CARMEN

and
Some Software Development Practices for Robotics

Today's Objectives

- Introduction to Carmen
- Introduction to programming in Carmen
- Software development and how you should program in Carmen
- Design decisions of Carmen and why things are they way they are

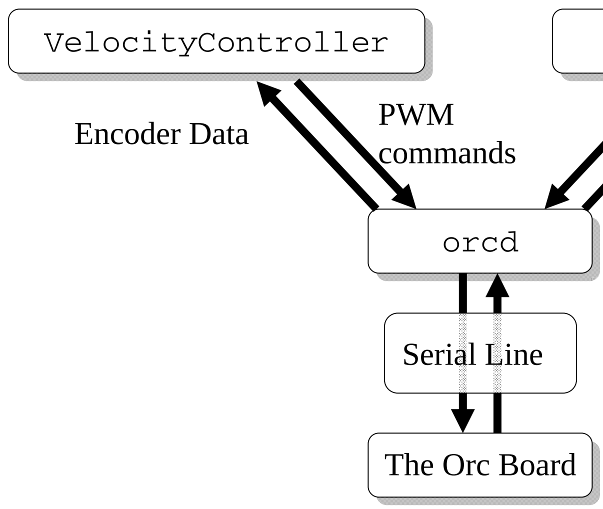
What is CARMEN?

- Open-source, modular toolkit for controlling mobile robots and sensors
- Originally primarily laser-based and map-based
- Provides end-to-end navigation capability using a core set of replaceable modules
  - Base control
  - Sensor control
  - Collision avoidance
  - Simulation
  - Localization
  - Navigation
  - Map building
  - Multi-robot support

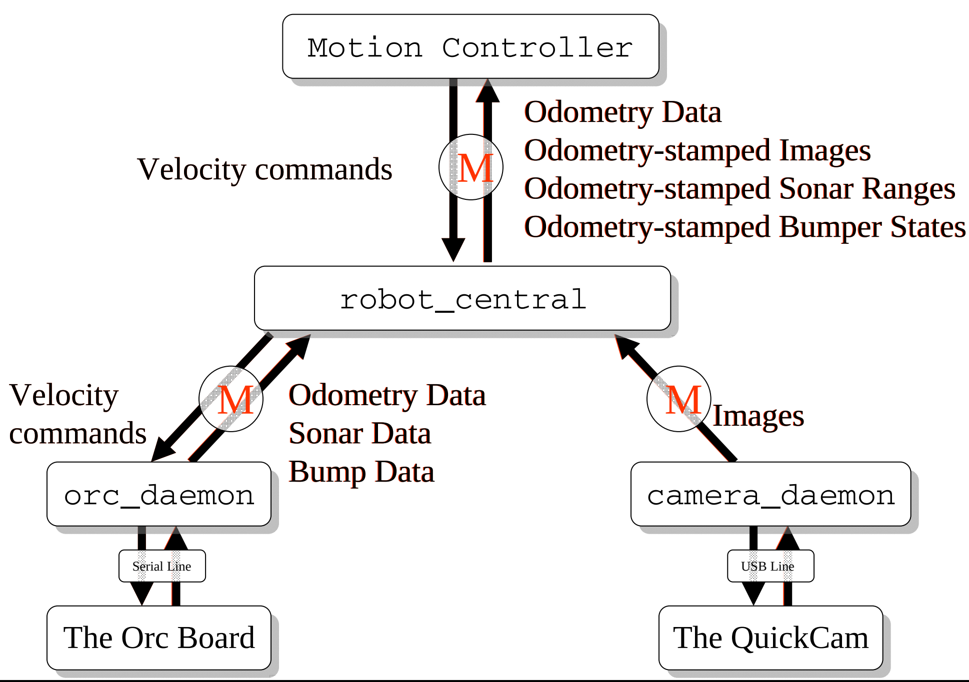
New set of Modules

- orc_daemon
  - Replaces orcd
  - Provides abstract interface to motors: no longer have to think in terms of PWM or left/right wheel velocities, or think about encoder integration
- camera_daemon
  - Provides abstract interface to camera
- robot_central
  - Tags sensor data (camera, sonar, etc) with odometry positions based on timestamps
- param_daemon
  - Provides each module with configuration data to be read at startup and during execution
- message_daemon
  - Communication managed by IPC package

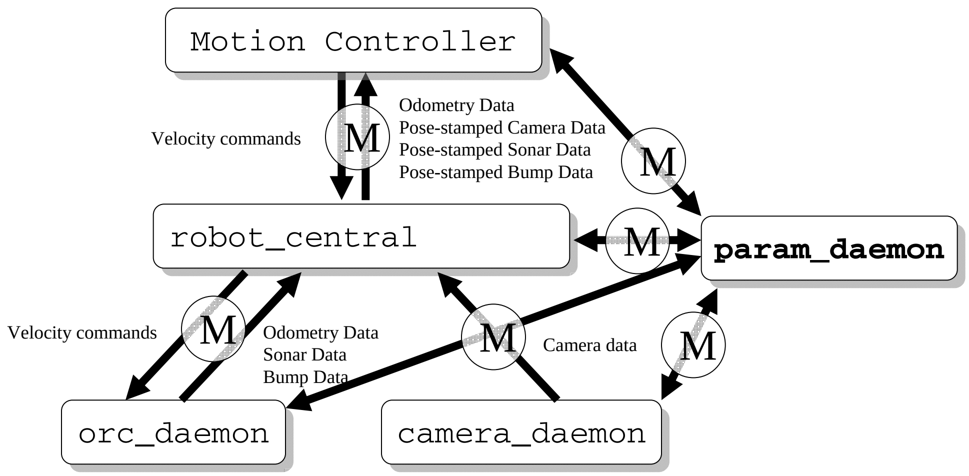
How Information Used to Flow (Labs 1-4)



The Basic Flow of Information



The Basic Flow of Information



Sequential Programming

- How (some of) you are used to thinking about programs:

```
x = getMyXPositionFromTheEncoderCounts();
y = getMyYPositionFromTheEncoderCounts();
goForward(1);
turnLeft(Math.PI/2);
Image image = camera.getImage();
double distance = computeServoDistance(image);
goForward(distance);
....
```

What happens if an obstacle appears while you are going forward?

What happens to the encoder data while you are turning?

What if someone else wants the data too?

Callbacks

- All execution occurs when an event happens
  - e.g., an image is read from the camera, the orc board reports odometry data
- Events are delivered in message form (typically consisting of sensor data)
- Program control flow:
  1) Connect to message_daemon
  2) Declare callbacks for different messages
  3) Dispatch
     - As each message arrives, the callback is called, the messages is processed, commands are issued, and the callback ends, returning control to the message processor

Carmen Programs

- Most Carmen programs will have the following structure:

```
import Carmen.*;
public class MyController implements DataHandler
{
    public void handleData ( DataMessage  msg)
    {
         processData();
         issueCommands();
    }
    public static void main(String args[])
    {
         MyController controller = new MyController();
         Robot.initialize();
         Robot.subscribeData(controller);
         Robot.dispatch();
    }
}
```

Carmen Programs

- For example:

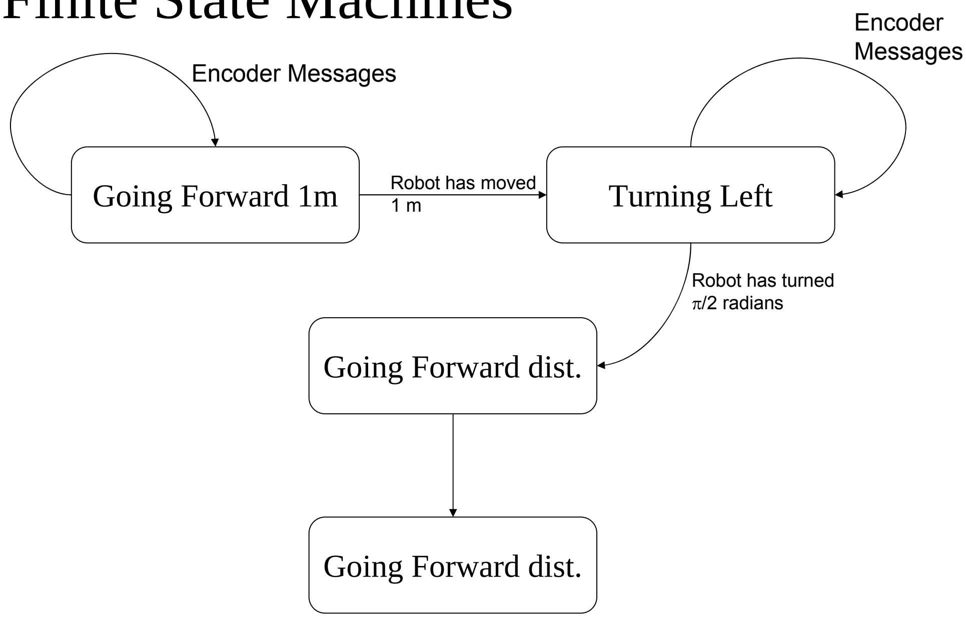
```
import Carmen.*;
public class MyController implements CameraHandler
{
    public void handleCamera(CameraMessage msg)
    {
        processImage();
        visualServo();
    }
    public static void main(String args[])
  {
        MyController controller = new MyController();
        Robot.initialize();
        Robot.subscribeCameraData(controller);
        Robot.dispatch();
}
```

The Real Big Change…

- How (some of) you are used to thinking about programs:

```
                x = getMyXPositionFromTheEncoderCounts();
                y = getMyYPositionFromTheEncoderCounts();
State 1 ---->   goForward(1);
State 2 ---->   turnLeft(Math.PI/2);
                Image image = camera.getImage();
                double distance = computeServoDistance(image);
State 3 ---->   goForward(distance);
                ....
```

Finite State Machines



Finite State Machines in Carmen

```
import Carmen.*;
public class MyController implements OdometryHandler
{
    int curState = 0;
    bool initialized = false;
    double goalX, goalY, goalTh;

    public void handleOdometry(OdometryMessage msg) {
      if (!initialized) {
        // initialize code
        return;
      }
      if (curState == 0) {
        // is the termination condition of state 0 true? if so, issue a command and advance to next state
        return;
      } else if (curState == 1) {
        // is the termination condition of state 0 true? if so, issue a command and advance to next state
      }
      .....
    }

    public static void main(String args[]) {
      MyController controller = new MyController();
      Robot.initialize();
      Robot.subscribeOdometryData(controller);
      Robot.dispatch();
    }
}
```

Finite State Machines in Carmen

```
import Carmen.*;
public class MyController implements OdometryHandler
{
    int curState = 0;
    bool initialized = false;
    double goalX, goalY, goalTh;

    public void handleOdometry(OdometryMessage msg) {
      if (!initialized) {
        goalX = msg.x; startY = msg.y +1; startTh = msg.Theta; initialized = true;
        Robot.setVelocity(1.0, 0.0);
        return;
      }
      if (curState == 0) {
        // is the termination condition of state 0 true? if so, issue a command and advance to next state
        return;
      } else if (curState == 1) {
        // is the termination condition of state 0 true? if so, issue a command and advance to next state
      }
      .....
    }

    public static void main(String args[]) {
      MyController controller = new MyController();
      Robot.initialize();
      Robot.subscribeOdometryData(controller);
      Robot.dispatch();
    }
}
```

Finite State Machines in Carmen

```
import Carmen.*;
public class MyController implements OdometryHandler
{
    int curState = 0;
    bool initialized = false;
    double goalX, goalY, goalTh;

    public void handleOdometry(OdometryMessage msg) {
      if (!initialized) {
        goalX = msg.x; startY = msg.y +1; startTh = msg.Theta; initialized = true;
        Robot.setVelocity(1.0, 0.0);
        return;
      }
      if (curState == 0) {
        if (Math.hypot(goalX-msg.X, goalY-msg.y) < .1) {
          curState++;
          Robot.setVelocity(0.0, Math.PI/8);
        }
        return;
      } else if (curState == 1) {
        // is the termination condition of state 0 true? if so, issue a command and advance to next state
      }
      .....
    }

    public static void main(String args[]) {
      MyController controller = new MyController();
      Robot.initialize();
      Robot.subscribeOdometryData(controller);
      Robot.dispatch();
```

The Anatomy of a Message

```
package RSS;

import Carmen.*;

public class MyMessage {
  [MESSAGE FIELDS]
  [MESSAGE NAME AND FORMAT]
  [MESSAGE CONSTRUCTOR]
  [INTERNAL MESSAGE HANDLER]
  [MESSAGE SUBSCRIBE METHOD]
  [MESSAGE PUBLICATION METHOD]
}
```

- Note that messages do not implement standard interfaces.
  - By convention, you should, however, implement a constructor, a message subscription method and a publication method.
  - You could also support query/response.
- Messages do, however, require a separate interface file to ensure type-safe message handling

```
public class MyMessageHandler {
  public void handleMyMessage(MyMessage message);
}
```

The Anatomy of a Message

```
package RSS;

import Carmen.*;

public class BlobMessage {
    public blobLocations[];
    public int numBlobs;
    public double timestamp;
    public String hostname;

  [MESSAGE NAME AND FORMAT]

  [MESSAGE CONSTRUCTOR]

  [INTERNAL MESSAGE HANDLER]

  [MESSAGE SUBSCRIBE METHOD]

  [MESSAGE PUBLICATION METHOD]
}
```

- Public fields have to come first in the message declaration.
- Every message **must** have a timestamp and hostname, and by convention, they **must** be the last two fields in the message.

The Anatomy of a Message

```
package RSS;

import Carmen.*;

public class BlobMessage {
    public int blobLocations[];
    public int numBlobs;
    public double timestamp;
    public String hostname;

    private final static String MESSAGE_NAME = “CARMEN_BLOB_MESSAGE”;
    private final static String MESSAGE_FMT = “{<int:2>,int,double,[char:10]}”;

  [MESSAGE CONSTRUCTOR]

  [INTERNAL MESSAGE HANDLER]

  [MESSAGE SUBSCRIBE METHOD]

  [MESSAGE PUBLICATION METHOD]
}
```

- The message format string is arcane, and easy to get wrong. Be careful to keep your messages simple.
- There is a formal definition in the IPC manual linked off the wiki,

The Anatomy of a Message

```
package RSS;

import Carmen.*;

public class BlobMessage {
    public int blobLocations[];
    public int numBlobs;
    public double timestamp;
    public String hostname;

    private final static String MESSAGE_NAME = “CARMEN_BLOB_MESSAGE”;
    private final static String MESSAGE_FMT = “{<int:2>,int,double,[char:10]}”;

    public BlobMessage(int blobLocations[]) {
        this.blobLocations = new int[blobLocations.length];
        System.arraycopy(blobLocations, 0, this.blobLocations, 0,
                         blobLocations.length);
        this.numblobs = blobLocations.length;
        this.timestamp = Util.getTime();
        this.hostname = Util.getHostName();
    }

  [MESSAGE SUBSCRIBE METHOD]

  [INTERNAL MESSAGE HANDLER]

  [MESSAGE PUBLICATION METHOD]
}
```

- Providing a constructor ensures that the module using your message does not have to remember to do things like fill in field lengths, or the timestamp and hostname.

The Anatomy of a Message

```
package RSS;

import Carmen.*;

public class blobMessage {
    public int blobLocations[];
    public int numBlobs;
    public double timestamp;
    public String hostname;

    private final static String MESSAGE_NAME = “CARMEN_BLOB_MESSAGE”;
    private final static String MESSAGE_FMT = “{<int:2>,int,double,[char:10]}”;

    public blobMessage(int blobLocations[]) {
      this.blobLocations = new int[blobLocations.length];
      System.arraycopy(blobLocations, 0, this.blobLocations, 0, blobLocations.length);
      this.numblobs = blobLocations.length;
      this.timestamp = Util.getTime();
      this.hostname = Util.getHostName();
    }

  public static void subscribe(blobHandler handler)
   {
     IPC.defineMsg(MESSAGE_NAME, MESSAGE_FMT);
     IPC.subscribeData(MESSAGE_NAME, new internalHandler(handler),
                       blobMessage.class);
     IPC.setMsgQueueLength(MESSAGE_NAME, 1);
   }

  [INTERNAL MESSAGE HANLER]

  [MESSAGE PUBLICATION METHOD]
}
```

The Anatomy of a Message

- Remember you have to define a separate interface class that handles your message
- The internal handler ensures that the handler that is called when a BlobMessage is received matches the handler type.

```
package RSS;

import Carmen.*;

public class BlobMessage {
    public int blobLocations[];
    public int numBlobs;
    public double timestamp;
    public String hostname;

    private final static String MESSAGE_NAME = “CARMEN_BLOB_MESSAGE”;
    private final static String MESSAGE_FMT = “{<int:2>,int,double,[char:10]}”;

    public blobMessage(int blobLocations[]) {
      this.blobLocations = new int[blobLocations.length];
      System.arraycopy(blobLocations, 0, this.blobLocations, 0, blobLocations.len
      this.numblobs = blobLocations.length;
      this.timestamp = Util.getTime();
      this.hostname = Util.getHostNam();
    }

  public static void subscribe(blobHandler handler)
  {
    IPC.defineMsg(MESSAGE_NAME, MESSAGE_FMT);
    IPC.subscribeData(MESSAGE_NAME, new internalHandler(handler),
                      blobMessage.class);
    IPC.setMsgQueueLength(MESSAGE_NAME, 1);
  }

   private static class internalHandler implements IPC.HANDLER_TYPE {
     private static MyHandler userHandler = null;
     PrivateMyHandler(MessageHandler userHandler) {
       this.userHandler = userHandler;
     }
     public void handle (IPC.MSG_INSTANCE msgInstance, Object callData) {
       MyMessage message = (MyMessage)callData;
       userHandler.handleMessage(message);
     }
   }

  [MESSAGE PUBLICATION METHOD]
}
```

The Anatomy of a Message

```
package RSS;

import Carmen.*;

public class BlobMessage {
    public int blobLocations[];
    public int numBlobs;
    public double timestamp;
    public String hostname;

    private final static String MESSAGE_NAME = “CARMEN_BLOB_MESSAGE”;
    private final static String MESSAGE_FMT = “{<int:2>,int,double,[char:10]}”;

    public blobMessage(int blobLocations[]) {
      this.blobLocations = new int[blobLocations.length];
      System.arraycopy(blobLocations, 0, this.blobLocations, 0, blobLocations.length);
      this.numblobs = blobLocations.length;
      this.timestamp = Util.getTime();
      this.hostname = Util.getHostNam();
    }

  public static void subscribe(blobHandler handler)
  {
    IPC.defineMsg(MESSAGE_NAME, MESSAGE_FMT);
    IPC.subscribeData(MESSAGE_NAME, new internalHandler(handler),
                      blobMessage.class);
    IPC.setMsgQueueLength(MESSAGE_NAME, 1);
  }

  private static class internalHandler implements IPC.HANDLER_TYPE {
    private static MyHandler userHandler = null;
    PrivateMyHandler(MessageHandler userHandler) {
      this.userHandler = userHandler;
    }
    public void handle (IPC.MSG_INSTANCE msgInstance, Object callData) {
      MyMessage message = (MyMessage)callData;
      userHandler.handleMessage(message);
    }
  }

    public void publish()
    {
      IPC.publishData(MESSAGE_NAME, this);
    }
}
```

“Good Practices”

- Ease of use
- Extensibility
- Robustness

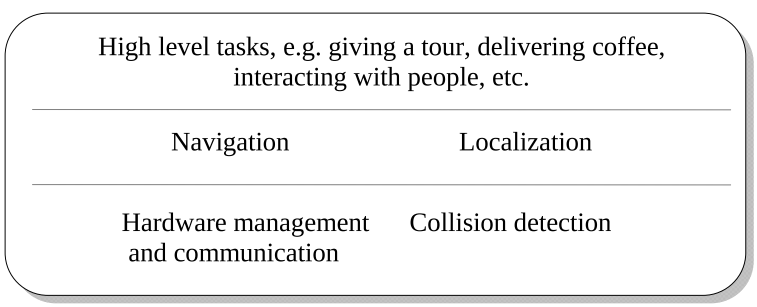
- CARMEN provides a framework for satisfying these principles.

Standardization

- Co-ordinate frame and unit standardization
  - Only 3 allowable co-ordinate frames
  - All units M-K-S
  - No left-handed co-ordinate systems
  - Always radians
  - Θ=0 is always along the +x-axis

Modularity

- Three rough groups of components
- Each component is a separate process
  - Enforces separability
  - Enforces robustness
  - Allows distribution of computation

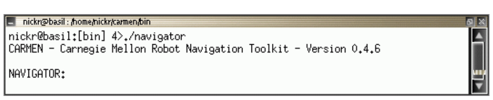


Alternatives: Monoliths, Threaded Architectures, ...

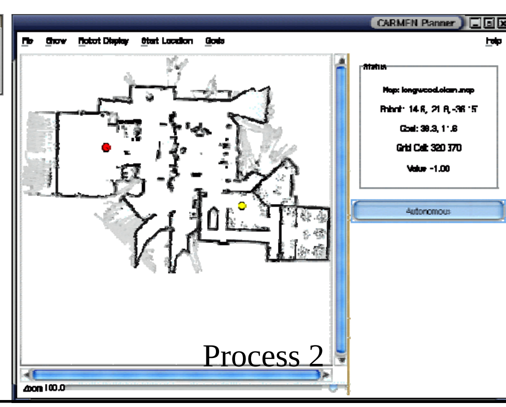
- Monolithic architecture
  - Debugging can be easier in a single process
  - No communication overhead
  - Control flow can get very messy when different tasks with different time constants need to be interleaved
  - Everything runs on the robot: need to have the computer horsepower onboard
  - Harder to extract components when not needed (e.g., displays)
- Threaded architectures
  - Control flow can become much cleaner
  - No communication overhead due to shared address space
  - Everything still can only run in a single place
  - Debugging multi-threaded implementations seem to be much harder than debugging multi-process implementations

Separation of Control and Display

- Model-View-Controller paradigm
- All data is accessible by other processes



Process 1



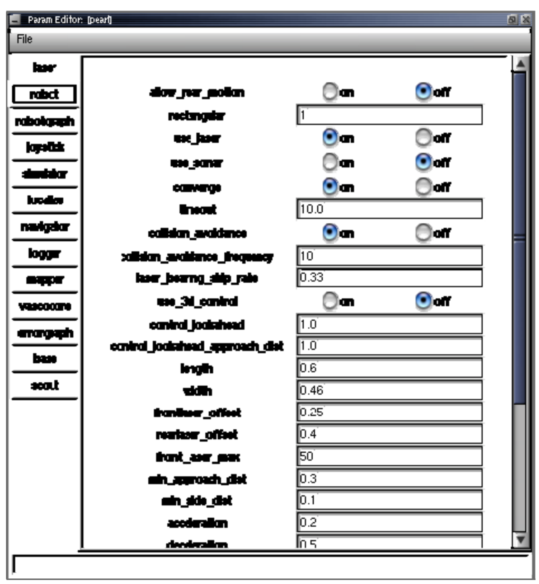
Process 2

Alternatives: Integrated Controller and View

- Cannot run controller on one machine (e.g., laptop) and the viewer on another machine (e.g., Sun workstation) without using X windows (high bandwidth)
- Any internal state of the controller that is shown in the view may not be accessible to other programs
- May result in not being able to run controller without display mechanism (e.g., headless display)

Centralized Model

- Ensures consistency across modules
- Programmatic interface allows run-time changes



Alternatives to the Centralized Model

- Distributed configurations
  - Every program gets its own command-line options, configuration file
  - Easier to implement
  - Separation of concerns means one process can't corrupt another's model
  - Extremely easy to have different processes with inconsistent models

Communication Abstraction

- Anonymous publish-and-subscribe
  - No module has to know *a priori* where any message comes from
  - Requires the message_daemon to know who is subscribed to a message and deliver the message appropriately (for n subscribers, requires n+1 network hops per message)
  - Callback mechanism provides a single point of entry for incoming data (clearer control flow)
  - Callback mechanism allows operating system to manage network polling (more efficient)
- Carmen encourages all modules to provide interface libraries that abstract away IPC details
  - Changes to the communication protocol at any time should be transparent to client modules

Alternative Communication Abstractions

- Point-to-point communication
  - Each module knows where each message comes from and subscribes to the source directly
  - More efficient in bandwidth: eliminates the need for the ipc_daemon, reduces the number of network hops for each message by 1
  - Requires each module know where to subscribe for each message
  - Prevents more efficient packet routing
- Query-response
  - Most communication protocols operate like this (including orcd)
  - cf. the UNIX file system
  - Meshes nicely with the “sequential” mental model of programming
  - Requires explicit polling, can leads to poor control flow
  - Requires each module to know where to query for each message

Implementing Tests

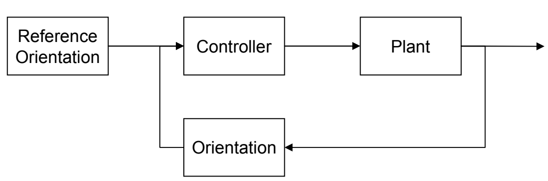
```
public class Pose {
  public double x, y, theta;

  public void updateHeading(double deltaTheta) {
    this.theta = this.theta+deltaTheta;

    if (theta >= -Math.PI && theta < Math.PI)
      return theta;

    if (theta >= Math.PI)
      theta -= 2*Math.PI;
    if (theta < -Math.PI)
      theta += 2*Math.PI;

    return theta;
   }
}
```

Implementing Tests

```
public class Pose {
  public double x, y, theta;

  public void updateHeading(double deltaTheta) {
    this.theta = this.theta+deltaTheta;

    if (theta >= -Math.PI && theta < Math.PI)
      return theta;

    if (theta >= Math.PI)
      theta -= 2*Math.PI;
    if (theta < -Math.PI)
      theta += 2*Math.PI;

    return theta;
  }

  public static void testUpdateHeading() {
    Pose p = new Pose(Math.random()*100, Math.random()*100,
   Math.random()*2*Math.PI);
    double deltaTheta = Math.random()*2*Math.PI);
    p.updateHeading(deltaTheta);
    assert(p.theta <= Math.PI);
    assert(p.theta > -Math.PI);
  }
}
```

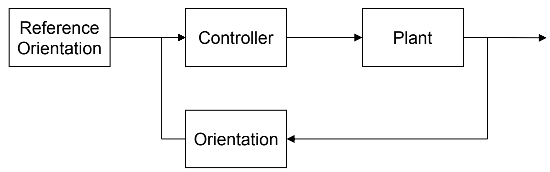
An external test to ensure that theta meets our bounds.

Test Data and Test Cases



- *Test data* Inputs which have been devised to test the system
- *Test cases* Inputs to test the system and the predicted outputs from these inputs if the system operates according to its specification
- Testing should be:
  - Repeatable:
    - If you find an error, you'll want to repeat the test to show others
    - If you correct an error, you'll want to repeat the test to check you did fix it
  - Systematic
    - Random testing is not enough
    - Select test sets that cover the range of behaviors of the program
    - are representative of real use
  - Documented
    - Keep track of what tests were performed, and what the results were

Preconditions, Postconditions and Invariants



- Preconditions/postconditions and invariants are commonly used in “design-by-contract” engineering
- Precondition - what must be true when a method is invoked. When a precondition fails, the method invoker has a fault.
- Postcondition - what must be true after a method completes successfully. When a postcondition fails, the method has a fault **or** the precondition was not met.
- Class Invariant - what must be true about each instance of a class after construction and after every method call. Also must true for static methods when there is no object of the class created. When an invariant fails, a fault could exist with the method invoker or the class itself.
- Another common kind of invariant is internal – conditions in the implementation we know must always hold

Implementing Preconditions

```
public class Pose {
  private double x, y, theta;
/**
  * Updates the heading.
  *
  * @param  deltaTheta heading change in radians.
  * @throws IllegalArgumentException if theta < -PI or
  * rate >= PI.
  */
  public void updateHeading(double deltaTheta) {
    if (deltaTheta < -Math.PI || deltaTheta >= Math.PI)
      throw new IllegalArgumentException("Invalid heading change: " + deltaTheta);
    this.theta = this.theta+deltaTheta;

    if (theta >= -Math.PI && theta < Math.PI)
      return theta;

    if (theta >= Math.PI)
      theta -= 2*Math.PI;
    if (theta < -Math.PI)
      theta += 2*Math.PI;

    assert result >= -Math.PI && result < Math.PI : this;

    return theta;
}`
```

We have explicit enforcement of the precondition here, but we would also write an external test to ensure this precondition is being enforced.

Implementing Preconditions

```
public class Pose {
  private double x, y, theta;
/**
  * Updates the heading.
  *
  * @param  deltaTheta heading change in radians.
  * @throws IllegalArgumentException if theta < -PI or
  * rate >= PI.
  */
  public void updateHeading(double deltaTheta) {
    if (deltaTheta < -Math.PI || deltaTheta >= Math.PI)
      throw new IllegalArgumentException("Invalid heading change: " + deltaTheta);
    this.theta = this.theta+deltaTheta;

    if (theta >= -Math.PI && theta < Math.PI)
      return theta;

    if (theta >= Math.PI)
      theta -= 2*Math.PI;
    if (theta < -Math.PI)
      theta += 2*Math.PI;

    assert result >= -Math.PI && result < Math.PI : this;

    return theta;
  }
  public static void testUpdateHeading() {
    Pose p = new Pose(Math.random()*100, Math.random()*100, Math.random()*2*Math.PI);
    double deltaTheta = 4*Math.PI;
    try {
      p.updateHeading(deltaTheta);
      assert(false);
    }
    catch (Exception e) { }
  }
}`
```

The test only succeeds if an exception is thrown before this point.

Guards

- Preconditions, postconditions and many internal invariants are properties that you can test in the method body itself. These internal tests we call "guards".
- We can also write external "black-box" tests to make sure the guards are upheld
- Including postcondition and internal invariant tests in the method body is part of a larger practice known as "defensive programming"
- Writing explicit tests for postconditions and invariants is somehow more "intuitive": you are checking to make sure the method worked correctly and the postconditions and invariants hold for every method
- There is an issue here with preconditions: you want to make sure that not only does the method accept reasonable arguments, but you want to test for failure of violated preconditions.

- In many cases, testing involves ensuring an exception is thrown.

Implementing Postconditions

```
public class Pose {
  private double x, y, theta;
/**
  * Updates the heading.
  *
  * @param  deltaTheta heading change in radians.
  * @throws IllegalArgumentException if theta < -PI or
  * rate >= PI.
  */
  public void updateHeading(double deltaTheta) {
  // Test precondition
    if (deltaTheta < -Math.PI || deltaTheta >= Math.PI)
      throw new IllegalArgumentException("Invalid heading change: " + deltaTheta);
    this.theta = this.theta+deltaTheta;

    if (theta >= -Math.PI && theta < Math.PI)
      return theta;

    if (theta >= Math.PI)
      theta -= 2*Math.PI;
    if (theta < -Math.PI)
      theta += 2*Math.PI;

    assert result >= -Math.PI && result < Math.PI : this;

    return theta;
}
```

We have explicit enforcement of the postcondition here, but we would also write an external test to ensure this postcondition is being enforced.

Class Invariants

```
public class Pose {
  private double x, y, theta;
/**
  * Updates the heading.
  *
  * @param  deltaTheta heading change in radians.
  * @throws IllegalArgumentException if theta < -PI or
  * rate >= PI.
  */
  public void updateHeading(double deltaTheta) {
  // Test precondition
    if (deltaTheta < -Math.PI || deltaTheta >= Math.PI)
      throw new IllegalArgumentException("Invalid heading change: " + deltaTheta);
    this.theta = this.theta+deltaTheta;

    if (theta >= -Math.PI && theta < Math.PI)
      return theta;

    if (theta >= Math.PI)
      theta -= 2*Math.PI;
    if (theta < -Math.PI)
      theta += 2*Math.PI;

    assert theta >= -Math.PI && theta < Math.PI : this;

    return theta;
}
```

This post-condition could be modelled as class invariant in other methods and the constructor. Can we write external tests to ensure that it holds after all method calls?

Internal Invariants

```
if (i % 2 == 0) {
  ...
} else { // i % 2 == 1?
   ...
}

switch(parity) {
  case Parity.EVEN:
      ...
      break;
  case Parity.ODD:
      ...
      break;
}

void method() {
  for (...) {
    if (...)
      return;
    }
  // We should never be here
}
```

This switch statement contains the (incorrect) assumption that parity can have one of only two values. To test this assumption, you should add the following default case:

```
default:
       assert false : parity;
```

Equivalence Partitioning

- Input data and output results often fall into different classes where all members of a class are related
- Each of these classes is an equivalence partition where the program behaves in an equivalent way for each class member
- Test cases should be chosen from each partition
- Example:
  - If input is a 5-digit integer between 10,000 and 99,999, equivalence partitions are <10,000, 10,000-99,999 and >99,999
  - Choose test cases at the boundary of these sets: 9999, 10000, 99999, 100000
  - Consider adding additional cases: 50000? -1? 0? Others?
- Input partitions:
  - Inputs which conform to the preconditions
  - Inputs where a pre-condition does not hold
  - Edge cases
- Other guidelines for preconditions
  - Test software with arrays which have only a single value
  - Use arrays of different sizes in different tests
  - Derive tests so that the first, middle and last elements of the array are accessed
  - Test with arrays of zero length

What You (Hopefully) Learned Today

- About the Carmen modules and what they do
- About callback-based programming
- Some of the design principles underlying Carmen, the tradeoffs we made and why
- Some good software development practices