

闘え、Robocode (ロボコード): 第2ラウンド

基礎を卒業して、高度なロボットの作成とチーム・プレイに進む


Sing Li
Author
Makawave

2002年 5月 01日

高度なロボットの作成とチーム・プレイを包括的に概観するこの記事をお読みにになり、Robocodeの世界にさらに挑戦する準備を整えてください。ベテランのJava開発者であり、Robocodeの熱狂的な信奉者に新しく転向したSing Li氏が、Robocodeのユニークで非常に面白い学習方法をフルに活用して、さらに高度なJavaプログラミング技法、アルゴリズム・デザイン、三角法の基礎を、そしてさらには分散コンピューティングの原理までも、読者に手ほどきします。敵ロボットは、何が自分たちを攻撃するのか知る由もないでしょう。

[その他の Robocode 関連記事はこちら](#)

Robocode Rumble (ロボコードの激闘)

 街で最新のRobocodeリーグで、Javaプログラミングのスキルに磨きをかけ、あなたの創造的エネルギーを解き放ってください。

今年1月の記事で、Robocodeシミュレーターの仕組みについて、[その舞台裏](#)を垣間見ました。その記事で作成したロボットは、ごく単純なもので、Robocodeに組み込まれている高度な機能は活用してはいませんでした。この記事では、Robocodeの紹介の完結編として、それらの高度な機能を取り上げます。その過程で、Javaプログラミング、数学、およびソフトウェア・デザインについて調べます。上級のロボット作成者にとっても記事の内容を興味深いものとするため、Robocodeの新しい「チーム・プレイ」機能を概観したり、数人のエキスパート設計者が作成したスーパー・ロボットから学べる点を調べたりします。

単純なロボットを卒業する: Javaクラスの継承

最初の記事で作成したロボットはすべて、Robotクラスから継承したものです。そのことは、次のステートメントからわかります。

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

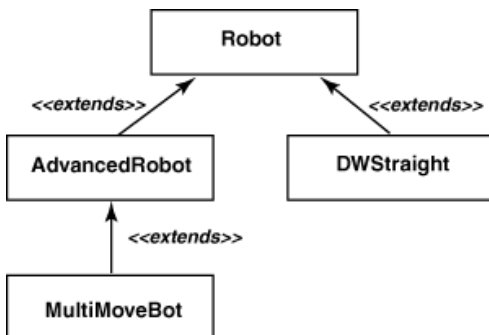
DWStraightは、Robot クラスを拡張しています。それは、そのクラスが提供しているすべてのメソッド (turnRight() や turnLeft() など) を使用できることを意味します。これらのメソッド

ドに関して制約の厳しい点を1つ挙げるとすれば、メソッドの操作が完了するまで制御をユーザー・コードに戻さないという点です。実際、これらのメソッドは、完了するまでにいくつもの動作 (turn) を必要とすることがあります。これらのメソッドを呼び出すと、実質的に、各動作ごとに判断を下す機会を手放すことになります。幸い、制御をユーザー・コードに戻してくれる、AdvancedRobot というクラスが用意されています。AdvancedRobot が提供しているすべてのメソッドを使用するのに必要なことは、作成するロボットをそのクラスのサブクラスにすることだけです。たとえば、AdvancedRobot から継承するMultiMoveBotというロボットを作成するには、次のようなコードを使用します。

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

ここで注目すべき重要な点として、AdvancedRobot は実際には Robot のサブクラスであり、私たちが作成する MultiMoveBot は AdvancedRobot のサブクラスです。その点を図示すると、図1のクラス階層のようになります。

図1. 継承の関係



クラスから継承する、つまりそのクラスのサブクラスになる、ということは、そのクラスのすべての public メソッドを使用できるようになるということです。したがって、私たちのロボットでは、AdvancedRobot クラスと Robot クラスの両方のメソッドを使用できるようになります。

次に、AdvancedRobot クラスから継承した新しい機能について調べてみましょう。

注: この記事には、関連するコード・ファイルがたくさんあります。ここで、[コードをダウンロード](#)することをお勧めします。この記事の内容は広範囲にわたっているため、記事を読みながらコードを分析することにも必ず時間を取るようにすれば、読者は最大限の益を得られます。コードは、記事の内容を補足するものとなるからです。

複数のアクションを同時に実行する: 非ブロッキング・メソッドの呼び出し

MultiMoveBot は AdvancedRobot のサブクラスなので、ロボットのアクションを1回の動作のたびに変更することができます。Robocodeにおける動作 (turn) は、刻時 (tick) (時計の針が1目盛り進むことに似ている) と呼ばれ、バトルフィールド上に表示されるグラフィカルなフレーム (frame) と関連しています。私たちがRobocodeのバトルを見ているとき、シミュレーション・エンジンは実際には、多数の静的なグラフィカル・フレームを順番に表示しています。これは、映画やテレビの映像の仕組みと同じです。これらのフレームそれぞれの間に、私たちのコードに制御が戻

り、ロボットをどのように動かすかについて判断することができます。最近のほとんどのPCは非常に高速であるため、1つのフレームつまり刻時ごとに多くのコードを実行する(つまり、非常に知的な判断をする)ことができます。

1刻時の間にロボットがどのように動くかを、どのようにして制御するのでしょうか。まず、Robot クラスで使用していたメソッドのほとんどすべてが、ブロッキング・メソッド呼び出しであることを確認しておきます。つまり、操作が完了するまでユーザー・コードに制御が戻りません。しかし、MultiMoveBot は AdvancedRobot のサブクラスなので、非ブロッキングの新しいメソッド群を使用できます。これらのメソッドは、直ちにユーザー・コードに制御を戻します。この後の表1に、私たちが使用したいいくつかのブロッキング・メソッドと、それに対応する非ブロッキング・メソッドを掲載します。

表1. ブロッキング・メソッドと非ブロッキング・メソッド

Robotから継承されるブロッキング・メソッド	AdvancedRobotから継承される非ブロッキング・メソッド
turnRight()	setTurnRight()
turnLeft()	setTurnLeft()
turnGunRight()	setTurnGunRight()
turnGunLeft()	setTurnGunLeft()
turnRadarRight()	setTurnRadarRight()
turnRadarLeft()	setTurnRadarLeft()
ahead()	setAhead()
back()	setback()

パターンに気づくとおもいますが、新しい非ブロッキング呼び出しはsetで始まります。これらの呼び出しを使用すると、ロボットに同時に複数のことを実行するように指示できます。たとえば、[ソース・コード](#)の配布に含まれているMultiMoveBot.javaファイルを見てみましょう。このコードのうち、リスト1の部分から分析を始めます。

リスト1. 非ブロッキング・メソッド呼び出しの利用

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

このコードは、MultiMoveBotに、右に曲がり、前進し、大砲を左に回転するよう指示しています。このすべてが、次の動作の間に同時に実行されます。

上記のメソッドはすべて、ロボットや大砲を回転したり移動したりせずに、直ちにユーザー・プログラムに制御を戻すことに注目してください。ユーザーが制御をRobocodeに戻すまで、何も実行されません。制御をRobocodeに戻すには、ブロッキング呼び出しを行う(つまり、上記の表の左欄にあるメソッドを呼び出す)か、特別なexecute()メソッドを使用します。

`execute()`メソッドは、ちょうど1刻時の間だけ、Robocodeエンジンに制御を戻します。ちょうどチェス・ゲームのように、今回の動作での動きを決定して指示すると、Robocodeが自動的に駒を動かしてくれるわけです。

実際のMultiMoveBotロボットでは、リスト2に示されている別の技法を使って、Robocodeに制御を戻しています。

リスト2. ブロッキング・メソッド呼び出しによってRobocodeに制御を戻す

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

刻時ごとの移動

ロボット、レーダー、および大砲は、1刻時あたりどの程度動くのでしょうか。実は、この質問に答えるのは、それほど易しいことではありません。1刻時あたり、大砲は20度回転し、レーダーは45度回転します。ロボットの回転速度は、その移動速度に応じて異なります。しかし、大砲はロボットの上に据え付けられており、レーダーは大砲の上に据え付けられているので、それらの回転速度は相互に影響しあいます。正確な数式を割り出す必要がある場合は、メニュー・オプション「Help」->「Robocode FAQ」を選択すれば、詳細な答えがわかります。

`waitFor()`メソッドは、ブロッキング呼び出しです。指定された条件が満たされるまでブロックします。このコードの場合は、車両の回転が完了した後で制御をユーザー・コードに戻すように、Robocodeに指示しています。もちろん、前進と回転を同時に行うようにロボットを設定したので、実際にはカーブを描いて動くことになります。

`toggleDirection()`メソッドは、我々が独自に作成したメソッドです。このメソッドは、呼び出されるたびに、ロボットの方向 (そして、大砲の方向) を反転させます。リスト3のとおりです。

リスト3. 曲線移動の方向と大砲の回転方向を反転させる

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

コードの中には、まだ説明していないメソッドがいくつかあります。好きな時にこのファイルをさらに調べてみることをお勧めします。MultiMoveBotを試してみるには、Robocodeバトルフィールド・ウィンドウで「Battle」->「Open」を選択して、showMultiMove.battleというバトルをロードします。このロボットの興味深い動きのパターンを観察してください。

カスタム・イベント: メソッドのオーバーライドと匿名クラス

DWStraightのような単純なロボットにさえ、イベントを処理するコードがありました。たとえば、自分のロボットに弾丸が命中すると、リスト4のコードが実行されます。

リスト4. イベント処理コード

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

ロボット作成者が処理できるすべてのイベントを調べるには、「Help」->「Robocode API」を選択します。RobotクラスAPIだけでも、次のようなイベントを処理できます。

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

イベントを処理するには、イベント・ハンドラー・メソッドを用意します。実は、私たちが継承したクラスには、既にこれらのメソッドが含まれています。しかし、私たちが自分でメソッドを作成すると、スーパークラスが提供しているメソッドの代わりに、その作成したメソッドが使用されます。このことを、仮想メソッドのオーバーライド、または単にオーバーライド (override) といいます。

AdvancedRobotクラスによって提供される付加的な柔軟性には、独自のカスタム・イベントを作成できることが含まれています。カスタム・イベントとは、Robocodeがユーザー・コードを呼び出す条件を定義できるイベントのことです。たとえば、リスト5に示されているCustomEventBotを見てください。

リスト5. CustomEventBotにおけるカスタム・イベントの作成

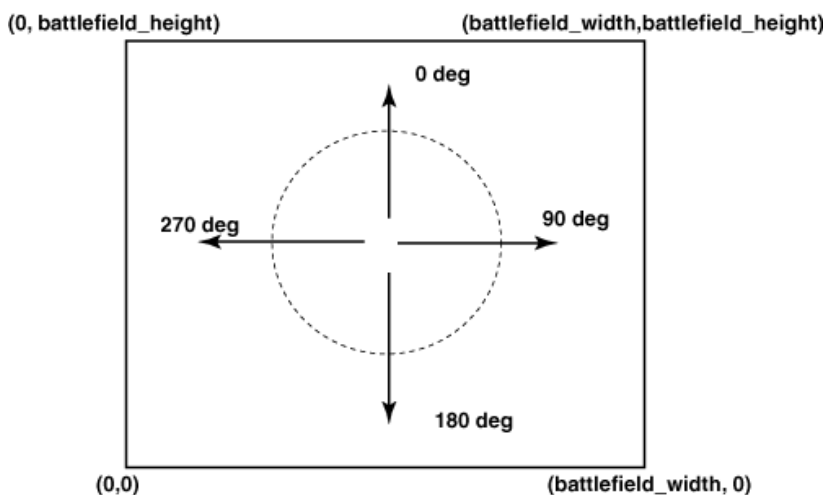
```
public class CustomEventBot extends AdvancedRobot
{
    ...
    public void run() {
        ...

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

このコードは、匿名 (anonymous) クラスというメカニズムを使って、2つの新しいカスタム・イベントを作成する方法を示しています。addCustomEvent()メソッドの中で、Conditionクラスの新しいサブクラスを定義して、そのtest()メソッドをオーバーライドしています。この新しいサブクラスは匿名です。つまり、名前がありません。"LeftLimit"などの引数は、Conditionクラスのコンストラクター・メソッドに対する引数として渡されます。

図2は、Robocodeで使用する座標と方向の規則を示しています。ここで定義した"LeftLimit"カスタム・イベントは、ロボットの正面が90度以下の向きになったときにトリガーされます。"RightLimit"条件は、正面が270度以上の向きになったときにトリガーされます。

図2. Robocodeの座標系



これらのカスタム・イベントを処理するために、AdvancedRobotクラスには、オーバーライド可能なonCustomEvent()メソッドがあります。私たちのコードでは、リスト6に示されているとおり、車両と大砲の回転方向を切り替えます。

リスト6. onCustomEventBot() によるカスタム・イベントの処理

```
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);
    }
}
```

これらのカスタム・イベントの定義とイベント・ハンドラーにより、ロボットの回転が90～270度の範囲に限定され、ロボットがツイスト・ダンスのような動きをします。これを試してみるには、twistergalore.battleファイルをロードして、開始してください。「Options」->「Preferences」->「Visible Scan Arcs」を選択すれば、「ツイスト」の様子がさらに劇的に表現されます。

ターゲットの所在: インターフェースと内部クラス

AdvancedRobotによって提供される新しい3つの重要な機能は、次のとおりです。

- 複数の動きを同時に実行できること。
- すべての刻時ごとにロボットのアクションまたは戦略を判断できること。
- カスタム・イベントを定義して処理できること。

これらの機能は、確かに、上級のロボット作成者にたくさんの機能を提供してくれますが、ターゲットがどこにいるか (ターゲットの所在) を見つける方法や、そこに近づく方法については何も教えてくれません。効率の良いロボットを作成したいのであれば、これはごく基本的な質問で、解決する必要があります。この問題には、少なくとも2つのアプローチがあります。最初に、簡単な方法を調べてみましょう。

リスト7に示されているDuckSeekerBotのコードをご覧ください。この単純なロボットは、じっと座っているカモ (Robocodeサンプル・コレクションの中で最もおとなしいロボット) に「目標を定め」、近づいて、“料理”します。

リスト7. DuckSeekerBotクラスの定義

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

予想通りのことですが、このロボットは、十分な柔軟性を実現するために、AdvancedRobotをサブクラス化しています。さらに、インターフェースDuckConstantsを実装して、そのインターフェースにある定数を共用しています。

DuckConstants.javaを見れば、そのインターフェースの定義がわかります。これまでずっと使用してきた定数 (halfTurn、fullTurn、その他) がすべて含まれています。したがっ

て、DuckConstantsインターフェースを実装することは、自分のクラスにこれらの定数すべてを組み込むための手軽な方法です。上記のコード断片の中では、`curTarget`という名前のTarget型の変数を作成したことに注目してください。しかし、TargetはRobocodeライブラリーには含まれていません。これは何なのでしょう。

DuckSeekerBot.javaの後半 (この後のリスト8に示されている部分) を注意深く調べれば、手掛かりが得られます。そこには、DuckSeekerBotの内側で定義されているまったく新しいクラスがあります。これは、内部クラス、さらに具体的には「メンバー・クラス」の定義です。

リスト8. Targetメンバー・クラス

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

DuckSeekerBotの中でTargetクラスをメンバー・クラスとして宣言すると、そのクラスをDuckSeekerBotの中で利用できるようになります。コードを調べるとわかるとおり、`curTarget`は、`onScannedRobot()`メソッドで発見した現在のターゲットを保持するために使用します。ターゲットを追尾するには、非常に単純な方法を使用します。つまり、ターゲットの方を向くように回転し、ターゲットの方へ移動してから、攻撃するだけです。リスト9に示されているのは、`onScannedRobot()`メソッドに含まれているコードで、獲物を料理するのに必要なコードです。

リスト9. ターゲットを追尾する

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

DuckSeekerBotは、新しいカモのスキャンと、ターゲットの追尾を交互に実行します。DuckSeekerBotの動きを実際に観察するには、「Battle」->「Open」を選択して、`duckoAducko.battle`ファイルを開きます。すると、DuckSeekerBotと、動かないカモの群れの対戦が始まります。

バトルフィールドの全体像を把握する: ベクトル、ポリモアフィズム、およびjava.Math

DuckSeekerBotは、次のような仕組みで動きます。

1. ターゲットのカモをスキャンする。
2. ターゲットに近づき、攻撃して“料理”する。

3. カモの群れがなくなるまで繰り返す。

同じ問題に対する別のアプローチとして、次のような方法も考えられます。

1. バトルフィールド内に見つかるすべてのカモをスキャンして、「軍事情報マップ」を作成する。
2. カモの群れに1つずつ近づき、その群れを全滅させる。
3. スキャン情報を利用して、「マップ」を定期的に更新する。

この第二のアプローチでも、最初のアプローチと同じ結果が達成されますが、より情報に基づいています。高度なロボットのほとんどは、この種の「全体像」の情報を利用して、瞬時に戦略的な判断を下しています。そのようなマップを保守する方法を学べば、より洗練された知能を備えたロボットを作成できるようになります。

この記事で紹介するFlockRoasterBotは、次のようなヘルパー・カスタム・クラスを使用して、全体像を保守します。

- Duck (Duck.javaに含まれている): DuckSeekerBotで定義をしたより単純なTargetクラスのように、このクラスはターゲットのカモについて収集できる情報をすべて保守します。
- Flock (Flock.javaに含まれている): このクラスは、見つかったすべてのカモを列挙します。これが、私たちの「軍事情報マップ」になります。このクラスを作成するために、実際には、標準のJavaライブラリー・クラスjava.util.Vectorをサブクラス化します。Vectorクラスは、リストの管理機能(リストへのカモの追加、リストからのカモの削除、その他)を提供してくれます。

FlockRoasterBot.javaソース・コードを調べれば、新しくスキャンされたロボットはすべて、onScannedRobot()イベント・ハンドラーで追加されていることがわかります。リスト10に示されているとおりです。

リスト10. スキャンされたロボットを群れ (flock) に追加する

```
public void onScannedRobot(ScannedRobotEvent evt) {  
    Duck nuDuck = new Duck(evt, getTime(), getX(), getY(), getHeading(),  
        getVelocity(), true, false);  
    if (!curFlock.contains(nuDuck)) {  
        curFlock.add(nuDuck);  
    }  
}
```

リスト10では、まず最初に、Duckの新しいインスタンスを作成します。これには、カモに関するスキャンで収集した情報をすべて格納します。次に、このカモをこれまでにスキャンしていないことを確認してから (Flock/Vectorクラスによって提供されているcontains()メソッドを使用する)、それを群れに追加します。Duck.javaを注意深く調べればわかるとおり、equals()メソッドの実装をオーバーライドすることにより、1つのDuckインスタンスを別のインスタンスと比較する方法を定義しています。この手順は、contains()メソッドが正しく機能するために不可欠です。

ここで興味深いのは、java.util.Vectorのcontains()メソッドが元々Java APIライブラリーの作成者たちによって記述されたとき、そのうちの誰も、私たちのDuckクラスについて知るよしもなかったということです。それでも、このメソッドは、私たちのDuckクラスに対して完べきに機能

します。これは、Java言語のポリモアフィズムを使用した一例です。つまり、contains()メソッド内のプログラミング・ロジックが、独自のequals()メソッドを適切に実装した任意のクラスについて、現在も将来も、正しく機能するということです。

最後に、避けられない物事として、数学 (さらに具体的には、三角法) にアプローチしなければなりません。Duck.javaのソース・コードを調べればわかるとおり、DuckRoasterBotクラスの古いTargetメンバー・クラスにあったすべてのメソッドに加えて、次の2つの興味深いメソッドが含まれています。

- bearingToDuck: 与えられた現在のx,y位置に対して、ターゲットのカモの方位を算出します。
- distanceToDuck: 与えられた現在のx,y位置に対して、ターゲットのカモまでの距離を算出します。

これらのメソッドにより、Duckクラスが非常に融通の利くものになります。上記の2つの数量を算出するのに使用される三角法の数式が、これらのメソッドによって隠される (Java言語の言い方ではカプセル化される) からです。難解であるとはいえ、非常に便利なこのトピックについて思い出すには、付属記事の「単位円の三角法」を参照してください。また、三角法について深い関心がある場合は、bearingToDuck()およびdistanceToDuck()メソッドのソース・コードを詳細に調べてみることもできます。これらのメソッドでは、java.Mathコード・ライブラリーをフルに活用しています。java.Mathライブラリーには、どんな数学者も満足させるだけの高度な数学メソッドが十分に収められています。

FlockRoasterBotは、distanceToDuck()メソッドを使って、攻撃して“料理”するための最も近いカモを判別します。FlockクラスのgetNextRoaster()メソッド (リスト11を参照)に、最も近いカモを選択するロジックが含まれています。

リスト11. getNextRoaster() メソッド

```
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
}
```

料理する次のカモが決まったら、bearingToDuck()メソッドを使ってそのカモに狙いを定めます。

FlockRoasterBotの動きを実際に観察するには、RoastedDucks.battleファイルをロードしてください。基本的な動きは、DuckRoasterBotと同じように「次から次へと攻撃する」スタイルです。ただし、「Visible Scan Arcs」オプションをオンにする (「Options」->「Preferences」->「Visible Scan Arcs」を選択する) とわかるとおり、FlockRoasterBotはそれぞれのカモを再スキャンすることをしません。バトルフィールドの全体像を把握しており、次にどのカモを料理したらよいかを「知っている」からです。

FlockRoasterBotをチーム・プレイ用に改造する

最新リリースのRobocodeは、待望のチーム・プレイ機能をサポートしています。チーム・プレイ・モードでは、ただ1つのロボットを設計するのではなく、様々なロボットから成るチーム全体を設計します。1つのチームには、いくつかの異なるクラスを参加させることも、同じクラスのいくつかのインスタンスを参加させることもできます。

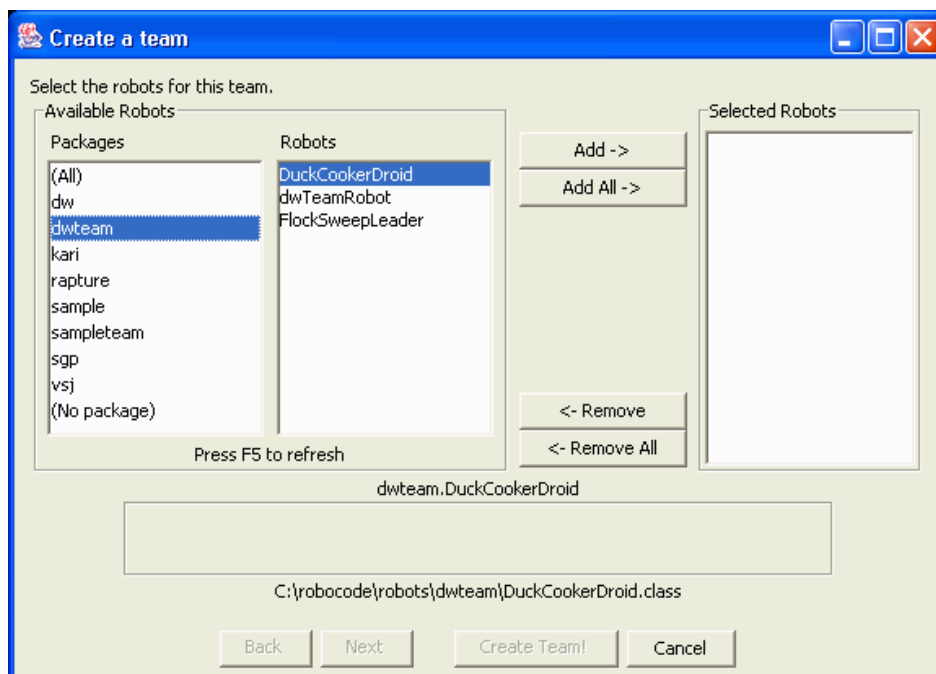
チーム・プレイで提供されるオプションの一部を概観すると、次のようになります。

- 1つのロボットを、チーム・リーダーとして指定できます (チームに追加した最初のロボット)。リーダーには、最初に余分のエネルギーが与えられます (合計で200ユニット)。万一、リーダーが破壊されてしまうと、すべてのチーム・メンバーは30ユニットずつ失います。
- チーム・ロボットには、付加的な機能があります。AdvancedRobotのサブクラスであるTeamRobotを使用して、互いにメッセージをやり取りすることができます。

独自のチーム・ロボットを作成するには、いつでもTeamRobotをサブクラス化する必要があります。さらに、ロボットにDroidインターフェースを実装することもできます。ドロイドにはリーダーがなく、スキャンを実行できません。しかし、すべてのドロイドは、非ドロイド・ロボットと比べて開始時に20ユニット多くのエネルギーを持っています。

チームを作成するには、「Robot」->「Team」->「Create Team」を選択します。図3に示されている「Create a Team」ダイアログが表示されます。

図3. 「Create a Team」ダイアログ



ここでは、先ほどのFlockRoasterBotを改造して、新しいチーム・プレイ・モードで動作するようにします。作成するチームは、1台の知的に活動するリーダーと、2台のドロイドで構成します。FlockRoasterBotにある「情報」を、チーム・リーダーとドロイドの間で分け合う必要があります。それを実行するために、次の2つのクラスを使用します。

- `FlockSweepLeader`: スキャンを実行し、バトルフィールドのインテリジェンス・マップを保守して、他のチーム・ドロイドにカモを攻撃するように指令を出します。さらに、攻撃にも参加します。
- `DuckCookerDroid`: `FlockSweepLeader`からの「攻撃指令」を監視し、指令が出たらターゲットのカモを探知し、追尾して、攻撃します。最後に、カモを仕留めることができたら、`FlockSweepLeader`に報告します。

`FlockRoasterBot`と一緒に以前に作成した`Duck.java`と`Flock.java`を再利用できれば理想的ですが、そうはいきません。チーム・プレイでは、群れの中のカモが生きていても、`FlockSweepLeader`によって追尾されていない、つまり、マップを保守するクラスの現在のターゲットになっていないことがあるからです。それは、問題のカモを攻撃するように既にドロイドに指令が出されている場合です。したがって、カモが、チーム・メンバーのいずれかによって既に狙われているかどうかを判別できなければなりません。`TeamDuck`および`TeamFlock`クラスは、汎用のカモに、この付加的な状態を追加します。これらのクラスは、それぞれ`Duck`および`Flock`クラスをサブクラス化して、それらのクラスに組み込まれている機能を再利用します。

チーム内のメッセージング: 通信プロトコルの設計

チーム・メンバー間でメッセージを送ることは、チーム内で連絡をとるための唯一の方法なので、勝つための戦略を練るには、明確に定義された通信プロトコルが不可欠です。混乱を避けるために、すべてのチーム・メンバーが、相互に通信する規則について合意していなければなりません。

私たちのチームは、`FlockSweepLeader`クラスという形での1台のリーダーと、任意の台数の`DuckCookerDroid`で構成します。これらのチーム・メンバーは、次のようなプロトコルで通信します。

Cook-a-duckプロトコル

Cook-a-duckプロトコル (表2を参照) は、リーダーがドロイドの位置を取得し、各ドロイドにカモの攻撃を委任するために使用されます。

表2. Cook-a-duckプロトコル

発信者	受信者	メッセージ	説明
リーダー	ドロイド	位置を報告 ("reppos". <code>TeamCommands.java</code> を参照)	チーム・メンバーのすべてのドロイドに対して、x,y位置情報の要求をブロードキャストします。
ドロイド	リーダー	x-y位置 (<code>DroidPosition.java</code> を参照)	x,y座標を報告します。リーダーは、この情報を利用して、群れの中でまだ生きているカモのうち、ドロイドに最も近いものを見つけます。
リーダー	ドロイド	直列化されたDuck (<code>TeamDuck.java</code> を参照)	指定したカモを標的にするようにドロイドに指令を出します。

Cook-a-duckプロトコルを開始するコードを調べるには、`FlockSweepLeader.java`の`assignMission()`メソッドを見てください (リスト12を参照)。

リスト12. assignMission() メソッド

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

`broadcastMessage()`メソッドは、`TeamRobot`クラスによって提供される新しい機能の一部です。`REPORT_POSITION`は、`TeamCommand.java`インターフェースの一部である定数です。このインターフェースには、私たちが送信するメッセージのためのプロトコル定数が含まれています。

ドロイド側では、`onMessageReceived()`メソッドを見てください。これは、チーム・ロボットがメッセージを受け取る部分です。メッセージの詳細は、引き渡される`MessageEvent`インスタンスの一部として入手可能であることに注目してください。メッセージそのものの内容は、任意の直列化されたJavaオブジェクト・インスタンスです。`REPORT_POSITION`および`FLOCK_GONE`メッセージは、どちらも`String`クラスをメッセージとして送信するのに対して、`FlockSweepLeader`からのカモの割り当てメッセージには、直列化された`TeamDuck`インスタンスが入っています。リスト13は、メッセージとして引き渡されたオブジェクトのクラス(型)を、`instanceof`演算子で判別する様子を示しています。

リスト13. ドロイドがメッセージを受け取る様子

```
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プロトコル

Duck-baggedプロトコル(表3を参照)は、ドロイドがカモを仕留めたことをリーダーに通知するために使用されます。リーダーは、保守している軍事情報マップを更新して、そのドロイドに攻撃を委任するべき別のカモがいるかどうかを調べます。もしいる場合は、リーダーはCook-a-duckプロトコルを使ってドロイドに指令を出します。

表3. Duck-baggedプロトコル

発信者	受信者	メッセージ	説明
-----	-----	-------	----

ドロイド	リーダー	直列化されたDuckBagged (詳細はDuckBagged.javaを参照)	任務が完了し、カモを仕留めたことを、リーダーに通知します。
------	------	--	-------------------------------

Flock-gone プロトコル

Flock-gone プロトコル (表4を参照) は、群れが全滅したことを、リーダーから残っているドロイドに通知するために使用されます。

表4. Flock-gone プロトコル

発信者	受信者	メッセージ	説明
リーダー	ドロイド	群れが全滅 ("flockgone". TeamCommands.javaを参照)	メッセージをチーム全体にブロードキャストします。

衝突の回避: ランダムな後退を実装する

バトルフィールド上でチームを戦わせる場合にしばしば起こる問題の1つは、チームメイトとの衝突を避けられないということです。私たちのチームでは、エネルギーを浪費する衝突が何度も繰り返し起こるのを回避するために、衝突の後に後退するようにしています。ただし、2台のチームメイトが衝突と回避を繰り返すデッドロック状態になるのを避けるため、3種類の後退の動きからランダムに選択します。pickRandAvoidance()メソッド (リスト14を参照) が、この振る舞いを実装しています。

リスト14. 後退の動き

```
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);
    }
}
```

pickRandAvoidance()メソッドは、追尾しているターゲットに向かう途中でロボットと衝突したときに呼び出されます。onHitRobot()イベント・ハンドラー (リスト15を参照) から、このメソッドが呼び出されます。

リスト15. onHitRobot() イベント・ハンドラー

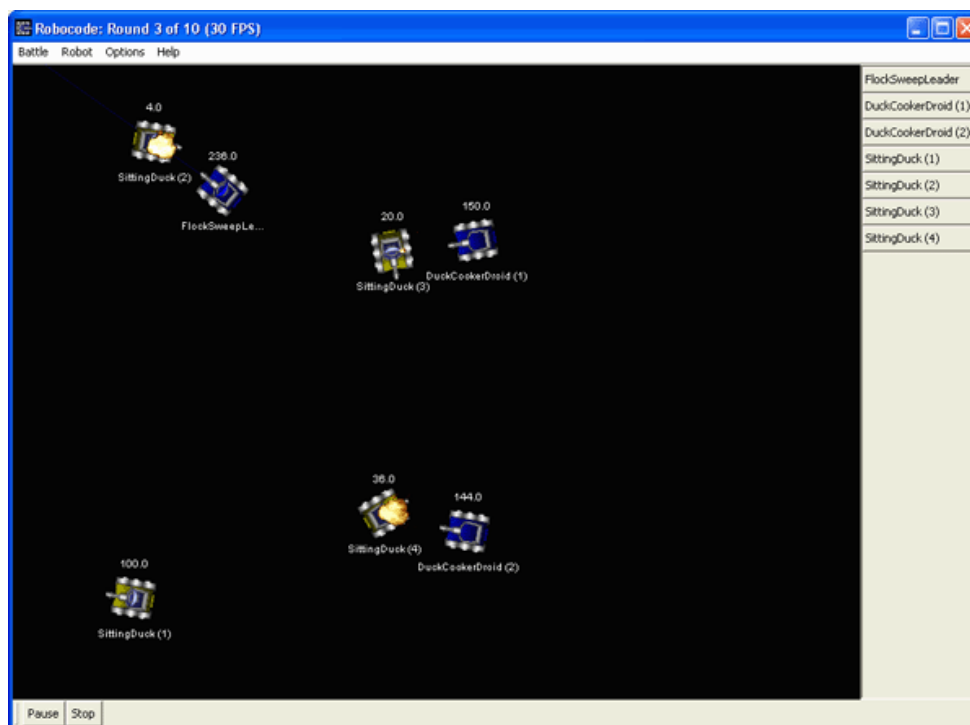
```
public void onHitRobot(HitRobotEvent e) {
    pickRandAvoidance();
    if (curTarget != null)
        curTarget.setLocked(false);
}
```

FlockSweepLeaderとDuckCookerDroidの両方がカモの攻撃に参加するため、その両方が、前述のランダムな後退とonHitRobot()の実装を必要とします。これらの共通のメソッドは、dwTeamRobotと

いう共通の基底クラスに抜き出しました(Javaプログラマーの言葉遣いでは、このことをリファクタリングするといいます)。FlockSweepLeaderとDuckCookerDroidは、どちらもdwTeamRobotのサブクラスです。要件や設計の変化に応じたこの種のリファクタリングはJavaプログラミングではごく一般的です。

私たちの分散型の「カモ一掃チーム」の動きを観察するには、「Battle」->「Open」を選択して、teamsweep.battleファイルをロードしてください。1台のリーダーと2台のドロイドが、4台の動かないカモのグループと対戦します。リーダーが最初にスキャンした後、攻撃を委任する指令を出す様子を見ることができます。衝突を回避する動きも、かなりの確率で観察されます。図4に、分散型のカモ一掃チームの対戦中の様子を示します。

図4. 分散型のカモ一掃チームが対戦しているところ



本物のRobocodeの世界: 上位ロボットの調査

動かないカモを攻撃して一掃するのも確かに面白いですが、実際には、最も平凡な敵ロボットでさえ、攻撃されるまでじっと動かずにいることはまずありません。動き回るロボットを相手にするときには、戦略を少し修正しなければなりません。たとえば、ターゲットのロボットの方へ移動したり、そのロボットを攻撃したりする前に、そのロボットが将来どこにいるかを予測することができるでしょう。このことを、標的の予測といいます。また、攻撃を返してくるロボットを相手にするときには、弾丸を検出して回避行動を取ることも考えなければなりません。さらに、戦略を練る際のベースとして、それぞれの時点での残りエネルギー量も考慮に入れたいと思います。

これらの点はどれも、優秀なロボットを作成するときに考慮に入れなければならない点です。しかし、残念ですが、ロボットの戦略の設計全般は、この記事で扱う内容の範囲をこえています。とはいえ、Robocode RumbleのWebサイト ([参考文献](#)を参照) には、優秀なRobocodeロボットを

作成するのに役立つ多くの優れた最新情報記事がまとめられています。この記事の残りの部分では、インターネット上の本物のRobocodeの世界で遭遇する上位ロボットのいくつかについて、その調査結果を取り上げます。この記事に掲載したロボット設計者は、その技術の面で頂点に立っている人たちで、世界中に散らばっています。

これらのロボットの作成者の皆様には、この記事のために設計上のアイデアを公開してくださったことを、この場をお借りして深く感謝したいと思います。

Nicator、Alisdair Owens作 (英国)

Nicatorは、その作成者のAlisdair Owensによれば、攻撃型のロボットで、猛烈な攻撃を仕掛けた末に若くして死滅するとのこと。したがって、グループ対戦の状況に特化したロボットです。このロボットの必勝法は、ロボットを最も有利なコーナーに配置するという独特の“反重力”移動と、待機状態を保ちながら敵を積極的に攻撃するという作戦です。Nicatorは、バトルフィールドの全体像をスキャンして保守します。さらに、AdvancedRobotクラスの複数の非ブロッキング・アクションをフルに活用して各刻時ごとに判断を行い、カスタム・クラスも十分に活用しています。(Alisdairは、彼の“反重力”技法を「Secrets from the Robocode masters」という記事に公開しています。 [参考文献](#)を参照。)

Wolverine、Jae Marsh作 (米国)

Nicatorとは対照的に、Wolverineは、1対1の対戦に特化したロボットです。Wolverineは、Jae Marsh (Robocodeコミュニティでのハンドル名はgraygoo) の独創的アイデアの結晶です。Wolverineは、標的を定めたり、相手の発砲を検出して弾丸を回避したりするために、高度なパターン・マッチングを利用します。Wolverineの戦略は本質的には攻撃型であり、最低でも50ポイント相手より優位に立ったことがわかると、攻撃を強めます。さらに、Nicatorとは違う別の点として、Wolverineは、バトルフィールドの全体像を常に最新に保つことはしません。その代わりに、バトルフィールドの一時的なスナップショットを利用して、そのパターン・マッチング・アルゴリズムを実現しています。Wolverineは、AdvancedRobotクラスのカスタム・イベント機能を活用して、イベントを記録するロジックを強化しています。

RayBot、Ray Vermette作 (カナダ)

RayBotは、標的の予測を行い、バトルフィールドの全体像を常に最新に保ちます。NicatorやWolverineとは違う点として、RayBotは基本的に防御型のロボットです。しかし、1対1の対戦シナリオで、相手よりはっきりと優位に立ったことがわかると、その性格を一変させて攻撃型になります。RayBotの作成者であるRay Vermette (Robocodeコミュニティでのハンドル名はRaymundo) によると、このロボットを他と差別化している主要な点は、スキャンしたデータとバトルフィールドの情報 (全体像) にあるとのこと。

JollyNinja、Simon Parker作 (オーストラリア)

JollyNinjaは、Simon Parkerの作品で、グループ対戦と1対1の対戦の両方に対応するように設計されています。Simonによれば、JollyNinjaの最も特徴的な機能は、その移動の戦略にあるとのこと。そのため、JollyNinjaは、相手にとって非常に攻撃しにくい、中程度の攻撃型ロボットになっています。JollyNinjaの核心は、ロボットの次の移動を決定する戦略的な評価機能にあります。

す。AdvancedRobotクラスの非ブロッキング呼び出し機能を利用して、各刻時ごとに判断を行います。JollyNinjaは、バトルフィールドに残された敵が1台だけになったことがわかると、攻撃のレベルを増加させます。JollyNinjaは、その戦略的な判断を実行するために、全体像を保守し、バトルフィールドの詳細な情報を収集します。(Simonは、敵の動きを追跡する戦略を、「Secrets from the Robocode masters」という記事に公開しています。[参考文献](#)を参照。)

TronおよびShadow、ABC作 (ポルトガル)

Robocodeコミュニティには、面白いハンドル名や魅力的なペンネームをもつメンバーが大勢います。ABCは、TronとShadowという2台の優秀なロボットの作成者です。Tronは、特徴的な移動パターンを持つ「なわばり」型のロボットですが、戦略上は主として防御に回ります。Shadowは、「なわばり」型ではない攻撃ロボットで、攻撃を回避する動きをします。Shadowのユニークな点は、グループ対戦と1対1の対戦で、モードを切り替えずに同じ戦略を使用しようとしている点です。それを達成するために、“重み付き中央周回 (weighted center orbiting)”戦略を採用しています。どちらのロボットも、AdvancedRobotクラスの同時に複数の動きをする機能をフルに活用しています。さらに、戦略実装への入力として、バトルフィールドの全体像の情報も収集します。

Robocodeによる学習は決して終わらない

Javaプログラミング、アルゴリズム・デザイン、三角法の基礎、そして分散コンピューティングの原理までをも守備範囲にする優れた教育ツールとしてのRobocodeの可能性を疑う向きがあったとすれば、この記事がその疑問を解いたはずです。Robocodeは、ロボット設計の初心者が、勝つことのできる高度なロボットを作成するべく「もうひと頑張り」するように自然と促します。そのことは、彼らがプログラミングとアルゴリズム・デザインの技術をいっそうマスターすることにつながります。Robocodeの対戦は、「ありふれたゲーム」とは程遠いもので、友人同士で切磋琢磨しながらその教育上の目標を達成できるのです。学習がいつでもこのように楽しければ、どんなにか良いことでしょう。

ダウンロード

内容	ファイル名	サイズ
	j-robocode2.zip	35KB

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。

© Copyright IBM Corporation 2002

(www.ibm.com/legal/copytrade.shtml)

商標

(www.ibm.com/developerworks/jp/ibm/trademarks/)