

The Development of U-Mart Agent with U-Mart System Version 2.0

Hajime Kita, Kyoto University kita @ media.kyoto-u.ac.jp

July 8, 2005

1 Introduction

Developing a trading agent for the U-Mart from the beginning may not be easy for novice users since it is required to communicate with the U-Mart server in SVMP protocol via the Internet. Even after the development, to evaluate and improve the developed agent, the U-Mart server must be executed, and it causes difficulty as well. For this reason, the U-Mart project provides a standard class so that a trading agent can be described easily. Although the functions provided by the class are more limited than those of the agent using the SVMP directly, the use of this class can facilitate the installation of trading strategies.

2 U-Mart System Version 2

The U-Mart System Version 2 contains the six programs in three types as shown in Table 1. Among them, the Market Server Network Version is used in the development of a trading strategy. The U-Mart System Version 2 deals with the agent using the class for a trading agent (hereinafter referred to as Strategy class) by the repetition of the following procedure.

1. The trading agent itself is installed in the market server and operated.
2. At the reception of each order, the market server sequentially inquires for orders from each installed agent first.
3. Next, for a certain period of time, it receives orders from agents (mainly the human agents using the trading terminal) through the network.
4. When the reception period closes, it performs *Itayose*¹ and deals with necessary transactions such as the execution of a contract etc.

¹An auction procedure used in U-Mart.

Table. 1 Programs Provided by U-Mart System Version 2

Program	English Version	Japanese Version	Functions/Other
Simulator Standalone Version	MarketSimulator_en.exe	MarketSimulator_jp.exe	The U-Mart simulator performing standalone environment
Market Server Network Version	MarketServer_en.exe	MarketServer_jp.exe	Exchange server receiving the connection from clients through the network
Trading Terminal Network Version	TradingTerminal_en.exe	TradingTerminal_jp.exe	Terminal to connect to the Market Server Network Version to trade

3 From Development to Operation of Agent

3.1 Development of Agent

An agent is written as a class code describing a trading strategy. More specifically, it is developed as a class extended from the `Strategy` class, and source code and compiled class file should be placed in the `strategy` folder of `UMartSystem`. The detailed description of the code will be discussed later.

3.2 To Compile Strategy Code

Here, an strategy code “`MySRandomStrategy.java`,” placed in the `strategy` folder, is used as an example. After moving to the `UMartSystem` folder, it is compiled as

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

It should be noted that it must be compiled in the `UMartSystem` folder.

When developing an original trading strategy class by using the attached sample strategy class as a model, the class name must be treated carefully. The following four need to be identical.

- Class Name in the source code,
- Constructor in the source code,
- File name of the source code,
- Description in Agent definition File (See below)

3.3 Preparation of Agent Definition File

Next the user needs to prepare an agent definition file for the server to read by the created class file. An example is shown in “ExampleForMyStrategyies.csv” Spreadsheet software such as Excel can be used to read this file for further adjustment². ²Text editors or other appropriate CSV format file editor can also be used.

The second line in this file is the description of the agent that uses “MySRandomStrategy” attached as a sample, and the third line is that of the agent using “MyDayTradeStrategy”. Both designate their strategy class in the sixth column, and the eighth column indicates the initial value of random numbers these agents use. The fourth line and below are description for the use of the agents built-in in the system previously.

3.4 Operation of Server

The server can be operated when compilation of the trading strategy class and preparation of the agent definition file are completed. The procedure to read the agent definition file and operate the server is the following.

1. Click “MarketServer_en.exe” to activate the server.
2. Click “Agent Setting” in “Setting” dialogue.
3. Click “File” in “Agent Setting” dialogue.
4. Select “ExampleForMyStrategies.csv” in the file selection display.
5. After confirming the details, click “OK” in “Agent Setting” dialogue.
6. Click “OK” in “Setting” dialogue.

With this procedure, the server will read the strategy class of the agent and be activated.

3.5 Tips for Performing Experiments

Here are a few tips for development of an agent strategy file and for performing of simulation.

- **Use of Standard Output in Strategy Class:** To confirm the strategy performance in strategy development etc., an output confirming the status should be generated by means of the `message` method of `Strategy` class. The output can be found by selecting the strategy from the strategy list in “Standard Output” tab in the server display.
- **Log:** When operating the server, folders named ‘UMART ...’ are automatically created under the `UMartSystem` folder, and the performance results are outputted there as log files. All results are outputted in CSV format and thus can be analyzed with EXCEL etc.
- **Random Numbers Sequence:** The results of the simulation depend on the sequence of the random numbers (used for tiebreak between the orders contracting) the exchange server uses as well as that of the random numbers each agent uses. To evaluate the agent, it is necessary to perform several simulations with different sequences of random numbers in use.
- **Sequence of Spot Prices:** The results of simulations also depend on the sequence of spot prices in use. The `UMartSystem` provides J30, which contains the actual data of the price index of stocks (closing quotation of each day) for about ten years. In an actual experiment, part of these data are used. The experiments should be performed with different parts of the sequence. The index that can be used as the beginning of spot information ranges from 120 to 2444-(the number of trade days) \times (the number of Itayose per day).



Fig. 1 J30 Time Series Data

- Composition of Agent: The agents described in the agent definition file are those provided as a standard agent set for the UMartSystem. The experiment can be performed with the different number of agents and different initial values of the random numbers etc.

3.6 Formal Description of Agent Strategies

In the U-Mart System Version 2, the agent strategies are based on the formal description as follows.

Input available to agents : the following five types of input are considered.

- Time series of spot price: The spot prices from 120 periods ago to the latest period.
- Time series of futures price: The futures prices from 60 periods ago to the latest period. However, in the period before opening of the market, the spot price is substituted. In addition, in case that exchange is not achieved, “-1” is set instead of the contract price.
- The agent 's current position: Net buying is indicated in positive value, and net selling is in negative value.
- The agent ' s cash balance.
- The remaining number of Itayose before closing/settlement.

Output that agent determines : The following three types of output are considered.

- Limit price (market order is not allowed).
- Order quantity.
- Types of order: Selling (the value is 1), buying (the value is 2), or no order (the value is 0).

Therefore, the agent strategy is considered a function or procedure that determines the output by the input available to agents. The information only in the relevant period can be used to determine the output. However, by storing the information in previous periods by defining instance variable within the agent, it can be taken into consideration to determine the output.

3.7 Class to Describe Agent Strategies

In the U-Mart System Version 2, the agent strategy based on the formal description the agent noted in the previous section can be described in Java source code `~Strategy.java`. The description example is indicated as List 1.

In case of placing one order per Itayose at the maximum, the trading strategy should be built in the method called `“getOrder”`.

“MySRandomStrategy” is the strategy to buy and sell randomly setting a limit price around the latest spot price. The following shows the overview of the source code.

- “`Package strategy;`” at the head indicates that this class belongs to the package named “strategy”. The name of the package and that of the folder in which the source code is placed need to be identical.
- The third line is the description of the class name. Since this class is browsed from outside, it should include the “`public`” modifier. This class is also indicated under “`extends Strategy`” that it is a subclass which is extended from the already-defined class “`Strategy`”.
- The following defines the constant number used within this strategy. In accordance with the coding convention of Java, the name indicating the constant number should be written in upper case (“_” is used between words). Since the constant number does not allow any change within the class, its name includes the “`static final`” modifier. Here various parametric default values used in the strategy are set out.
- Next, the instance variables in this class are defined. The default value previously defined is set as an initial value. The instance variables include the “`private`” modifier so that outsiders cannot access them directly.

- The three lines beginning with “`public MySRandomStrategy(int seed) {`” is the constructor. When creating an instance, they create the instances in the `Random` class giving the initial value “`seed`”.
- Next the method “`getOrder()`” is defined. This method is called by the server when it inquires each agent for orders. This method should be defined including parameters in this format.
- The returned value of the “`getOrder()`” method is the instance in the “`Order`” class. The instance in the “`Order`” class is created to give the returned value.
- The latest price is brought out from the spot price sequence “`spotPrices`”. The method “`getLatestPrice`” is designed for this procedure and defined in the super class, “`Strategy`”. When the price is undefined, “`-1`” is set out, so a nominal price is set out here.
- The buying and selling orders to be placed are randomly set out in “`order.buysell`”. “`fRandom.nextInt(2)`” is to bring out the next random number as a uniform random number among 0 and 1 from the random number sequence. One (1) is added to make it selling (1) or buying (2).
- When the buying and selling of the orders make the position greater than the stipulated value, no order is placed and the result is returned.
- A limit price (`order.price`) is determined by adding the regular random number to the latest spot price. The limit price may be in negative value because the regular random number is used. The while loop is repeated until the limit price becomes positive.
- The quantity (`order.quant`) is set as a uniform random number between minimum and maximum values.
- The determined orders and the latest spot price being browsed at the time

- of the order are written on the display.
- The method is completed with the order as a returned value.
- “`setParameters()`” is the method to set the parameters brought over from the agent definition file as an instance variable. The details are omitted here.

Notes:

- To display the information within a strategy, write the calling of the method “message (character string); in the method “getOrder of the “TestStrategy.java. The class for the parameters is “String.
- When using the random number, use the parameter of the constructor “seed to initialize. It is given a value that is equivalent of the seed of random number given in the operating command.

3.8 MySRandomStrategy.java

In the following lists, line numbers are provided just for readability, and they should not be included in Java source code.

```
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:
```

```

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) {

```

```
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: }
```


3.9 MyDayTradeStrategy.java

```
1: /*
2: A class describing an agent's strategy, which simultaneously submits a buying
3: order with a limit price lower than and a selling order higher than the
4: previous price by a prescribed spread.
5: The quantity of the order is set randomly within a prescribed range.
6: Position of the agent is also considered in decision making.
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:
```

```

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.  */

```

```

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

```

```

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: }

```


3.10 To Write a More Complicated Strategy

- Use of Instance Variables: When constructing a complicated strategy, introduce and use the instance variables in the trading strategy class to store the information in the agent for multiple periods. For example, the quantity of the previous order that reached the execution of a contract can be obtained through storing the previous position and calculating the difference between the previous and latest positions. In order to do so, the previous position should be stored as an instance variable, and the calculation of the quantity of the previously contracted orders and previous position are updated with the “`action`” method, “`getOrder`” method etc.
- Placing Multiple Orders: It is possible to place multiple orders in one Itayose. “`MyDayTradeStrategy.java`” is shown as an example. In the “`MySRandomStrategy.java`”, the “`getOrder`” method is defined, but here the server’s procedure to inquire for the orders to the agent is the following:
 1. The server calls the “`doActions()`” method defined in the “`Strategy`” class.
 2. The “`doActions()`” method obtains the necessary information such as price sequence from the server and calls the “`action()`” method defined also in the “`Strategy`” class.
 3. In the “`Strategy`” class, after calling the “`getOrder()`” method (this “`getOrder()`” method is defined e.g., in the “`MySRandomStrategy`” class) in the “`action()`” method, the “`orderRequest()`” method is called and the order is delivered to the server.

The “`Strategy`” class inherits an abstract class “`UBaseStrategy`”. These inheritance relations and the “`public`” methods defined in each are shown

in Figure 2 ³.

In order to place multiple orders in an Itayose, the “`action()`” method defined in the “`Strategy`” class should be overridden. In the “`MyDayTradeStrategy`” class, the “`getSellOrder()`” and “`getBuyOrder()`” methods are called within the “`action()`” method to create the buying and selling orders, and each is delivered to the server with the “`orderRequest()`” method.

- Writing new methods and classes: When the trading strategy becomes complicated, it should be dealt with by adding new methods and creating a new classes.

3.11 Notes on Writing Strategies

In the U-Mart, an experiment is conducted with multiple participants who bring the self-made trading strategies. For this reason, the strategies and classes added on their own may provoke disorder when an experiment is conducted without an establishment of naming convention and without following the convention.

³Such graphical expression is called UML class diagram.

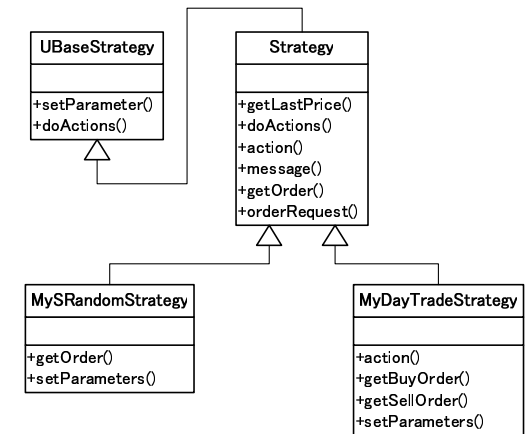


Fig. 2 Inheritance Relations of Strategy Classes

3.12 Specification of U-Mart Experiment

The experiment of U-Mart possesses several parameters. This kit assumes the following setting conditions.

- Members' fee: none
- Underlying securities: J30⁴
- The number of trade days: 30 days
- The number of Itayose: 8 times a day
- Initial Cash: 1,000,000,000 JPY per member
- Contracted Price: The 1000 fold of J30 is designated as a unit of contracted Price.
- Deposit: 300,000 JPY per position unit.
- Exchange Loan Limit: 30,000,000 JPY.
- Exchange Loan Interest Rate: 10%. Daily installments.

⁴J30 is the price index of stocks of Tokyo Stock Exchange that Mainichi Newspapers Co., Ltd. creates. It is calculated based on the data from thirty representative brands of Japan.

Appendix Execution of the U-Mart System from Command Line

In other OS than Windows, the U-Mart System needs to be executed from the command line. The following commands can execute the system:

- Trade Terminal Network Version (Japanese Mode)
`Java -cp .:UMartSystem.jar gui.UNetGUI`
- Market Server Network Version (Japanese Mode)
`Java -cp .:UMartSystem.jar gui.UsServerManager`
- Trade Terminal Network Version (English Mode)
`Java -Duser.language=en -cp .:UMartSystem.jar gui.UNetGUI`
- Market Server Network Version (English Mode)
`Java -Duser.language=en -cp .:UMartSystem.jar gui.UsServerManager`