

# U-Mart System Version 2 による U-Mart エージェントの作成

京都大学 喜多 一  
kita@media.kyoto-u.ac.jp

平成 17 年 7 月 8 日

## 1 はじめに

U-Mart の取引エージェントは U-Mart サーバとインターネットを経由して SVMP プロトコルで通信をする必要があるなど、これを最初から作ることは初心者には決して容易ではありません。また、作ったエージェントを評価し、改良するためには手元で U-Mart サーバを動かす必要があることも U-Mart のエージェント作成を難しくしています。このため、U-Mart プロジェクトでは、より簡単にエージェントを記述するために取引エージェントのための標準クラスを提供しています。このクラスを用いることにより、直接 SVMP を用いるエージェントことに比べれば機能は限定されますが、取引戦略を簡単に実装することができます。

## 2 U-Mart System Version 2 について

U-Mart System Version 2 では表 1 に示す 3 種類 6 本のプログラムを提供しています。このうち取引戦略の開発には「ネットワーク版市場サーバ」を使います。

U-Mart System Version 2 では取引エージェントのためのクラス（以下、戦略クラスと呼びます）を用いるエージェントについては以下の手順を繰り返すことで取り扱います。

1. 取引エージェント自身は市場サーバに組み込んで動作させます。
2. 毎回の注文受け付けの際に市場サーバは、まず組み込まれた各エージェントからの注文を順次照会します。
3. 次に一定時間、ネットワーク経由でのエージェント（主に取引端末を利用するヒューマンエージェント）の注文を受け付けます。
4. 注文受け付け時間がすぎると板寄せを行い約定など必要な処理を行ないます。

## 3 エージェントの作成から実行まで

### 3.1 エージェントの作成

エージェントはその取引戦略を書いたクラスのコードとして記述します。具体的には Strategy クラスを継承したクラスとして作成し、ソースコードおよびコンパイルした class ファイルを UMartSystem の strategy フォルダに置きます。コードの具体的記述については後述します。

表 1: U-Mart System Version 2 が提供するプログラム

プログラム	英語版	日本語版	機能・備考
スタンドアロン版シミュレータ	MarketSimulator_en.exe	MarketSimulator_jp.exe	単独で動作する U-Mart シミュレータです
ネットワーク版市場サーバ	MarketServer_en.exe	MarketServer_jp.exe	ネットワーク経由でクライアントからの接続を受ける取引所サーバです。
ネットワーク版取引端末	TradingTerminal_en.exe	TradingTerminal_jp.exe	ネットワーク版市場サーバに接続して取引するための端末です。

## 3.2 コンパイル

ここでは strategy フォルダの下にサンプルとして提供されている MySRandomStrategy.java を例に示します。コンパイル方法は、UMartSystem フォルダに移動したのち、

```
javac -classpath UMartSystem.jar strategy/MySRandomStrategy.java
```

とします。かならず UMartSystem フォルダでコンパイルすることに注意してください。

例として添付されている戦略クラスを雛形として独自の取引戦略クラスを作る際にはクラス名の扱いに注意してください。以下の 4 つが一致する必要があります。

- クラス名
- コンストラクタ
- ソースコードのファイル名
- エージェント定義ファイル（後述）での記述

## 3.3 エージェント定義ファイルの準備

次に出来上がった class ファイルをサーバで読み込むための準備を行ないます。このためにはサーバに読み込ませるエージェント定義のファイルを準備しなければなりません。例を ExampleForMyStrategies.csv に示します。エクセルでこのファイルを読み込んで調整します<sup>1</sup>。

このファイルの 2 行目が例として添付されている MySRandomStrategy を、3 行目が MyDayTradeStrategy を使うエージェントの記述です。それぞれ 6 列目に戦略クラスを指定しています。また 8 列目はこれらのエージェントが使う乱数の初期値です。4 行目以降はシステムに予め組み込まれているエージェントを使用するための記述です。

## 3.4 サーバの実行

取引戦略のクラスのコンパイル、エージェント定義ファイルの準備ができたならサーバを実行します。エージェント定義ファイルを読み込んでサーバを実行する手順は以下のとおりです。

1. MarketServer\_ja.exe をクリックしてサーバを起動します。

<sup>1</sup>もちろんテキストエディタでもかまいませんし、適当な CSV 形式ファイル用のエディタでもかまいません。

2. 「設定」ダイアログにて、「エージェント設定」ボタンをクリックします。
3. 「Agent Setting」ダイアログにて、「ファイル」ボタンをクリックします。
4. ファイル選択画面にて、ExampleForMyStrategies.csv を選択します。
5. 「Agent Setting」ダイアログにて、内容を確認後、「OK」ボタンをクリック。
6. 「設定」ダイアログにて、「OK」ボタンをクリック

これでエージェントの戦略クラスを読み込んでサーバが起動します。

### 3.5 実験の実施の Tips

エージェントの戦略ファイルを開発したり、シミュレーションを実行したりする際のヒント (Tips) をいくつか紹介します。

- 戦略クラスでの標準出力の利用：戦略の開発などで、その動作を確認するためには Strategy クラスの message メソッドを使って状況を確認する出力を生成します。出力は、サーバ画面の「標準出力」というタブにおいて、エージェントリストからエージェントを選択することにより見ることができます。
- ログ：サーバを実行すると UMartSystem フォルダの下に 'UMART...' というフォルダが自動的に生成され、そこに実行結果がログファイルとして出力されます。すべて CSV 形式で出力されますのでエクセルなどで分析できます。
- 乱数系列：シミュレーションの結果は取引所が使う乱数（約定する注文のタイプレックに使用します）、各エージェントが使用する乱数の系列に依存します。エージェントの評価には用いる乱数の系列を変えたいいくつかのシミュレーションを行う必要があります。
- 現物系列：シミュレーション結果は用いる現物の系列にも依存します。UMartSystem では J30 と呼ばれる株価指数の実際のデータ（日々の終値）を 10 年分程度提供しています。実際の実験ではこのデータの一部を用います。系列のどの部分を利用するかを変えながら実験します。  
現物情報のスタート点として取り得る値は  $120$  から  $2444 - (\text{取引日数}) \times (1 \text{ 日あたりに板寄せ回数})$  です。
- エージェントの組成：エージェント定義ファイルに書かれているエージェントは UMartSystem の標準エージェントセットとして提供されているものです。これらのエージェント数や乱数の初期値などを変えて実験することもできます。

### 3.6 エージェントの戦略の形式的記述

UMartSystem Version 2 では U-Mart エージェントの戦略は以下のような形式的な記述に基づいています。

エージェントが毎期に観測できる入力：以下の 5 種類の入力を考えます。

- 現物価格の時系列情報：直前の期から 120 期前の期までの現物価格。
- 先物価格の時系列情報：直前の期から 60 期前の期までの先物価格。ただし、市場が開く前の期では現物価格が代入されている。また、取引が成立しなかった場合は  $-1$  が代入されている。



図 1: J30 時系列データ

- そのエージェントの現在のポジション．買い越している時は正值，売り越しているときは負値です．
- そのエージェントの余裕金額．
- 期日までに残っている板寄せ回数．

エージェントが決定する出力：以下の 3 種類の出力です．

- 指し値 (成行き注文はできません)
- 注文数量
- 注文の種類 (「売り (値は 1)」か「買い (値は 2)」か「注文しない (値は 0)」か)

したがって，エージェントの戦略とはこれらの観測入力から出力を決定する関数あるいは手続きであると考えられることになります．

もちろん，当該期の情報だけを用いて出力を決定してもよいし，エージェント内にインスタンス変数を確保してそれまでの期の情報を何らかの形で保持し，それも考慮して出力を決めてもかまいません．

### 3.7 エージェントの戦略を記述するクラス

U-Mart System Version 2 では前節で述べたような形式的なエージェントの記述法に基づいた戦略を `~Strategy.java` という Java のソースコードに記述します．記述例をリスト 1 として示します．

1 回の板寄せ当たり，最大 1 つの注文を出す場合は `getOrder` というメソッドに取引戦略を実装します．

`MySRandomStrategy` という戦略は直近の現物価格 (Spot Price) の周辺にランダムに指値を設けながらランダムに売り買いする戦略です．以下，ソースコードの概要を見てゆきます．

- 先頭の `'package strategy;'` はこのクラスが `strategy` というパッケージに属することを示します．パッケージ名とソースコードが置かれるフォルダ名を一致させる必要があります．
- 3 行目がクラス名の記述です．このクラスは外部から参照されるため `public` という修飾子をつけます．また，このクラスは既に定義されているクラス `Strategy` を拡張したサブクラスであることを `'extends Strategy'` で示します．

- 以下、この戦略内で用いる定数を定義しています。Java のコード化の 慣習で定数を表す名前は大文字（単語間は ‘\_’）で書きます。定数は クラス内で唯一でその変更を許さないの修飾子 ‘static final’ を付けます。ここでは戦略が用いるさまざまなパラメータなデフォルト値を設定しています。
- 次にこのクラスのインスタンス変数が定義されます。先に定義した デフォルト値を初期値として設定しています。インスタンス変数は外部からの参照を許さないように private 修飾子を付けています。
- ‘public MySRandomStrategy(int seed) {’ で始まる 3 行が コンストラクタです。インスタンスの生成に際して、Random クラスのインスタンスを 初期値 seed を与えて生成しています。
- 次に ‘getOrder()’ というメソッドを定義しています。このメソッド はサーバが各エージェントが注文を照会する際に呼ばれます。引数を含めて、この 形で定義してください。
- メソッド getOrder() の戻り値は Order クラスのインスタンスです。戻り値を返すために Order クラスのインスタンスを生成します。
- 現物価格系列の配列 spotPrices から直近の価格を取り出します。メソッド getLatestPrice() はこのためのメソッドで親クラス Strategy で定義されているものです。価格が未定義の場合は -1 が設定されるので この場合には名目的な価格を設定します。
- 出すべき注文の売り買いを order.buysell にランダムに設定します。fRandom.nextInt(2) は乱数系列から次の乱数を 0 以上 2 未満（すなわち 0 か 1 か）の一樣乱数として取り出すものです。売り (1) か買い (2) にするために 1 を足します。
- 注文の売買がポジションを規定値以上に大きくするものである場合は注文を出さないことにして戻ります。
- 指値 (order.price) を直近の現物価格に正規乱数を加えて決定します。正規乱数を用いるため指値が負になることがあります。指値が正になるまで while ループを繰り返すようにしてあります。
- 数量 (order.quant) を最小値、最大値間の一樣乱数として定めます。
- 決定した注文や、その際に参照している直近の現物価格を画面に書き出します。
- order を戻り値としてメソッドを終了します。
- ‘setParameters()’ メソッドはエージェント設定ファイルから引き渡されたパラメータをインスタンス変数に設定するメソッドです。詳細は省略します。

## 注意

- 戦略のなかでの情報を表示するには “message(文字列);” というメソッドの呼び出しを TestStrategy.java の中のメソッド getOrder の中に書き込みます。引数のクラスは String です。
- 乱数を使う際にはその初期化にコンストラクタの引数 “seed” を使ってください。これは実行コマンドで与えられる乱数の種に相当する値が渡されています。

```

1:  /*
2:  SRandomStrategy.java                2002/01/31
3:
4:  A class describing an agent's strategy, which buys or sells randomly.
5:  The limit price is set randomly around the latest spot price, and
6:  quantity of the order is set randomly within a prescribed range.
7:  Position of the agent is also considered in decision making.
8:  */
9:
10: package strategy;
11:
12: import java.util.*;
13:
14: public class MySRandomStrategy extends Strategy {
15:
16:     /** Default value of Order Price Width */
17:     public static final int DEFAULT_WIDTH_OF_PRICE = 20;
18:
19:     /** Default value of Maximum Order Volume */
20:     public static final int DEFAULT_MAX_QUANT = 50;
21:
22:     /** Default value of Minimum Order Volume */
23:     public static final int DEFAULT_MIN_QUANT = 10;
24:
25:     /** Default value of Maximum Position */
26:     public static final int DEFAULT_MAX_POSITION = 300;
27:
28:     /** Default Value of the Price when All the Previous Prices Are Not Defined */
29:     public static final int DEFAULT_NOMINAL_PRICE = 3000;
30:
31:     /** Random Number Generator */
32:     private Random fRandom;
33:
34:     /** Width of Order Price */
35:     private int fWidthOfPrice = DEFAULT_WIDTH_OF_PRICE;
36:
37:     /** Maximum Value of Order Volume */
38:     private int fMaxQuant = DEFAULT_MAX_QUANT;
39:
40:     /** Minimum Value of Order Volume */
41:     private int fMinQuant = DEFAULT_MIN_QUANT;
42:
43:     /** Maximum Position */
44:     private int fMaxPosition = DEFAULT_MAX_POSITION;
45:

```

リスト 1 ( MySRandomStrategy.java )

```

46:  /** Price when All the Previous Prices Are Not Defined */
47:  private int fNominalPrice = DEFAULT_NOMINAL_PRICE;
48:
49:
50:  public MySRandomStrategy(int seed) {
51:      fRandom = new Random(seed);
52:  }
53:
54:  public Order getOrder(int[] spotPrices, int[] futurePrices,
55:                        int pos, long money, int restDay) {
56:      Order order = new Order();
57:      // Scan a well defined latest futures price. If it is not available,
58:      // a nominal value is used.
59:      int prevPrice = getLatestPrice(spotPrices);
60:      if (prevPrice == -1) {
61:          prevPrice = fNominalPrice;
62:      }
63:      // Submit a random order with a random price around the latest spot price.
64:      order.buysell = fRandom.nextInt(2) + 1;
65:      // Cancel decision if it may increase absolute value of the position
66:      if (order.buysell == Order.BUY) {
67:          if (pos > fMaxPosition) {
68:              order.buysell = Order.NONE;
69:              return order;
70:          }
71:      } else if (order.buysell == Order.SELL) {
72:          if (pos < -fMaxPosition) {
73:              order.buysell = Order.NONE;
74:              return order;
75:          }
76:      }
77:      while (true) {
78:          order.price = prevPrice + (int) (fWidthOfPrice * fRandom.nextGaussian());
79:          if (order.price > 0)
80:              break;
81:      }
82:      order.quant = fMinQuant + fRandom.nextInt(fMaxQuant - fMinQuant + 1);
83:
84:      message("MySRandomStrategy: buysell = " + order.buysell
85:             + ", price = " + order.price
86:             + ", volume = " + order.quant
87:             + " (prevSpotPrice = " + prevPrice + " )");
88:
89:      return order;
90:  }
91:
92:  /* (non-Javadoc)
93:   * @see strategy.UBaseStrategy#setParameters(java.lang.String[])
94:   */
95:  public void setParameters(String[] args) {

```

## リスト1 ( MySRandomStrategy.java ) のつづき

```
96:     super.setParameters(args);
97:     for ( int i = 0; i < args.length; ++i ) {
98:         StringTokenizer st = new StringTokenizer(args[i], "=" );
99:         String key = st.nextToken();
100:        String value = st.nextToken();
101:        if ( key.equals("WidthOfPrice") ) {
102:            fWidthOfPrice = Integer.parseInt(value);
103:        } else if ( key.equals("MinQuant") ) {
104:            fMinQuant = Integer.parseInt(value);
105:        } else if ( key.equals("MaxQuant") ) {
106:            fMaxQuant = Integer.parseInt(value);
107:        } else if ( key.equals("MaxPosition") ) {
108:            fMaxPosition = Integer.parseInt(value);
109:        } else {
110:            message("Unknown parameter:" + key + " in SRandomStrategy.setParameters");
111:        }
112:    }
113: }
114:
115: }
```

リスト 1 ( MySRandomStrategy.java ) のつづき



```

1:  /*
2:  2: A class describing an agent's strategy, which simultaneously submits a buying
3:  3: order with a limit price lower than and a selling order higher than the
4:  4: previous price by a prescribed spread.
5:  5: The quantity of the order is set randomly within a prescribed range.
6:  6: Position of the agent is also considered in decision making.
7:  7:  */
8:
9:  package strategy;
10: import java.util.*;
11: public class MyDayTradeStrategy extends Strategy {
12:
13:     /** Default value of Order Price Width */
14:     public static final int DEFAULT_WIDTH_OF_PRICE = 20;
15:
16:     /** Default value of Maximum Order Volume */
17:     public static final int DEFAULT_MAX_QUANT = 50;
18:
19:     /** Default value of Minimum Order Volume */
20:     public static final int DEFAULT_MIN_QUANT = 10;
21:
22:     /** Default value of Maximum Position */
23:     public static final int DEFAULT_MAX_POSITION = 300;
24:
25:     /** Default value of Spread Ratio */
26:     public static final double DEFAULT_SPREAD_RATIO = 0.01;
27:
28:     /** Random Number Generator */
29:     private Random fRandom;
30:
31:     /** Width of Order Price */
32:     private int fWidthOfPrice = DEFAULT_WIDTH_OF_PRICE;
33:
34:     /** Maximum Value of Order Volume */
35:     private int fMaxQuant = DEFAULT_MAX_QUANT;
36:
37:     /** Minimum Value of Order Volume */
38:     private int fMinQuant = DEFAULT_MIN_QUANT;
39:
40:     /** Maximum Position */
41:     private int fMaxPosition = DEFAULT_MAX_POSITION;
42:
43:     /** Spread Ratio */
44:     private double fSpreadRatio = DEFAULT_SPREAD_RATIO;
45:

```

リスト 2 ( MyDayTradeStrategy.java )

```

46:
47:  /* Constructor for Agent Initialization */
48:
49:  public MyDayTradeStrategy(int seed) {
50:      /* If you need some initialization of e.g., instance variables
51:         write here.
52:      */
53:      fRandom = new Random(seed); // Initialization of random sequence.
54:  }
55:
56:  /* Method of making order. If you submit a single order in one period,
57:     keep it as it is. If you want to submit several orders in one period,
58:     prepare methods other than getOrder and repeat orderRequest.
59:  */
60:  public void action(int[] spotPrices, int[] futurePrices,
61:                    int pos, long money, int restDay) {
62:      /* Class order has instance variables as follows
63:         order.price: limit price,
64:         order.quant: quantity,
65:         order.buysell: no order(0), sell(1) or buy(2)
66:      */
67:
68:      /* decide 'order' (here we use method getOrder described below),
69:         and call method orderRequest to submit order.
70:         If you want to submit several orders, repeat such procedure
71:      */
72:
73:      Order buyOrder =
74:          getBuyOrder(spotPrices, futurePrices, pos, money, restDay);
75:      Order sellOrder =
76:          getSellOrder(spotPrices, futurePrices, pos, money, restDay);
77:      orderRequest(buyOrder);
78:      orderRequest(sellOrder);
79:  }
80:  /* Method of making single order. Implement your trading strategy rewriting
81:     This method */
82:
83:  public Order getBuyOrder(int[] spotPrices, int[] futurePrices,
84:                          int pos, long money, int restDay) {
85:
86:      /* Information available for trading:
87:         spotPrices[]: Time series of spot prices. It provides 120 elements from spotPrices[0]
88:         to spotPrices[119]. The element spotPrices[119] is the latest.
89:         futurePrices[]: Time series of futures price. It provides 60 elements from futurePrices[0]
90:         to futurePrices[59]. The element futurePrices[59] is the latest.
91:         Before opening the market, same values with spot prices are given.
92:         If no contract is made in the market, value -1 is given.
93:         pos: Position of the agent. Positive is buying position. Negative is selling.
94:         money: Available cash. Note that type 'long' is used because of needed precision.
95:         restDay: Number of remaining transaction to the closing of the market.  */

```

## リスト2 ( MyDayTradeStrategy.java ) のつづき

```

96:
97:     Order order = new Order(); // Object to return values.
98:
99:     order.buysell = Order.BUY;
100:    // Cancel decision if it may increase absolute value of the position
101:    if (pos > fMaxPosition) {
102:        order.buysell = Order.NONE;
103:        return order;
104:    }
105:
106:    int latestFuturePrice = futurePrices[futurePrices.length - 1];
107:    if (latestFuturePrice < 0) {
108:        order.buysell = Order.NONE;
109:        return order;
110:    } // If previous price is invalid, it makes no order.
111:    order.price = (int) ((double) latestFuturePrice * (1.0 - fSpreadRatio));
112:
113:    if (order.price < 1)
114:        order.price = 1;
115:
116:    order.quant = fMinQuant + fRandom.nextInt(fMaxQuant - fMinQuant + 1);
117:
118:    // Message
119:    message("MyDayTradeStrategy: Buy" + ", price = " + order.price
120:           + ", volume = " + order.quant + " ");
121:
122:    return order;
123: }
124:
125: /* Method of making single order. Implement your trading strategy rewriting
126:    This method */
127:
128: public Order getSellOrder(int[] spotPrices, int[] futurePrices,
129:                          int pos, long money, int restDay) {
130:
131:    /* Information available for trading:
132:       spotPrices[]: Time series of spot prices. It provides 120 elements from spotPrices[0]
133:       to spotPrices[119]. The element spotPrices[119] is the latest.
134:       futurePrices[]: Time series of futures price. It provides 60 elements from futurePrices[0]
135:       to futurePrices[59]. The element futurePrices[59] is the latest.
136:       Before opening the market, same values with spot prices are given.
137:       If no contract is made in the market, value -1 is given.
138:       pos: Position of the agent. Positive is buying position. Negative is selling.
139:       money: Available cash. Note that type 'long' is used because of needed precision.
140:       restDay: Number of remaining transaction to the closing of the market. */
141:
142:    Order order = new Order(); // Object to return values.
143:
144:    order.buysell = Order.SELL;
145:    // Cancel decision if it may increase the position

```

## リスト2 ( MyDayTradeStrategy.java ) のつづき

```

146:     if (pos < -fMaxPosition) {
147:         order.buysell = Order.NONE;
148:         return order;
149:     }
150:
151:     int latestFuturePrice = futurePrices[futurePrices.length - 1];
152:     if (latestFuturePrice < 0) {
153:         order.buysell = Order.NONE;
154:         return order;
155:     } // If previous price is invalid, it makes no order.
156:
157:     order.price = (int) ((double) latestFuturePrice * (1.0 + fSpreadRatio));
158:
159:     order.quant = fMinQuant + fRandom.nextInt(fMaxQuant - fMinQuant + 1);
160:
161:     // Message
162:     message("MyDayTradeStrategy: Sell" + ", price = " + order.price
163:           + ", volume = " + order.quant + " ");
164:
165:     return order;
166: }
167:
168: /* (non-Javadoc)
169:  * @see strategy.UBaseStrategy#setParameters(java.lang.String[])
170:  */
171: public void setParameters(String[] args) {
172:     super.setParameters(args);
173:     for (int i = 0; i < args.length; ++i) {
174:         StringTokenizer st = new StringTokenizer(args[i], "=");
175:         String key = st.nextToken();
176:         String value = st.nextToken();
177:         if (key.equals("WidthOfPrice")) {
178:             fWidthOfPrice = Integer.parseInt(value);
179:         } else if (key.equals("MinQuant")) {
180:             fMinQuant = Integer.parseInt(value);
181:         } else if (key.equals("MaxQuant")) {
182:             fMaxQuant = Integer.parseInt(value);
183:         } else if (key.equals("MaxPosition")) {
184:             fMaxPosition = Integer.parseInt(value);
185:         } else if (key.equals("SpreadRatio")) {
186:             fSpreadRatio = Double.parseDouble(value);
187:         } else {
188:             message("Unknown parameter:" + key
189:                   + " in RandomStrategy.setParameters");
190:         }
191:     }
192: }
193:
194: }

```

## リスト2 ( MyDayTradeStrategy.java ) のつづき

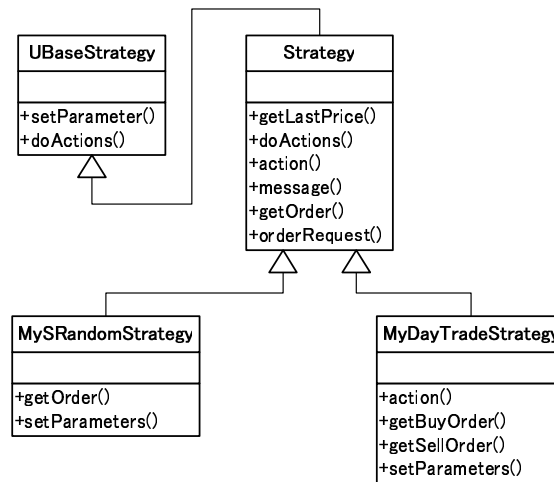


図 2: 戦略クラスの継承関係

### 3.8 より複雑な戦略を記述するために

- インスタンス変数の利用: 複雑な戦略を構成する際にはエージェントの中で情報を多期にわたって保存するために取引戦略クラスの中でインスタンス変数を導入して利用してください。

例えば前回約定した注文量は前回のポジションを記憶しておき、最新のポジションからの差を取ることで得られます。このためには前回のポジションをインスタンス変数とし、`action` メソッドや `getOrder` メソッドなどで、約定した注文量の計算の計算と前回のポジションの更新を行います。

- 複数の注文を出す: 1 回の板寄せ当りに複数の注文を出すこともできます。その例として `MyDayTradeStrategy.java` を示します。`MySRandomStrategy.java` では `getOrder` メソッドを定義しましたが、サーバがエージェントに注文を照会する手順は以下のようになります。

1. サーバは `Strategy` クラスで定義されている `doActions()` メソッドを呼びます。
2. よばれた `doActions()` メソッドは価格系列など必要な情報をサーバから得て同じく `Strategy` クラスで定義されている `action()` メソッドを呼びます。
3. `Strategy` クラスでは `action()` メソッドの中で `getOrder()` メソッド (`MySRandomStrategy` クラスではこの `getOrder()` メソッドを定義しました。)を読んだ後、`orderRequest()` メソッドを呼んで注文をサーバに伝えます。

`Strategy` クラスは抽象クラス `UBaseStrategy` を継承しています。これらのクラスの継承関係とそれぞれで定義されている `public` なメソッドを図 2 に示します<sup>2</sup>。

一回の板寄せ時に複数の注文を出すためには `Strategy` クラスで定義されている `action()` メソッドを再定義 (オーバーライド) します。`MyDayTradeStrategy` クラスでは `action()` メソッド内で `getSellOrder()`, `getBuyOrder()` メソッドを呼んで売り注文、買い注文を生成し、それぞれを `orderRequest()` メソッドでサーバに伝えています。

<sup>2</sup>このような図示の方法は UML のクラス図と呼ばれるものです。

- 新しいメソッドやクラスを書く：取引戦略が複雑になってきた場合には、新しいメソッドを追加したり、新しいクラスを作ったりして対応します。

### 3.9 戦略記述上の注意

U-Mart では複数の参加者がそれぞれ開発した取引戦略を持ち寄って実験を行ないます。このため、戦略や独自に追加するクラスについては実験に際して互いに命名規則を設けてこれを遵守しないと混乱を生じます。

### 3.10 U-Mart 実験の仕様

U-Mart の実験はいくつかのパラメータを持っています。このキットは以下の設定を仮定しています：

- 取引所会費：なし。
- 原証券：J30<sup>3</sup>
- 取引日数：30 日。
- 板寄せ回数：1 日 8 回。
- 初期所持金：各会員 1,000,000,000 円。
- 約定金額：J30 の 1000 倍を 1 単位の約定金額とします。
- 証拠金：ポジション 1 単位あたり 300,000 円。
- 取引所ローン限度額：30,000,000 円。
- 取引所ローン利率：10%。日払い。

## 付録 U-Mart System のコマンドラインからの起動

Windows 以外の OS では U-Mart System はコマンドラインから起動する必要があります。以下のような命令で起動できます。

- ネットワーク版取引端末（日本語モード）

```
java -cp .:UMartSystem.jar gui.UNetGUI
```

- ネットワーク版市場サーバ（日本語モード）

```
java -cp .:UMartSystem.jar gui.UTServerManager
```

- ネットワーク版取引端末（英語モード）

```
java -Duser.language=en -cp .:UMartSystem.jar gui.UNetGUI
```

- ネットワーク版市場サーバ（英語モード）

```
java -Duser.language=en -cp .:UMartSystem.jar gui.UTServerManager
```

<sup>3</sup> J30 は毎日新聞社が作成している東京証券取引所の株価指数。日本を代表する産業 30 銘柄から計算されます。