Introduction to Multithreaded Real-Time Systems and Debugging

Jonah Caplan

McGill University

jonah.caplan@mail.mcgill.ca

October 24, 2015

Structure of tutorial

1. Review of threads and important methods
2. How to structure a real-time system using threads
3. How to write a finite state machine
4. How to properly log debugging and sensor info

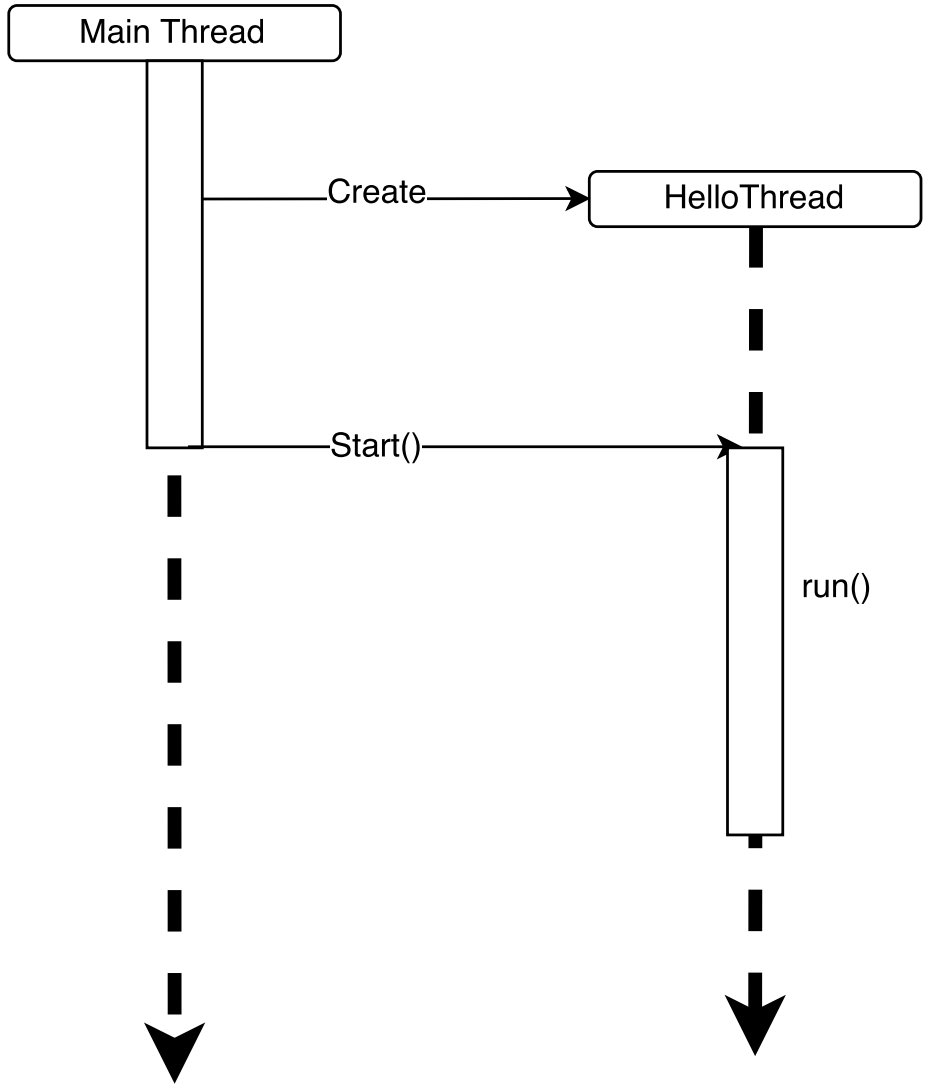
Review of threads and important methods

- There are two ways to use threads, but you should always extend the Thread class.
- Your thread class needs a `run()` method (line 3-5).
- When you create a thread you need to start it (line 8).

```
1  public class HelloThread extends Thread {
2
3      public void run() {
4          System.out.println("Hello from a thread!");
5      }
6
7      public static void main(String args[]) {
8          (new HelloThread()).start();
9      }
10
11 }
```

NOTE: taken from Oracle tutorials.

What actually happened



New to sequence diagrams? →
https://en.wikipedia.org/wiki/Sequence_diagram

Join method

- In the last example, main does not wait for HelloThread to finish running.

```
1  public class ThreadJoin {
2
3      public void run() {
4          System.out.println("Hello from a thread!");
5      }
6
7      public static void main(String args[]) {
8          (new HelloThread()).start();
9          System.out.println("Hello from main!");
10     }
11
12 }
```

outputs:

```
1  Hello from main!
2  Hello from a thread!
```

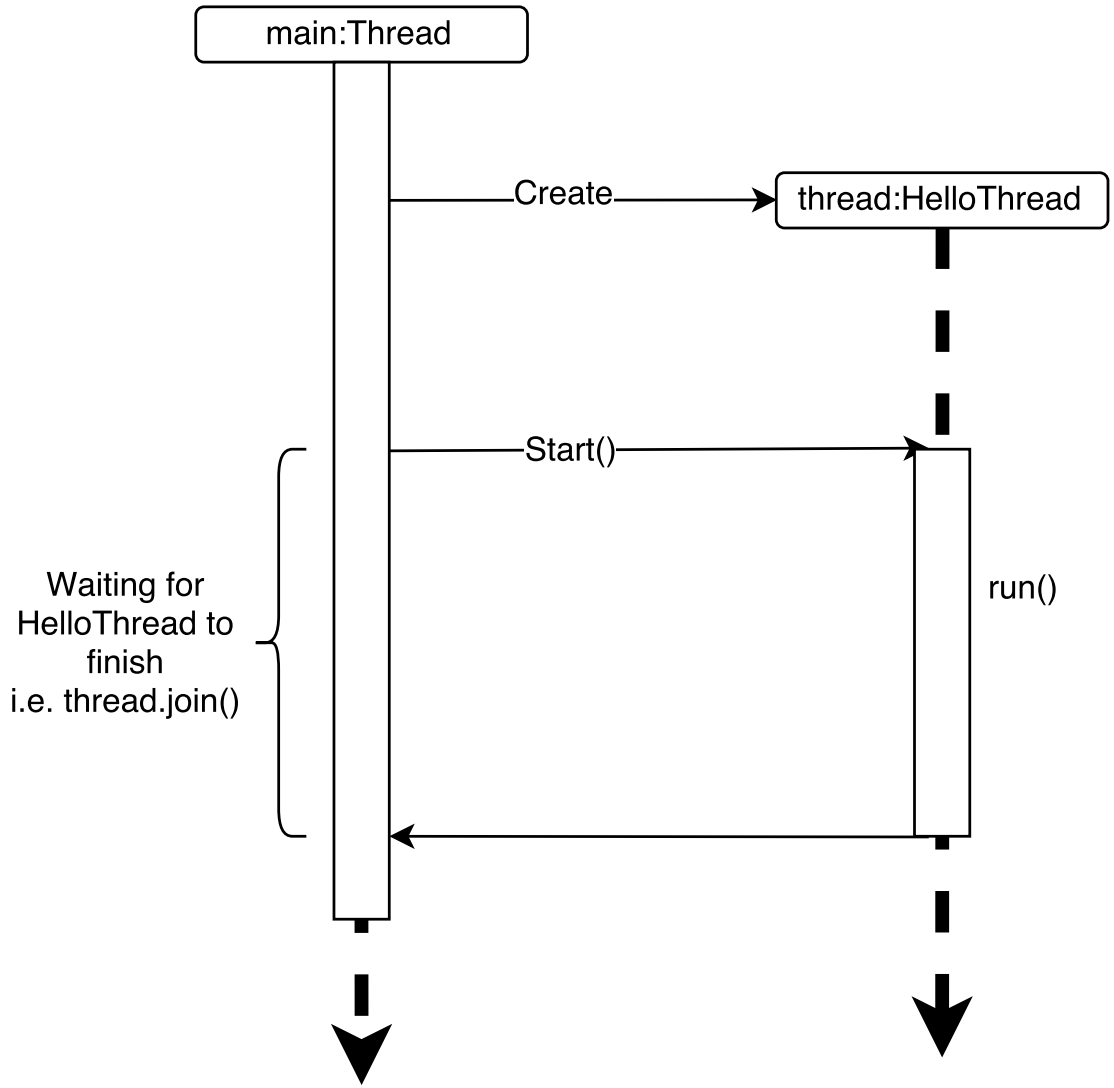
The join method forces main thread to wait for the spawned thread.

```
public class ThreadJoin {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        HelloThread thread = new HelloThread();
        thread.start();
        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Hello from main!");
    }

}
```

outputs:

```
Hello from a thread!
Hello from main!
```

The corresponding sequence diagram:



Real-time systems

There are two ways to effectively use threads in Java for a real-time application corresponding to two types of jobs:

- A single long sequential workload that needs to be done once.
- A periodic task that executes control flow or a state machine.

First we will examine how to structure a real problem in these terms without considering the Java implementation details.

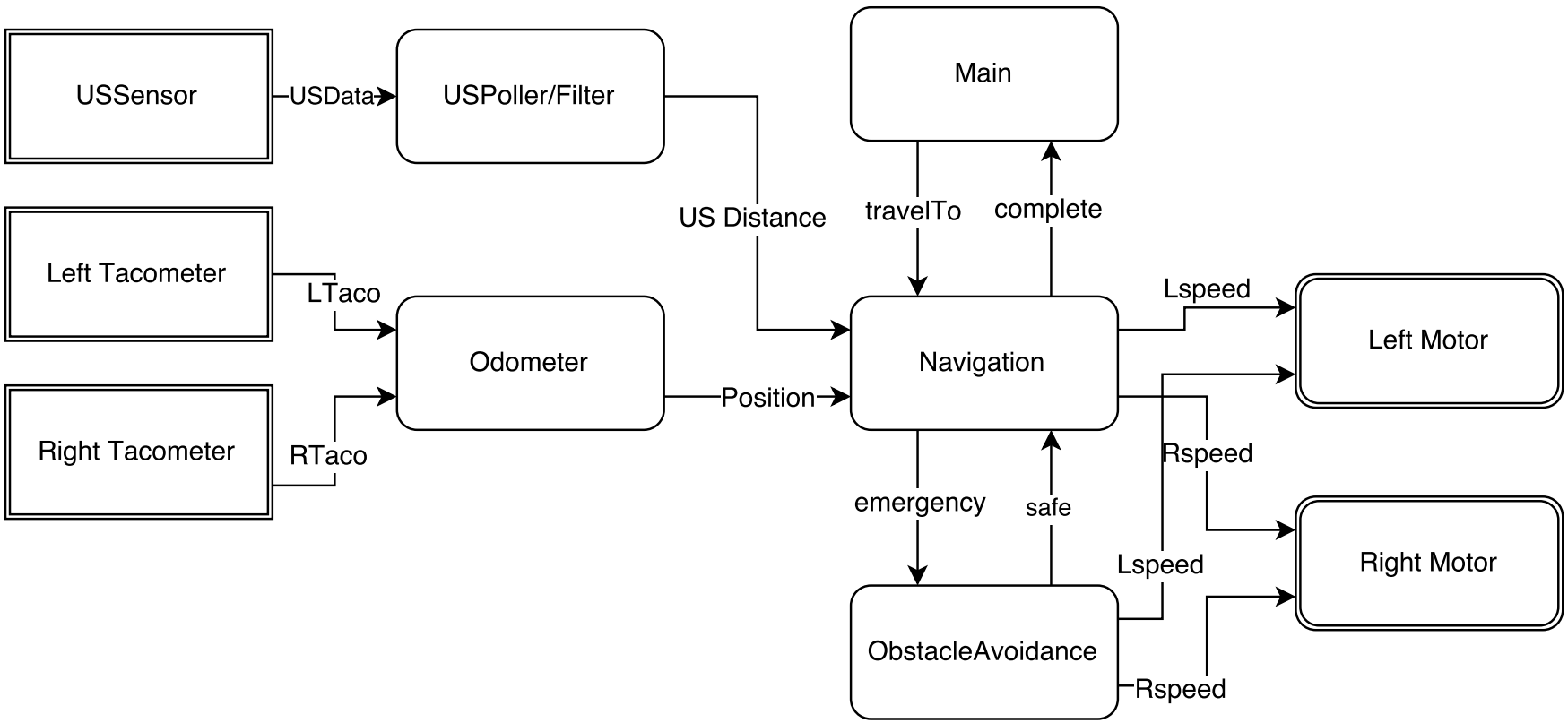
Problem: Navigation and obstacle avoidance

Design a robot that can:

- Travel to an arbitrary destination
- Avoid any obstacles along the way
- Sound familiar?

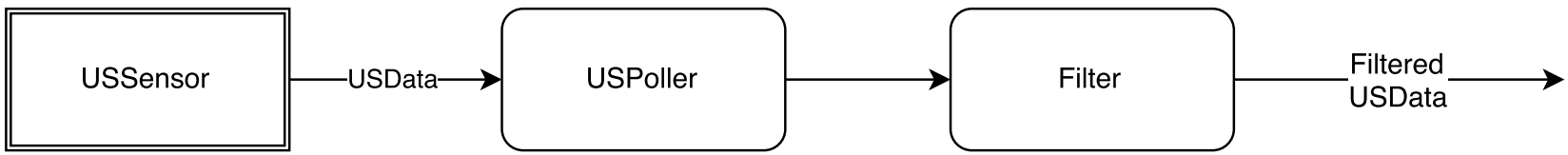
Start with a block diagram of dataflow (e.g. Simulink or LabVIEW)

Shows key data that must be passed between different functions in system.



Collecting sensor data

- Each sensor should have its own thread.
- For simple cases the filtering of data may go in the polling thread.
- For complex filters it is better to represent them as separate entities.



Problem with dataflow

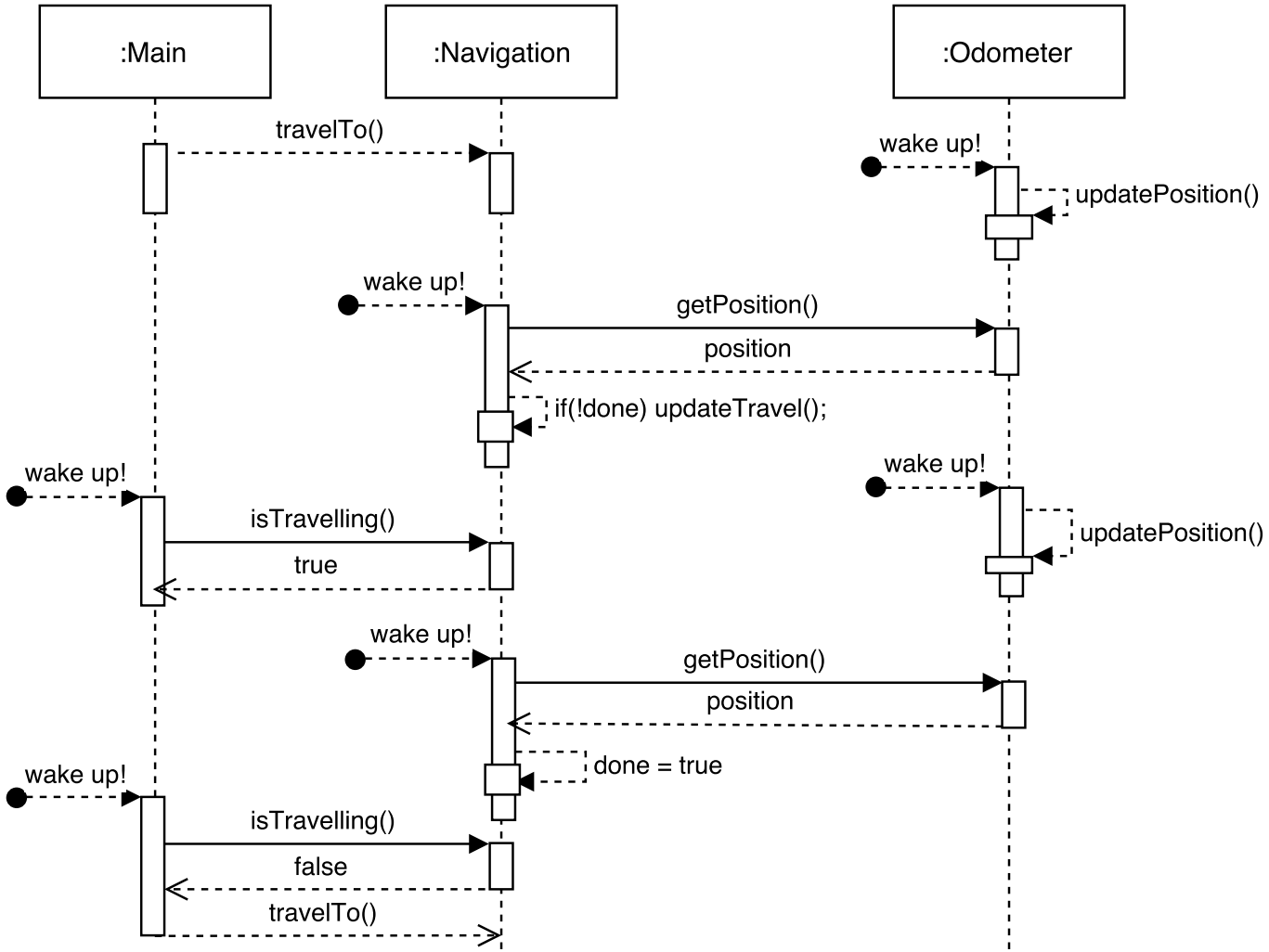
Dataflow doesn't tell us everything.

- What order do events happen in?
- How often is data updated?
- Which events are independent and which events are causal?
- Arrows show direction of information flow but not good representation of how code will look.

We need **sequence diagrams** to get a better picture of how to implement this system with Java threads.

Incremental design - First no obstacle detection

Here is the sequence diagram for the three main threads without obstacle detection.



Review

Q: Which type of threads are these?

Review

Q: Which type of threads are these?
A: Periodic tasks that execute control flow.

Review

Q: Which type of threads are these?
A: Periodic tasks that execute control flow.
Q: How to implement this behaviour in Java (in a way that's easy to integrate with future modifications/additions)?

Review

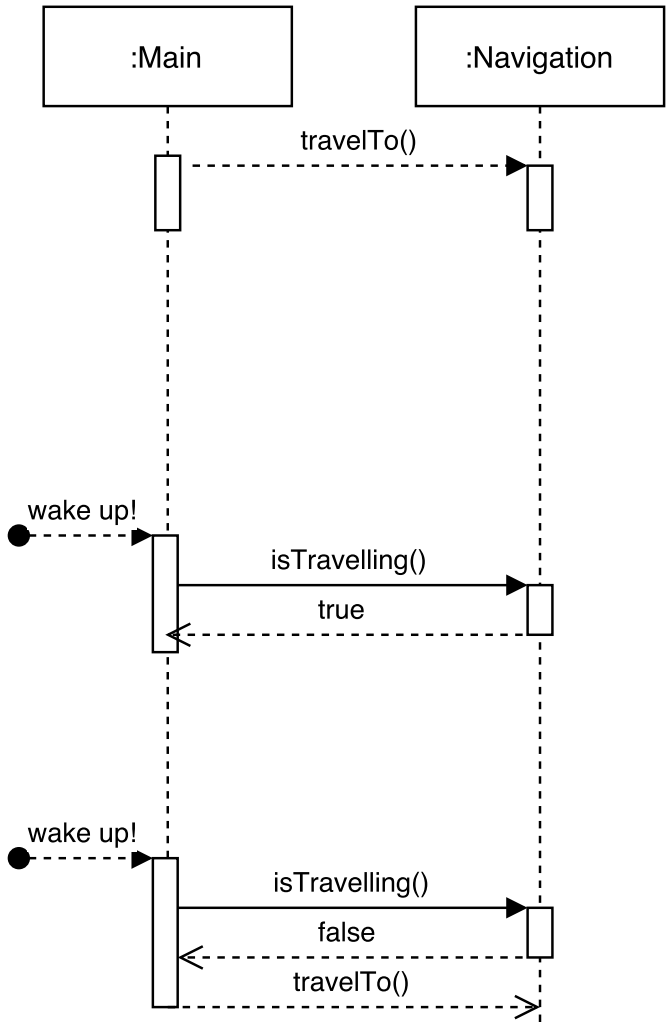
Q: Which type of threads are these?
A: Periodic tasks that execute control flow.
Q: How to implement this behaviour in Java (in a way that's easy to integrate with future modifications/additions)?
A: We shall see!

Things to note

- `travelTo()` is asynchronous! (i.e. it returns before the travelling is complete)
- The fact that the objects are threads are represented by the **wake up!** signals representing when the sleep time has expired.
- The Navigation `run()` method (which is where you are after the **wake-up**) calls `updateTravel()` to update the wheel speeds.
- Main is also a thread and goes to sleep just like the others.
- The timing of the main functionality of each thread is independent (comes from the sleep time instead of other threads).

Main class

```
public class Main {
  //Stuff omitted...
  public static void main(String[] args) throws
    InterruptedException {
    odometer = new Odometer();
    odometer.start();
    nav = new Navigation(odometer,leftMotor,rightMotor,
    usPoller);
    nav.start();
    completeCourse();
  }

  private static void completeCourse() throws
    InterruptedException {
    int[][] waypoints = {{60,30},{30,30},{30,60},{60,0}};
    for(int[] point : waypoints){
      nav.travelTo(point[0],point[1]);
      while(nav.isTravelling()){
        Thread.sleep(500);
      }
    }
  }

}
```




```
1  private static void completeCourse() throws
       InterruptedException {
     int[][] waypoints =
       {{60,30},{30,30},{30,60},{60,0}};
3    for(int[] point : waypoints){
       nav.travelTo(point[0],point[1]);
5      while(nav.isTravelling()){
         Thread.sleep(500);
7      }
     }
9  }
```

Convince yourself that this code and this picture mean the same thing!

Navigator class

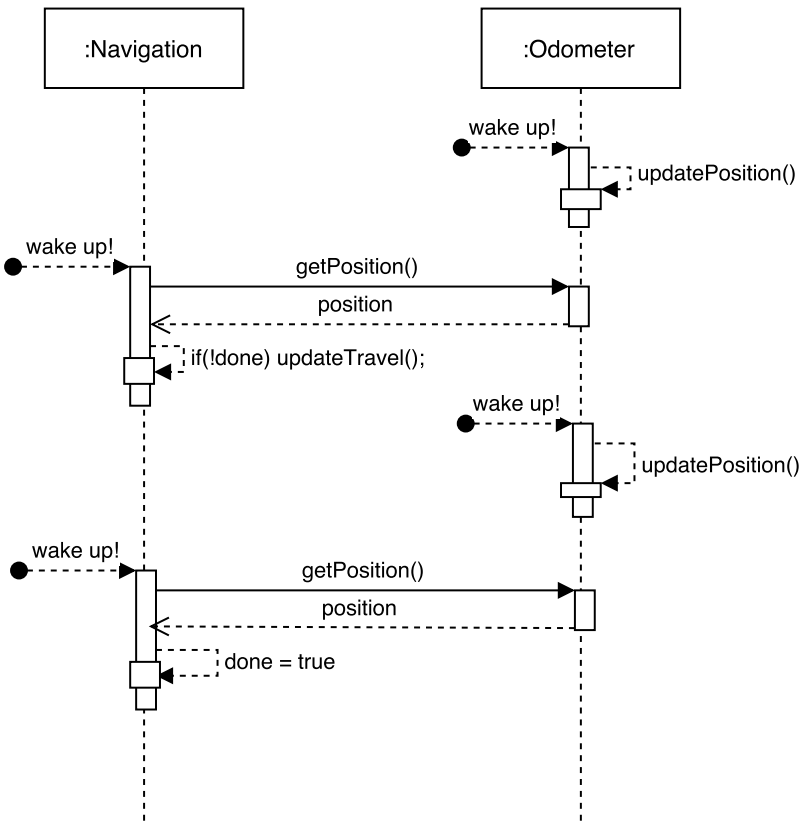
For the Navigator class let's start with `travelTo()`:

```
1 public void travelTo(double x, double y) {
    destx = x;
3   desty = y;
    destAngle = getDestAngle();
5   isNavigating = true;
}
```

- First the destination is stored and the initial angle is calculated
- The `isNavigating` flag is initialized

Navigator class

Next up is the `run()` method:

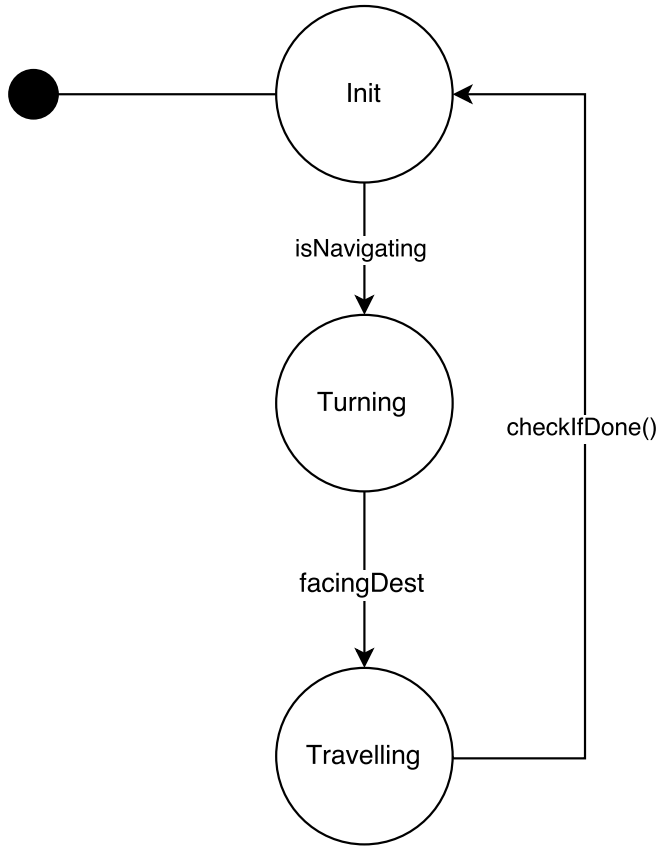



```
public void run() {
  while (true) {
    if (isNavigating) {
      // First turn to your destination
      if (!facingDest) {
        double angleDest = getDestAngle();
        turnTo(angleDest);
      } else {
        // facing your destination -> time to drive
        double[] distance = odometer.getPosition();
        if(!checkIfDone(distance)){
          updateTravel();
        } else {
          stopDriving();
        }
      }
    }
    try {
      Thread.sleep(30);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

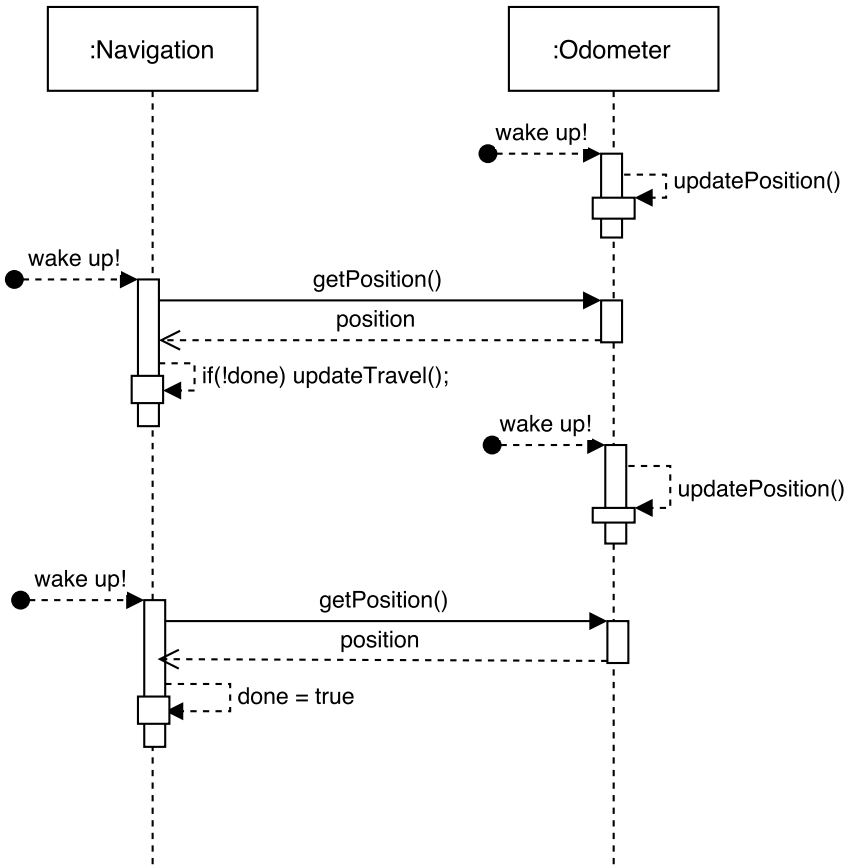
Some logic has been added to turn to the destination at start.
Some flags must be set in the helper methods.
Q: Is there a cleaner way to accomplish the same thing?

State machines

The same control flow could be represented by a state machine:



- Writing periodic code as a state machine will make future modifications easier.
- Get rid of all those large and difficult nested if blocks.



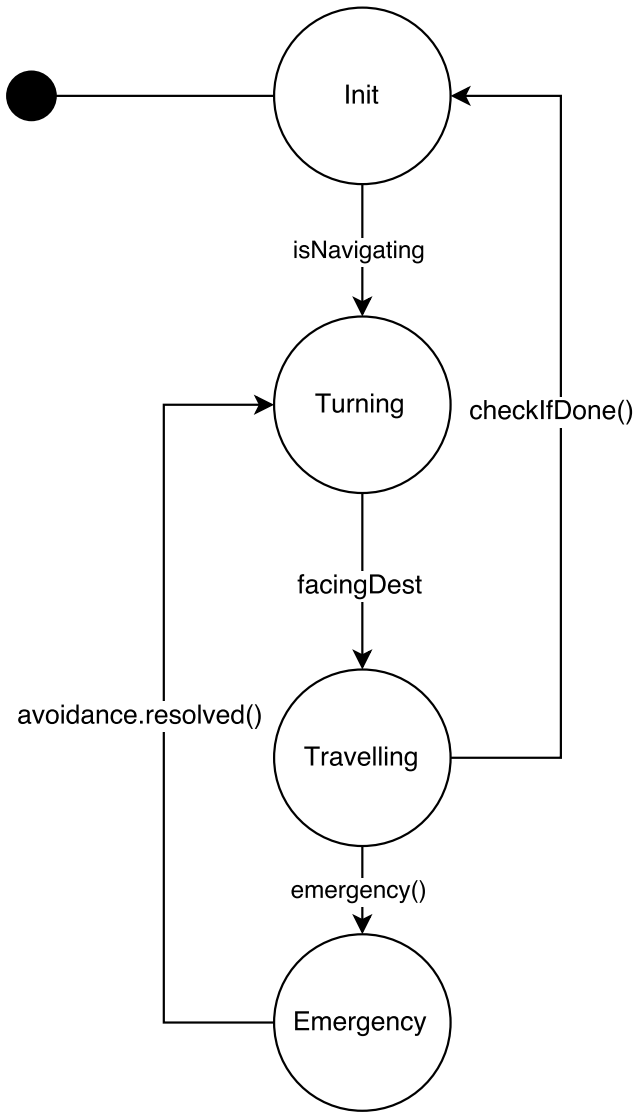

```
enum State {INIT,TURNING,TRAVELLING};

public void run(){
  State state = State.INIT;
  while(true){
    switch(state){
    case INIT:
      if(isNavigating){
        state = State.TURNING;
      }
      break;
    case TURNING:
      turnTo(destAngle);
      if(facingDest(destAngle)){
        state = State.TRAVELLING;
      }
      break;
    case TRAVELLING:
      double[] distance = odometer.getPosition();
      if(!checkIfDone(distance)){
        updateTravel();
      } else { //Arrived!
        setSpeeds(0,0);
        isNavigating = false;
        state = State.INIT;
      }
      break;
    }
    try {
      Thread.sleep(30);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

A few more lines, but more clear and easier to modify in future (as more states and possible branches are added)

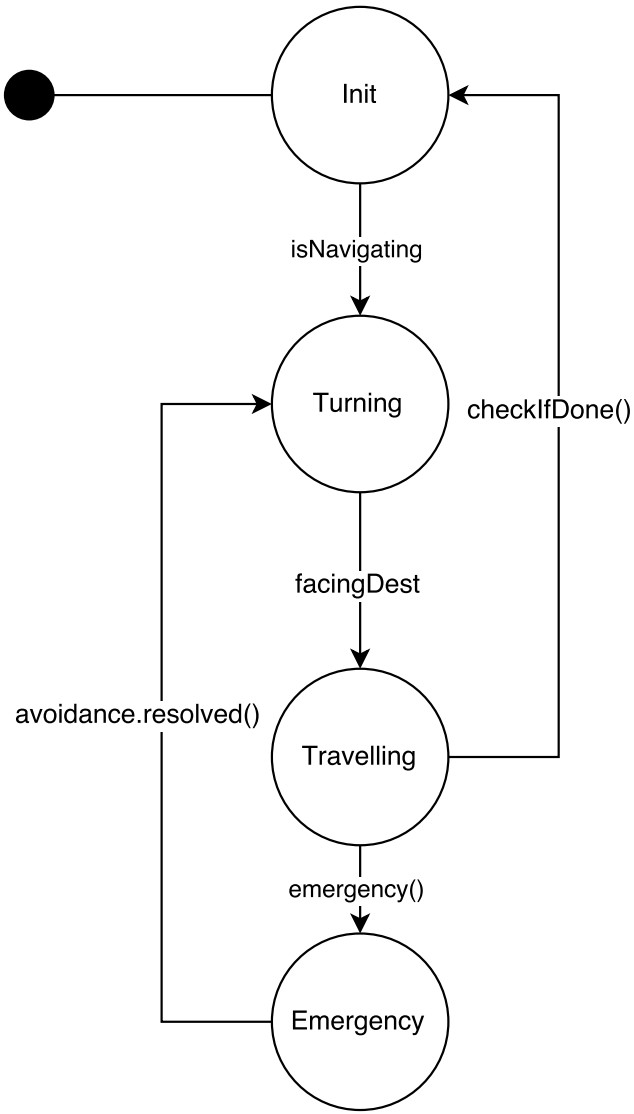
Less nesting $\rightarrow$ less bugs!

Adding in obstacle avoidance

Here's what happens to the state machine:



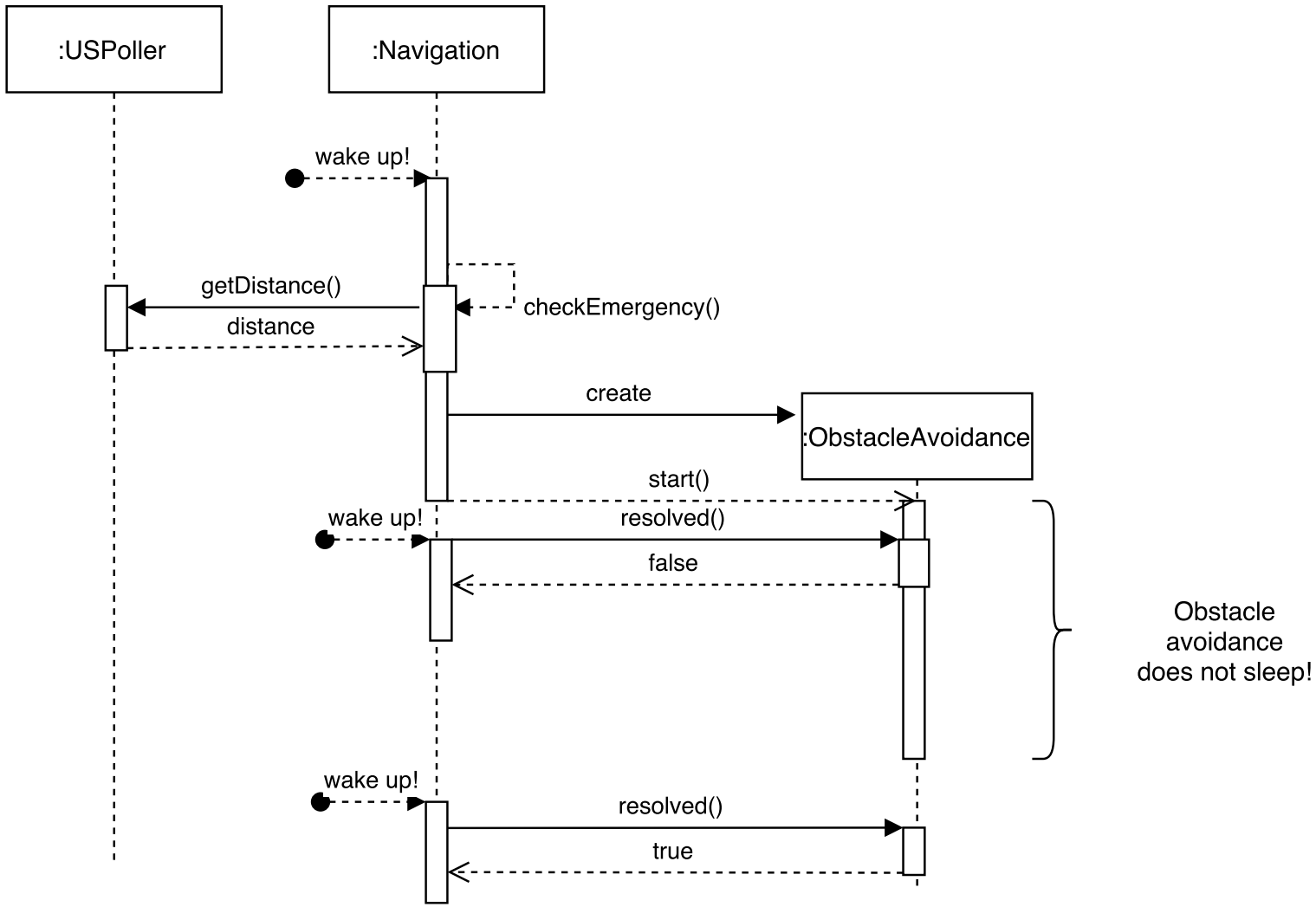
Add new state to state machine code:



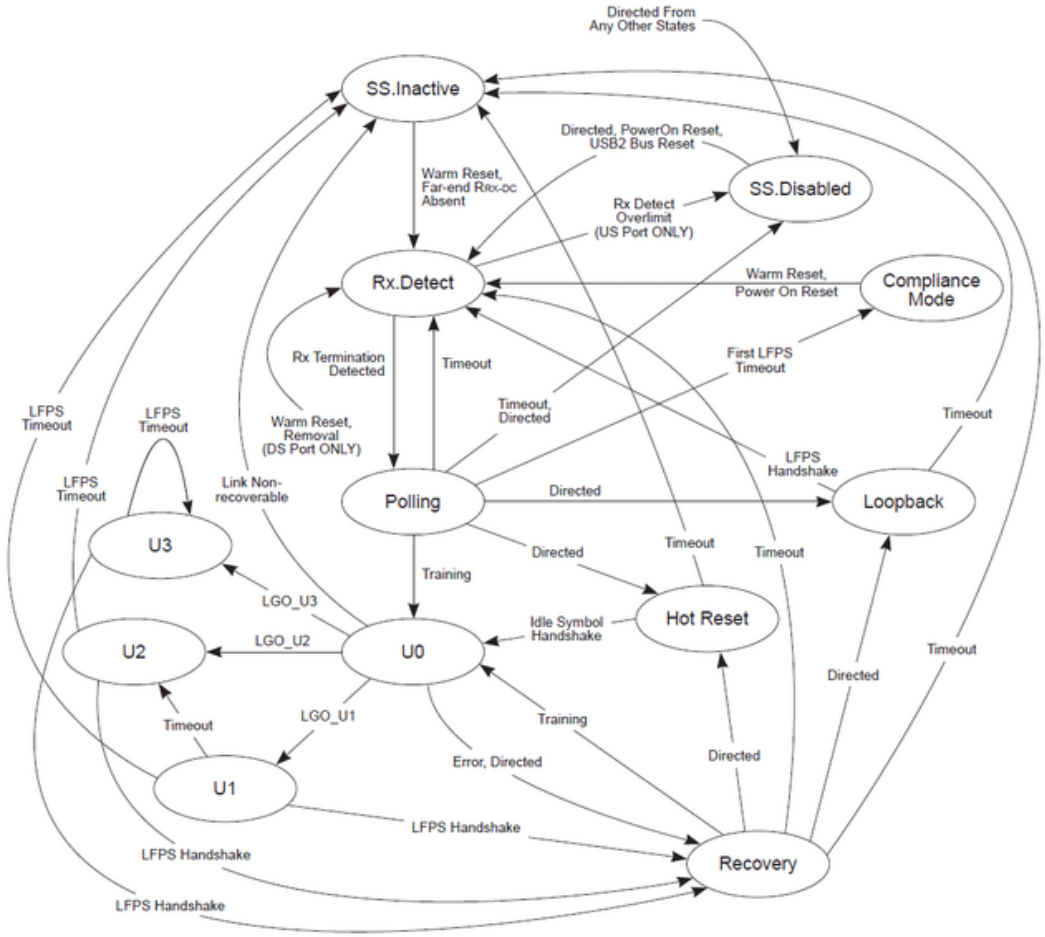

```
case TRAVELLING:
  double[] distance = odometer.getPosition();
  if (checkEmergency()) { //order matters!
    state = State.EMERGENCY;
    avoidance = new ObstacleAvoidance(this);
    avoidance.start();
  } else if (!checkIfDone(distance)) {
    updateTravel();
  } else { // Arrived!
    setSpeeds(0, 0);
    isNavigating = false;
    state = State.INIT;
  }
  break;
case EMERGENCY:
  if (avoidance.resolved()) {
    state = State.TURNING;
  }
  break;
```

```java
public class ObstacleAvoidance extends Thread{

  Navigation nav;
  boolean safe;

  public ObstacleAvoidance(Navigation nav){
    this.nav = nav;
    this.safe = false;
  }

  public void run(){
    //No infinite loop!!
    //Do some obstacle avoidance here
    //This is likely a single long sequential
    //list of instructions based on Lab 1.

    //After the instructions are complete
    safe = true;
  }

  public boolean resolved(){
    return safe;
  }

}
```

Q: Can you draw a sequence diagram for the Navigation and ObstacleAvoidance class?


:USPoller
:Navigation
wake up!
getDistance()
distance
checkEmergency()
create
:ObstacleAvoidance
start()
wake up!
resolved()
false
Obstacle
avoidance
does not sleep!
wake up!
resolved()
true

As life gets more complicated (branching increases in dynamic system) state machines are a very useful design pattern:



http://www.totalphase.com/support/articles/200349256-USB-Background

Good logging practices for debugging

- Write to a file
- Make the filename a function of the current time in milliseconds
- Use a Log class
- Provide a mechanism to choose which threads are printed (Threads always send messages but the logger may ignore them).
- Use `scp` to get the log files off of your brick.

```java
public class Log {

  static PrintStream writer = System.out;

  public static enum Sender {
    odometer, Navigator, usSensor, avoidance
  }

  static boolean printOdometer;
  static boolean printNavigator;
  static boolean printUsSensor;
  static boolean printAvoidance;

  public static void log(Sender sender, String message) {
    long timestamp = System.currentTimeMillis() % 100000;

    if (sender == Sender.Navigator && printNavigator) {
      writer.println("NAV::" + timestamp + ": " + message);
    }
    if (sender == Sender.odometer && printOdometer) {
      writer.println("ODO::" + timestamp + ": " + message);
    }
    if (sender == Sender.usSensor && printUsSensor) {
      writer.println("US::" + timestamp + ": " + message);
    }
    if (sender == Sender.avoidance && printAvoidance){
      writer.println("OA::" + timestamp + ": " + message);
    }

  }

  public static void setLogging(boolean nav, boolean odom, boolean us,boolean avoid) {
    printNavigator = nav;
    printOdometer = odom;
    printUsSensor = us;
    printAvoidance = avoid;
  }

  public static void setLogWriter(String filename) throws FileNotFoundException {
    writer = new PrintStream(new File(filename));
  }
}
```

In Navigation use the Log class intead of System.out:

```
public void run(){
  State state = State.INIT;
  while(true){
    String message = "message";
    Log.log(Log.Sender.Navigator, message);
    //etc
```

In main choose your logging settings:

```
public static void main(String[] args) throws
    InterruptedException {
  //choose which threads print to the log
  Log.setLogging(false,true,false,true);
  //Want to write to a file instead of System.out
  Log.setLogWriter(System.currentTimeMillis() + ".log");
  //etc
```