Robocodeの達人たちが明かす秘訣: 弾丸を追跡する

大学教育のニーズを満たす

Ray Vermette

2001年 7月 01日

敵に合わせて最善の標的合わせ技法を使用するには、どの技法が有効で、どの技法が有効でないかについて、正確な統計を得る必要があります。このヒントでは、Ray Vermetteが、まさにそのことを行う BulletTracker クラスを実装する方法を紹介します。

ロボットの機能と複雑さが増してくると、おそらく、いくつかの異なる標的合わせの技法を採用するようになるでしょう。敵に合わせて最善の標的合わせ技法を選ぶには、発射した弾丸と弾丸の命中を追跡する必要があります。これは、 `AdvancedRobot` クラスの `onBulletHit` イベント・ハンドラーをオーバーライドすることによって実行できます。しかし、この記事では、カスタム・イベントを利用することによって、まったく同じことを、より少ない行数で、かつ、より高い精度で実現する方法を紹介します。

onBulletHit: 優れた方法ではない

`onBulletHit` イベント・ハンドラーを使用することの問題点は、弾丸がもともとの標的に命中したかどうかを教えてくれないということです。たとえば、自分のロボットがsample.Wallsに向けて弾丸を発射したとき、代わりにsample.SpinBotに命中したとすると、sample.SpinBotの `BulletHitEvent` を受け取ります。しかし、その場合に本当に知りたいのは、sample.Wallsに命中しなかったということではないでしょうか。 `Bullet` オブジェクトと標的の名前を追っていけば、弾丸が狙いどおりに命中したのか、偶然に当たったのかを判別できます。しかし、そのためには、弾丸と標的の名前を格納するために余分のデータ構造を用意し、弾丸と標的の名前を挿入し、検索し、削除するために余分のコードを追加する必要があります。これは1つの方法ですが、決してエレガントな解決策ではありません。

より良い方法

その代わりに、同じことを実行するカスタム・イベントを使用してみましょう。

カスタム・イベントを作成するには、まず `Condition` クラスが必要です。 `Condition` クラスには、 `public boolean test()` という抽象メソッドがあり、これをオーバーライドする必要があります。 `test()` メソッドは、カスタム・イベントを発生させるべきときには真を返し、そうでない

ときには偽を返します。リスト1にある `BulletTracker` クラスは、必要な `Condition` クラスの考えられる実装の1つを示しています。

リスト1. BulletTrackerクラス

```
package bt;
import robocode.*;

public class BulletTracker extends Condition {

    private long                expectedTimeOfImpact;
    private String              targetName;
    private Bullet              bullet;
    private AdvancedRobot       myRobot;
    private int                 aimMethod;
    private boolean             hitTarget;


    public BulletTracker(
            AdvancedRobot ar,
            Bullet b,
            String targetName,
            long timeToImpact,
            int aimMethod) {

        if (b != null) {

            this.myRobot                = ar;
            this.bullet                 = b;
            this.targetName             = targetName;
            this.expectedTimeOfImpact   = ar.getTime() + timeToImpact;
            this.aimMethod              = aimMethod;
            this.hitTarget              = false;
            myRobot.addCustomEvent(this);
        }

    }


    public long     getExpectedTimeOfImpact() { return expectedTimeOfImpact; }
    public String   getTargetName()           { return targetName; }
    public Bullet   getBullet()               { return bullet; }
    public int      getAimMethod()            { return aimMethod; }
    public boolean  hitTarget()               { return hitTarget; }


    public boolean test() {

        if (targetName.equals(bullet.getVictim())) {
            hitTarget = true;
            myRobot.removeCustomEvent(this);
            return true;
        }

        if (bullet.getVictim() != null) {
            myRobot.removeCustomEvent(this);
        }

        if (expectedTimeOfImpact <= myRobot.getTime()) {
            hitTarget = false;
            myRobot.removeCustomEvent(this);
            return true;
        }

        return false;
    }
```

```
}
```

BulletTracker 条件は、 AdvancedRobot への参照、追跡するべき弾丸への参照、狙っている標的の名前、計算した弾丸が命中するまでの時間、および使用した標的合わせ技法に対応する整数を必要とします。弾丸がヌルでなければ (ヌル弾丸が発生するのは、大砲の熱が0より大きいとき、またはラウンドの開始時に、ロボットが弾丸を発射しようとした場合です)、これらのパラメーターが格納されて BulletTracker オブジェクトがカスタム・イベントとして追加されます。

それ以降、Robocodeエンジンは、この BulletTracker オブジェクトの test() メソッドを各動作ごとに呼び出して、カスタム・イベントを発生させるべきかどうかを調べます。弾丸が狙いどおりの標的に命中すると、 hitTarget にtrueが設定され、カスタム・イベントが削除され、 test() メソッドはtrueを返します。その結果、カスタム・イベントが AdvancedRobot に送られます。弾丸が当たったロボットの名前が狙った標的の名前と違っており、ヌルでない場合は、その弾丸は、狙った標的に向かう途中で偶然に別のロボットに当たったということです。 timeToImpact の時間が既に経過し、弾丸が狙った標的にまだ命中していない場合は、 BulletTracker は弾丸が外れたと見なします。 hitTarget にfalseが設定され、カスタム・イベントが削除され、メソッドはtrueを返します。その結果、カスタム・イベントが AdvancedRobot に送り返されます。弾丸が外れたことをチェックするために、この BulletTracker クラスはタイムアウト (expectedTimeOfImpact) を使用していますが、その代わりに bullet.isActive() をチェックしてもまったく問題ありません。タイムアウトを使用することの利点については、後ほど説明します。

カスタム・イベントを組み込む

BulletTracker クラスを使用するには、 AdvancedRobot 内の2箇所にコードを追加する必要があります。まず、ロボットが弾丸を発射するセクションに、次のようなコードを追加します。

リスト2. BulletTrackerのインスタンス生成

```
Bullet bullet = fireBullet(firePower);
BulletTracker bt =
  new BulletTracker(this, bullet, targetName, timeToImpact, aimMethod);
```

次に、onCustomEvent ハンドラーに、次のようなコードを追加します。

リスト3. OnCustomEventハンドラー

```
Condition condition = e.getCondition();
if (condition instanceof BulletTracker) {

    BulletTracker bt = (BulletTracker)condition;

    if (bt.hitTarget()) {
        if (bt.getAimMethod() == AIM_TECHNIQUE_1) {
            /* Increment technique 1's hit count */;
        }
        if (bt.getAimMethod() == AIM_TECHNIQUE_2) {
            /* Increment technique 2's hit count */;
        }
    }
}
```

`BulletTracker` オブジェクトは、カスタム・イベントとしての自らの追加と削除を処理するため、私たちが気を遣う必要があるのは、弾丸を発射する時点で `BulletTracker` を作成することと、 `BulletTracker` カスタム・イベントが発生した時点で命中か外れかに応じて処理することだけです。弾丸と標的の名前のベクトルまたはリストを維持する必要はありません。`BulletTracker` クラスと、Robocodeのイベント・メカニズムが、そのあたりの厄介な作業を自動的に処理してくれるのです。そのため、こんなにシンプルです。

さらにもう1歩先へ進める

この例の `BulletTracker` では、 `expectedTimeOfImpact` を使って弾丸が外れたかどうかを判断しているため、このことを利用して、弾丸をかわすロボットを検知して反撃することができます。たとえば、次のようにします。

1. 200ピクセル離れた位置にいる完全に停止した標的に向けて弾丸を発射する。
2. 計算された `timeToImpact` は15ゲーム刻時 (tick) であり、標的の現在位置に向けてまっすぐに弾丸を発射する。
3. `BulletTracker` オブジェクトを作成して、弾丸のゆくえを追跡する。
4. 弾丸が発射された後、標的は左方へ移動して、やってくる弾丸をかわす。
5. 15ゲーム刻時が経過した時点で、 `BulletTracker` はこの弾丸が外れたと判断する。

このとき、標的の最初のxおよびy座標を `BulletTracker` クラスに追加しておけば、攻撃を受けたときの反応として標的が実際に移動した方向と速度を計算できます。標的が弾丸を同じ方法でかわすと予想される場合は、これらの反応を格納しておき、後ほどその情報を使って標的合わせをすれば、弾丸をかわす敵ロボットに命中させることができます。

また、命中と外れのカウンターをインクリメントするコードを `BulletTracker` クラスに移せば、`onCustomEvent` ハンドラーに必要なコードを減らすことができます。

それで、どうしますか?

サンプル・ロボットをダウンロードして、 `BulletTracker` クラスの動きを観察してください。物事を簡単にするために、サンプル・ロボットは移動せず、1対1の対戦でのみ機能します。しかし、反復によって標的に当てる方式と、直線方式の標的合わせの、2通りの標的合わせ技法を実装しています。 `BulletTracker` によって、発射された弾丸が追跡され、それぞれの標的合わせ技法ごとの命中カウンターが更新されます。各ラウンドが終了すると、発射された弾丸の数と、命中した数が、コンソールに出力されます。

それでは、BulletTrackerを動かして、何かのロボットを攻撃してみましょう!

関連トピック

- BulletTrackerの実例のソース・コードをダウンロードできます。
- Robocode関連記事リンク集 の全記事をお読みください。このページは、新しいヒントが入手可能になるたびに更新されます。
- Robocodeの生みの親、Mathew Nelson氏が、 Robocodeの公式サイトを開設しています。Robocodeを真剣に検討したい人にとって、このサイトは良い入門となります。このサイトからは、Mathew Nelson氏が進行役をしている ディスカッション・グループ にも参加できます。要望の高い機能の「最新状況」をリストしたto-do list (今後の計画リスト) もチェックしてください。
- Christian Schnellによる RoboLeague は、Robocodeのためのリーグおよびシーズン・マネージャーです。これを利用すると、どんなグループ分けでも実際に試合を実行し、その結果を管理し、HTMLの状況報告書を生成できます。
- 「闘え、Robocode (ロボコード)」(developerWorks、2002年1月) では、Robocodeの武装を解きほぐし、カスタマイズされた軽量で平均的な戦闘マシンの構築を始める手ほどきをします。
- 「闘え、Robocode (ロボコード): 第2ラウンド」(developerWorks、2002年5月)では、Sing Li氏が、高度なロボットの構築とチーム・プレイに目を向けます。
- その他のJava関連の参考資料は、 developerWorks の Java technologyゾーンで参照できます。