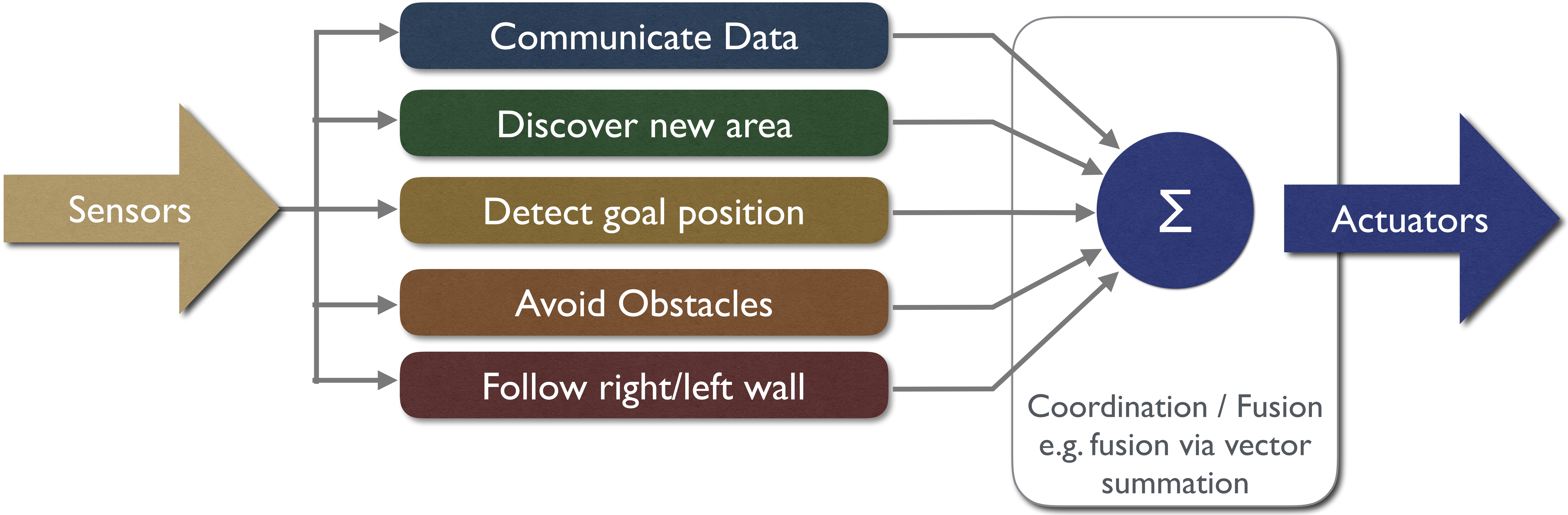
Motion Control

Actuators

The classic “Sense/Plan/Act” approach breaks it down serially like this

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Behaviours

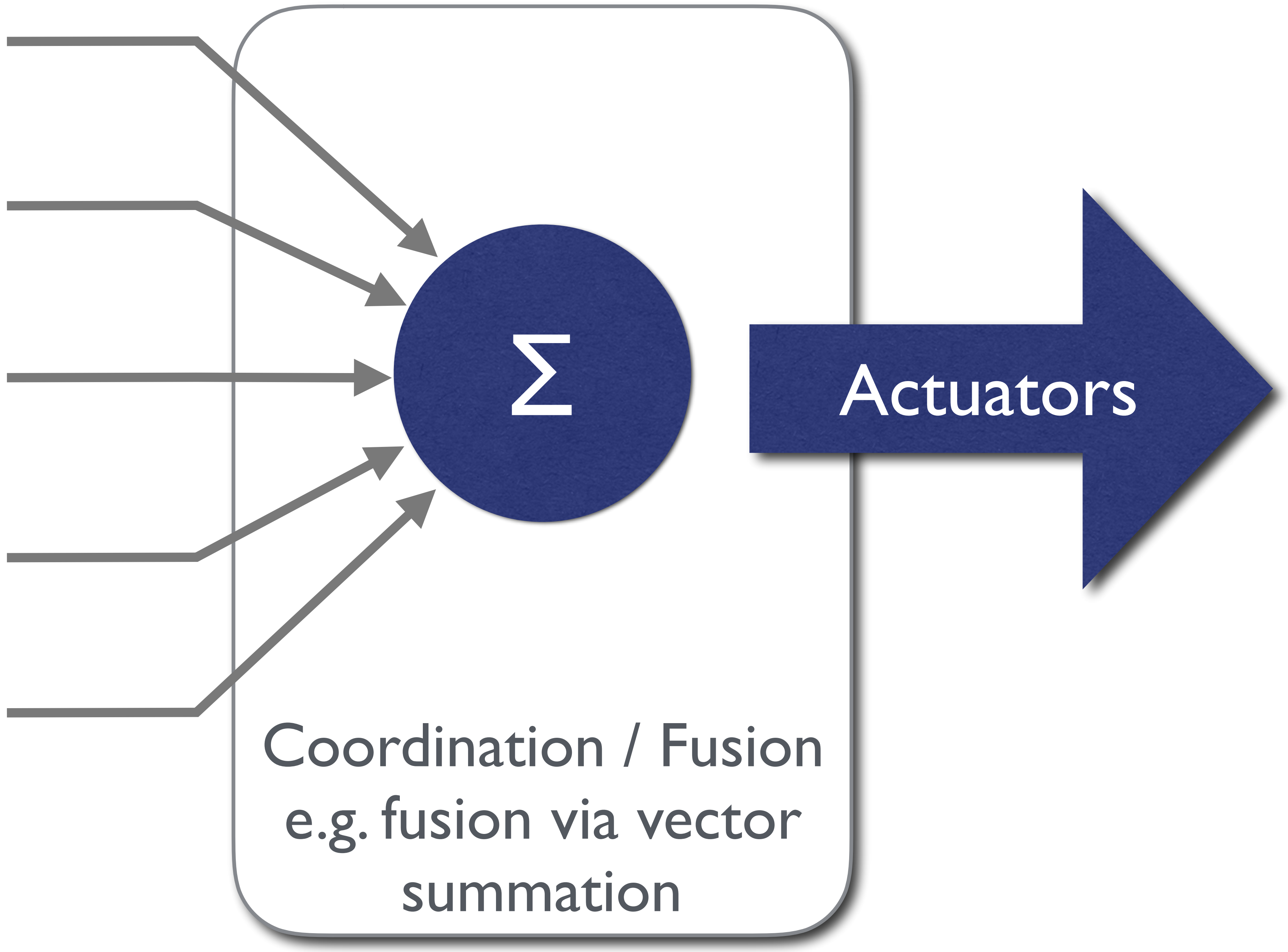


- Behaviour based control sees things differently
  - Behavioural chunks of control each connecting sensors to motors
  - Implicitly parallel

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Behaviours

- 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.



Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

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.
- The resulting systems are, in terms of the amount of computation they do, ***extremely*** simple.
  - Some of the robots do tasks that would be impressive if they were accomplished by symbolic AI systems.

Rodney Brooks "subsumption architecture" was originally developed open Genghis

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Brooks Behavioural Languages

- Brooks proposed the following three theses:
    1. Intelligent behaviour can be generated ***without*** explicit ***representations*** of the kind that symbolic AI proposes.
    2. Intelligent behaviour can be generated ***without*** explicit ***abstract reasoning*** of the kind that symbolic AI proposes.
    3. Intelligence is an ***emergent*** property of certain complex systems.

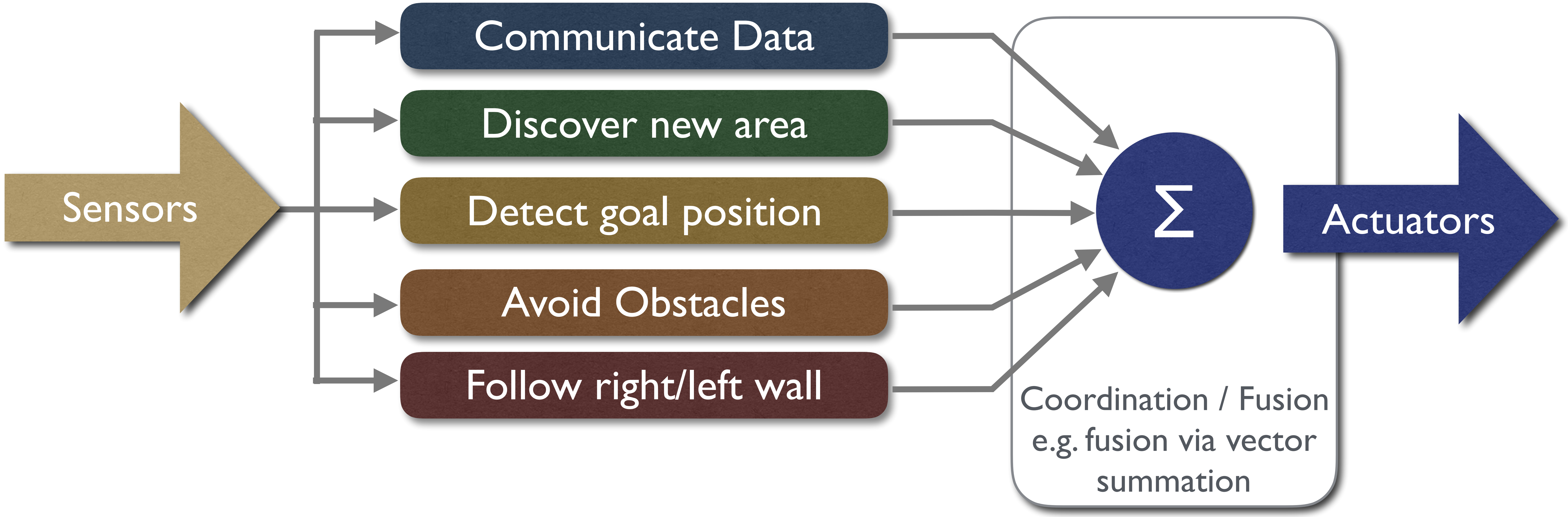
Brooks Behavioural Languages

- He identified two key ideas that have informed his research:
    1. ***Situatedness and embodiment***: ‘Real’ intelligence is situated in the world, not in disembodied systems such as theorem provers or expert systems.
    2. ***Intelligence and emergence***: ‘Intelligent’ behaviour arises as a result of an agent’s interaction with its environment. Also, intelligence is ‘in the eye of the beholder’; it is not an innate, isolated property.
- Brooks built several agents (such as Genghis) based on his ***subsumption architecture*** to illustrate his ideas.

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Subsumption Architecture

- It is the piling up of layers that gives the approach of its power.
  - Complex behaviour emerges from simple components.
  - Since each layer is independent, each can independently be:
    - Coded / Tested / Debugged
  - Can then assemble them into a complete system.



Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Real World Example: Stanley

- Won the 2005 DARPA Grand Challenge
  - Used a combination of the subsumption architecture with deliberative planning
  - Consists of 30 different independently operating modules across 6 layers

Global Services Layer

User Interface Layer

Vehicle Interface Layer

Planning and Control layer

Perception layer

Sensor interface layer

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

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.

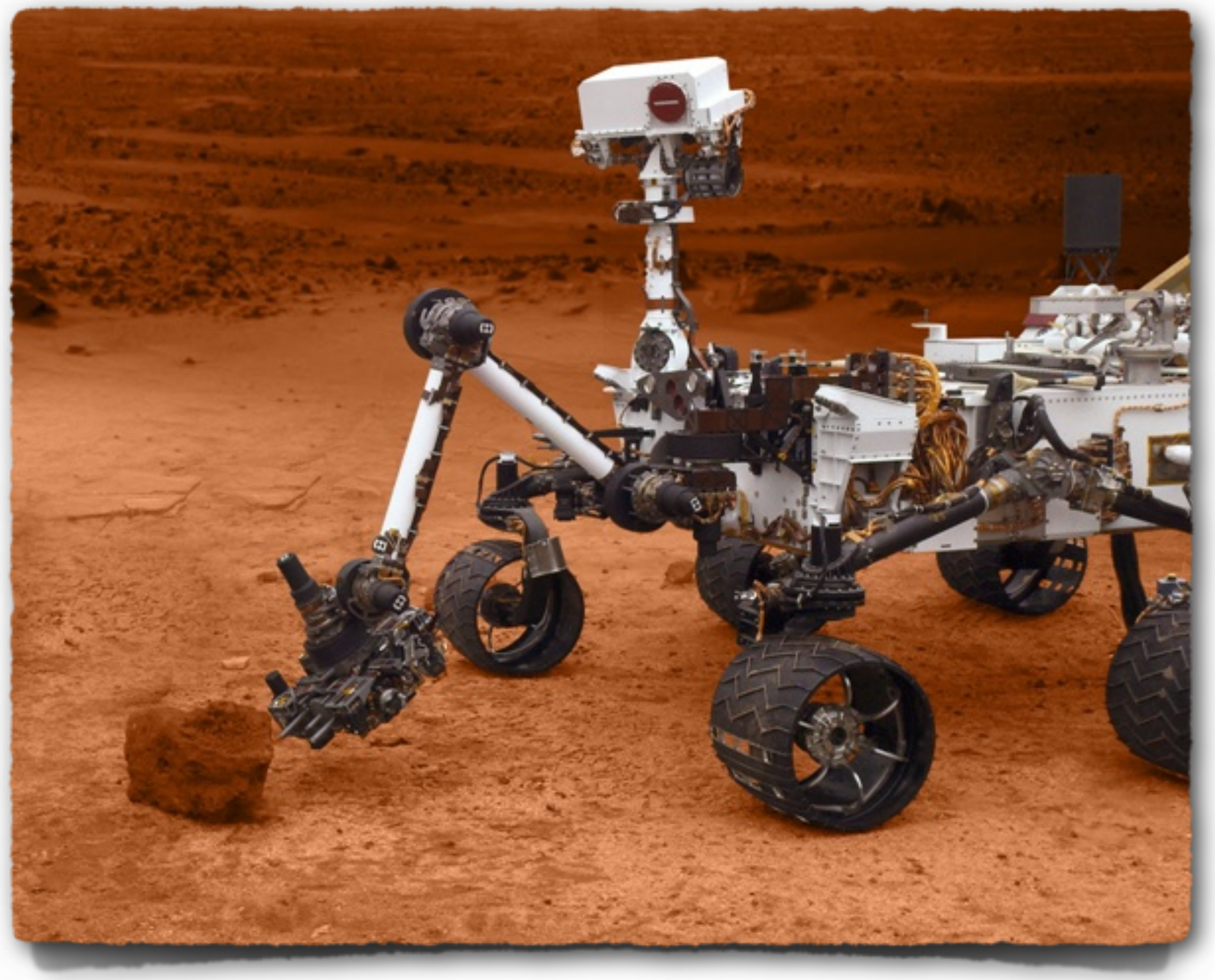
Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Steel’s Mars Explorer System

- Steels’ Mars explorer system, using the ***subsumption*** architecture, achieves near-optimal cooperative performance in simulated ‘rock gathering on Mars’ domain
  - ***Individual behaviour*** is governed by a set of simple rules.
  - ***Coordination between agents*** can also be achieved by leaving “markers” in the environment.

Objective

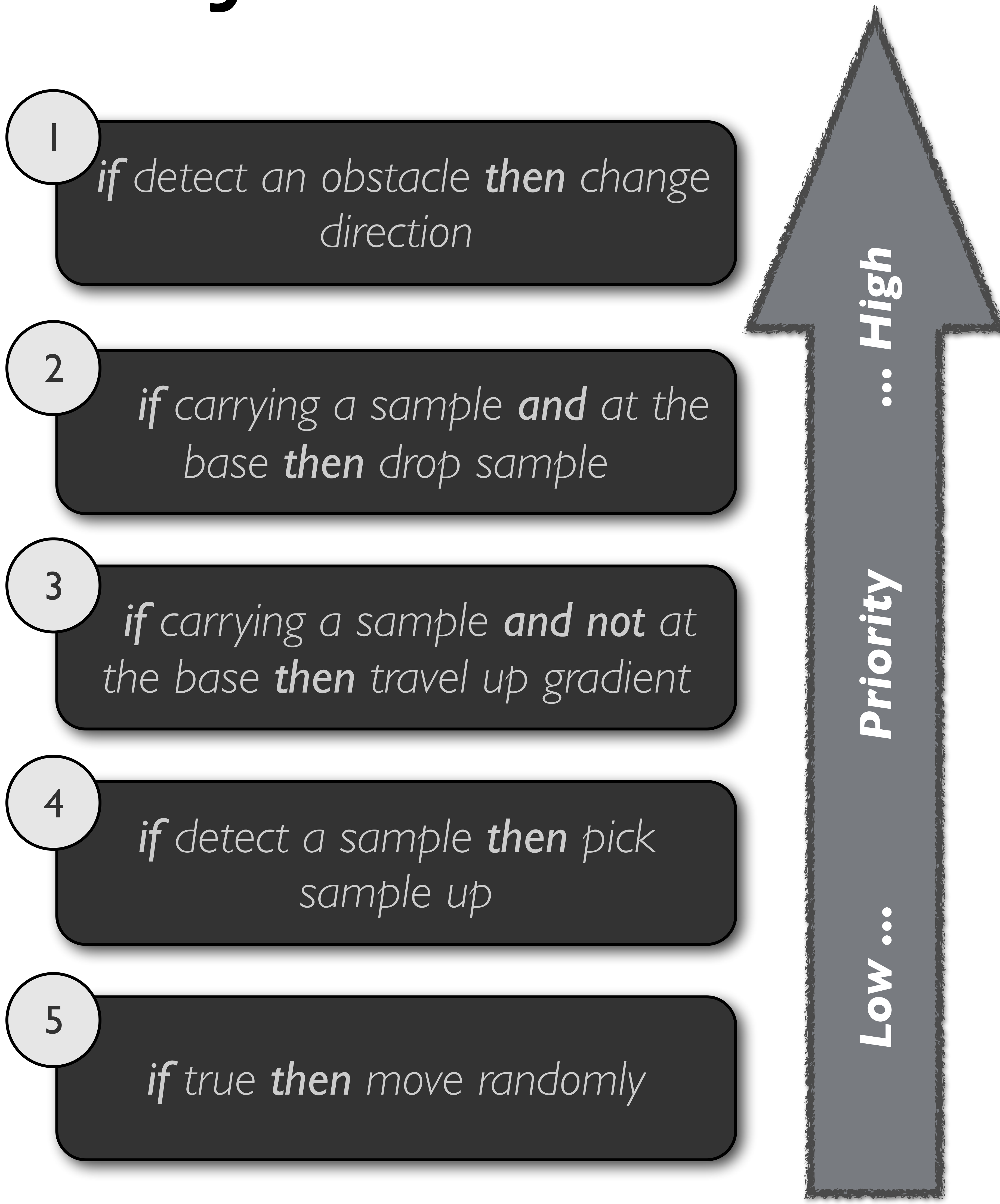
To explore a distant planet, and in particular, to collect sample of a precious rock. The location of the samples is not known in advance, but it is known that they tend to be clustered.



Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Steel’s Mars Explorer System

1. For individual (non-cooperative) agents, the lowest-level behavior, (and hence the behavior with the highest “priority”) is obstacle avoidance.
2. Any samples carried by agents are dropped back at the mother-ship.
3. If not at the mother-ship, then navigate back there.
    - The “gradient” in this case refers to a virtual “hill” radio signal that slopes up to the mother ship/base.
4. Agents will collect samples they find.
5. An agent with “nothing better to do” will explore randomly. This is the highest-level behaviour (and hence lowest level “priority”).



1 *if* detect an obstacle **then** change direction

2 *if* carrying a sample **and** at the base **then** drop sample

3 *if* carrying a sample **and not** at the base **then** travel up gradient

4 *if* detect a sample **then** pick sample up

5 *if* true **then** move randomly

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Steel’s Mars Explorer System

- Existing strategy works well when samples are distributed randomly across the terrain.
  - However, samples are located in clusters
  - Agents should ***cooperate*** with each other to locate clusters
- Solution to this is based on foraging ants.
  - Agents leave a “radioactive” trail of crumbs when returning to the mother ship with samples.
    - If another agent senses this trail, it follows the trail back to the source of the samples
    - It also picks up some of the crumbs, making the trail fainter.
    - If there are still samples, the trail is reinforced by the agent returning to the mother ship (leaving more crumbs)
    - If no samples remain, the trail will soon be erased.

⋮

3’ *if carrying samples **and not** at the base **then** drop 2 crumbs **and** travel up gradient.*

⋮

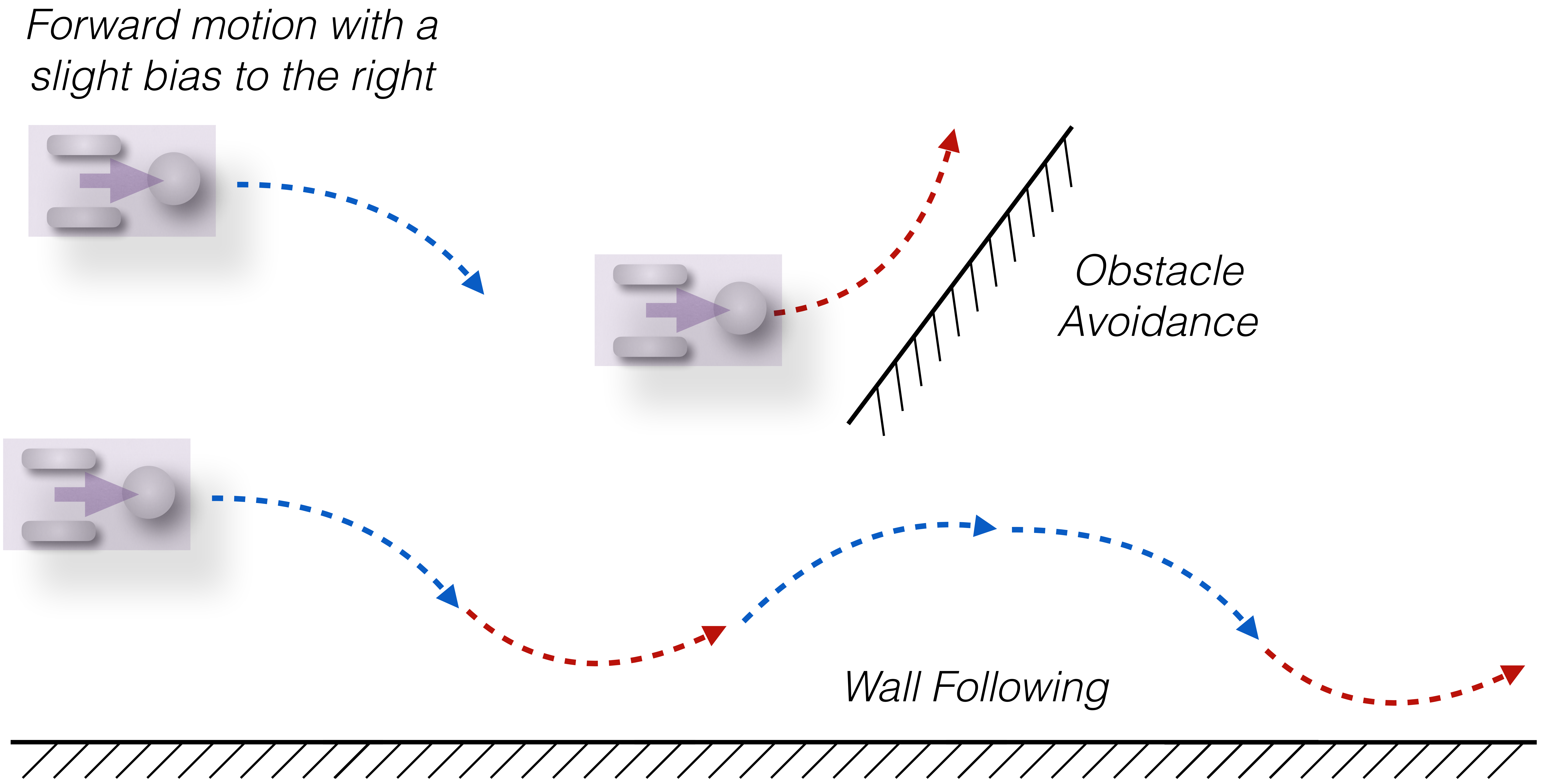
4.5 *if sense crumbs **then** pick up 1 crumb **and** travel down gradient*

⋮

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Emergent Behaviour

- Putting simple behaviours together leads to synergies



Forward motion with a slight bias to the right

Obstacle Avoidance

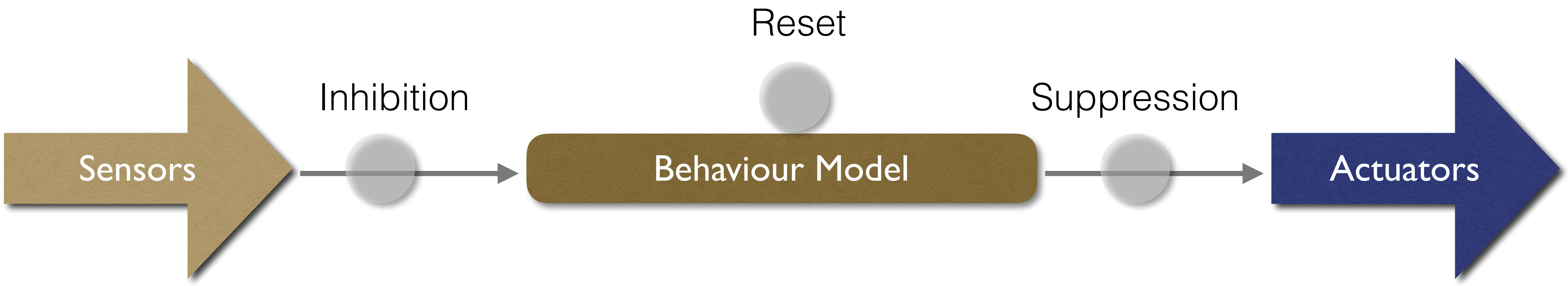
Wall Following

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Abstract view of a Subsumption Machine

- Layered approach based on levels of competence
  - Higher level behaviours ***inhibit*** lower levels

- Augmented finite state machine:



Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Toto

- Maja Mataric´’s Toto is based on the subsumption architecture
  - Can map ***spaces*** and ***execute plans*** without the need for a ***symbolic*** representation.
  - Inspired by “…*the ability of insects such as bees to identify shortcuts between feeding sites*…”

- Each feature/landmark is a set of sensor readings
  - Signature

- Recorded in a behaviour as a triple:
  - Landmark type
  - Compass heading
  - Approximate length/size

- Distributed topological map.

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

ToTo

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

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Behaviours in LeJOS

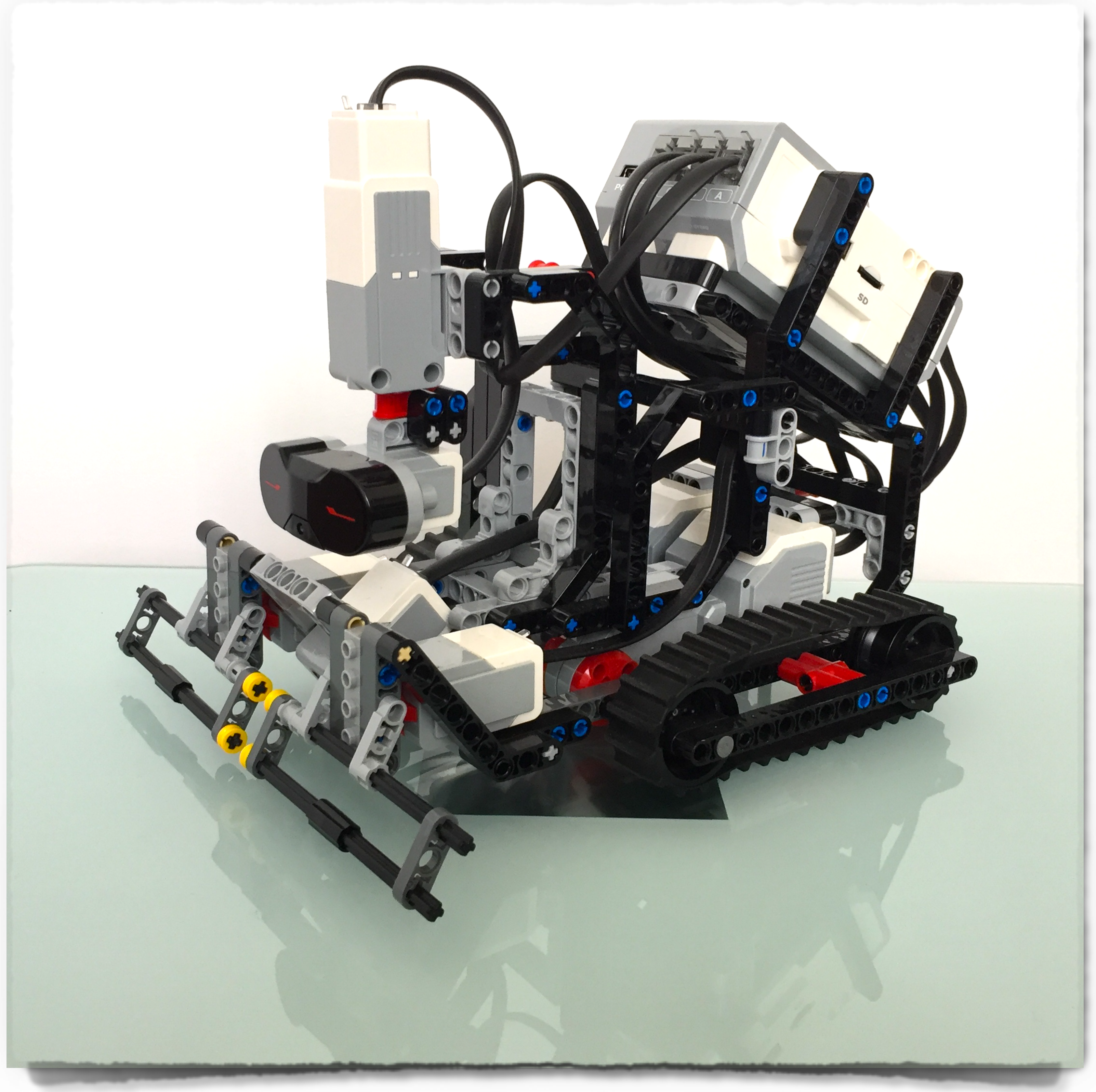
- LeJOS has the `Behavior` class which provides support for implementing behaviour-based systems.
  - Not quite a subsumption architecture, but clearly inspired by it.

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

- We'll look at a simple use of it to create a controller for the EV3.
  - We’ll build on the notion of a “standard robot”, which will be illustrated later

Behaviours in LeJOS

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



Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Behaviour

- Each behaviour 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 behaviour thinks it should take control.
  - `void action()`
    - the code executed when the behaviour is in control.
  - `void suppress()`
    - called to terminate the code in the `action()` method.
- Unlike the full subsumption architecture there is no ``inhibit''.

Note the spelling of the Behavior class, which follows the US spelling!!!

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Writing the `action()` method

- Typical example of an action
  - `suppressed` is a flag set by the method `suppress()`

- Recommended design patterns
  - `action()` should quit quickly when `suppress()` is called.
  - `action()` should leave the robot "clean" (i.e. no motors running)

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

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Writing the **suppress()** method

- With **action()** method as given above, the method **suppress()** could be as simple as:
  - Since this will immediately disable the loop in **action()**

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

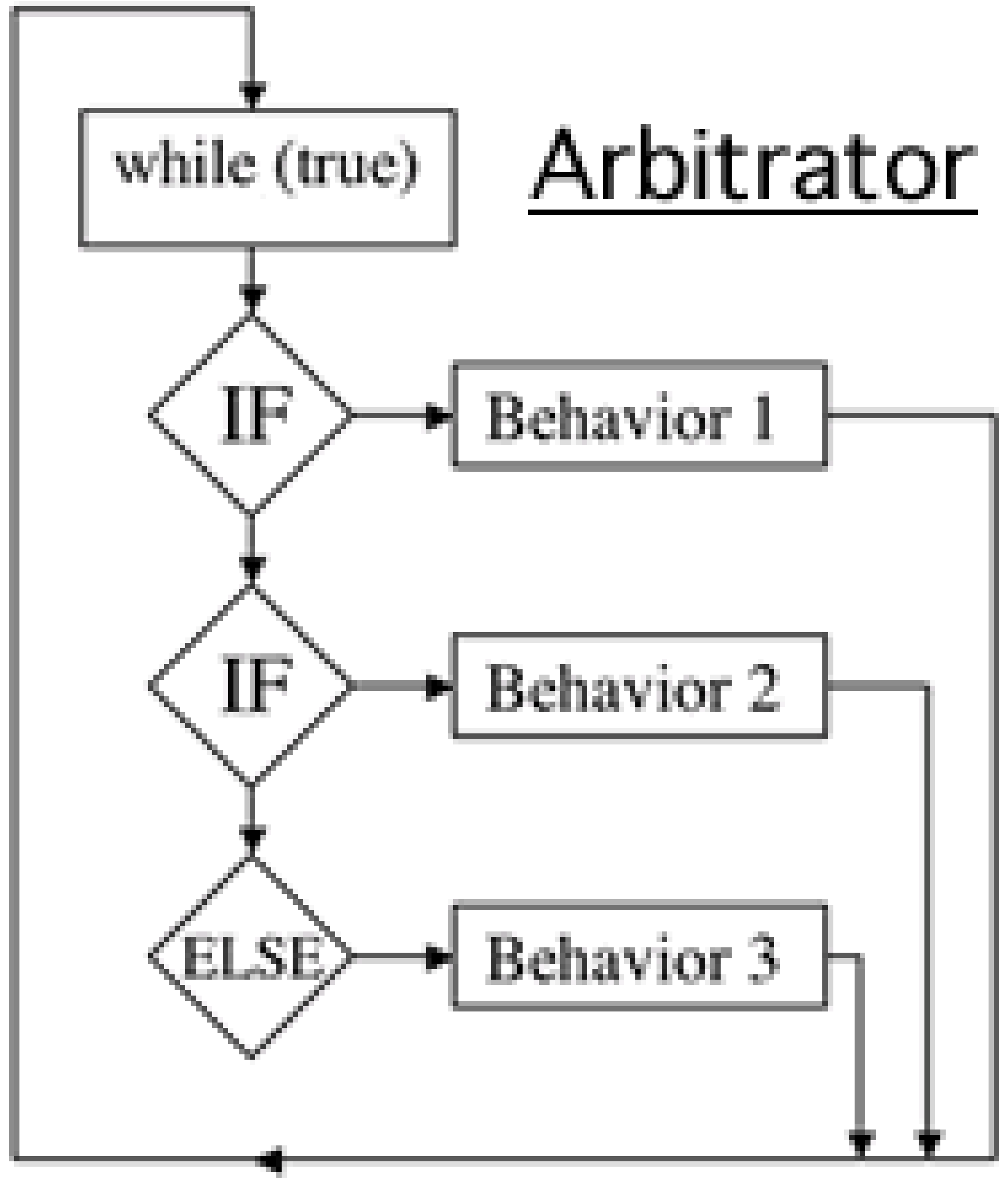
Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Arbitrator

- The `Arbitrator` class allows us to select between competing behaviours
  - 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 behaviour wanting to take control.
  - Method: `public void start()`

- `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 behaviours are considered before lower indexed ones…

List of behaviours is ordered



Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Code Example: Forward/Avoid

- The code examples on the website include…
  - ***ForwardBehaviour.java***: moves the robot forward
  - AvoidBehaviour.java: reacts to an obstacle
- Uses the StandardRobot and RobotMonitor classes we will visit when looking at Threading.
  - `yield()` is being used here to allow other threads to run --- effectively a `sleep()` that doesn't have a fixed duration.
    - Not clear if this is the best way to solve the problem here….

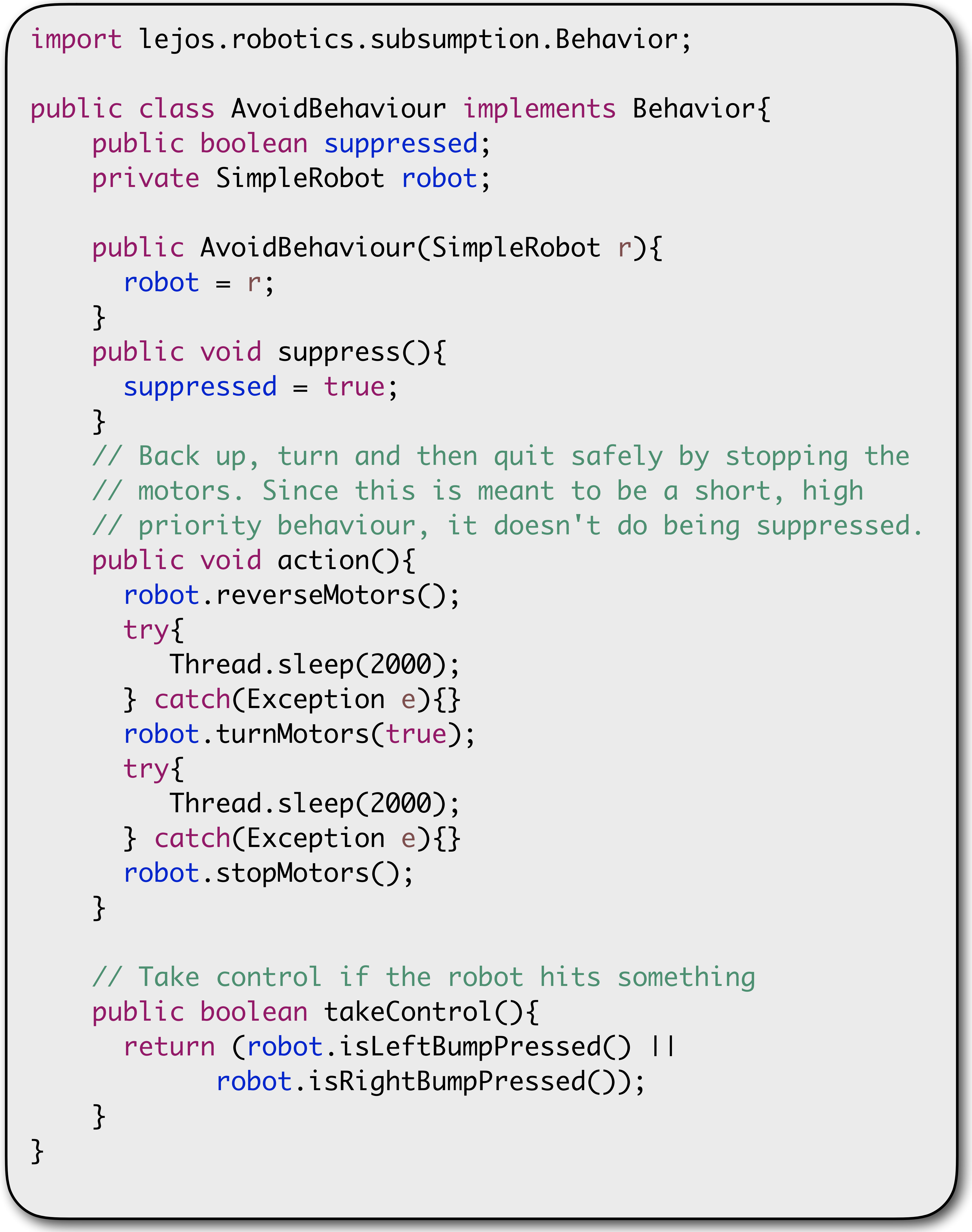
```java
import lejos.robotics.subsumption.Behavior;

public class ForwardBehaviour implements Behavior{
    public boolean suppressed;
    private SimpleRobot robot;

    public ForwardBehaviour(SimpleRobot r){
      robot = r;
    }

    public void suppress(){
      suppressed = true;
    }

    // Start driving and then yield (for a non-busy wait).
    // If suppressed, then stop the motors and quit.
    public void action(){
        suppressed = false;
        robot.startMotors();

        while(!suppressed){
           Thread.yield();
        }
        robot.stopMotors();
    }

    // Take control if the robot hits something
    public boolean takeControl(){
      return true;
    }
}
```

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Code Example: Forward/Avoid

- The code examples on the website include…
  - ForwardBehaviour.java: moves the robot forward
  - ***AvoidBehaviour.java***: reacts to an obstacle

- The `Arbitrator` will let the `AvoidBehaviour` method take over when an obstacle is detected
  - Note that this code doesn't allow the behaviour to be suppressed



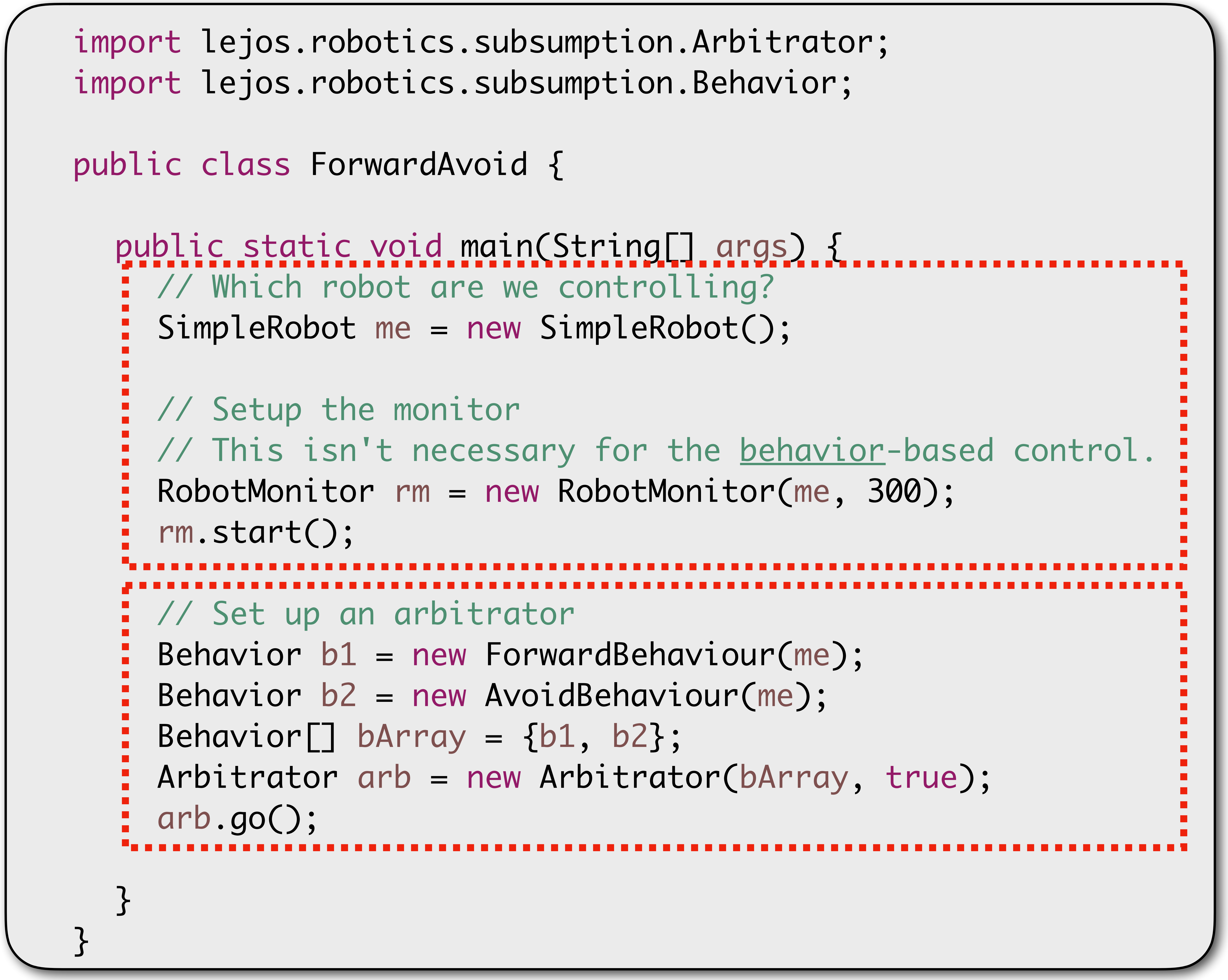
```java
import lejos.robotics.subsumption.Behavior;

public class AvoidBehaviour implements Behavior{
    public boolean suppressed;
    private SimpleRobot robot;

    public AvoidBehaviour(SimpleRobot r){
      robot = r;
    }
    public void suppress(){
      suppressed = true;
    }
    // Back up, turn and then quit safely by stopping the
    // motors. Since this is meant to be a short, high
    // priority behaviour, it doesn't do being suppressed.
    public void action(){
      robot.reverseMotors();
      try{
         Thread.sleep(2000);
      } catch(Exception e){}
      robot.turnMotors(true);
      try{
         Thread.sleep(2000);
      } catch(Exception e){}
      robot.stopMotors();
    }

    // Take control if the robot hits something
    public boolean takeControl(){
      return (robot.isLeftBumpPressed() ||
            robot.isRightBumpPressed());
    }
}
```

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

How we combine them…

The **SimpleRobot** class provides a basic robot with abstract controls. The **RobotMonitor** class is a headed class that gives status information of the various sensors on the robot.

The **Arbitrator** is created by passing an ordered array of behaviours, with the lowest priority ones (in this case, **ForwardBehaviour**) being earlier in the array. The **Arbitrator** is initialised by calling the method **go()**



```
import lejos.robotics.subsumption.Arbitrator;
import lejos.robotics.subsumption.Behavior;

public class ForwardAvoid {

  public static void main(String[] args) {
    // Which robot are we controlling?
    SimpleRobot me = new SimpleRobot();

    // Setup the monitor
    // This isn't necessary for the behavior-based control.
    RobotMonitor rm = new RobotMonitor(me, 300);
    rm.start();

    // Set up an arbitrator
    Behavior b1 = new ForwardBehaviour(me);
    Behavior b2 = new AvoidBehaviour(me);
    Behavior[] bArray = {b1, b2};
    Arbitrator arb = new Arbitrator(bArray, true);
    arb.go();

  }
}
```

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17

Summary

- This lecture looked at behaviour based robots.
  - We looked at the basic principle of the subsumption architecture and emergent behaviour
  - We looked that LeJOS support for programming this way.
- The next lecture will look at Maps, mapping, and models

Original Source: M. Wooldridge, S.Parsons, D.Grossi - updated by Terry Payne, Autumn 2016/17