Rock 'em, sock 'em Robocode!

Learning Java programming is more fun than ever with this advanced robot battle simulation engine

Level: Introductory

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

01 Jan 2002

Is it possible to learn inheritance, polymorphism, event handling, and inner classes, all while dodging bullets and executing precision attack maneuvers? A surprisingly addictive teaching-tool-turned-game-craze called Robocode is about to make this a reality for Java developers worldwide. Follow along as Sing Li disarms Robocode and starts you on your way to building your own customized lean, mean, fighting machine.

Robocode is an easy-to-use robotics battle simulator that runs across all platforms supporting Java 2. You create a robot, put it onto a battlefield, and let it battle to the bitter end against opponent robots created by other developers. Robocode comes with a set of pre-fab opponents to get you started, but once you outgrow them, you can enter your creation against the world's best in one of the leagues being formed worldwide.

Each Robocode participant creates his or her own robot using elements of the Java language, enabling a range of developers -- from rank beginners to advanced hackers -- to participate in the fun. Beginning Java developers can learn the basics: calling API code, reading Javadocs, inheritance, inner classes, event handling, and the like. Advanced developers can tune their programming skill in a global challenge to build the best-of-breed software robot. In this article, we will introduce Robocode and start you on your way to conquering the world by building your very first Robocode robot. We will also take a peek at the fascinating "behind the scenes" machinery that makes Robocode tick.

Downloading and installing Robocode

Robocode is the brainchild of Mathew Nelson, a software engineer in the Advanced Technology, Internet division at IBM. First, head to the Robocode page. Here, you will find the latest executables of the Robocode system. Once you have downloaded the distribution, which is in a self-contained installation file, you can use the following command to get the package installed on your system (assuming you have a Java VM (JDK 1.3.x) pre-installed on your machine, of course):

```
java -jar robocode-setup.jar
```

During installation, Robocode will ask you if you'd like to use this external Java VM for robot compilations. The other alternative is the Jikes compiler that is supplied as part of the Robocode distribution.

After your installation, you can start the Robocode system from either the shell script (robocode.sh), batch file (robocode.bat), or icon on the desktop. At this point, the battlefield will appear. From here, you can invoke the Robot Editor and compiler using the menu.

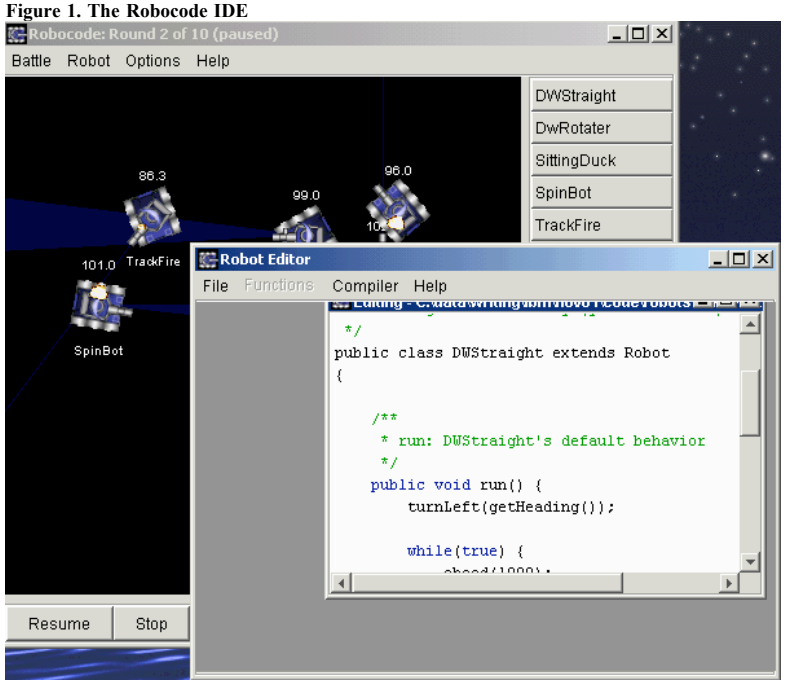
Components of the Robocode system

When you activate Robocode, you will see two interrelated GUI windows, which form Robocode's IDE:

- The battlefield
- The Robot Editor

Figure 1 shows the battlefield and the Robot Editor in action.

Figure 1. The Robocode IDE



The battlefield is where the battle between the robots plays itself out. It houses the main simulation engine and allows you to create, save, and open new or existing battles. You can pause and resume the battle, terminate the battle, destroy any individual robot, or get the statistics of any robot using the controls available in the arena. Furthermore, you can activate the Robot Editor from this screen.

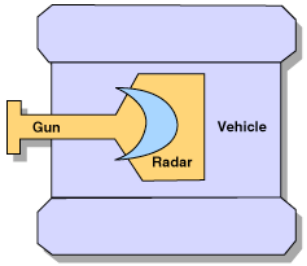
The Robot Editor is a customized text editor for editing the Java source files that make up a robot. It integrates both the Java compiler (for compiling robot code) and the customized Robot packager in its menu. Any robot created with the Robot Editor and successfully compiled is in a ready-to-deploy location for the battlefield.

A robot in Robocode consists of one or more Java classes. These classes can be archived into a JAR package. The latest version of Robocode provides a "Robot Packager" that can be activated from the battlefield GUI window, for just this purpose.

The anatomy of a Robocode robot

At the time of this writing, a Robocode robot is a graphical tank. Figure 2 illustrates a typical Robocode robot.

Figure 2. Anatomy of a Robocode robot



Note that the robot has a rotating gun, and on top of the gun is a rotating radar. The robot vehicle, the gun, and the radar can all rotate independently: at any moment in time, the robot's vehicle, the gun, and radar can be turned in different directions. By default, these items are aligned, facing the direction of the vehicle movement.

Robot commands

The set of commands for a Robocode robot are all documented in the Javadoc of the Robocode API. You will find them as public methods of the `robocode.Robot` class. In this section, we'll cover each of the available commands, by category.

Moving the robot, gun, and radar

Let's begin with the basic commands to move the robot and its accoutrements:

- `turnRight(double degree)` and `turnLeft(double degree)` turn the robot by a specified degree.
- `ahead(double distance)` and `back(double distance)` move the robot by the specified pixel distance; these two methods are completed if the robot hits a wall or another robot.
- `turnGunRight(double degree)` and `turnGunLeft(double degree)` turn the gun, independent of the vehicle's direction.

- `turnRadarRight(double degree)` and `turnRadarLeft(double degree)` turn the radar on top of the gun, independent of the gun's direction (and the vehicle's direction).

None of these commands will return control to the program until they are completed. Furthermore, when the vehicle is turned, the direction of the gun (and radar) will also move, unless indicate differently by calling the following methods:

- `setAdjustGunForRobotTurn(boolean flag)`: If the flag is set to true, the gun will remain in the same direction while the vehicle turns.
- `setAdjustRadarForRobotTurn(boolean flag)`: If the flag is set to true, the radar will remain in the same direction while the vehicle (and the gun) turns.
- `setAdjustRadarForGunTurn(boolean flag)`: If the flag is set to true, the radar will remain in the same direction while the gun turns. It will also act as if `setAdjustRadarForRobotTurn(true)` has been called.

Obtaining information about the robot

Many methods exist for getting information about the robot. Here is a short list of frequently used method calls:

- `getX()` and `getY()` get the current coordinate of the robot.
- `getHeading()`, `getGunHeading()`, and `getRadarHeading()` get the current heading of the vehicle, gun, or radar in degrees.
- `getBattleFieldWidth()` and `getBattleFieldHeight()` get the dimension of the battlefield for the current round.

Firing commands

Once you have mastered how to move the robot and its associated weaponry, it's a good time to consider the tasks of firing and controlling damage. Each robot starts out with a default "energy level," and is considered destroyed when its energy level falls to zero. When firing, the robot can use up to three units of energy. The more energy supplied to the bullet, the more damage it will inflict on the target robot. `fire(double power)` and `fireBullet(double power)` are used to fire a bullet with the specified energy (fire power). The `fireBullet()` version of the call returns a reference to a `robocode.Bullet` object that can be used in advanced robots.

Events

Whenever the robot moves or turns, the radar is always active, and if it detects any robots within its range, an event is triggered. As the robot creator, you can choose to handle various events that can occur during the battle. The basic `Robot` class has default handlers for all of these events. However, you can override any of these "do nothing" default handlers and implement your own custom actions. Here are some of the more frequently used events:

- `ScannedRobotEvent`. Handle the `ScannedRobotEvent` by overriding the `onScannedRobot()` method; this method is called when the radar detects a robot.
- `HitByBulletEvent`. Handle the `HitByBulletEvent` by overriding the `onHitByBullet()` method; this method is called when the robot is hit by a bullet.

- `HitRobotEvent`. Handle the `HitRobotEvent` by overriding the `onHitRobot()` method; this method is called when your robot hits another robot.
- `HitWallEvent`. Handle the `HitWallEvent` by overriding the `onHitWall()` method; this method is called when your robot hits a wall.

That's all we need to know to create some pretty complex robots. You can find the rest of the Robocode API in the Javadoc, which can be accessed from either the battlefield's help menu or the Robot Editor's help menu.

Now it's time to put our knowledge to use.

Creating a robot

To create a new robot, start the Robot Editor and select **File->New->Robot**. You will be prompted for the name of the robot, which will become the Java class name. Enter **DWStraight** at this prompt. Next, you will be prompted for a unique initial, which will be used for the name of the package that the robot (and potentially its associated Java file) will reside in. Enter **dw** at this prompt.

The Robot Editor will display the Java code that you need to write to control the robot. Listing 1 is an example of the code that you will see:

Listing 1. Robocode-generated Robot code

```
package dw;
import robocode.*;

/**
 * DWStraight - a robot by (developerWorks)
 */
public class DWStraight extends Robot
{
        ...  // <<Area 1>>
        /**
         * run: DWStraight's default behavior
         */
        public void run() {
                ... // <<Area 2>>
                while(true) {
                ... // <<Area 3>>
                }
        }
      ... // <<Area 4>>
        public void onScannedRobot(ScannedRobotEvent e) {
                fire(1);
        }
}
```

The highlighted areas are those places where we can add code to control the robot:

Area 1
In this space we can declare class scope variables and set their value. They will be available within the robot's `run()` method, as well as any other helper methods that you may create.

Area 2
The `run()` method is called by the battle manager to start the robot's life. It typically consists of two areas (designated Area 2 and Area 3 in Listing 1) where you can add code. Area 2 is where you will place code that will run only once per robot instance. It is often used to get the robot into a pre-determined state before starting repetitive action.

Area 3
This is the second part of a typical `run()` method implementation. Here, within an endless `while` loop, we'll program the repetitive action that a robot may be involved in.

Area 4
This is the area where you add helper methods for the robot to use within its `run()` logic. It's also where you add any event handlers that you wish to override. For example, the code in Listing 1 handles the `ScannedRobot` event and simply fires directly at the robot whenever one is detected by the radar.

For our first robot, DWStraight, we'll update the code as shown (in red) in Listing 2.

Listing 2. DWStraight robot code additions

```
package dw;
import robocode.*;

public class DWStraight extends Robot
{
        public void run() {
                turnLeft(getHeading());
                while(true) {
                        ahead(1000);
                        turnRight(90);

                }
        }

        public void onScannedRobot(ScannedRobotEvent e) {
                fire(1);
        }
        public void onHitByBullet(HitByBulletEvent e) {
                turnLeft(180);
        }

}
```

Here's what this first robot will do, area by area:

Area 1
We don't specify any class scope variables in this robot.

Area 2
To get the robot into a known state, we turn it so that it faces 0 degrees using `turnLeft(getHeading())`.

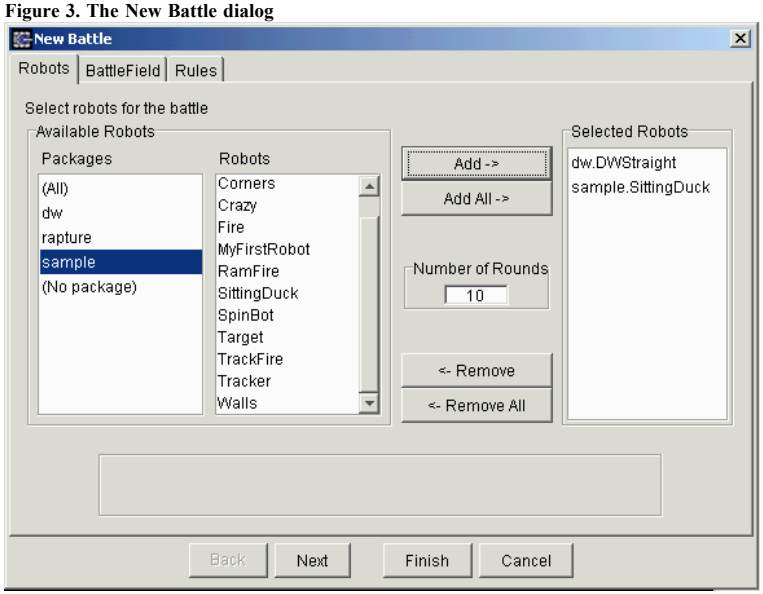
Area 3
In this repetitive section, we move the robot forward as far as it will go using `ahead(1000)`. It will stop when it hits a wall or robot. Then we turn right using `turnRight(90)`. As this is repeated, the robot will basically trace out the walls in a clockwise direction.

Area 4
Here, in addition to handling the auto-generated `ScannedRobot` event and firing at the robot that is found directly, we also detect the `HitByBullet` event and turn 180 degrees (going clockwise and counterclockwise, alternately) when we get hit.

Compiling and testing the robot

From the Robot Editor menu, select **Compiler->Compile** to compile your robot code. We are now ready to try our first battle. Switch back to the battlefield and select menu **Battle->New** to display a dialog similar to the one in Figure 3.

Figure 3. The New Battle dialog



Add our robot, dw.DWStraight to the battle, then add an opponent robot, such as sample.SittingDuck. Click **Finish**, and the battle will begin. Admittedly, doing battle with SittingDuck is not too exciting, but you get to see what the DWStraight robot does by default. Experiment with other robots in the sample collection, and see how DWStraight fares against them.

When you're ready to examine the coding of another robot, check out the dw.DWRotater robot code that is supplied with the code distribution in Resources. This robot will, by default:

- Move to the center of the battlefield
- Keep spinning its gun until it detects a robot
- Fire slightly ahead of the detected robot, trying different angles each time
- Move rapidly back and forth whenever it is hit by another robot.

The code is straightforward and we will not analyze it here, but I encourage you to try it out. The sample package included with Robocode provides code for many other robots, as well.

Additional robot support classes

As you become more competent in robot design, the body of code that you can include with the robot can increase substantially. A modular way to handle the code is to decompose it into separate Java classes and then bundle them into a single package (JAR file), using the packager, to include as part of your robot distribution. Robocode will automatically find robot classes within packages placed in its robots directory.

Other Robot subclasses

Anyone can create subclasses of `Robot` and add new functionalities that can be used to build robots. Robocode supplies a subclass of `Robot`, called `AdvancedRobot`, which enables asynchronous API calls. A description of the `AdvancedRobot` class is beyond the scope of this article, but I encourage you to experiment with this advanced class when you are comfortable with the operation of the basic `Robot` class.

The motivation behind Robocode design

I caught up with Mathew Nelson, Robocode's creator, and asked him about his original motivation for creating Robocode. Here's what Mat had to share: "Part of the motivation for writing Robocode was to prove to the world that the statements like 'Java is slow' and 'You can't write games in Java' are no longer true. I think I did it."

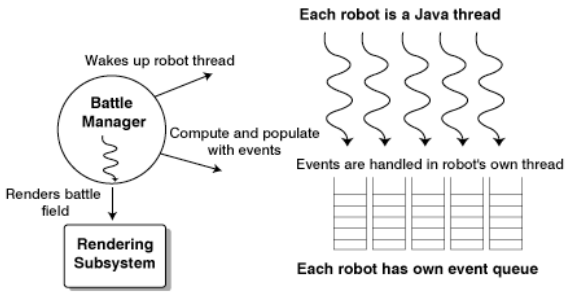
The architecture of a battle simulator

A look behind the scenes at Robocode reveals a sophisticated simulation engine that is both high performance (in order to render the battle at realistic speed) and flexible (enabling the creation of complex robotics logic without getting in the way). A special thanks to Robocode creator Mathew Nelson for graciously providing the inside information on the architecture of the simulation engine.

A design that leverages the Java platform

This simulation engine, shown in Figure 4, leverages the non-preemptive threading that most modern Java VMs offer, and couples it with the rendering capabilities provided by the JDK GUI and 2D graphics libraries.

Figure 4. Robocode simulation engine architecture



Notice that each robot being simulated is on its own Java thread, leveraging the VM's native thread mapping wherever applicable. A battle manager thread is the controller of the system: it orchestrates the simulation and drives the graphical rendering subsystem. The graphical rendering subsystem itself is based on Java 2D and AWT.

Loose thread coupling

To alleviate potential problems with shared resources (and thus potentially deadlocking or choking the simulation engine), a very loose coupling is required between the battle manager thread and the robot threads. To implement this loose coupling, each robot thread is given its own event queue. The events for each robot are then fetched and processed in the robot's very own thread. This per-thread queuing effectively eliminates any potential contention between battle manager thread and robot thread, or between robot threads themselves.

Robocode internals

You can view the Robocode simulator engine as a simulator program that takes a set of plug-ins (custom robots) during run time; this set of plug-ins can make use of the API supplied (the `robocode.Robot` class's methods). Physically, each robot is an independent Java thread, and the `run()` method contains the logic that will be executed on the thread.

At any time, a robot thread can call an API supplied by its parent, the `robocoode.Robot` class. This will typically block the robot thread via an `Object.wait()` call.

The battle manager thread

A battle manager thread manages the robots, bullets, and rendering on the battlefield. The simulation "clock" is marked by the number of frames rendered on the battlefield. The actual frame rate is adjustable by the user.

In a typical turn, the battle manager thread wakes up each robot thread, and then waits for the robot to complete its turn (that is, calling a blocking API again). This wait interval is typically tens of milliseconds, and even the most complex robot tends to use only 1 or 2 milliseconds for strategy and computation with today's typical system speed.

Here is the pseudo-code for the logic that the battle manager thread performs:

Listing 3. Pseudo-code logic for battle manager

```
while (round is not over) do
   call the rendering subsystem to draw robots, bullets, explosions
   for  each robot do
       wake up the robot
       wait for it to make a blocking call, up to a max time interval
   end for
   clear all robot event queue
   move bullets, and generate event into robots' event queue if applicable
   move robots, and generate event into robots' event queue if applicable
   do battle housekeeping and generate event into robots' event queue
        if applicable
   delay for frame rate if necessary
end do
```


Note that in the inside for loop, the battle manager thread will not wait beyond the maximum time interval. It will go on with the battle if the robot thread does not call a blocking API in time (typically due to some application logic error or endless loop). A `SkippedTurnEvent` is generated into a robot's event queue to notify advanced robots.

Replaceable rendering subsystem

The rendering subsystem in the current implementation is simply an AWT and Java 2D thread that takes commands from the battle manager and renders the battlefield. It is adequately decoupled from the rest of the system. It is foreseeable that it can be replaced in a future revision (with, for example, a 3-D renderer). In the current implementation, rendering is disabled whenever the Robocode application is minimized, allowing the simulation to proceed at a faster rate.

The future of Robocode

Mathew Nelson is in a tight feedback loop with the Robocode user community via a discussion group hosted at the alphaWorks Robocode site (see Resources). Much of the feedback is incorporated into the actual code. Some upcoming enhancements Mathew has planned are:

- Custom battlefield map with different object and obstacles
- Team-based battles
- Integrated support for tournaments or leagues
- User-selectable style of tank body/gun/radar/weapon

The unstoppable Robocode momentum

For a project that debuted as recently as July 12, 2001, Robocode's climb to fame is nothing short of phenomenal. While the latest version available has yet to hit 1.0 (at the time of writing it is version 0.98.2), it is already becoming a very popular pastime on university campuses and corporate computers throughout the world. Robocode leagues (or *roboleagues*), in which people pit their custom creations against each other over the Internet, are springing up fast. University professors are tapping Robocode's educational properties and have incorporated it into their computer science curriculum. Robocode user groups, discussion list, FAQs, tutorials, and Webrings can be found throughout the Internet.

Evidently, Robocode has filled a void in the popular gaming and educational arena -- supplying a simple, fun,

non-intimidating, yet competitive way for students and midnight engineers to unleash their creative energy and potentially fulfill their fantasy to conquer the world.

Download

Name	Size	Download method
j-robocode.zip		**FTP**

→ Information about download methods Get Adobe® Reader®

Resources

- Download the source code for the two simple robots featured in this article.
- Robocode's creator, Mathew Nelson, maintains the official Robocode site. This should be the first stop for anyone serious about Robocode. From here, you can also participate in the discussion group.
- "Rock 'em, sock 'em Robocode: Round 2" (*developerWorks*, May 2002) ventures into advanced robot construction and team play.
- For league play, RoboLeague is concurrently being developed with Robocode.
- A Yahoo Robocode group is available for sharing Robocode information if you are already a Yahoo member.
- Check out the other interesting early-access technologies for Java developers at IBM alphaWorks.
- Find other Java resources from 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.