

Rock 'em, sock 'em Robocode: Round 2

Go beyond the basics with advanced robot building and team play

Level: Introductory

Sing Li (westmakaha@yahoo.com), Author, Wrox Press

01 May 2002

Get ready to venture further into the realm of Robocode with this comprehensive look at advanced robot construction and team play. Veteran Java developer and newly converted Robocode fanatic Sing Li capitalizes on Robocode's unique -- and wildly fun -- approach to learning to walk you through more advanced Java programming techniques, algorithm design, basic trigonometry, and even distributed computing principles. Your opponents won't know what hit them.

Back in January, we presented a [behind-the-scenes glimpse](#) at how the Robocode simulator works. The robots we created were very simple and did not take advantage of the advanced features built into Robocode. In this article, we complete our introduction to Robocode by working with these advanced features. Along the way, we'll explore Java programming, math, and software design. To make things fun -- even for more advanced robot builders -- we'll look at the new "team play" feature in Robocode and survey some expert designers to learn about the super robots they have built.

Going beyond simple robots: Java class inheritance

The robots we created in the first article are all inherited from the `Robot` class. We can see that by the statement:

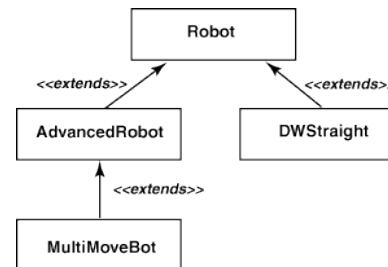
```
public class DWStraight extends Robot {
```

`DWStraight` extends `Robot` class, meaning we get to use all the methods provided by the class, such as `turnRight()` and `turnLeft()`. One very restrictive thing about these methods is that they do not return control to our code until they finish operation. In fact, these operations can take many turns to complete. By calling these methods, we are essentially passing up on the ability to make decisions on every turn. Fortunately, there is a class, called `AdvancedRobot`, that gives this control back to us. To use all the methods provided by `AdvancedRobot`, all we need to do is make our robot a subclass of it. For example, if we want to create a robot called `MultiMoveBot` that inherits from `AdvancedRobot`, we'd use the following code:

```
public class MultiMoveBot extends AdvancedRobot {
```

It is important to note that `AdvancedRobot` is actually a subclass of `Robot`, and our own `MultiMoveBot` is in turn a subclass of `AdvancedRobot`, as illustrated in the class hierarchy shown in Figure 1.

Figure 1. Inheritance relationships



Remember, inheriting from a class -- or becoming a subclass of it -- means we get to use all its `public` methods. As a result, we'll be able to use the methods from both the `AdvancedRobot` class and the `Robot` class in our robot.

Next we'll explore the new capabilities we inherited from the `AdvancedRobot` class.

Note: This article has many associated code files. I recommend that you [download the code](#) now. The article itself covers a lot of ground, so be sure to take time to analyze the code as you read along for maximum benefit, as the code is supplementary to the discussion.

Performing multiple simultaneous actions: Non-blocking method calls

Because `MultiMoveBot` is a subclass of `AdvancedRobot`, we have the ability to change the robot's action on every single turn. A *turn* in Robocode is called a *tick* (as in a clock tick), and relates to a graphical *frame* that is displayed on the battlefield. When we watch battles on Robocode, the simulation engine is actually displaying many static graphical frames in sequence, similar to how movies or television work. During each of these frames, our code can get control back and decide what to do with the robot. Because these days most PCs are extremely fast, we can execute a lot of code (or make very intelligent decisions) during every single frame or tick.

How do we control what goes on with the robot during a tick? First, realize that almost all of the methods we have encountered with the `Robot` class are blocking method calls: they do not return control to us until they complete their operations. Now, because `MultiMoveBot` is a subclass of `AdvancedRobot`, we can use a new set of methods that are non-blocking: they return control immediately to us. Table 1 below shows some of the blocking calls that we are used to and their new non-blocking equivalents:

Table 1. Blocking versus non-blocking methods

Blocking methods inherited from <code>Robot</code>	Non-blocking methods inherited from <code>AdvancedRobot</code>
<code>turnRight()</code>	<code>setTurnRight()</code>
<code>turnLeft()</code>	<code>setTurnLeft()</code>
<code>turnGunRight()</code>	<code>setTurnGunRight()</code>
<code>turnGunLeft()</code>	<code>setTurnGunLeft()</code>
<code>turnRadarRight()</code>	<code>setTurnRadarRight()</code>

turnRadarLeft()	setTurnRadarLeft()
ahead()	setAhead()
back()	setback()

Notice the pattern here: all of the new non-blocking calls begin with *set*. Using these calls, we can tell our robot to do multiple things at once. For example, take a look at the MultiMoveBot.java file in the [source code](#) distribution. Our analysis of this source file begins with Listing 1:

Listing 1. Working with non-blocking method calls

```
public class MultiMoveBot extends AdvancedRobot
{
    ....
    public void run() {
        ....
        setTurnRight(fullTurn);
        setAhead(veryFar);
        setTurnGunLeft(fullTurn);
    }
}
```

This code instructs MultiMoveBot to turn right, move ahead, then turn the gun left -- all at the same time during the next turn.

Note that all of the above methods return control to your program immediately without turning or moving the robot and the gun. Nothing will happen until you give control back to Robocode. You can give control back by making a blocking call (that is, any in the left column of the table above), or by using the special `execute()` method.

The `execute()` method gives control back to the Robocode engine for exactly one tick. Like a chess game, you are indicating that you have decided on your move for this turn, and Robocode will now move your piece for you.

Our MultiMoveBot robot actually uses another technique, shown in Listing 2, to give control back to Robocode:

Listing 2. Giving control back to Robocode with a blocking method call

```
while(true) {
    waitFor(new TurnCompleteCondition(this));
    toggleDirection();
}
```

The `waitFor()` method is a blocking call. It will block until the condition specified is satisfied. In this case, we are telling Robocode to return control to us after it has completed the vehicle turn. Of course, since we have set up the robot to move ahead and turn at the same time, it will actually be moving in a curved motion.

The `toggleDirection()` method is actually our own method. It reverses the robot's direction (and gun direction) each

Movement per clock tick

How far does the robot, radar, or guns

time it is called, as shown in Listing 3:

Listing 3. Reversing the direction of curved movement and gun turn

```
private void toggleDirection() {
    if (clockwise) {
        setTurnLeft(fullTurn);
        setback(veryFar);
        setTurnGunRight(fullTurn);
    } else {
        setTurnRight(fullTurn);
        setAhead(veryFar);
        setTurnGunLeft(fullTurn);
    }
    clockwise = ! clockwise;
}
```

There are a couple of other methods we haven't covered; I encourage you to further review the file at your leisure. To try out MultiMoveBot, select **Battle->Open** from the Robocode battlefield window and load the battle named showMultiMove.battle. Observe the interesting movement pattern that this robot makes.

How far does the robot, radar, or guns turn per tick? It turns out that this question is not quite so simple to answer. The gun turns 20 degrees and the radar turns 45 degrees per tick. The turn rate of the robot depends on its velocity. But because the gun is mounted on the robot, and the radar is mounted on the gun, their turn rates do affect one another. If you need to figure out the precise math, select the menu option **Help->Robocode FAQ** for a more detailed answer.

Custom events: Method override and anonymous classes

We've seen code that handles events even with simple robots like DWStraight. For example, when a bullet hits your robot, the code in Listing 4 may be executed.

Listing 4. Event handling code

```
public void onHitByBullet(HitByBulletEvent e) {
    turnLeft(90 - e.getBearing());
}
```

To find out all the events that you as a robot creator can handle, select **Help->Robocode API**. Just the Robot class API allows you to handle the following events:

- onBulletHit()
- onBulletHitBullet()
- onBulletMissed()
- onDeath()
- onHitByBullet()
- onHitRobot()
- onHitWall()
- onRobotDeath()
- onScannedRobot()
- onWin()

We handle an event by providing an event handler method. In fact, the class that we inherit from already contains

these methods. However, if we create our own method, that one will be used instead of the one provided by our superclass. This is sometimes called a virtual method override, or just *override*.

The added flexibility provided by the `AdvancedRobot` class includes the ability to create our own *custom* event. A custom event is one in which we can define the condition when Robocode will call us. For example, take a look at `CustomEventBot` shown in Listing 5:

Listing 5. Creating custom events in `CustomEventBot`

```
public class CustomEventBot extends AdvancedRobot
{
    ...
    public void run() {
        ...

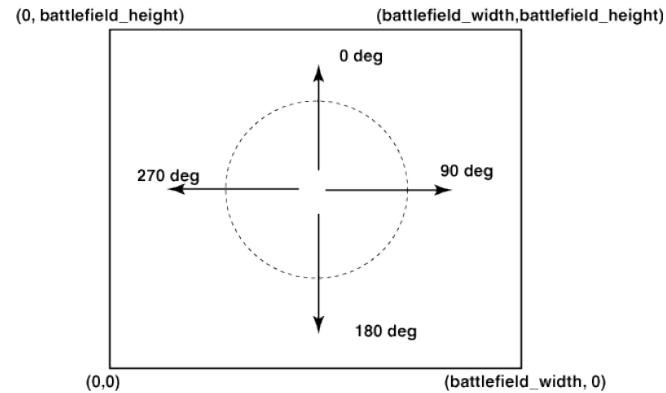
        addCustomEvent(
            new Condition("LeftLimit") {
                public boolean test() {
                    return (getHeading() <= quarterTurn);
                }
            });

        addCustomEvent(
            new Condition("RightLimit") {
                public boolean test() {
                    return (getHeading() >= threeQuarterTurn);
                }
            });
    }
}
```

This code shows how to create two new custom events using a mechanism known as an *anonymous* class. Inside the `addCustomEvent()` method, we are defining a new subclass of the `Condition` class and overriding its `test()` method. This new subclass is anonymous, meaning it has no name. The argument, such as `"LeftLimit"`, is passed as an argument to the constructor method of the `Condition` class.

Figure 2 shows the coordinates and direction conventions used by Robocode. The `"LeftLimit"` custom event that we have defined will be triggered whenever our robot has a heading of 90 degrees or less. The `"RightLimit"` condition will be triggered whenever the heading is 270 degrees or higher.

Figure 2. Robocode coordinates system



To handle these custom events, the `AdvancedRobot` class has an `onCustomEvent()` method that we can override. In our case, we switch the turn direction of the vehicle and gun, as shown in Listing 6:

Listing 6. Handling custom events with `onCustomEventBot()`

```
public void onCustomEvent(CustomEvent ev) {
    Condition cd = ev.getCondition();
    System.out.println("event with " + cd.getName());
    if (cd.getName().equals("RightLimit")) {
        setTurnLeft(fullTurn);
        setTurnGunRight(fullTurn);
    }
    else {
        setTurnRight(fullTurn);
        setTurnGunLeft(fullTurn);
    }
}
```

Together, our custom event definition and handler restrict the turn of the robot between 90 and 270 degrees, sending the robot into a twisting motion. Try this out by loading the `twistergalore.battle` file, and start it. Select **Options->Preferences->Visible Scan Arcs** for a more dramatic presentation of the "twist."

Target location: Interfaces and inner classes

The three key new features provided by `AdvancedRobot` are the ability to:

- Carry out multiple movements simultaneously
- Decide on the robot's action or strategy at every clock tick
- Define and handle custom events

While these features certainly provide a lot of capabilities for advanced robot builders, they do not tell us how to go about finding out where a target is (target location) and how to get close to it. This is a very basic question, but one that needs to be resolved if we want to create effective robots. There are at least two ways to approach this problem. We will first take a look at a simple way.

Take a look at the `DuckSeekerBot` code shown below in Listing 7. This simple bot will "target locate" sitting ducks (the most docile robot in the Robocode samples collection) on the battlefield, move in, and roast them.

Listing 7. DuckSeekerBot class definition

```
public class DuckSeekerBot extends AdvancedRobot implements DuckConstants
{
    boolean targetLocked = false;
    Target curTarget = null;
```

Not surprisingly, this robot subclasses `AdvancedRobot` for extra flexibility. It also implements the interface `DuckConstants` and shares those constants.

If you take a look at `DuckConstants.java`, you will find the interface definition. In this case, it contains all the constants that we have been using all along (`halfTurn`, `fullTurn`, and so on.). Therefore, implementing the `DuckConstants` interface is a shorthand way to include all the constants into our class. In the code fragment above, notice that we created a variable called `curTarget`, of the type `Target`. But `Target` isn't part of the Robocode library, so what is it?

If you carefully examine the latter part of `DuckSeekerBot.java`, shown below in Listing 8, it will provide a clue. We see the definition of a completely new class inside the `DuckSeekerBot` class. This is an inner class definition, or more specifically a "member class."

Listing 8. The Target member class

```
class Target {
    ...
    public Target(String inname, boolean inalive, boolean inlocked) {
        ...
    }
    public boolean isAlive() {
        ...
    }
    public boolean isLocked() {
        ...
    }
} // of Target
```

By declaring the class `Target` inside `DuckSeekerBot` as a member class, we have made it available for use inside `DuckSeekerBot`. As you examine the code, you will see that `curTarget` is used to hold the current target that is detected with the `onScannedRobot()` method. We use a very simplistic means of homing in on the target: we turn toward it, move in, then fire. Listing 9 shows the code inside the `onScannedRobot()` method that we need to roast our prey.

Listing 9. Homing in on our target

```
stop();
    turnRight(evt.getBearing());
    if (evt.getDistance() > safeDistance)
        ahead(evt.getDistance() - safeDistance);
```

The `DuckSeekerBot` alternates between scanning for new ducks and locking in on a target. To see the `DuckSeekerBot` in action, select **Battle->Open** and open the `duckoAducko.battle` file. This will pit our `DuckSeekerBot` against a flock of sitting ducks.

Getting the big picture on the battlefield: Vector, polymorphism, and java.Math

The `DuckSeekerBot`:

1. Scans for a duck target
2. Zooms in and roasts the target
3. Repeats until the entire flock is gone

An alternative approach to the same problem is this:

1. Scan for all the ducks that can be seen in the battlefield and build an "intelligence map"
2. Zoom in on the flock one at a time to eliminate them
3. Update the "map" constantly from scan information

This second approach achieves the same result as the first, but uses more intelligence. Most advanced robots use this sort of "big picture" information in formulating an instantaneous strategic decision. Learning how to maintain such a map will allow us to create robots with more sophisticated intelligence.

Our `FlockRoasterBot` uses the following helper custom classes to maintain the big picture:

- **Duck** (contained in `Duck.java`): Like the simpler `Target` class in `DuckSeekerBot`, this class maintains all the information we can gather on a target duck.
- **Flock** (contained in `Flock.java`): This class lists all of the ducks we have found; it is our "intelligence map." We actually subclass a standard Java library class, `java.util.Vector`, to create our flock. The `Vector` class provides us with all the list management (add a duck to the list, remove a duck from the list, and so on) capabilities.

If you take a look at the `FlockRoasterBot.java` source code, you'll see that all newly scanned robots are added in the `onScannedRobot()` event handler, as shown in Listing 10:

Listing 10. Adding scanned robots to our flock

```
public void onScannedRobot(ScannedRobotEvent evt) {
    Duck nuDuck = new Duck(evt, getTime(), getX(), getY(), getHeading(),
```

```

    getVelocity(), true, false);

    if (!curFlock.contains(nuDuck)) {
        curFlock.add(nuDuck);
    }
}

```

In Listing 10, we first create a new instance of `Duck`, storing all the information we gathered from the scan about the duck. Then, we make sure that we have not previously scanned this duck (using the `contains()` method provided by the `Flock/Vector` class) before we add it to the flock. If you examine `Duck.java` carefully, you'll see that we have defined how to compare one `Duck` instance against another by overriding the implementation of the `equals()` method. This step is essential in ensuring that the `contains()` method works properly.

It is interesting to note that `java.util.Vector`'s `contains()` method was originally written by the Java API library creators, none of whom had any way of knowing about our `Duck` class. Yet it works perfectly with our `Duck` class. This is an example of Java language's use of polymorphism: the programming logic within the `contains()` method works across any class(es), present or future, that implements its own `equals()` method properly.

Finally, we must approach the inevitable: mathematics (more specifically, trigonometry). If you examine the source code to `Duck.java`, you'll notice that in addition to all the methods from the old `Target` member class of the `DuckRoasterBot`, it also has two other interesting methods:

- `bearingToDuck`: Given a current x,y position, determine the bearing to the targeted duck.
- `distanceToDuck`: Given a current x,y position, determine the distance to the targeted duck.

These methods make the `Duck` class quite versatile, as they hide (or *encapsulate* in Java language parlance) the trigonometric mathematics used to determine these quantities. See the accompanying sidebar [Unit circle trigonometry](#) to refresh your memory on this arcane but incredibly useful topic. If you are keen and serious, you may also consider examining the detailed source code to the `bearingToDuck()` and `distanceToDuck()` methods. These methods make extensive use of the `java.Math` code library. The `java.Math` library contains enough sophisticated mathematical methods to make any mathematician proud.

`FlockRoasterBot` uses the `distanceToDuck()` method to determine the nearest duck to roast. In the `getNextRoaster()` method of the `Flock` class, shown in Listing 11, is the logic that selects the nearest duck:

Listing 11. getNextRoaster() method

```

public Duck getNextRoaster(double x,double y) {
    Iterator it = this.iterator();
    double curMin = 9999.0;
    while (it.hasNext()) {
        Duck tpDuck = (Duck) it.next();
        double tpDist = tpDuck.distanceToDuck(x,y);
        // check for non-alive ducks
        if ((tpDuck.isAlive() && (tpDist < curMin))) {
            curMin = tpDist;
            curDuck = tpDuck;
        } // of if
    } // of while
}

```

```

}

```

After we've determined the next duck to roast, the `bearingToDuck()` method can be used to home in on it.

To see `FlockRoasterBot` in action, load in the `RoastedDucks.battle` file. You can see how similar the action is to the "from-the-hip" style of the `DuckRoasterBot`. However, if you turn on the **Visible Scan Arcs** option (select **Options->Preferences->Visible Scan Arcs**), you will see that `FlockRoasterBot` does not rescan between each duck; it just "knows" who to cook next through its big picture of the battlefield.

Retrofitting FlockRoasterBot for team play

The latest release of Robocode supports a long-awaited feature: team play. In the team play mode, rather than designing just one robot, you design a whole team of different robots. You can use as many different classes as you want in a team or have many robot instances of the same class.

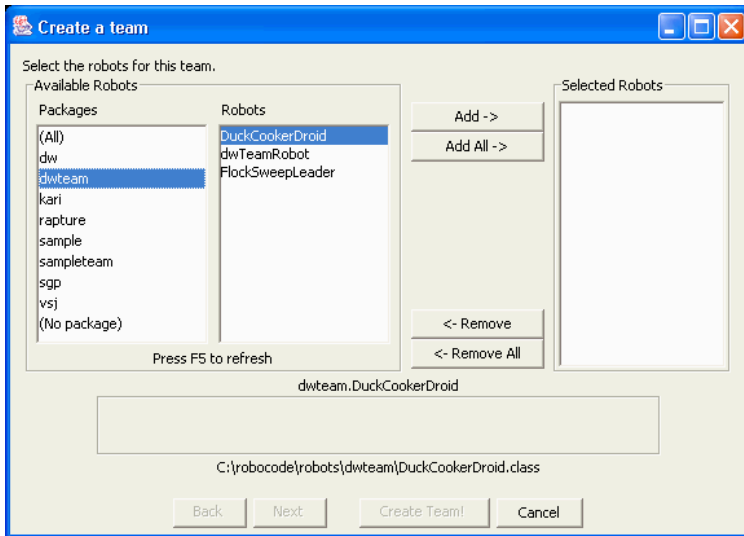
Here's a quick look at some of the options team play offers:

- You can designate a robot as a team leader (the first one you add to a team). Leaders get extra energy to start -- 200 units total. If the leader ever gets toasted, every team member loses 30 units.
- Team robots have the additional capability -- using the `TeamRobot` class, a subclass of `AdvancedRobot` -- to send messages to one another.

To create your own team robot, you should always subclass `TeamRobot`. In addition, you may also want to implement the `Droid` interface with your robots. A droid has no radar, and it cannot scan. However, all droids start off with an extra 20 units of life when compared to non-droid robots.

To create a team, select **Robot->Team->Create Team**. The Create Team dialog will appear as shown in Figure 3.

Figure 3. The Create Team dialog



We're going to revise our `FlockRoasterBot` to work in the new team play mode. Our team will consist of one intelligent leader and two droids. We need to distribute the "intelligence" contained in `FlockRoasterBot` among the team leader and the droids. To do this, we'll use the following two classes:

- **FlockSweepLeader**: Performs the scanning and maintains the intelligence map of the battlefield and tells other team droids to sweep up the ducks. It will also participate in the roasting.
- **DuckCookerDroid**: Monitors for "command to roast" from the `FlockSweepLeader`, then locks in, homes in on, and roasts the target duck. Finally, it reports back to `FlockSweepLeader` once the duck is bagged.

While it would be ideal if we could reuse `Duck.java` and `Flock.java`, which we created earlier with the `FlockRoasterBot`, we cannot. In team play, a duck in a flock may be alive, but not locked in by the `FlockSweepLeader` -- and is, thus, not the current target for the map maintainer class. That is, the duck in question may be delegated to a droid for roasting. This means that we must be able to determine whether a duck is claimed by one of our team members. The `TeamDuck` and `TeamFlock` classes add this additional state to the common duck. They subclass `Duck` and `Flock`, respectively, to reuse their built-in capabilities.

Intra-team messaging: Communications protocol design

Because sending messages between team members is the only way for a team to communicate, a winning strategy requires a well-defined communication protocol. All team members must agree on a convention to communicate with one another to avoid chaos.

In our team, we have one leader in the form of the `FlockSweepLeader` class, and any number of `DuckCookerDroid`. They communicate using the following protocols:

Cook-a-duck protocol

The Cook-a-duck protocol, shown in Table 2, is used by the leader to obtain a droid's position and to delegate a duck for the droid's roasting pleasure.

Table 2. Cook-a-duck protocol

From	To	Message	Description
Leader	Droids	Report Position ("reppos," see <code>TeamCommands.java</code>)	Broadcasts a request to all team members for x,y information from all droids.
Droid	Leader	x-y position (see <code>DroidPosition.java</code>)	Reports the x,y coordinates. The leader uses this information to find the nearest duck to the droid that is still alive in the flock.
Leader	Droid	Serialized Duck (see <code>TeamDuck.java</code>)	Tells the droid to target the specified duck.

To see the code that starts the Cook-a-duck protocol, take a look at the `assignMission()` method in `FlockSweepLeader.java`, shown in Listing 12:

Listing 12. assignMission() method

```
public void assignMission() {
    try {
        broadcastMessage(new String(REPORT_POSITION));
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

The `broadcastMessage()` method is part of the new capability offered by the `TeamRobot` class. `REPORT_POSITION` is a constant that is part of the `TeamCommand.java` interface. This interface contains the protocol constants for the messages we send.

On the droid side, take a look at the `onMessageReceived()` method. This is where a team robot will receive messages. Note that the message details are available as part of the `MessageEvent` instance passed in. The content of the message itself can be of any serializable Java object instance, and both the `REPORT_POSITION` and `FLOCK_GONE` messages send a `String` class as message, while the duck assignment message from the `FlockSweepLeader` has a serialized `TeamDuck` instance in the message. Listing 13 demonstrates the `instanceof` operator to determine the class (type) of object that is passed in the message:

Listing 13. How a droid receives messages

```
public void onMessageReceived(MessageEvent e)
{
    Object msg = e.getMessage();
    if (msg instanceof String) {
        if (((String) msg).equals(REPORT_POSITION))
            try {
                ourLeaderTheGreat = e.getSender();
                sendMessage(e.getSender(), new DroidPosition(getX(), getY()));
            }
            catch (IOException ex) {
            }
    }
}
```

```

        ex.printStackTrace();
    }
    if (((String) msg).equals(FLOCK_GONE))
        flockGone = true;
    } // if instanceof String

    if (msg instanceof TeamDuck)
    {
        onAssignment = true;
        curTarget = (TeamDuck) msg;
    }
}

```

Duck-bagged protocol

The Duck-bagged protocol, detailed in Table 3, is used by the droid to signal the leader whenever it bags a duck. The leader should update its intelligence map and check to see if it has another duck for the droid. If it does, the leader uses the Cook-a-duck protocol to tell the droid.

Table 3. Duck-bagged protocol

From	To	Message	Description
Droid	Leader	Serialized DuckBagged (see DuckBagged.java for information)	Notifies the leader that the assignment is completed and the duck is in the bag.

Flock-gone protocol

The Flock-gone protocol, detailed in Table 4, is used by the leader to tell all the remaining droids that the flock is gone.

Table 4. Flock-gone protocol

From	To	Message	Description
Leader	Droids	Flock Gone ("flockgone," see TeamCommands.java)	Broadcasts the message to the entire team.

Collision avoidance: Implementing random backoff

One problem that occurs frequently when you have a team on the battlefield is the inevitable collision with teammates. Our team avoids embarrassing repeated, energy-draining collisions by backing up after a collision. However, to avoid repeated maneuvers that may place two teammates in deadlocks, it selects one of three different backup moves randomly. The `pickRandAvoidance()` method, shown in Listing 14, implements this behavior:

Listing 14. Backup moves

```

protected void pickRandAvoidance() {
    double tpRnd = Math.random() * 10;
    int rndInt = (int) Math.ceil(tpRnd);
    tpRnd = tpRnd % 3;
    switch (rndInt) {
        case 0: back(100);
                break;
        case 1: back(10);
                turnRight(90);
    }
}

```

```

        ahead(50);
        break;
    case 2: back(10);
            turnLeft(90);
            ahead(50);
        }
    }
}

```

The `pickRandAvoidance()` method is called whenever we hit a robot on our way to a locked-in target. The `onHitRobot()` event handler, shown in Listing 15, calls this method:

Listing 15. onHitRobot() event handler

```

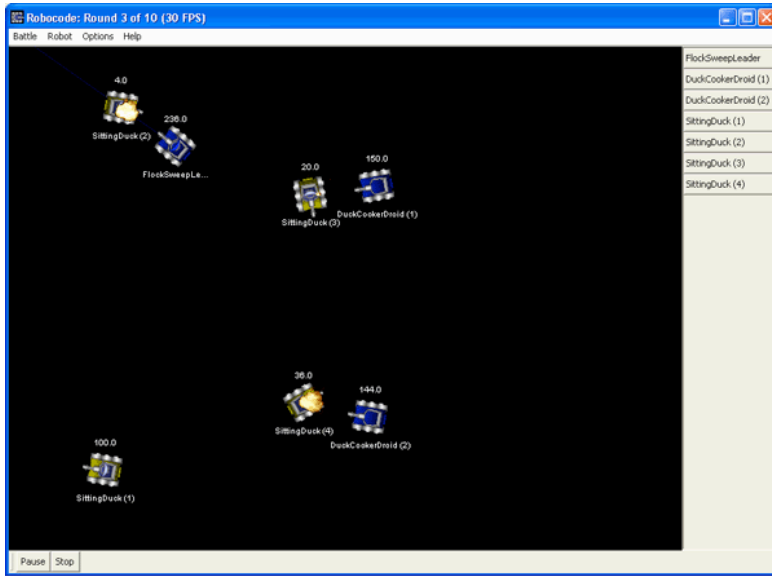
public void onHitRobot(HitRobotEvent e) {
    pickRandAvoidance();
    if (curTarget != null)
        curTarget.setLocked(false);
}

```

Because both `FlockSweepLeader` and `DuckCookerDroid` will perform duck removal, they both need the random back off and `onHitRobot()` implementations above. These common methods are extracted (*refactored* in Java programmer's lingo) into a common base class called `dwTeamRobot`. Both `FlockSweepLeader` and `DuckCookerDroid` are subclasses of `dwTeamRobot`. This type of refactoring is very common in Java programming as requirements or designs change.

To see our distributed "duck sweeping team" in action, select **Battle->Open** and load the `teamsweep.battle` file. This will pit our team consisting of one leader and two droids against a group of four sitting ducks. You will be able to observe the delegation of command after the initial scan by the leader. The collision-avoidance moves will also be quite visible. Figure 4 shows an ongoing distributed duck sweep team in action.

Figure 4. A distributed duck sweep team does battle



In the real Robocode world: A survey of top robots

While sweeping and roasting sitting ducks is certainly fun, even the most trivial robot rival doesn't typically stay still waiting to be wiped out. When dealing with moving robots, we must modify our strategies slightly. For example, we may anticipate where a targeted robot will be in the future before we head toward it or fire at it. This is called *predictive targeting*. When working with robots that will fire back, we must start thinking about detecting and dodging bullets. We may also want to base our strategy on the amount of energy remaining at any time.

These are all thoughts that must be considered when creating a great robot. Unfortunately, general robot strategy design is out of the scope of this article. However, you will find many excellent contemporary tips articles on creating great Robocode robots on the Robocode Rumble Web site (see [Resources](#)). We will spend the remaining portion of this article on a survey of some of the top robots that you may encounter today in the real Robocode world: out there on the Internet. The robot designers profiled in this article are at the pinnacle of their art and come from all over the world.

I would like to acknowledge and sincerely thank the creators of these robots for sharing their design ideas.

Nicator by Alisdair Owens (U.K.)

Nicator is an offensive robot that hits hard and dies young, according to Alisdair Owens, the creator of Nicator. It is a specialized robot for melee (group) situations. Its winning strategy is an original "anti-gravity" move that

places the robot at the most advantageous corner and maintains a holding pattern while aggressively attacking the opponents. Nicator scans and maintains a big picture of the battlefield. It also makes extensive use of the multiple non-blocking action of the `AdvancedRobot` class to make decisions on every tick, and uses custom classes extensively. (Alisdair shares his anti-gravity technique in "Secrets from the Robocode masters" -- see [Resources](#).)

Wolverine by Jae Marsh (U.S.)

Unlike Nicator, Wolverine is a specialized one-on-one robot. Wolverine is the brainchild of Jae Marsh, or graygoo in the Robocode community. Wolverine uses advanced pattern matching for targeting, and firing detection to dodge bullets. Wolverine's strategy is offensive in nature, and it will step up its offense once it determines that it has at least a 50-point advantage. Also unlike Nicator, Wolverine does not build an up-to-date big picture of the battlefield. Instead, it uses temporal snapshots of the battlefield to implement its pattern-matching algorithm. Wolverine makes use of the custom event feature of the `AdvancedRobot` class to consolidate its event-logging logic.

RayBot by Ray Vermette (Canada)

RayBot uses predictive targeting, and maintains an up-to-date big picture of the battlefield. Unlike Nicator and Wolverine, RayBot is basically a defensive robot. It will, however, switch personalities and become offensive once it determines that it definitely has the upper hand on a one-on-one scenario. The creator of RayBot, Ray Vermette -- or Raymundo among the Robocode community -- says that scanned data and battlefield intelligence (the big picture) is the key differentiator for this robot.

JollyNinja by Simon Parker (Australia)

Simon Parker is the creator of the JollyNinja robot, which is designed for both melee and one-on-one play. According to Simon, JollyNinja's most distinguishing feature is its movement strategy. It makes JollyNinja a mid-level offensive opponent that is very difficult to hit. At the core of JollyNinja is a strategic evaluation function that determines the robot's next move: it uses the `AdvancedRobot` class's non-blocking calls ability to make decisions at every clock tick. JollyNinja increases its offensive level once it determines that only one opponent is left in the battlefield. JollyNinja maintains a big picture and collects extensive battlefield information to execute its strategy. (Simon shares his strategy for tracking his opponents' movements in "Secrets from the Robocode masters" -- see [Resources](#).)

Tron and Shadow by ABC (Portugal)

The Robocode community is full of members with interesting handles and fascinating pseudonyms. ABC is the author of two excellent robots: Tron and Shadow. Tron is a territorial robot with a distinctive movement pattern, but is largely defensive in strategy. Shadow is an offensive opponent that is not territorial and uses movement to avoid being hit. The unique thing about Shadow is that it attempts to use the same strategy for both melee and one-on-one battles without switching modes. It uses a "weighted center orbiting" strategy in accomplishing this. Both robots use the multiple simultaneous movements of the `AdvancedRobot` class extensively. They also collect information for a big picture of the battlefield as input to the strategy implementation.


Learning never stops with Robocode

If there was ever any doubt on the ability of Robocode to serve as a great teaching tool for Java programming, algorithm design, basic trigonometry, or even distributed computing principles, this article should settle it. Robocode naturally challenges the beginning robot designer to go "the extra mile" to create winning, advanced

robots that reflect their mastery of the programming and algorithm design art. Far from being "just another game," Robocode delivers on its educational goals in the name of friendly competition. If only learning could always be this much fun!

Download

Name	Size	Download method
j-robocode2.zip		FTP

→ [Information about download methods](#)  [Get Adobe® Reader®](#)

Resources

- Download the complete [source code](#) for all the robots analyzed in this article.
- See the first article in this series for a basic [introduction to Robocode](#) (*developerWorks*, January 2002).
- [Download Robocode](#).
- Read all of the [Secrets from the Robocode masters](#). This page will be updated as new tips become available.
- Robocode's creator, Mathew Nelson, maintains the [official Robocode site](#). This should be the first stop for anyone serious about Robocode. During the creation of this article, Mat graciously provided the inside information on the architecture of the simulation engine. From the Robocode site, you can also participate in the [discussion group](#).
- For league play, Christian Schnell's [RoboLeague](#) is concurrently being developed with Robocode.
- An excellent alternative Robocode site, with a robot upload and download area, as well as lively discussion forum, can be found at [the Robocode Repository](#). The latest versions of the top robots that we have

mentioned in this article may be downloaded there.

- New to Java? Check out "[Introduction to Java programming](#)" (*developerWorks*, November 2004), a tutorial that steps you through the fundamentals of Java language programming.
 - A [Yahoo Robocode group](#) is available for sharing Robocode information if you are already a Yahoo member.
 - Be sure to visit [alphaWorks](#) for other interesting early-access technologies.
 - Find other Java-related resources on the [developerWorks Java technology zone](#).
-

About the author



Sing Li is the author of *Early Adopter JXTA* and *Professional Jini*, as well as numerous other books with Wrox Press. He is a regular contributor to technical magazines, and is an active evangelist of the P2P evolution. Sing is a consultant and freelance writer, and can be reached at westmakaha@yahoo.com.
