闘え、Robocode (ロボコード) !

高機能のロボット格闘シミュレーション・エンジンで、Javaプログラミング学習がずっと楽しくなる

Sing Li
Author
Makawave

2002年 1月 01日

弾丸から身をかわしたり、高精度攻撃訓練を行ったりしながら、継承、多相性 (ポリモフィズム)、イベント処理、内部クラスなどを学習することは果たして可能でしょうか。Robocodeという名の学習ツール兼大流行のゲームのおかげで、世界中のJava開発者にとってそれが現実のことになっています。著者Sing Li氏がRobocodeの武装を解きほぐし、軽量で簡単な戦闘マシンを読者が独自にカスタマイズできるようにします。
その他の Robocode 関連記事はこちら

Robocodeは、Java 2をサポートするあらゆるプラットフォームで実行できる、使いやすいロボット戦闘シミュレーターです。ユーザーはロボットを作成し、それをバトルフィールドに配置して、他の開発者が作成した敵ロボットを相手に徹底的に闘わせます。Robocodeには初心者用にあらかじめ作成された敵がいくつか含まれていますが、これらの敵に勝てるようになったら、あなたが作ったロボットを世界のいずれかのリーグに参加させて、世界最強の相手と闘わせることができます。

Robocodeのそれぞれの参加者は、Java言語の部品を使って独自のロボットを作成します。こうして、まったくの素人からすご腕のハッカーに至るまで、さまざまな開発者が遊びに参加できるのです。初歩のJava開発者は、APIコードの呼び出し、Javadocの閲覧、継承、内部クラス、イベント処理などの基礎を学ぶことができます。上級の開発者は、世界を舞台にして「最高の血統」のソフトウェア・ロボットを構築する課題に取り組むことにより、プログラミング技術を磨くことができます。この記事ではRobocodeについて紹介し、初歩のRobocodeロボットを作成して、読者が世界制覇の道を踏み出すのを支援します。さらに、Robocodeを動かす背後にあるしくみについても、簡単に説明します。

ダウンロードとインストール

Robocodeは、IBMのAdvanced Technology (Internet部門) に属するソフトウェア・エンジニア MathewNelson氏によって発案されました。まずは、IBM alphaWorksのRobocode ページに移動し

てください ("Learn more"ボタンをクリックして移動したページにある "install Robocode."リンクをクリックしてください) 。ここには、Robocodeシステムの最新の実行可能ファイルがあります。配布版ファイル (自己完結型インストール・ファイル) をダウンロードしたら、以下のコマンド行を使用して、パッケージを読者のシステムにインストールします。当然ですが、Java VM (JDK 1.3.x) がマシンにあらかじめインストールされていることが前提です。

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

インストール中にRobocodeは、この外部Java VMをロボットのコンパイル用に使用するかどうかを問い合わせます。使用しない場合は、もう1つの選択肢として、Robocode配布版ファイルに付属しているJikesコンパイラーを使用できます。

インストールした後、シェル・スクリプト (robocode.sh)、バッチ・ファイル (robocode.bat)、またはデスクトップ・アイコンのいずれかからRobocodeシステムを開始することができます。開始した時点では、バトルフィールドが表示されます。ここから、メニューを使ってRobot Editorとコンパイラーを起動できます。

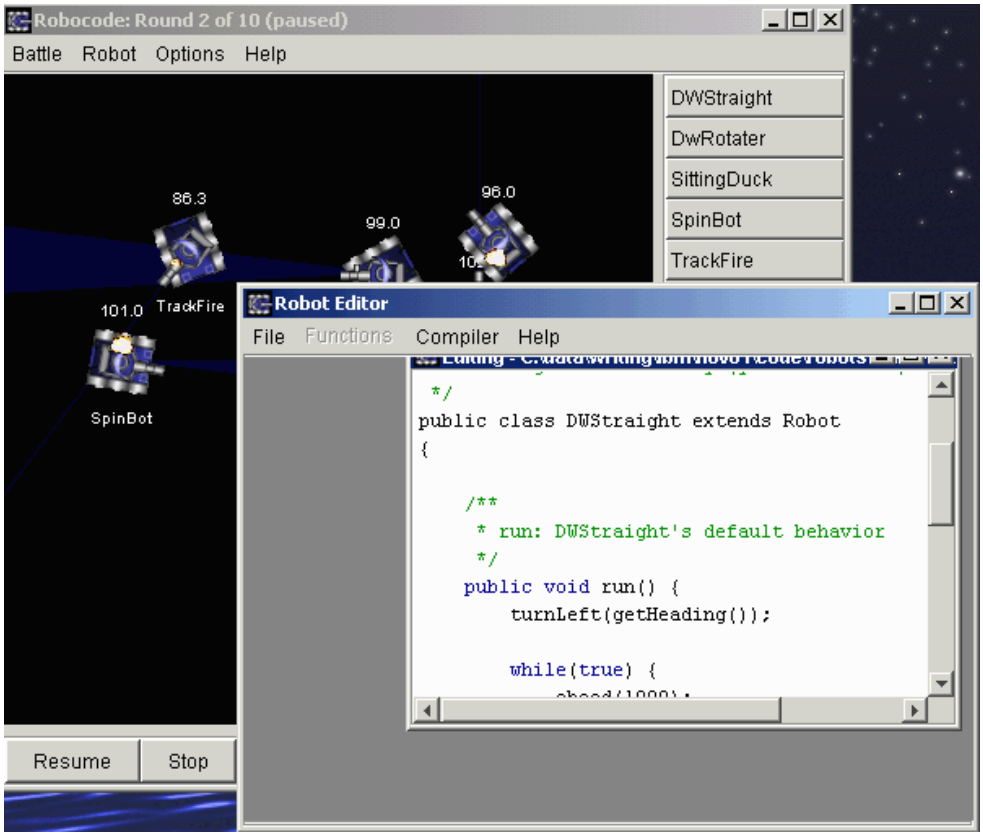
Robocodeシステムのコンポーネント

Robocodeを起動すると、互いに関連する以下の2つのGUIウィンドウが表示されます。この2つのウィンドウがRobocodeのIDE (統合開発環境) を形成します。

- バトルフィールド
- Robot Editor

図1は、実行中のバトルフィールドおよびRobot Editorを示しています。

図1. RobocodeのIDE



バトルフィールドでは、ロボット間の格闘が繰り広げられます。ここにはメイン・シミュレーション・エンジンがあり、新規バトルを作成して保管したり、新規または既存のバトルを開くことができます。競技場内のコントロールを使って、バトルの一時停止と再開、バトルの終了、任意のロボットの破棄、任意のロボットに関する統計の取得が可能です。さらに、この画面からRobot Editorを起動することもできます。

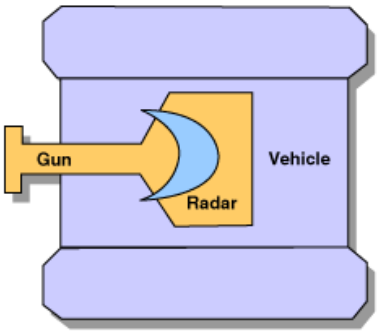
Robot Editorは、ロボットを構成するJavaソース・ファイルの編集用にカスタマイズされたテキスト・エディターです。(ロボット・コードをコンパイルするための) Javaコンパイラーと、カスタマイズされたRobotパッケージャーがメニューに組み込まれています。Robot Editorを使って作成され、正常にコンパイルされたすべてのロボットは、バトルフィールドへの配置準備が完了した状態に置かれます。

Robocodeのロボットは、1つまたは複数のJavaクラスから成ります。これらのクラスは、1つのJARパッケージにアーカイブできます。最新バージョンのRobocodeにはアーカイブ用の "Robot Packager" が付属しており、バトルフィールドGUIウィンドウから起動できます。

Robocodeのロボットを解剖する

これを書いている時点では、Robocodeロボットはグラフィック表現された戦車です。図2は典型的なRobocodeロボットを示しています。

図2. Robocodeロボットの構造



ロボットには回転する大砲があり、大砲の上には回転するレーダーがあります。ロボットの車両、大砲、レーダーはそれぞれ独立して回転できます。つまり、任意の時点でロボットの車両、大砲、レーダーを互いに別々の方向に回転させることができます。デフォルトでは、これらのアイテムは互いに連動し、車両の動きと同じ方向を向くようになっています。

Robotのコマンド

Robocodeロボット用の一連のコマンドは、Robocode APIのJavadocにすべて説明されています。これらのコマンドは、`robocode.Robot` クラスのpublicメソッドです。以下では、利用できるそれぞれのコマンドを種類別に説明します。

ロボット、大砲、レーダーを動かす

まず、ロボットとその装備を動かす以下のような基本的なコマンドがあります。

- `turnRight(double degree)` と`turnLeft(double degree)` は、指定された角度 (double degree) だけロボットを回転させます。
- `ahead(double distance)` と`back(double distance)` は、指定されたピクセル距離 (double distance) だけロボットを移動させます。この2つのメソッドは、ロボットが壁または別のロボットに当たった場合には完了します。
- `turnGunRight(double degree)` と`turnGunLeft(double degree)` は、車両の向きとは無関係に大砲 (Gun) を回転させます。
- `turnRadarRight(double degree)` と`turnRadarLeft(double degree)` は、大砲の上にあるレーダー (Radar) を回転させます。大砲の向き、および車両の向きとは無関係です。

これらのコマンドはすべて、完了するまでプログラムに制御を戻しません。加えて、車両を回転させるとき、大砲とレーダーの方向も一緒に変わります。ただし、以下のメソッドを呼び出すことによって別の指示を与えた場合は、そうではありません。

- `setAdjustGunForRobotTurn(boolean flag)`: フラグ (boolean flag) をtrueに設定すると、車両が回転している間、大砲の向きは変わりません。
- `setAdjustRadarForRobotTurn(boolean flag)`: フラグをtrueに設定すると、車両 (および大砲) が回転している間、レーダーの向きは変わりません。
- `setAdjustRadarForGunTurn(boolean flag)`: フラグをtrueに設定すると、大砲が回転している間、レーダーの向きは変わりません。さらに、このメソッドは`setAdjustRadarForRobotTurn(true)` が呼び出されたときと同じ動作をします。

ロボットに関する情報を得る

ロボットに関する情報を取得するためのメソッドは多数あります。以下に示すのは、よく使われるごく一部のメソッドです。

- `getX()` と`getY()` は、ロボットの現在の座標を取得します。
- `getHeading()`、`getGunHeading()`、および`getRadarHeading()` は、車両、大砲、レーダーの現在の向き (角度) をそれぞれ取得します。
- `getBattleFieldWidth()` と`getBattleFieldHeight()` は、現在のラウンドでのバトル・フィールドの大きさを取得します。

発射コマンド

ロボットとその装備を動かす方法がわかったところで、次は発射およびダメージ制御のタスクについて見ていきます。各ロボットは最初に、デフォルトの「エネルギー・レベル」を保有しています。このエネルギー・レベルがゼロになったとき、ロボットは破壊されたと見なされます。弾丸を発射するとき、ロボットは最大で3ユニット (単位) までのエネルギーを使用できます。弾丸により多くのエネルギーを投入するほど、相手ロボットにより大きなダメージを与えます。弾丸を発射するには、`fire(double power)` および`fireBullet(double power)` を使用します。その際、エネルギーつまり発射パワー (double power) を指定します。`fireBullet()` 呼び出しは`robocode.Bullet` オブジェクトへの参照を戻すので、高機能ロボットはこれを活用できます。

イベント

ロボットが動いたり回転したりするとき、常にレーダーがアクティブになります。レーダーが範囲内に別のロボットを発見した場合、イベントがトリガーされます。ロボット作成者は、バトル中に発生するさまざまなイベントをどのように処理するかを選択できます。基本的な`Robot` クラスには、これらすべてのイベントを処理するデフォルト・ハンドラーがあります。作成者は、この「何もしない」デフォルト・ハンドラーのいずれかをオーバーライドして、独自のカスタム・アクションを実装することができます。よく使われるイベントは、次のとおりです。

- `ScannedRobotEvent` (ロボット発見)。`ScannedRobotEvent` を処理するには、`onScannedRobot()` メソッドをオーバーライドします。このメソッドは、レーダーがロボットを検出したときに呼び出されます。
- `HitByBulletEvent` (弾丸がこのロボットに命中)。`HitByBulletEvent` を処理するには、`onHitByBullet()` メソッドをオーバーライドします。このメソッドは、弾丸がこのロボットに当たったときに呼び出されます。
- `HitRobotEvent` (弾丸が他のロボットに命中)。`HitRobotEvent` を処理するには、`onHitRobot()` メソッドをオーバーライドします。このメソッドは、このロボットの弾丸が別のロボットに当たったときに呼び出されます。
- `HitWallEvent` (壁に衝突)。`HitWallEvent` を処理するには、`onHitWall()` メソッドをオーバーライドします。このメソッドは、このロボットが壁に当たったときに呼び出されます。

以上を理解するだけで、ある程度複雑なロボットを作成できます。このほかのRobocode APIに関する説明は、バトルフィールドのヘルプ・メニューまたはRobot Editorのヘルプ・メニューからアクセスできるJavadocを参照してください。

では、知識を実践してみましょう。

ロボットの作成

新規ロボットを作成するには、「File (ファイル)」->「New (新規作成)」->「Robot (ロボット)」を選択します。ロボットの名前を指定するようプロンプトが出されます。この名前は、Javaクラス名になります。今回の例では、DWStraight と入力しましょう。次に、固有のイニシャル (頭文字) を指定するようプロンプトが出されます。このイニシャルは、ロボット (および関連するJavaファイル) が格納されるパッケージ名として使用されます。今回の例ではdw と入力します。

ロボットを制御するために作成しなければならないJavaコードを、Robot Editorが表示します。たとえば、リスト1のようなコードが表示されます。

リスト1. Robocodeによって生成されるロボット・コード

```
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);
 }
}
```

以下のような強調表示された領域 (Area) に、ロボット制御用のコードを追加することができます。

Area 1 (領域1)
この場所には、クラス・スコープの変数とそれらの値を宣言できます。これらの変数は、ロボットの`run()` メソッド、および作成者が書く他のすべてのヘルパー・メソッドで使用できます。

Area 2
ロボットの生命を開始するために、`run()` メソッドがバトル・マネージャーによって呼び出されます。これには、通常、(リスト1でArea 2およびArea 3と表示された) 2つの領域を使用してコードを追加します。Area 2では、1つのロボット・インスタンスにつき一度だけ実行するコードを作成します。ここでよく行われる操作は、繰り返しアクションを開始する前に、ロボットをあらかじめ決められた状態に設定することです。

Area 3
ここは、典型的な`run()` メソッド実装の2番目の部分です。今回は、エンドレスな`while` ループの中に、ロボットが行う可能性のある繰り返しアクションをプログラミングします。

Area 4
この領域には、ロボットが`run()` ロジック内で使用するヘルパー・メソッドを追加します。さらに、作成者が希望する場合、イベント・ハンドラーをここに追加してオーバーライドすること

もできます。たとえば、リスト1のコードにおける`ScannedRobot` イベントの処理は、ロボットがレーダーによって発見されると、そのロボットに向けてただちに発射するという単純なものです。

この例の最初のロボットDWStraightの場合、リスト2に赤字で示されているようにコードを更新することにします。

リスト2. ロボット・コードDWStraightへの追加

```
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);
 }

}
```

この最初のロボットの機能は、以下のとおりです。

Area 1 (領域1)
このロボットでは、クラス・スコープの変数を1つも指定しません。

Area 2
ロボットを既知の状態に設定するために、`turnLeft(getHeading())` を使用して、ロボットが0度の方向を向くように回転させます。

Area 3
この繰り返しセクションでは、`ahead(1000)` を使用してロボットをできる限り遠くに移動させます。壁または他のロボットに衝突すると、停止します。その後、`turnRight(90)` を使って右に回転します。これを繰り返すと、ロボットは基本的に時計回りで壁の位置をトレースしてゆきます。

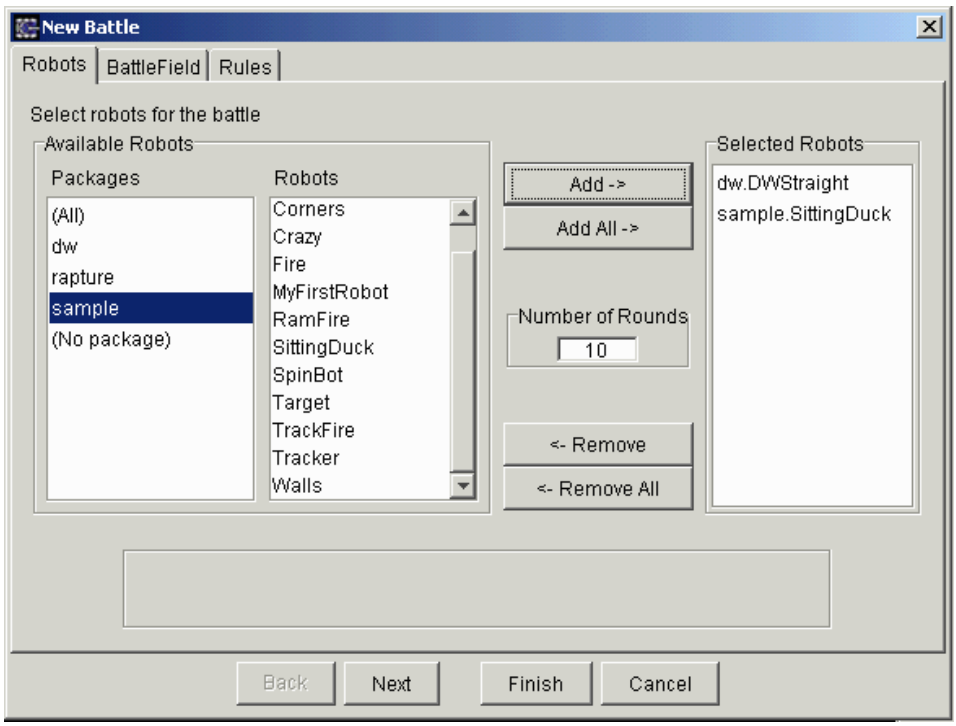
Area 4
ここでは、自動生成される`ScannedRobot` イベントを処理し、発見されたロボットに対して発射する操作のほかに、`HitByBullet` イベントを検出して、弾丸が自分に命中したときに180度回転するようにします (時計回り、または反時計回りに)。

ロボットのコンパイルとテスト

Robot Editorのメニューから、「Compiler (コンパイラー)」->「Compile (コンパイル)」を選択してロボット・コードを完成させます。これで初めてのバトルを闘う準備ができました。バトル

フィールドに戻って、メニューの「Battle (バトル)」->「New (新規作成)」を選択すると、図3のようなダイアログが表示されます。

図3. 新規バトルのダイアログ



この例のロボットdw.DWStraightをバトルに追加した後、敵ロボット (たとえばサンプルのsample.SittingDuck) を追加します。「Finish (完了)」をクリックすると、バトルが始まります。確かにSittingDuck (じっと座っているカモ) との戦闘はつまらないでしょうが、DWStraightロボットがデフォルトでどんな動きをするか確かめられます。サンプル・コレクション内の他のロボットと闘わせて実験すると、それらのロボットと比較したDWStraightの強さがわかります。

別のロボットのコードを分析する用意ができた読者は、参考文献のコード配布版に含まれているdw.DWRotaterロボットのコードをご覧ください。このロボットは、デフォルトで以下の動作をします。

- バトルフィールドの中心に移動する
- 他のロボットを発見するまで、大砲を回転し続ける
- 発見されたロボットよりも少し先に向けて、いくつかの異なる角度を試しながら発射する
- 他のロボットからの弾丸が命中したら、すばやく前後に動く

コードは単純なのでここでは分析しませんが、ぜひこれを試してください。さらに、Robocodeに付属のサンプル・パッケージには、この他にも多数のロボットが含まれています。

ロボットをサポートする追加のクラス

ロボット設計の腕が上がるにつれて、作成者がロボットに組み込むコード本体の量も増えるでしょう。モジュール性よくコードを扱う方法は、コードを複数のJavaクラスに分割し、パッケー

ジャーを使ってそれらを1つのパッケージ (JARファイル) にバンドルし、ロボットの配布に含めることです。Robocodeは、ロボット・ディレクトリー内のパッケージに含まれるさまざまなロボット・クラスを自動的に検出します。

その他のRobotサブクラス

誰でも`Robot` のサブクラスを作成して、ロボット構築用の機能を新しく追加することができます。Robocodeには`Robot` のサブクラス (非同期API呼び出しを可能にする`AdvancedRobot`) が1つ付属しています。この記事では`AdvancedRobot` クラスについて説明しませんが、基本的な`Robot` クラスの操作に慣れた読者は、この高機能のクラスをぜひ試してください。

バトル・シミュレーターのアーキテクチャー

Robocodeを設計した理由

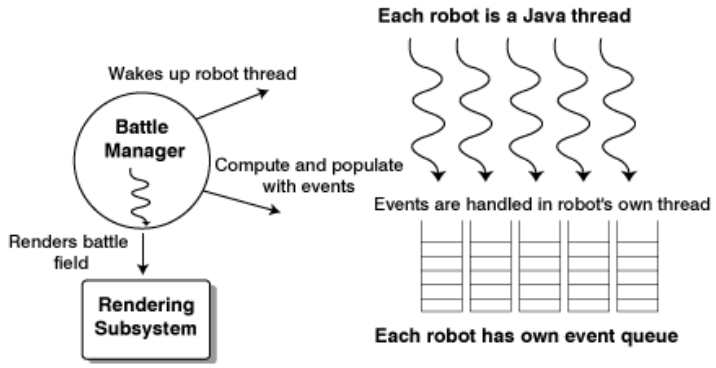
私はRobocode作成者であるMathew Nelson氏に会って、そもそもどんな動機からRobocodeを作成したのか尋ねてみました。Mathewは次のように語ってくれました。「Robocodeを作った動機の1つは、『Javaは遅い』とか『Javaではゲームは書けない』などという意見がもはや正しくないことを世界に証明したかったのです。この目的は達成されたと思います。」

Robocodeの中身を調べると、洗練されたシミュレーション・エンジンの存在に気付かされます。このエンジンは (現実的なスピードでバトルを描画するために) ハイパフォーマンスであり、しかも (複雑なロボット・ロジックを簡単に作れるよう) 柔軟性に富んでいます。寛大にもシミュレーション・エンジンのアーキテクチャーに関する内部情報を提供してくれたRobocode作成者Mathew Nelson氏に対して、特別な感謝を表します。

Javaプラットフォームを活用した設計

このシミュレーション・エンジン (図4を参照) は、最近のほとんどのJava VMが提供するノンプリエンプティブなスレッド化を利用しています。さらに、JDK GUIおよび2Dグラフィックス・ライブラリーの提供する描画機能も備えています。

図4. Robocodeシミュレーション・エンジンのアーキテクチャー



シミュレーション対象の各ロボットは、それぞれ独自のJavaスレッドになっていることに注意してください。それらは、可能であればVMのネイティブ・スレッド・マッピングを利用します。バトル・マネージャー・スレッドはこのシステムのコントローラーです。このスレッドはシミュレー

ションを統括し、グラフィック・レンダリング・サブシステムを管理します。グラフィック・レンダリング・サブシステムそのものは、Java 2DおよびAWTに基づいています。

ゆるやかなスレッド結合

共用リソースに関する潜在的な問題 (そして、シミュレーション・エンジンをデッドロックあるいは窒息させるという潜在的な問題) を軽減するために、バトル・マネージャーと各ロボット・スレッドとの間に非常にゆるやかな結合が必要です。このゆるやかな結合を実装するために、各ロボット・スレッドに独自のイベント・キューが与えられます。こうして、各ロボットへのイベントは、そのロボット自身のスレッドにおいて取り出されて処理されます。このスレッドごとのキューイングによって、バトル・マネージャー・スレッドとロボット・スレッドの間、またはロボット・スレッドどうしの間に競合が起こる可能性を効果的に回避しています。

Robocodeの内部

Robocodeのシミュレーター・エンジンは、実行時に一連のプラグイン (つまりカスタム・ロボット) を導入するシミュレーター・プログラムと見なすことができます。これらのプラグインは、提供されているAPI (`robocode.Robot` クラスのメソッド) を利用できます。物理的には、各ロボットは互いに独立したJavaスレッドであり、`run()` メソッドにはそのスレッドに対して実行されるロジックが組み込まれています。

任意の時点で、ロボット・スレッドは親である`robocoode.Robot` クラスの提供するAPIを呼び出すことができます。通常、その際には`Object.wait()` 呼び出しを介してロボットがブロックされます。

バトル・マネージャー・スレッド

バトル・マネージャー・スレッドは、バトルフィールド上のロボット、弾丸、描画を管理します。シミュレーションの「クロック (刻時)」は、バトルフィールド上に描画されるフレーム数で決まります。実際のフレーム・レートはユーザーによる調整が可能です。

バトル・マネージャーの典型的な1動作は、各ロボット・スレッドをwake upし、それぞれのロボットが自分の1動作 (つまり、ブロッキングAPIを再び呼び出すこと) を完了するのを待ちます。この待ち時間は、通常、ミリ秒の数十倍です。一般的に、現在のシステム速度のもとでは、最も複雑なロボットでさえ、戦略およびコンピューター処理にほんの1、2ミリ秒しか使用しません。

以下は、バトル・マネージャー・スレッドが実行するロジックの疑似コードです。

リスト3. バトル・マネージャーの疑似コード・ロジック

```
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
```

ループ内では、バトル・マネージャー・スレッドは最大時間間隔を超えて待機しないことに注意してください。(おそらく何らかのアプリケーション論理エラーや永久ループが原因で) ロボットが時間内にブロッキングAPIを呼び出さない場合、バトル・マネージャーはバトルを続行します。高機能のロボットに対しては、動作 (Turn) がスキップされたことを通知する`SkippedTurnEvent`がロボットのイベント・キュー内に生成されます。

置き換えられるレンダリング・サブシステム

現在の実装でのレンダリング・サブシステムは、バトル・マネージャーからのコマンドを受け取ってバトルフィールドを描画する単純なAWTおよびJava 2Dスレッドです。このサブシステムは、システムの他の部分から適切に切り離されています。将来の改訂版では、サブシステムが置換されることが予想されます (たとえば3-Dレンダラーに)。現在の実装では、Robocodeアプリケーションが最小化されるときには、シミュレーションが高速で進行するよう描画が常に使用不可になります。

今後のRobocode

Mathew Nelson氏は、Robocodeのユーザー・コミュニティーとの間で緊密なフィードバック・ループを保っています。alphaWorks Robocodeサイトのディスカッション・グループをご覧ください (参考文献を参照)。フィードバックのかなりの部分が、実際のコードに取り入れられています。Mathewが計画している機能強化には、次のようなものがあります。

- さまざまなオブジェクトや障害が存在するカスタム・バトルフィールド・マップ
- チームによるバトル
- トーナメントやリーグのサポートを取り入れること
- ボディ、大砲、レーダー、武器のスタイルをユーザーが選択できるようにすること

Robocodeの止まらぬ勢い

2001年7月12日に世間に公表されて間もないRobocodeがこれほど流行しているのは、まさに1つの現象です。最新バージョンはまだ1.0にも達していません (これを書いている時点では0.98.2) が、すでに世界中の大学のキャンパスや企業のPCでは、大変人気のある娯楽となっています。さまざまなユーザーが自分の作成したロボットをインターネット上で互いに闘わせるRobocodeリーグ (roboleagues) が、急速に広まっています。大学教授たちもRobocodeの教育的な特色に注目し、自分たちのコンピューター・サイエンスのカリキュラムに取り入れています。Robocodeのユーザー・グループ、ディスカッション・リスト、FAQ、チュートリアル、Webringなどもインターネット上に広まっています。

Robocodeは、人気ゲーム兼教育という分野の空白を埋めたようです。自分の創造的エネルギーを発散し、世界制覇の夢を追い求める学生たちや真夜中のエンジニアにとって、Robocodeは単純ながらも面白くてとっつき易く、しかも効果的な手段となっています。

著者について

Sing Li

Sing LiはWrox Pressから出版されている多数の本の著者で、Professional Apache Tomcat 、Early Adopter JXTA 、Professional Jini などを執筆しています。技術雑誌に頻繁に寄稿しており、P2P発展に関する熱心なエバンジェリスト（伝道者）でもあります。