Developing Software in Carmen

6.189/2.994/16.401

October 5th, 2005

3 Topics

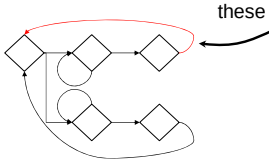
- Writing New Modules
- Writing New Message Classes
- Writing Test Harnesses

State Machines

- You are used to thinking of programs like this:

```
....
start_manipulator();
brick_num = next_closest_brick();
set_planner_goal(brick_num);
start_planner();
.....
```

- Instead, think of this:



The Anatomy of a Module

- Initialize IPC Connection
- Create handlers
- Set subscriptions
- Dispatch
  - Handle a message
  - Do some work
  - Update the internal state
  - Go back to dispatching

The Anatomy of a Module

```
package RSS;

import Carmen.*;

public class BrickFinder implements CameraHandler {
  public void handleCamera (CameraMessage message) {
    System.out.println("Please tell me the brick colour: ");
    String colourName = System.in.readline();
    int bricks [] = processImage(message, colourName);
    BrickMessage msg = new BrickMessage(bricks);
    msg.publish();
  }

  public static void main (String args[]) {
    Robot.initialize("BrickFinder", argv[0]);
    BrickFinder finder = new BrickFinder();
    Robot.subscribeCamera(finder);
    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 BrickMessage {
    public brickLocations[];
    public int numBricks;
    public double timestamp;
    public char hostname[10];

  [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 BrickMessage {
    public int brickLocations[];
    public int numBricks;
    public double timestamp;
    public char hostname[10];

    private final static String MESSAGE_NAME = "CARMEN_BRICK_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 BrickMessage {
    public int brickLocations[];
    public int numBricks;
    public double timestamp;
    public char hostname[10];

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

    public BrickMessage(int brickLocations[]) {
        this.brickLocations = new int[brickLocations.length];
        System.arraycopy(brickLocations, 0, this.brickLocations, 0,
                         brickLocations.length);
        this.numBricks = brickLocations.length;
        this.timestamp = Util.getTime();
        this.hostname = Util.getHostName();
        IPC.
    }

  [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 BrickMessage {
    public int brickLocations[];
    public int numBricks;
    public double timestamp;
    public char hostname[10];

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

    public BrickMessage(int brickLocations[]) {
      this.brickLocations = new int[brickLocations.length];
      System.arraycopy(brickLocations, 0, this.brickLocations, 0, brickLocations.length);
      this.numBricks = brickLocations.length;
      this.timestamp = Util.getTime();
      this.hostname = Util.getHostName();
    }

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

  [INTERNAL MESSAGE HANLER]

  [MESSAGE PUBLICATION METHOD]
}
```

The Anatomy of a Message

```
package RSS;

import Carmen.*;

public class BrickMessage {
    public int brickLocations[];
    public int numBricks;
    public double timestamp;
    public char hostname[10];

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

    public BrickMessage(int brickLocations[]) {
      this.brickLocations = new int[brickLocations.length];
      System.arraycopy(brickLocations, 0, this.brickLocations, 0, brickLocations.length);
      this.numBricks = brickLocations.length;
      this.timestamp = Util.getTime();
      this.hostname = Util.getHostName();
    }

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

  private static class internalHandler implements IPC.HANDLER_TYPE {
    private static BrickHandler userHandler = null;
    PrivateBrickHandler(MessageHandler userHandler) {
      this.userHandler = userHandler;
    }
    public void handle (IPC.MSG_INSTANCE msgInstance, Object callData) {
      BrickMessage message = (BrickMessage)callData;
      userHandler.handleBrick(message);
    }
  }

  [MESSAGE PUBLICATION METHOD]
}
```

- 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 BrickMessage is received matches the handler type.

```
public class BrickHandler {
  public void handleBrick(BrickMessage message);
}
```

The Anatomy of a Message

```
package RSS;

import Carmen.*;

public class BrickMessage {
    public int brickLocations[];
    public int numBricks;
    public double timestamp;
    public char hostname[10];

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

    public BrickMessage(int brickLocations[]) {
      this.brickLocations = new int[brickLocations.length];
      System.arraycopy(brickLocations, 0, this.brickLocations, 0, brickLocations.length);
      this.numBricks = brickLocations.length;
      this.timestamp = Util.getTime();
      this.hostname = Util.getHostName();
    }

  public static void subscribe(BrickHandler handler)
  {
    IPC.defineMsg(MESSAGE_NAME, MESSAGE_FMT);
    IPC.subscribeData(MESSAGE_NAME, new internalHandler(handler),
                      BrickMessage.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.defineMsg(MESSAGE_NAME, MESSAGE_FMT);
    IPC.publishData(MESSAGE_NAME, this);
  }
}
```

- We IPC.define'd the message in the subscribe. Why do we need to define it here as well?
- Is this a good idea? Could we do it better?

Issues

- Concurrency
  - What if a handler needs to run for a long time?
  - I warn you now: threads will **not** help you.
- Graphical Displays
  - How to display the internal state of a program for debugging?
  - Use a timer, and call Robot.listen() for a few ms.
- Initialization
  - How to make sure every module agrees on the size of the robot?
  - Use the param_daemon to store/get parameters.

Some useful IPC methods

```
public static int disconnect () ;
public static boolean isConnected ();
public static boolean isModuleConnected (String moduleName);
public static int defineMsg (String msgName, String
   formatString);
public static boolean isMsgDefined (String msgName);
public static int listen (long timeoutMSecs);
public static int listenClear (long timeoutMSecs);

public static Object queryResponseData
(String msgName, Object data, Class responseClass, long
   timeoutMSecs);
public static int respondData (MSG_INSTANCE msgInstance,
   String msgName, Object data)
```

Unit Testing

- How can things go wrong between the design review and implementation?
  - We forget what we promised messages would look like.
  - We forget what messages we promised to send.
  - We forget what messages we promised to subscribe to.
  - We forget what we said was reasonable for a message to contain.
  - We forget what we said was the right order of things (i.e., what our state machine is supposed to look like).
  - We forget some important edges cases.
  - We forget what happens to us if we use degrees

Unit Testing for Message Sending

- "If a program feature lacks an automated test, we assume it doesn't work. This seems much safer than the prevailing assumption, that if a developer assures us a program feature works, then it works now and forever." www.junit.org
- We can build unit tests to automate some simple co ordination tests
- Can we automate everything?
- Many projects dictate that unit testing is part of the build process. Can we do this? Is it a good idea?

The Anatomy of a UnitTest

```
package RSS;

import Carmen.*;
import junit.*;

public class [TESTCLASSNAME] {
  public void test[FIRSTTESTNAME]() {
     .....
  }

 public void test[SECONDTESTNAME]() {
     .....
  }

  public static void main(String args[])
  {
    TestSuite suite = new TestSuite(TESTCLASSNAME.class);
    junit.textui.TestRunner.run(suite);
  }

}
```

- 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

A Unit Test for the BrickFinder

```
package RSS;

import Carmen.*;
import junit.*;

public class BrickFinderTest implements BrickHandler {
  public void test[FIRSTTESTNAME]() {
     .....
  }

 public void test[SECONDTESTNAME]() {
     .....
  }

  public static void main(String args[])
  {
    TestSuite suite = new TestSuite(BrickFinderTest.class);
    junit.textui.TestRunner.run(suite);
  }
}
```

A Unit Test for the BrickFinder

```
package RSS;

import Carmen.*;
import junit.*;

public class BrickFinderTest implements BrickHandler {

  private gotABrick = false;

  public void handleBrick(BrickMessage message) {
    gotABrick = true;
  }

  public void testBrickFinderSubscribe() {
     Robot.initialize("BrickFinderTest", "localhost");
     CameraMessage message = constructFakeImage();
     message.publish();
     BrickMessage.subscribe(this);
     Robot.listen(1000);
     assertTrue(gotABrick);
  }

 public void test[SECONDTESTNAME]() {
    .....
  }

  public static void main(String args[])
  {
    TestSuite suite = new TestSuite(BrickFinderTest.class);
    junit.textui.TestRunner.run(suite);
  }
}
```

- Unit tests are not an exact science.
- What can go wrong with our test during these three statements?
- What about during just this statement alone?

Additional Carmen Modules

- Laser
- Laser simulator (no vision, manipulator simulator)
- Localize (laser-based, map-based localization)
- Navigator (laser-based, map-based numerical potential field motion planner) and navigator_panel (map gui)
- Map builder (vasco)
- Documentation exists at http://www.cs.cmu.edu/~carmen

- Some gotchas in using these modules
  - Many of them do not have Java libraries yet. (That's ok -- you're replacing these modules anyhow. But they may help in bootstrapping your particular section.)
  - You must have a map. (There is one checked into rssII/data, called "longwood.cmf".)
  - The simulator subsumes orc, laser and robot_central. You do not need to run any of these with the simulator.
  - You do need to run localize, navigator and navigator_panel to position the robot graphically.