



```
import robocode.*;

public class DodgeBot extends
AdvancedRobot {
    double previousEnergy=100;
    int movementDirection = 1;
    int gunDirection = 1;
    public void run() {
        setTurnGunRight(99999);
    }
    public void
onScannedRobot(ScannedRobotEvent e) {
        // Stay at right angles to the opponent

        setTurnRight(e.getBearing()+90-30*movementDirection);

        // If the bot has small energy drop,
        assume it fired
        double changeInEnergy =
previousEnergy-e.getEnergy();
        if (changeInEnergy>0 &&
changeInEnergy<=3) {
            // Dodge!
            movementDirection =
-movementDirection;

        setAhead((e.getDistance()/4+25)*movementDirection);
        }
        // When a bot is spotted, sweep the gun
        and radar
        gunDirection = -gunDirection;
        setTurnGunRight(99999*gunDirection);

        // Fire directly at target
        fire(2);

        // Track the energy level
        previousEnergy = e.getEnergy();
    }
}
```

## リスト 1. DodgeBot のコード

さらに大きな問題点は、この妙手により、標的を狙う一般的な方法をかく乱することができるとはいえ、横方向にステップする動きは、実際のところかなり予想しやすいということです。この方策を利用する最善の方法は、おそらく、得られた情報によって動きを制御するのではなく、その情報を回避行動の手引きとすることでしょう。

この方策は単純なので、読者の中には、自分でも思いつくことができただろうと考えている人もいるかもしれません。それは立派なことです。実のところ、これがこのゲームの遊び方であり、これほど病みつきになってしまう理由もそこにあります。Robocode はちょうどチェス・ゲームのようなもので、駒の新しい動かし方を思いつくたびに、その背

後に新しいアイデアがあるのです。

**Jae Marsh** 氏は、InfoStructure Systems の開発者です。趣味はロック・クライミングとピアノの即興演奏で、今は Robocode にも夢中です！



## レーダーによるスキャン Eivind Bjarte Tjore 著

Robocode で成功するためのキー・ポイントの 1 つは、すべての敵ロボットについて常に最新の情報を得ておくことです。そのためには、必要最低限のスキャンを実行するようにレーダーを活用しなければなりません。たとえば、ロボットが競技場のコーナーにいる場合は、周囲の 360 度を完全にスキャンしても無意味です。

このヒントでは、レーダーを制御するために私が et.Predator で使用しているコードについて説明します。前提条件は、ロボットが AdvancedRobot であり、Enemy クラス (ScannedRobotEvent からの情報を保持するクラス) と、EnemyMap クラス (Enemy のコレクション) を持っていることです。私が使用している EnemyMap は HashMap を拡張したのですが、Hashtable やその他の適切なクラスを使用することもできます。

Enemy クラスには、isUpdated() 関数が必要です。この関数は、直前の情報が更新されて以降の時間 (Robot の getTime() 値から ScannedRobotEvent の getTime() 値を引いた時間) が一定の制限値以内であるかどうかをチェックします。現在のところ、私は、16 という制限値を使っています。

ロボットの初期化部分 (run() の中の、ループの前) では、次のようなコードを使用します。

```
addCustomEvent(new
RadarTurnCompleteCondition(this));
setAdjustRadarForGunTurn(true);
setTurnRadarRight(360);
```

次に、カスタム・イベントのイベント・ハンドラーを追加します。

```
public void onCustomEvent(CustomEvent e) {
    if (e.getCondition() instanceof
        RadarTurnCompleteCondition) sweep();
}
```

さらに、それぞれの周回において getScannedRobotEvents() を使用して、すべての ScannedRobotEvent が記録されるようにする必要があります。

最後に、レーダー・スキャンを制御するために、**リスト 2** のコードを追加します。このコードでは、数値 n の符号 (-1 または 1) を返す int sign(double n)

というメソッドを使用していることに注意してください。さらに、入力値の角度を -180 ~ 180 度の範囲に正規化する normalRelativeAngle(double angle) も使用しています。

Sweep() 関数では、他のロボットへの方位角の絶対値のうち最大値を判別します。すべてのロボットが更新済みの場合は、自分からの方位角が最大の敵ロボットの方へレーダーを向けます。更新済みでないロボットがある場合は、現在の方向で引き続きスキャンするようにレーダーを設定します。

スキャンに 22.5 度を加算していることに注目してください。これは、前回のスキャン以降ロボットが移動したことを考慮に入れた、安全マージンです。22.5 という値を選んだのは、レーダーの回転速度がフレーム当たりおよそ 45 度だからです。この値に定数を使わずに、方位角が最大のロボットまでの距離を取得して、正しい方位角として考えられる最大値を計算することにより、このアルゴリズムを改善する余地があります。

**Eivind Bjarte Tjore** 氏は、ノルウェーのベルゲン出身で、トロンハイムにあるノルウェー科学技術大学でコンピューター科学を学んでいます。



## RoboLeague Christian Schnell 著

遅かれ早かれ、多くの積極的なロボット作製者たちは、他のロボットについて学びたいと考えるはずで、他にどんなロボットがあるのだろうか？ どのロボットなら私のロボットを負かすことができるだろうか？ 私のロボットに挑戦しないほうがよいロボットはどれで、それはなぜだろうか？ Robocode を使い始めてしばらく後、私はこのように感じたため、いろいろと見て回って、RoboLeague の必要性を見い出しました。

## 設計

リーグを構築するには、いくつかの方法があります。私の最初の考えは、グローバルで制限のないロボット・リーグを開催することでした。ロボットの集合は変更できるようにする必要があり、個々のロボットも、それぞれのランクに影響を与えずに更新できるようにする必要があります。200 を超えるロボット・クラスが見つかったので、スポーツ・リーグで試合数 (リーグの大きさに応じて指数関数的に数が増える) を減らすためによく利用される、複数のディビジョン (部) に分かれたリーグ・モデルを採用することにしました。このモデルでは、リーグを複数のディビジョンに分割し、そのディビジョンを通

```

private int radarDirection=1;

private void sweep() {
    double maxBearingAbs=0, maxBearing=0;
    int scannedBots=0;
    Iterator iterator = theEnemyMap.values().
        Iterator();
;
    while(iterator.hasNext()) {
        Enemy tmp = (Enemy)iterator.next();

        if (tmp!=null && tmp.isUpdated()) {
            double bearing=normalRelativeAngle
                (getHeading() + tmp.getBearing()
                 - getRadarHeading());
            if (Math.abs(bearing)>maxBearingAbs) {
                maxBearingAbs=Math.abs(bearing);
                maxBearing=bearing;
            }
            scanned Bots++;
        }
    }
    double radarTurn=180*radarDirection;
    if (scannedBots==getOthers())
        radarTurn=
            maxBearing+sign(maxBearing)*22.5;

    setTurnRadarRight(radarTurn);
    radarDirection=sign(radarTurn);
}

```

## リスト 2. レーダー・スキャンのメソッド

してプレイします。近接したディビジョン間での昇格と降格があります。そして、このプレイ方式を、シーズンと呼びます。このモデルは、トーナメント・モデルと比べると制限が少ないですが、将来的にはトーナメント・モデルがサポートされるようになる可能性もあります。

Mathew Nelson 氏 (Robocode の作成者) と私は協力して作業し、Robocode ゲーム・エンジンを、単にプロセスとしてではなく、オブジェクトとして使用するための適切なインターフェースを作り上げました。このことが (その他の改善とあわせて)、1 回のバトルに要する時間を約 10 倍ほど短縮し、より大きいディビジョン・サイズに向けた展望を明確にしました。私が直面した第二の障害は、ロボット名に一貫性がないことでした。これを解決するため、RoboLeague では、編集可能な名前をロボットに付けることにしました。

設計の初期段階では、HTML 形式でユーザーに結果を提示して、それをインターネットで公開しやすくすることにしていました。しかし、Java プログラミングで Extensible Markup Language (XML) を使うという機会を生かすよう、その設計上の決定を転換しました。その結果、RoboLeague によって生成される永続的なデータはすべて XML 形式とし、HTML ページは XSL Transformations (XSLT) を利

用して生成することになりました。(XSLT ファイルはユーザーが編集できるので、HTML の「スキン」のようなものになっています。) 永続的な XML 構造の更新 (どうしても避けられないもの) を実装するためには、XSLT の XML から XML への変換を利用して、データの損失をできる限り避けるつもりです。内部的には、RoboLeague は次のようなコンポーネントで構成されます。

### • ロボット・ディレクトリー。

シンボリック名と、実際の Robocode ロボットのクラス名およびバージョンとの間のマップ。図 2 は、ロボット・ディレクトリーのウィンドウです。現在インストールされているロボットの名前、クラス名、およびバージョンが表示されています。

### • グループ分けエンジン。

Robocode ゲーム・エンジンを使用して、その上でのグループ分け (シンボリック名による単純なバトル) の実装を実行します。

### • ディビジョン・エンジン。

グループ分けエンジンを使用して、その上でのディビジョンのシーズンの実行を実装します。

### • リーグ・エンジン。

ディビジョン・エンジンを使用して、その上でのリーグのシーズンの実行と、GUI クラスを実装します。

グループ分け、ディビジョン、およびリーグは、別々の競技モデルであることに注意してください。それぞれは、エンジン・クラス、エンジン・リスナー・インターフェース、およびエンジンの設定データと結果データをカプセル化するデータ・モデル・クラスとして設計されています。RoboLeague では、これをデータ・モデル・オブジェクトを永続的に格納するいくつかの方法の 1 つと考えます。このアプローチにより、外部のプロジェクトを各エンジンに容易に組み込むことができます。将来的には、トーナメントのような競技モデルが追加されるかもしれません。

絶対ランキング (すべてのディビジョンを連結した結果) とは別に、各ロボットごとの結果ページが生成されます。このページでは、過去の 1 対 1 の対戦が考慮に入れられ、敵ロボットごとの合計が表示されます。図 3 は、リーグをセットアップするための GUI の一部です。

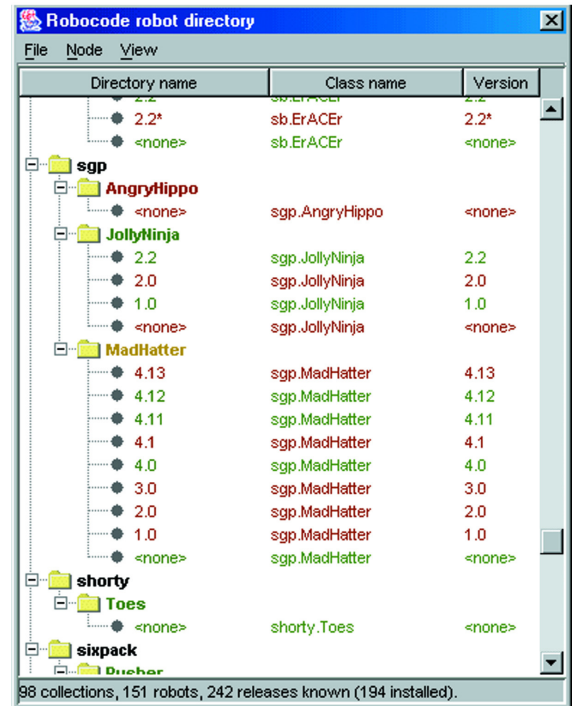


図 2. ロボット・ディレクトリー・ウィンドウ。よく知られたロボットが表示されている

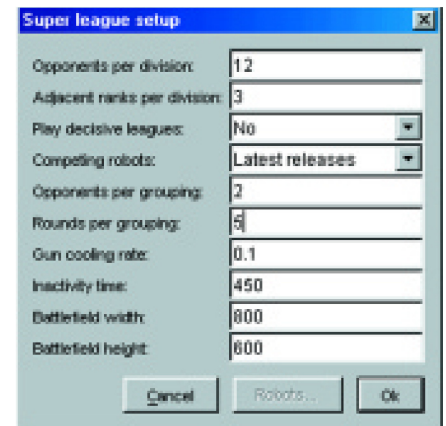


図 3. スーパー・リーグのセットアップ

図 4 と図 5 は、それぞれ、シーズンの結果と、個々のロボットの結果を示しています。すべてのロボットにそれぞれ強み (すべてというか、ほとんど) と弱点があることがはっきりわかります。結局、どのロボットもそれなりのちょっとした課題をもっています。

## リーグの主催またはリーグへの参加

www.cs.tu-berlin.de/~lulli/roboleague/ の Web ページから RoboLeague をダウンロードすれば、自分独自のプライベート・リーグをすぐに開始できます。別の方法として、Gladiatorial League などのリーグに参加できますし、自分が公式のリーグの管理者で



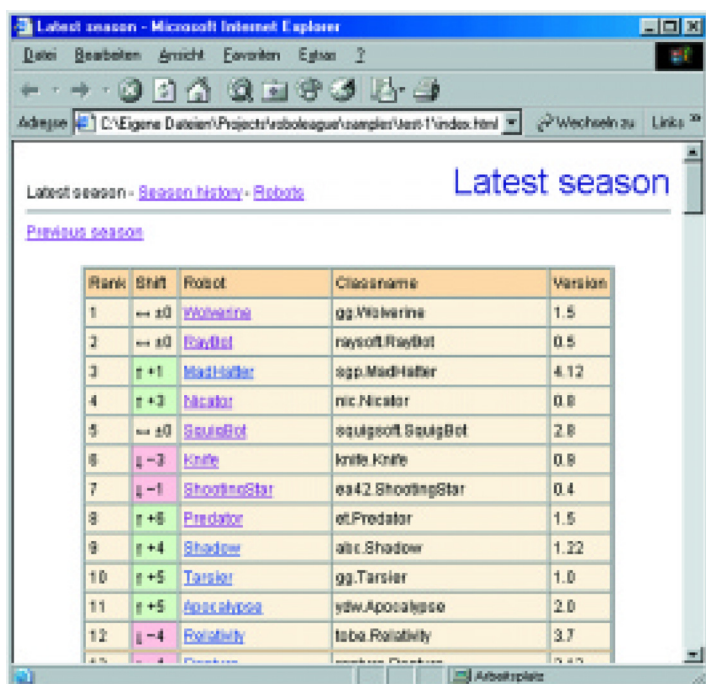


図 4. HTML による結果のフロント・ページ。最新のシーズンが表示されている。

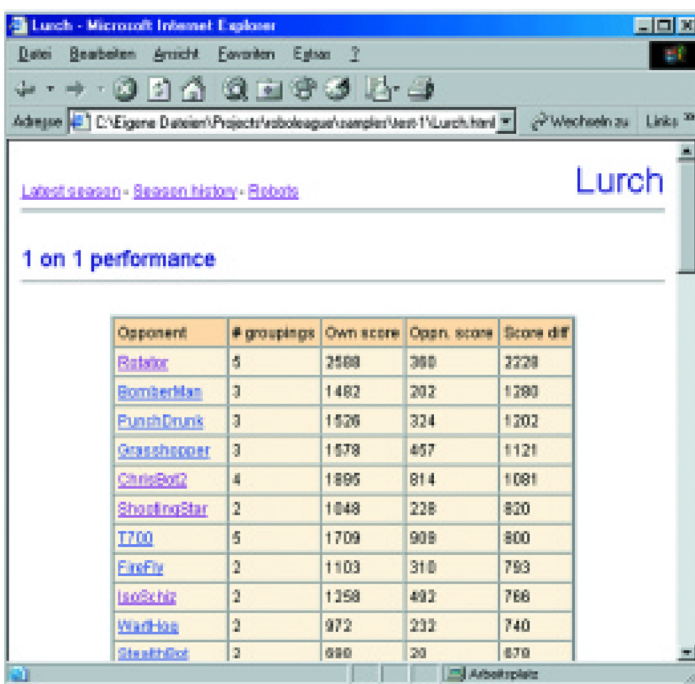


図 5. ロボット cs.Lurch についての HTML 詳細ページ

あることを宣言して、参加者にたとえば次のようなルールを配布すれば、独自のトーナメントを開始することもできます。

1. IBM alphaWorks の Robocode の Web サイトから Robocode をダウンロードする。
2. ロボットを作成する。
3. 自分のロボットをトーナメント・サイトにアップロードする。
4. 1 週間くらいのうちに記録を調べて、現在の結果と自分のロボットの地位を確かめる。
5. 自分のロボットを打ち負かした敵ロボットをダウンロードして、自分のロボットの戦略を微調整する。
6. 改良版のロボットをトーナメント・サイトにアップロードして、ロボットの地位が改善されるかどうか確かめる。

トーナメントに参加して、自分のロボットが競技でどんな成績を収めるか見てみたいと思いますか？ IBM alphaWorks の Robocode の Web ページを参照して、最新のトーナメントを探し、それに参加する方法を調べてください。

**Christian Schnell** 氏は、現在、ベルリン技術大学での勉強を終えようとしているところで、RDBM やその他のプロジェクトで働いてきました。

## 拡張可能/再利用可能なロボット Ray Vermette 著

私が最初に作ったロボットは、1 つの大きなクラス・ファイルとして記述しました。最初のうちはこれでもうまく動いていましたが、ロボットにいろいろな機能を追加していくにつれて、管理するのが困難になるどころまで膨れ上がってしまいました。というわけで、上位ロボットと対戦するチャンスを得たければ、もっと良い設計を考え出さなければならぬことがすぐにわかりました。

私が思い付いた解決策は、ロボットをモジュールに分割することでした。各モジュールは、再利用可能なコード断片で、それぞれロボットの動作の異なる側面を管理します。移動、レーダー、大砲の制御、そして標的を狙うことのそれぞれについて、1 つのクラスを作成しました。また、敵ロボットの情報を追跡するためのクラスや、すべての敵ロボットについての情報を格納するクラスも作成しました。間もなく、その他のクラスも加わっていきました。

各クラスを記述する際の私の目標は、それぞれのクラスをできる限り独立したものにすることでした。そうすれば、異なるロボットで各クラスを再利用することも、スキャンや移動や標的を狙うことなどの機能を書き直す必要なしに、メイン・ロボット・クラスの全体的な設計を変更することもできます。

このようなモジュラー設計の場合でも、メイン・ロボット・クラスのファイルには依然として多くのコードが散らばっています。コードのかかなりの部分は単に各種のモジュールを組み合わせるためのものであり、ロボットが実際に何を実行しているかが不明瞭になっています。

この状況をどのように解決したのでしょうか。私

は、AbstractBot という抽象ロボット・クラスを作成して、組み合わせるためのコードや、標準的な初期化コードや、その他の厄介な詳細部分を隠蔽することにしました。

AbstractBot クラスには、いくつかの便利なメソッドがあって、レーダーの操作、移動、標的の選択、狙いを定めること、弾丸を発射することなどを簡略化できます。今では、すべての考えられる標的を評価し、最善の標的を選び出し、レーダーを回転し、どれだけのエネルギーを使用するかを計算し、どのように動くかを判断する、といったコードを記述する代わりに、AbstractBot クラスを拡張して、次のようなコーディングをするだけでよいのです。

```
EnemyBot target = pickTarget();
scan(target);
ram(target);
blast(target);
```

あるいは、次のように書きます。

```
EnemyBot target = pickTarget();
scan(rammingBot);
evadeRam(rammingBot);
aimAndShoot(target);
```

ロボットが何を実行しているかは明白ですが、どのようにしてそれを実行しているかはわかりません。詳細な点は、AbstractBot や、レーダー、移動、標的の選択、弾丸の発射などを制御するその他のクラ

```

public class Intercept {
    public Coordinate impactPoint = new Coordinate(0,0);
    public double bulletHeading_deg;

    protected Coordinate bulletStartingPoint = new Coordinate();
    protected Coordinate targetStartingPoint = new Coordinate();
    public double targetHeading;
    public double targetVelocity;
    public double bulletPower;
    public double angleThreshold;
    public double distance;

    protected double impactTime;
    protected double angularVelocity_rad_per_sec;

    public void calculate (
        // Initial bullet position x coordinate
        double xb,
        // Initial bullet position y coordinate
        double yb,
        // Initial target position x coordinate
        double xt,
        // Initial target position y coordinate
        double yt,
        // Target heading
        double tHeading,
        // Target velocity
        double vt,
        // Power of the bullet that we will be firing
        double bPower,
        // Angular velocity of the target
        double angularVelocity_deg_per_sec
    )
    {
        angularVelocity_rad_per_sec =
            Math.toRadians(angularVelocity_deg_per_sec);

        bulletStartingPoint.set(xb, yb);
        targetStartingPoint.set(xt, yt);

        targetHeading = tHeading;
        targetVelocity = vt;
        bulletPower = bPower;
        double vb = 20-3*bulletPower;

        double dX,dY;

        // Start with initial guesses at 10 and 20 ticks
        impactTime = getImpactTime(10, 20, 0.01);
        impactPoint = getEstimatedPosition(impactTime);

        dX = (impactPoint.x - bulletStartingPoint.x);
        dY = (impactPoint.y - bulletStartingPoint.y);

```

```

        distance = Math.sqrt(dX*dX+dY*dY);

        bulletHeading_deg = Math.toDegrees(Math.atan2(dX,dY));
        angleThreshold = Math.toDegrees
            (Math.atan(ROBOT_RADIUS/distance));
    }

    protected Coordinate getEstimatedPosition(double time) {

        double x = targetStartingPoint.x + targetVelocity * time *
            Math.sin(Math.toRadians(targetHeading));
        double y = targetStartingPoint.y + targetVelocity * time *
            Math.cos(Math.toRadians(targetHeading));
        return new Coordinate(x,y);
    }

    private double f(double time) {

        double vb = 20-3*bulletPower;

        Coordinate targetPosition = getEstimatedPosition(time);
        double dX = (targetPosition.x - bulletStartingPoint.x);
        double dY = (targetPosition.y - bulletStartingPoint.y);

        return Math.sqrt(dX*dX + dY*dY) - vb * time;
    }

    private double getImpactTime(double t0,
        double t1, double accuracy) {

        double X = t1;
        double lastX = t0;
        int iterationCount = 0;
        double lastfX = f(lastX);

        while ((Math.abs(X - lastX) >= accuracy) &&
            (iterationCount < 15)) {

            iterationCount++;
            double fX = f(X);

            if ((fX-lastfX) == 0.0) break;

            double nextX = X - fX*(X-lastX)/(fX-lastfX);
            lastX = X;
            X = nextX;
            lastfX = fX;
        }

        return X;
    }
}

```

### リスト 3. Intercept クラス

スに都合よく隠蔽されているわけです。コードはずっと読みやすくなっており、その結果として、ずっと変更しやすくなっています。こうして、使命は達成されました。私は、Predator を打ち負かすことに専念すればよいだけです。

**Ray Vermette** 氏は、昼間は温厚なメインフレーム・プログラマー兼ローラー・プレーダー乗りで、夜は熱心な Robocode および Java の愛好者です。



### 標的の予測

Simon Parker 著

敵ロボットに狙いを定めて弾丸を命中させるには、将来の時点で敵ロボットがいると予測される場所に向けて弾丸を発射するアルゴリズムが必要です。このアルゴリズムは、直線方式 (Linear)、円形方式 (Circular)、および振動方式 (Oscillating) の標的予測 (predictive targeting) のために使用できます。実際、将来の時点で敵ロボットの位置を返す関数があれば、このアルゴリズムを使用できます。

このアルゴリズムでは、衝突ポイント、発射角度、衝突位置、および衝突時間を計算します。

このアルゴリズムは、割線法を実装して、衝突時間を数値的に解きます。衝突時間がわかれば、予測関数によって衝突位置を取得できます。そこで、その位置に向けて弾丸を発射します。

```

public class CircularIntercept extends
Intercept {
    protected Coordinate
        getEstimatedPosition(double time) {
        if(Math.abs(angularVelocity_rad_per_sec)
        <= Math.toRadians(0.1)) {
        return super.getEstimatedPosition(time);
        }

        double initialTargetHeading =
            Math.toRadians(targetHeading);
        double finalTargetHeading =
            initialTargetHeading +
            angularVelocity_rad_per_sec * time;
        double x = targetStartingPoint.x
            - targetVelocity /
            angularVelocity_rad_per_sec
            * (Math.cos(finalTargetHeading) -
            Math.cos(initialTargetHeading));
        double y = targetStartingPoint.y
            - targetVelocity /
            angularVelocity_rad_per_sec *
            (Math.sin(initialTargetHeading) -
            Math.sin(finalTargetHeading));

        return new Coordinate(x,y);
    }
}

```

リスト 4. CircularIntercept クラス

**リスト 3** の Intercept クラスは、ロボットが現在の位置から現在の速度で直線上を移動していることを前提としています。

Intercept クラスの優れている点は、円形の動きに対する発射角度を計算するために容易に再利用できることです。そのためには、Intercept クラスを継承する CircularIntercept クラスを記述して、getEstimatedPosition メソッドを上書きします。

**リスト 4** に、CircularIntercept クラスのコードを示します。

**リスト 5** は、Intercept クラスの使用例を示しています。このコードは、標的の現在位置、進行方向、および速度を既に計算しており、さらに発射する弾丸のパワーも計算してあることを前提としています。この発射方法が非常に優れていることは、私の 2 つのロボット (JollyNinja および MadHatter) で証明されています。Intercept を独自のクラスにカプセル化し、別の推測アルゴリズムの場合にもそれをサブクラス化することにより、他のロボットの弾丸が私のロボットに命中するよりも多く、他のロボットにこれらの弾丸を命中させることができます。



## ポリモアフィックな敵ロボットのキャッシュ

Christian Schnell 著

成功しているロボットは、情報のストアを保守しており、バトル中に重要な決定を下すためにいつでもアクセスできるようにしています。

これは、様々な理由で便利です。敵ロボットの移動パターンの分析から、敵の距離と残りエネルギーに基づく攻撃対象の決定に至るまで、様々な用途に利用できるからです。このヒントでは、ポリモアフィズム (多様性) を利用して、効率的で高速な敵ロボットのキャッシュを実装すると同時に、いつでも最新のオブジェクトを取得できるという便利さを実現する方法を説明します。このコードは、Rapture 3.0 ロボットで使用する予定です。

ポリモアフィズムは、適切なオブジェクト指向プログラミングのキー・コンポーネントであり、ロボットを作成するときにも利用できる強力なツールです。ポリモアフィズムとは、基本的に、別々の仕方で実装された 1 つの共通のメソッドがあり、呼び出し先の実際のクラスを気にせずに、単純にそのメソッドを呼び出せるということです。言い換えると、たとえば food.eat() を呼び出した場合、それは実際には steak.eat() または salad.eat() の呼び出しであるかもしれませんが、その振る舞いは異なっていますが、そのことを心配する必要はないということです。そのどちらも、food 型のクラスだからです。

この後のコード・サンプルでは、Robocode においてポリモアフィズムを実装して、強力な機能性と大幅な柔軟性を付加する方法をお見せします。

**図 6** は、このフレームワークで使用するオブジェクト構造の概要を示しています。

この構造の利点は明白です。Enemy オブジェクトに対する参照は、EnemyManager を通して取得できますが、そのとき、実際には EnemyLog キヤストが Enemy として返されます。EnemyLog オブジェクトは、Enemy インターフェースによって継承されたすべての責務を、ログ内の最初のオブジェクトに委譲するので、このオブジェクトを操作しているときには、最新の情報にアクセスしていることがわかります。

**リスト 6~9** では、この構造を利用しています。通常なら、Enemy インターフェースをもっと精巧なものにし、EnemyManager にもっと多くの機能を追加したいと考えるでしょうが、学習用という目的のために、ここで紹介するインターフェースは最小限のものにしてあります。それでは、楽しいロボット狩りを！

```

Intercept intercept = new Intercept();

intercept.calculate
(
    ourRobotPositionX,
    ourRobotPositionY,
    currentTargetPositionX,
    currentTargetPositionY,
    currentTargetHeading_deg,
    currentTargetVelocity,
    bulletPower,
    0 // Angular velocity
);

// Helper function that converts any angle
// into an angle between +180 and -180
// degrees.

double turnAngle = normalRelativeAngle
(intercept.bulletHeading_deg -
robot.getGunHeading());

// Move gun to target angle

robot.setTurnGunRight(turnAngle);
if (Math.abs(turnAngle) <=
intercept.angleThreshold) {
    // Ensure that the gun is pointing
    // at the correct angle

    if (
        (intercept.impactPoint.x > 0) &&
        (intercept.impactPoint.x <
        getBattleFieldWidth()) &&
        (intercept.impactPoint.y > 0) &&
        (intercept.impactPoint.y <
        getBattleFieldHeight())
    ) {

        // Ensure that the predicted impact
        // point is within the battlefield

        fire(bulletPower);
    }
}
}

```

リスト 5. Intercept クラスの使い方

```

package example.enemy;
// This class is used as an interface
// to define the common methods needed
// to define enemy information.

public interface Enemy {
    // Returns the enemy's name
    public String getName();

    // Gets the current energy of our enemy
    public double getEnergy();

    // Gets the velocity of our enemy
    public double getVelocity();
}

```

リスト 6. Enemy インターフェース



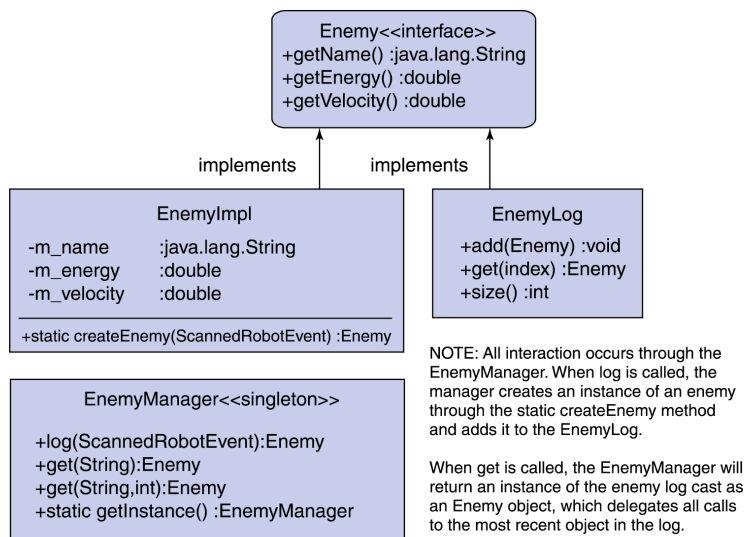


図6. UML ダイアグラム

Chris Shorrock 氏は、カナダのバンクーバーにある Make Technologies の上級 Java 開発者で、過去 4 年間にわたって Java プログラマーとして活動してきました。

## Robocode の戦略 Simon Parker 著

Robocode の戦略に関するこの手引きでは、Gladiator League の誕生の背景を紹介し、移動、攻撃、および情報収集に関するいくつかの戦略レベルについて説明します。

### Gladiatorial League

私が Robocode を見つけた時には、どのロボットが最高かを示す定期的な競技会はありませんでした。存在したトーナメントは、新しいバージョンのロボットがリリースされると、すぐに古くなってしまいました。私自身のロボットを開発する過程で私が実施したテストは、公に入手できるどの競技データよりも完全かつ最新でした。その頃、Christian Schnell 氏が RoboLeague をリリースし、対抗試合を自動化できるようになったので、私は独自にリーグ制の対抗試合を実施しました。私自身のロボットをテストする中で、他の人が Robocode Repository にアップロードした 50 を超えるロボットをダウンロードしました。それらのロボットを 10 体ごとのディビジョンに分割し、各ディビジョンについて、剣闘士 (Gladiator) スタイルの試合を 100 ラウンド実行しました。上位 2 つのロボットと、下位 2 つのロボットは、次のシーズンにはそれぞれ上位ディビジョンに昇格、および下位ディビジョンに降格させました。生成された HTML 形式の結果をシンプ

ルな Web サイトにアップロードするのは簡単な作業であり、次第に大きくなっていった Robocode コミュニティーのメンバーはそのサイトを見て、自分たちのロボットが当時の他のロボットに比べてどのような強さなのかを調べることができました。こうして、Gladiatorial League が誕生したのです。

リーグ制の競技形式には、積極的な特徴があります。第一に、これは、大量のロボットについて比較的小さい数のバトルで成り立つ対抗試合を実行するための方法です (各バトルは、実施するのにかなり時間がかかる)。第二に、機能の類似したロボットが結果的に同じリーグに入ることが多くなります。したがって、まだ始めたばかりのロボット作成者は、同じようなレベルのロボットと対戦して互角の戦いをするのができ、初心者レベルのロボットよりはるかに進化したロボットと戦って完敗するということがありません。

### 移動の戦略

移動の戦略には、次のようなレベルがあります。

1. **じっと動かない。**  
攻撃を受けやすくなります。一般には、良い戦略ではありません。
2. **直線上を移動する。**  
「定位置」標的合わせによる攻撃を回避できます。
3. **曲線上を移動する。**  
「定位置」および「直線方式」の標的合わせによる攻撃を回避できます。
4. **前後に移動して、振動するような動きをする。**  
「直線方式」および「円形方式」の標的合わせでは攻撃しにくくなりますが、「定位置」標的合わせでうまく攻撃できることがあります。

```

package example.enemy;

import robocode.*;

// A class to hold this package's local
// implementation of the enemy interface.

class EnemyImpl extends Object implements
    Enemy {
    // Members
    private String m_name = null;
    private double m_energy = 0;
    private double m_velocity = 0;

    // Constructor
    private EnemyImpl() {}

    // Set methods
    private void setName(
        String in_newName )
    { m_name = in_newName; }
    private void setEnergy(
        double in_newEnergy )
    { m_energy = in_newEnergy; }
    private void setVelocity(
        double in_newVelocity )
    { m_velocity = in_newVelocity; }

    // Get methods
    public String getName() {
        return m_name; }
    public double getEnergy() {
        return m_energy; }
    public double getVelocity() {
        return m_velocity; }

    // Method used to create an instance
    // of an enemy impl object.
    public static final Enemy createEnemy(
        ScannedRobotEvent in_event ) {

        EnemyImpl out_enemy = new EnemyImpl();

        out_enemy.setName( in_event.getName() );
        out_enemy.setEnergy(
            in_event.getEnergy() );
        out_enemy.setVelocity(
            in_event.getVelocity() );

        return out_enemy;
    }
}

```

リスト 7. EnemyImpl クラス

```

package example.enemy;

// A class that maintains a stack of all the enemy information
// objects that we have collected for a certain enemy. It is
// managed by the EnemyManager and can be referenced as an
// Enemy to maintain a reference to the most recent
// information for a certain enemy.

class EnemyCollection implements Enemy {
    /* constant that holds our maximum stack size - the number
    of Enemy information objects to maintain */
    protected static final int MAX_STACK_SIZE = 500;

    private Enemy[] m_information = new Enemy[ MAX_STACK_SIZE ];
    private int next = 0;
    private int size = 0;

    // our constructor, which takes in the first object in
    our collection; this ensures that any collection always
    has at least one entry. */
    public EnemyCollection( Enemy in_newEnemy ) {
        add( in_newEnemy );
    }

    // Adds an Enemy to our stack of objects.
    public void add( Enemy in_new ) {
        size = (size == MAX_STACK_SIZE) ? size : size + 1;
        m_information[next] = in_new;
        next = (next + 1) % MAX_STACK_SIZE;
    }

    // Method used to get a enemy represented by an index,
    // where 0 is the last element inserted.
    public Enemy get( int in_index ) {
        int index = next - in_index - 1;
        if( index < 0 ) index += MAX_STACK_SIZE;

        return m_information[ index ];
    }

    // Method that returns the number of Enemy objects
    // that we currently are storing.
    public int size() {
        return size;
    }

    // Enemy interface methods

    /* NOTE: By having this class implement enemy, we can use it
    * as an enemy object and be ensured that whenever it is
    * updated, we always have the most recent information */

    public String getName() {
        return get(0).getName();
    }

    public double getEnergy() {
        return get(0).getEnergy();
    }

    public double getVelocity() {
        return get(0).getVelocity();
    }
}

```

リスト 8. EnemyCollection クラス (ポリモアフィックな敵ロボット・キャッシュ)

```

package example.enemy;

import robocode.*;
import java.util.*;

// a class which is used to manage and maintain
// all of our enemy information objects.

public class EnemyManager {
    /* holds a reference to a SINGLETON instance of this class
    to ensure that it only ever gets instantiated once */
    private static EnemyManager SINGLETON = new EnemyManager();

    /* holds a map of all our enemy information objects
    as values with the enemies name's used as keys. */
    private Map m_allInformation = null;

    // private constructor used to hide unnecessary instantiation
    private EnemyManager() {
        // We want to use a Hashtable, since it is synchronized so
        // we need not worry about data integrity for the most part.
        m_allInformation = new Hashtable();
    }

    // Method that is used to get an instance of this class, as
    // it exists.
    public static EnemyManager getInstance()
    { return SINGLETON; }

    // Method used to log an enemy object in the manager
    public Enemy log( ScannedRobotEvent in_event ) {
        Enemy newEnemy = EnemyImpl.createEnemy( in_event );
        EnemyCollection col = (EnemyCollection)m_allInformation.get(
            newEnemy.getName() );
        if( null == col ) {
            col = new EnemyCollection( newEnemy );
            m_allInformation.put( newEnemy.getName(), col );
        } else {
            col.add( newEnemy );
        }

        return col;
    }

    // Method used to get a dynamic enemy object. By dynamic,
    // this means that calls to any of the methods will always
    // get the most recent data. This does not necessarily imply
    // that the data is current, it's just the most current.
    public Enemy get( String in_name ) {
        return (Enemy)m_allInformation.get( in_name );
    }

    // Method used to get a reference to an enemy at a certain
    // point in time.
    public Enemy get( String in_name, int in_time ) {
        EnemyCollection col = (EnemyCollection)m_allInformation.get(
            in_name );

        return (null == col) ? null : col.get( in_time );
    }

    // Method used to return the maximum number of cached items
    // we store on any Enemy at any given time.
    public static final int getCacheSize() {
        return EnemyCollection.MAX_STACK_SIZE;
    }
}

```

リスト 9. EnemyManager クラス (ポリモアフィックな敵ロボット・キャッシュ)



#### 5. ランダムな方向に移動する。

あらゆるレベルの標的合わせによる攻撃を減らすのに効果的ですが、攻撃を受けないような仕方では移動するのは困難です。

#### 6. 高度な移動

他のロボットに関するあらゆる種類のデータを利用して、移動すべき最善の位置を選択します。これは、ロボットの戦略において間違いなく最大の関心分野であり、同時に最も理解が限られている分野です。これは、最高のロボットとそれほど良くないロボットの違いを生む、主要な要因の1つです。

## 攻撃

標的合わせには、次のようなレベルがあります。

1. **定位**。標的ロボットの現在の位置に向けて攻撃します。これは、標的合わせの戦略として最も単純ですが、効率は最も悪く、敵ロボットが少しでも動けば弾丸は命中しません。
2. **直線方式**。標的が一定の速度で直線上を移動していると仮定して、標的の予測位置に向けて攻撃します。この方式の標的合わせは、特に標的が比較的近くにある場合に、非常に効果的です。ロボットのあらゆる動きは、短時間で見れば、直線で近似できるからです。とはいえ、この近似は、かなり早く信頼性が落ちます。
3. **円形方式**。標的が一定の速度かつ一定の角速度で移動していると仮定して、標的の予測位置に向けて攻撃します。これは、“直線方式”の標的合わせより少し精度が上がります。この標的合わせは、停止と移動を繰り返したり、前後に素早く移動したりするなど、速度を素早く変化させることにより回避できます。
4. **振動方式**。この戦略では、標的ロボットが継続的に前後に移動しているものと仮定します。この種の標的合わせを採用しているロボットは数少ないため、この種の動きをするロボットに対して、この標的合わせは非常に効果的です。
5. **移動パターン・マッチング**。このアプローチでは、ロボットの過去の動きを記録して、ロボットが反復されるパターンで動くことを仮定します。そして、予測される将来の位置に向けて攻撃します。この種の標的合わせは、これまでに取り上げたどの方式よりも命中する可能性が大きくなります。ただし、考えられる欠点として、しらみつぶしのサーチに要する処理時間が長くなることが挙げられます。

## 情報の収集

スキャンには、次のようなレベルがあります。

1. **単純スキャン**。単にレーダーが大砲と同じ方向を指すようにします。
2. **円形スキャン**。レーダーを継続的に回転させます。
3. **追跡スキャン**。レーダーを単一の標的にロックして追跡します。
4. **最速スキャン**。すべてのロボットをできるだけ素早くスキャンできるような仕方ではレーダーを動かします。

さらに高度なロボットでは、バトル・フィールド上の他のロボットについて収集できるデータが、移動と攻撃について高度に戦略的な判断を下す上で非常に重要になります。他のロボットについて取得できるデータには、たとえば次のようなものがあります。

- 位置
- 移動方向
- 移動速度
- 移動の角速度
- エネルギー
- 標的に与えたダメージの量 (バトル・フィールド上の他のロボットと比較して、最も簡単に破壊できそうなロボット (または、破壊するのが最も難しそうなロボット) に標的を合わせるのに役立つ)
- 敵ロボットが自分に与えたダメージの量

- 敵ロボットが破壊された回数と時期
- 敵ロボットが弾丸を発射した時期 (エネルギーの差を調べるという特殊な技法を使う)

この戦略リストは、決して完全なものではありません。Gladiatorial League 競技会の上位ロボットは、この記事で取り上げた戦略のほとんどをある程度利用しています。しかし、しばしば誰かが新しい考え方やシンプルな戦略を考案し、他のロボットはそれに対抗する戦略を考え出さなければならなくなります。最もシンプルなアイデアでも、大成功を収めることがあるのです。

**Simon Parker** 氏は、オーストリアのシドニーにある Neopraxis Pty. Ltd. の設計技術者で、生物医学システムの組み込みコードを記述しています。

## 関連情報

IBM alphaWorks Robocode:

[ibm.com/alphaWorks/tech/robocode](http://ibm.com/alphaWorks/tech/robocode)

IBM developerWorks Java テクノロジー・ゾーン:

[ibm.com/developerWorks/java](http://ibm.com/developerWorks/java)

Robocode Central:

[robocode.net](http://robocode.net)

RoboLeague:

[user.cs.tu-berlin.de/~lulli/roboleague](http://user.cs.tu-berlin.de/~lulli/roboleague)

## WebSphere Developer Domain - WebSphere のすべての必要のために

IBM WebSphere Developer Domain (WSDD) は、WebSphere ソフトウェア・プラットフォームの学習曲線のすべての段階を経験するように開発者を手引きするべく計画された、オンライン技術資料を提供しています。WSDD ライブラリーには、初心者向けから熟練者向けまで、何百という技術記事、チュートリアル、ホワイトペーパー、最良事例、レッドブック、FAQ、技術上のヒント、および製品ドキュメンテーションが収録されています。

クイック入門のために、初心者には WebSphere for Newcomers ゾーンを活用できます。このゾーンは、基本的な説明とチュートリアルを提供しています。新しい課題にチャレンジする用意のできている熟練者は、Lessons from the Experts シリーズや、さらに高度なチュートリアルから益を得られます。

WebSphere に関する最新の話題のスクープ記事を読みたい場合は、WebSphere Developer Technical Journal を調べてください。この月刊オンライン・マガジンは、WebSphere ソフトウェア・プラットフォームを使用する開発についての詳細なハウ・ツー記事を提供します。このオンライン・マガジンは、e-business アプリケーションを作成するために IBM ツールを使用している開発者向けに発行されています。WebSphere Developer Technical Journal は、世界中の開発者が無料で WSDD から入手できます。

WSDD は、製品ダウンロード、ハウ・ツー情報、Web セルフ・サポートなど、あらゆる技術情報や資料を 1 か所で入手できるサイトです。

さっそく、[ibm.com/WebSphere/developer](http://ibm.com/WebSphere/developer) の WebSphere Developer Domain にアクセスしてみてください。

## WebSphere Developer Domain からのメッセージ