Variations sur les agents "chartist" dans ATOM

M. Gaciarz, P. Mathieu, Y. Secq

Résumé

Le package `chartists` d'ATOM permet de peupler facilement vos simulations avec différents types d'agents chartistes et d'agents à raisonnement interne. Cette documentation explique comment les utiliser et comment créer vos propres agents à partir des modèles proposés.

Table des matières

1 Présentation des agents

Sur ATOM, une simulation est une succession de tours de parole (ou *ticks*) durant lesquels chaque agent peut passer un ordre. Lorsqu'un agent a la parole, celui-ci doit être capable de prendre une décision, c'est à dire envoyer un ordre ou choisir de ne rien faire. Sur un marché dirigé par les ordres, et donc sur ATOM, un ordre limite contient les informations suivantes :

- l'émetteur de l'ordre
- le nom du titre à échanger
- la direction souhaitée (achat ou vente)
- la quantité de titres à échanger
- le prix limite (prix maximal accepté pour un ordre d'achat, prix minimal accepté pour un ordre de vente)

1.1 La classe `StrategicAgent`

Les deux premières informations (émetteur et titre) étant triviales, un agent doit être capable de fixer la direction de l'ordre, son prix et la quantité à échanger. Nous proposons dans le package chartists un squelette d'agent permettant de remplir ces 3 fonctions, le StrategicAgent. Celui-ci est une coquille vide qu'il faut remplir avec plusieurs modules pour obtenir différents types d'agents.

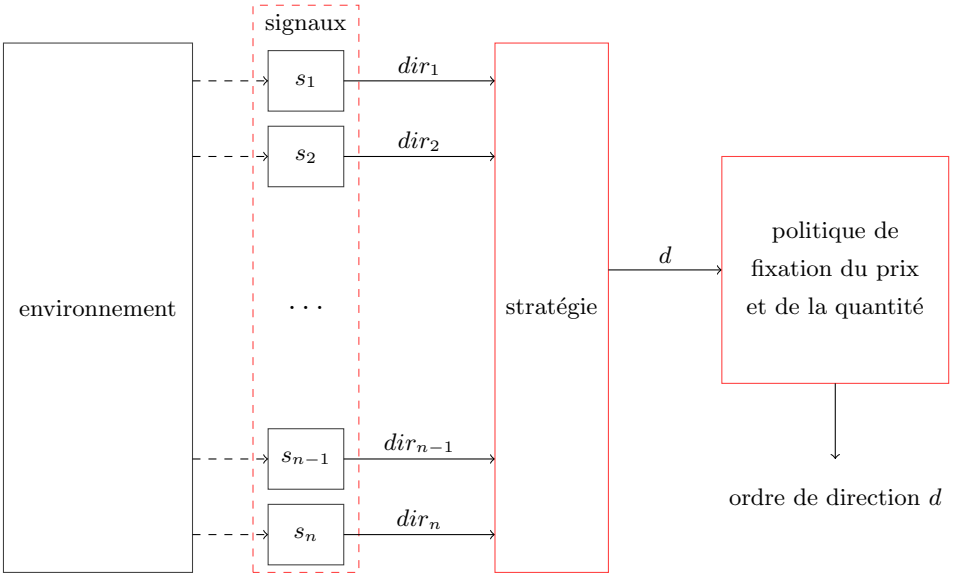


FIGURE 1 – Fonctionnement d'un `StrategicAgent`

Pour définir le comportement de trading d'un agent, il est nécessaire d'utiliser les 4 notions suivantes :

- une direction (`Signal.Direction`) représente tout simplement la direction d'un ordre. Elle peut valoir acheter (`BUY`), vendre (`SELL`) ou ne rien faire

(`HOLD`).
- un signal (`Signal`) produit une direction en fonction de l'état du marché et de son propre raisonnement interne.
- une stratégie (`Strategy`) transforme les résultats d'un ou plusieurs signaux en une seule direction. Cela permet par exemple de suivre ou non un signal unique, ou d'agréger plusieurs signaux pouvant être contradictoires.
- une politique (`OrderPolicy`) produit un ordre à partir d'une direction. Elle determine son prix et sa quantité.

Pour créer un `StrategicAgent`, il faut donc le doter des modules suivants :
- une liste de signaux
- une stratégie
- une politique

Le `StrategicAgent` fonctionne ainsi : à chaque prise de décision, les différents signaux de l'agent produisent chacun une direction en fonction de l'environnement. Ces différentes directions sont agrégées par la stratégie, qui produit à son tour une direction unique d. Si d vaut `HOLD`, il n'y a pas besoin de produire d'ordre. Sinon, la politique de l'agent produit un prix et une quantité, et retourne un ordre de direction d ayant ce prix et cette quantité (cf Fig. 2.5.3).

1.2 Différentes possibilités de création d'un `StrategicAgent`

Plusieurs constructeurs sont proposés pour faciliter la création de différents types d'agents stratégiques.

Agent vide (sans signal, stratégie, ni politique) :

```
public StrategicAgent(String name);

public StrategicAgent(String name, long cash);
```

Agents avec un seul signal (on suit ce signal par défaut) :

```
public StrategicAgent(String name, long cash, Signal signal,
                      OrderPolicy policy);

public StrategicAgent(String name, long cash, Signal signal,
                      Strategy strategy, OrderPolicy policy);
```

Ce constructeur est le plus complexe mais permet de définir le comportement le plus complet : un ensemble de signaux, leur agrégation par la stratégie et la détermination des prix et quantités.

```
public StrategicAgent(String name, long cash, List<Signal> signals,
                      Strategy strategy, OrderPolicy policy);
```

2 La notion de signal et les signaux standards

Un signal produit une direction à partir de l'état du marché. La fonction d'un signal est donc la même que celle d'un agent dans un marché fixant les prix de manière équationnelle. Cette interface est définie ainsi :

```
public interface Signal extends java.io.Serializable {
    public Direction update(OrderBook asset, Day day);
    [...]
}
```

Pour créer un signal, il suffit de créer une classe implémentant l'interface `Signal`, en implémentant la méthode `update` et en s'assurant que la classe est `Serializable`.

Pour créer facilement des agents sur ATOM, plusieurs types de signaux sont proposés par défaut, et chacun de ces signaux correspond à un type d'agent connu. Par exemple le signal `RandomDirection` produit une direction au hasard, ce qui correspond au comportement d'un ZIT (*Zero Intelligence Trader*).

Certains des signaux disponibles utilisent uniquement un raisonnement interne pour décider d'acheter ou de vendre (signaux `Periodic` et `RandomDirection`). D'autres, dits "*chartists*", s'appuient sur les prix passés pour tenter de prédire la manière dont ceux-ci vont évoluer, et donc quelle action est la plus profitable (signaux `Indicator`, `Momentum`, `MovingAverage`, `MixedMovingAverage`, `Variation` et `Rsi`). Ceux-ci sont basés sur le temps. On note t l'instant courant. $t-1$ représente l'instant précédent pour l'agent, c'est-à-dire la dernière fois qu'il a eu la parole. Ainsi, $p(t)$ représente le prix courant et $p(t-n)$ représente le prix qui était en vigueur lorsque l'agent avait la parole il y a n ticks. Nous détaillons dans cette section le fonctionnement de chacun de ces signaux.

2.1 `Indicator`

Signal comparant la variation des prix entre l'instant t et l'instant $t-n_1$ et celle entre l'instant t et l'instant $t-n_2$.

2.1.1 Paramètres et usage

- un entier n_1
- un entier $n_2 \geq n_1$
- un pourcentage *seuil* (1 signifie 1%), 0 par défaut

```
public Indicator(int n1, int n2);

public Indicator(int n1, int n2, double seuil);
```

2.1.2 Fonctionnement

Le signal compare $p(t)$ à $p(t-n_1)$. Il fait de même avec n_2.
Si ce sont deux hausses de prix d'au moins *seuil*%, c'est un signal d'achat.
Si ce sont deux baisses de prix d'au moins *seuil*%, c'est un signal de vente.
Pas de signal dans les autres cas.

$(\frac{p(t)}{p(t-n_1)} > 1 + \frac{seuil}{100}) \wedge (\frac{p(t)}{p(t-n_2)} > 1 + \frac{seuil}{100}) \Rightarrow BUY$
$(\frac{p(t)}{p(t-n_1)} < 1 - \frac{seuil}{100}) \wedge (\frac{p(t)}{p(t-n_2)} < 1 - \frac{seuil}{100}) \Rightarrow SELL$

2.1.3 Exemple

Exemple avec $n1 = 2$, $n2 = 5$, $seuil = 1.0$.

temps	...	$t-5$	$t-4$	$t-3$	$t-2$	$t-1$	t
prix	...	101.1	101.6	101.5	102.1	103.5	104.2

$\frac{104.2}{102.1} > 1.01$ et $\frac{104.2}{101.1} > 1.01$, c'est donc un signal d'achat.

2.2 `Momentum`

Signal raisonnant sur le momentum, c'est-à-dire la différence entre le prix à l'instant t et le prix à $t-n$.

2.2.1 Paramètres et usage

- un entier n
- un pourcentage *seuil* (1 signifie 1%), 0 par défaut

```
public Momentum(int n);

public Momentum(int n, double seuil);
```

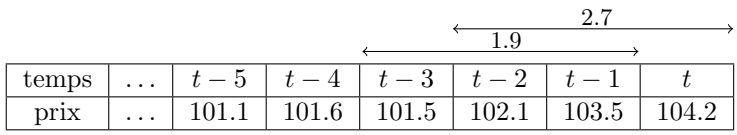
2.2.2 Fonctionnement

Le signal calcule le momentum, qui est la différence entre le prix à l'instant courant t et le prix à l'instant $t-n$.
Si le momentum est supérieur à celui de l'instant précédent d'au moins *seuil*% du prix, c'est un signal d'achat.
Si le momentum est inférieur au momentum de l'instant précédent d'au moins *seuil*% du prix, c'est un signal de vente.
Pas de signal dans les autres cas.

$momentum(t_i) = p(t_i) - p(t_i - n)$
$momentum(t) - momentum(t-1) > \frac{p(t-1) \times seuil}{100} \Rightarrow BUY$
$momentum(t) - momentum(t-1) < -\frac{p(t-1) \times seuil}{100} \Rightarrow SELL$

2.2.3 Exemple

Exemple avec $n = 3$, $seuil = 0.5$.



temps	...	$t-5$	$t-4$	$t-3$	$t-2$	$t-1$	t
prix	...	101.1	101.6	101.5	102.1	103.5	104.2

$momentum(t) = 104.2 - 101.5 = 2.7$
$momentum(t-1) = 103.5 - 101.6 = 1.9$
$momentum(t) - momentum(t-1) = 0.8$
$momentum(t) - momentum(t-1) > 0.5$, c'est donc un signal d'achat.

2.3 `MovingAverage`

Signal raisonnant sur les coupures entre la courbe des prix et la courbe de la moyenne des n derniers prix.

2.3.1 Paramètre et usage

– un entier n

```
public MovingAverage(int n);
```

2.3.2 Fonctionnement

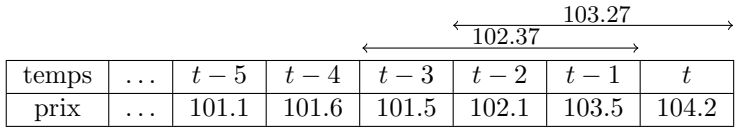
Le signal observe l'évolution du prix et de la moyenne des n derniers prix. Lorsque ces deux valeurs se coupent, un signal d'achat ou de vente est lancé.
Si le prix coupe la moyenne vers le haut, c'est un signal d'achat.
Si le prix coupe la moyenne vers le bas, c'est un signal de vente.
Pas de signal dans les autres cas.

$$moyenne(t_i) = \frac{\sum_{x=t_i+1-n}^{t_i} p(x)}{n}$$

$(p(t-1) < moyenne(t-1)) \wedge (p(t) > moyenne(t)) \Rightarrow BUY$
$(p(t-1) > moyenne(t-1)) \wedge (p(t) < moyenne(t)) \Rightarrow SELL$

2.3.3 Exemple 1

Exemple avec $n = 3$.

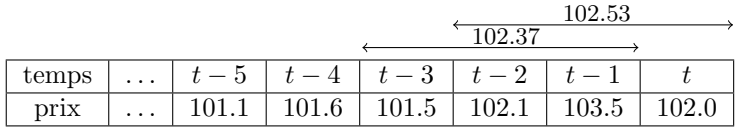


temps	...	$t-5$	$t-4$	$t-3$	$t-2$	$t-1$	t
prix	...	101.1	101.6	101.5	102.1	103.5	104.2

$moyenne(t-1) = \frac{103.5+102.1+101.5}{3} \simeq 102.37$
$moyenne(t) = \frac{104.2+103.5+102.1}{3} \simeq 103,27$
$p(t-1) > moyenne(t-1)$ et $p(t) > moyenne(t)$, donc on ne fait rien.

2.3.4 Exemple 2

Exemple avec $n = 3$.



temps	...	$t-5$	$t-4$	$t-3$	$t-2$	$t-1$	t
prix	...	101.1	101.6	101.5	102.1	103.5	102.0

$moyenne(t-1) = \frac{103.5+102.1+101.5}{3} \simeq 102.37$
$moyenne(t) = \frac{102.0+103.5+102.1}{3} \simeq 102.53$
$p(t-1) > moyenne(t-1)$ et $p(t) < moyenne(t)$, c'est donc un signal de vente.

2.4 `MixedMovingAverage`

Signal raisonnant sur les coupures entre la courbe de la moyenne "courte" des n_1 derniers prix et celle de la moyenne "longue" des n_2 derniers prix.

2.4.1 Paramètres et usage

- un entier n_1
- un entier $n_2 \geq n_1$
- un pourcentage *seuil* (1 signifie 1%), 0 par défaut

```
public MixedMovingAverage(int n1, int n2);

public MixedMovingAverage(int n1, int n2, double seuil);
```

2.4.2 Fonctionnement

Le signal observe l'évolution des n_1 derniers prix (moyenne courte) et celle des n_2 derniers prix (moyenne longue). Lorsque ces deux valeurs se coupent, un signal d'achat ou de vente est lancé.

- Si la moyenne courte coupe la moyenne longue vers le haut en étant supérieure d'au moins *seuil*%, c'est un signal d'achat.
- Si la moyenne courte coupe la moyenne longue vers le bas en étant inférieure d'au moins *seuil*%, c'est un signal de vente.
- Pas de signal dans les autres cas.

$$moyCourte(t_i) = \frac{\sum_{x=t_i+1-n_1}^{t_i} p(x)}{n_1}$$

$$moyLongue(t_i) = \frac{\sum_{x=t_i+1-n_2}^{t_i} p(x)}{n_2}$$

$$(moyCourte(t-1) < moyLongue(t-1)) \wedge (moyCourte(t) > moyLongue(t)) \wedge$$

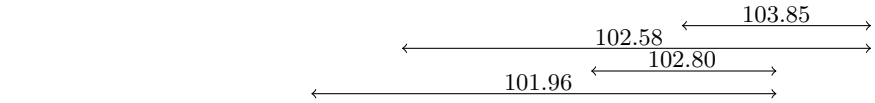
$$\left(\frac{moyCourte(t)+moyLongue(t-1)-moyCourte(t-1)-moyLongue(t)}{moyLongue(t)} > \frac{seuil}{100}\right) \Rightarrow BUY$$

$$(moyCourte(t-1) > moyLongue(t-1)) \wedge (moyCourte(t) < moyLongue(t)) \wedge$$

$$\left(\frac{moyCourte(t-1)+moyLongue(t)-moyCourte(t)-moyLongue(t-1)}{moyLongue(t)} > \frac{seuil}{100}\right) \Rightarrow SELL$$

2.4.3 Exemple1

Exemple avec $n_1 = 2$, $n_2 = 5$ et $seuil = 0.1$.



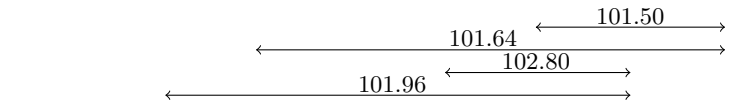
temps	...	$t-5$	$t-4$	$t-3$	$t-2$	$t-1$	t
prix	...	101.1	101.6	101.5	102.1	103.5	104.2

$$moyCourte(t) = \frac{104.2+103.5}{2} = 103.85$$
$$moyCourte(t-1) = \frac{103.5+102.1}{2} = 102.80$$
$$moyLongue(t) = \frac{104.2+103.5+102.1+101.5+101.6}{5} = 102.58$$
$$moyLongue(t-1) = \frac{103.5+102.1+101.5+101.6+101.1}{5} = 101.96$$

$moyCourte(t-1) > moyLongue(t-1)$ et $moyCourte(t) > moyLongue(t)$, donc on ne fait rien.

2.4.4 Exemple 2

Exemple avec $n_1 = 2$, $n_2 = 5$ et $seuil = 0.1$.



temps	...	$t-5$	$t-4$	$t-3$	$t-2$	$t-1$	t
prix	...	101.1	101.6	101.5	102.1	103.5	99.5

$moyCourte(t) = \frac{99.5+103.5}{2} = 101.50$
$moyLongue(t) = \frac{99.5+103.5+102.1+101.5+101.6}{5} = 101.64$
$moyCourte(t-1) = \frac{103.5+102.1}{2} = 102.80$
$moyLongue(t-1) = \frac{103.5+102.1+101.5+101.6+101.1}{5} = 101.96$

On a :
$moyCourte(t-1) > moyLongue(t-1) \wedge moyCourte(t) < moyLongue(t) \wedge \frac{moyCourte(t-1)+moyLongue(t)-moyCourte(t)-moyLongue(t-1)}{moyLongue(t)} \simeq 0,0096 > \frac{0.1}{100}$

C'est donc un signal de vente.

2.5 `Variation`

Signal raisonnant sur la quantité de hausses ou de baisses successives des prix.

2.5.1 Paramètres et usage

- un entier $nbHausses$
- un entier $nbBaisses$
- un entier n

```
public Variation(int nbHausses, int nbBaisses, int n);
```

2.5.2 Fonctionnement

Le signal observe l'évolution de la somme des n derniers prix.
- Si cette somme a connu $nbHausses$ hausses successives, c'est un signal d'achat.
- Si cette somme a connu $nbBaisses$ baisses successives, c'est un signal de vente.
- Pas de signal dans les autres cas.

En considérant que l'on définit la fonction somme ainsi :

$$somme(t_i) = \sum_{x=t_i+1-n}^{t_i} p(x)$$

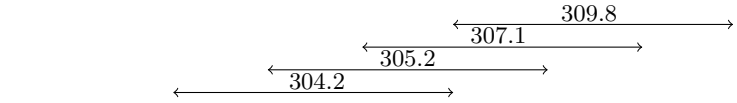
alors l'algorithme de `Variation` se définit ainsi :

```
hausse ← true;
baisse ← true;
for i : 0 to (max(nbHausses, nbBaisses) − 1) do
    d ← somme(t − i) − somme(t − i − 1);
    if (i < nbHausses) and (d < 0) then
        hausse ← false;
    if (i < nbBaisses) and (d > 0) then
        baisse ← false;
end
if hausse then
    if baisse then
        return(HOLD);
    return(BUY);
end
if baisse then
    return(SELL);
return(HOLD);
```

Algorithme 1 : Fonctionnement de `Variation`

2.5.3 Exemple

Exemple avec $nbHausses = 3$, $nbBaisses = 2$ et $n = 3$.



temps	...	$t-5$	$t-4$	$t-3$	$t-2$	$t-1$	t
prix	...	101.1	101.6	101.5	102.1	103.5	104.2

$somme(t) = 104.2 + 103.5 + 102.1 = 309.8$
$somme(t-1) = 103.5 + 102.1 + 101.5 = 307.1$
$somme(t-2) = 102.1 + 101.5 + 101.6 = 305.2$
$somme(t-3) = 101.5 + 101.6 + 101.1 = 304.2$

$somme(t-3) < somme(t-2) < somme(t-1) < somme(t-0)$, on a donc 3 hausses successives de la somme des prix, c'est un signal d'achat.

2.6 `Rsi`

Ce signal s'appuie sur l'indicateur d'analyse technique RSI (*Relative Strength Index*). Celui-ci est basé sur les nombres de hausses et de baisses de prix récentes. Il est calculé ainsi : $RSI = \dfrac{1}{1 + \frac{moyHausse}{moyBaisse}}$. Celui-ci a pour but de reconnaître les situations de sur-achat ou de sur-vente.

2.6.1 Paramètres et usage

- un entier n
- un pourcentage *seuil* (1 signifie 1%), 0 par défaut

```
public Rsi(int n);

public Rsi(int n, double seuil);
```

2.6.2 Fonctionnement

Le signal calcule le RSI sur les n derniers tours.

- Si celui-ci est inférieur à $(50 - seuil)\%$, c'est un signal d'achat.
- Si celui-ci est supérieur à $(50 + seuil)\%$, c'est un signal de vente.
- Pas de signal dans les autres cas, en particulier s'il n'y a pas eu de hausse des prix dans les n derniers tours, ou s'il n'y a pas eu de baisse.

$$moyHausse = \frac{\sum_{x=t_i+1-n}^{t_i} p(x) - p(x-1) \text{ tq } p(x) > p(x-1)}{nbHausses}$$

$$moyBaisse = \frac{\sum_{x=t_i+1-n}^{t_i} p(x-1) - p(x) \text{ tq } p(x-1) > p(x)}{nbBaisses}$$

$$RSI = \frac{1}{1 + \frac{moyHausse}{moyBaisse}}$$

$$RSI < \frac{50 - seuil}{100} \Rightarrow BUY$$

$$RSI > \frac{50 + seuil}{100} \Rightarrow SELL$$

2.6.3 Exemple

Exemple avec $n = 5$et $seuil = 1.0$.

temps	...	$t-5$	$t-4$	$t-3$	$t-2$	$t-1$	t
prix	...	101.1	101.6	101.5	102.1	103.5	104.2

$p(t) - p(t-1) = 104.2 - 103.5 = 0.7$ (hausse)
$p(t-1) - p(t-2) = 103.5 - 102.1 = 1.4$ (hausse)
$p(t-2) - p(t-3) = 102.1 - 101.5 = 0.6$ (hausse)
$p(t-3) - p(t-4) = 101.5 - 101.6 = -0.1$ (baisse)
$p(t-4) - p(t-5) = 101.6 - 101.1 = 0.5$ (hausse)

$moyHausses = \frac{0.7+1.4+0.6+0.5}{4} = 0.8$
$moyBaisses = \frac{0.1}{1} = 0.1$

$RSI = \frac{1}{1 + \frac{0.8}{0.1}} \simeq 0.11$
$RSI < 0.49$, c'est donc un signal d'achat.

2.7 `Periodic`

Le signal périodique est un signal interne, c'est-à-dire qui ne base pas son calcul sur des informations de l'environnement. Ce signal se contente d'alterner des périodes d'achats et des périodes de vente.

2.7.1 Paramètres et usage

- un entier $pAchat$
- un entier $pVente$

```
public Periodic(int pachat, int pvente);
```

2.7.2 Fonctionnement

Achète pendant $pAchat$ tours, vend pendant $pVente$ tours.

2.8 `RandomDirection`

Le signal RandomDirection est aussi un signal interne qui retourne aléatoirement une direction (`HOLD`/`BUY`/`SELL`).

2.8.1 Paramètre et usage

- un réél (entre 0 et 1) $holdProbability$, 0.5 par défaut

```
public RandomDirection();

public RandomDirection(double holdProbability);
```

2.8.2 Fonctionnement

Signal qui choisit de ne rien faire avec une probabilité de $holdProbability$, et qui décide aléatoirement d'acheter ou de vendre sinon (avec une probabilité de 1/2).

3 La notion de stratégie d'agrégation de signaux

Les signaux produisent des directions, qui peuvent être des choix mauvais ou contradictoires. Pour utiliser et agréger un ou plusieurs signaux, on utilise une stratégie. L'interface `Strategy` est définie comme suit :

```
public interface Strategy extends java.io.Serializable {
    public Signal.Direction aggregate(List<Signal.Direction> directions,
                                      Agent a);
}
```

Pour créer une stratégie, il suffit de créer une classe implémentant l'interface `Strategy`, en implémentant la méthode *aggregate* et en s'assurant que la classe est `Serializable`. Quelques stratégies simples sont proposées sur ATOM.

3.1 `SingleSignalStrategy`

La stratégie `SingleSignalStrategy` ne peut être utilisé que lorsqu'un agent n'utilise qu'un seul signal. Cette stratégie est paramétrée par un booléen qui indique si la stratégie suit la direction proposée par le signal ou au contraire prend la direction inverse. Lorsque le signal retourne HOLD, la stratégie aussi.

3.1.1 Paramètre et usage

– un booléen *regular*, vrai par défaut

```
public SingleSignalStrategy();

public SingleSignalStrategy(boolean regular);
```

3.1.2 Fonctionnement

Si *regular* est vrai, on retourne la direction produite par le signal, sinon on retourne la direction opposée. Si la direction produite par le signal est `HOLD`, on retournera `HOLD` dans les deux cas.

3.2 `Majority`

La stratégie `Majority` retourne la direction majoritairement (ou minotairement) choisie parmi un ensemble de signaux.

3.2.1 Paramètre et usage

– un booléen *regular*, vrai par défaut

```
public Majority();

public Majority(boolean regular);
```

3.2.2 Fonctionnement

Si l'une des direction achat ou vente est absolument majoritaire, on retourne cette direction (ou la direction contraire si $\neg regular$). On retourne `HOLD` sinon.

$regular \wedge nb(BUY) > nb(SELL) + nb(HOLD) \Rightarrow BUY$
$regular \wedge nb(SELL) > nb(BUY) + nb(HOLD) \Rightarrow SELL$

$\neg regular \wedge nb(BUY) > nb(SELL) + nb(HOLD) \Rightarrow SELL$
$\neg regular \wedge nb(SELL) > nb(BUY) + nb(HOLD) \Rightarrow BUY$

$nb(HOLD) \geq nb(BUY) - nb(SELL) \wedge nb(HOLD) \geq nb(SELL) - nb(BUY) \Rightarrow HOLD$

4 Les politiques de fixation de prix et de quantités

Décider d'acheter ou de vendre n'est pas suffisant pour produire un ordre qui puisse être passé sur le marché. Nos agents utilisent pour cela une politique, qui est un outil permettant de fixer le prix limite d'un ordre (prix minimum de vente ou prix maximum d'achat) et la quantité de titres à échanger.

`OrderPolicy` est une interface implémentée par les différentes politiques proposées sur ATOM. Cette interface est définie ainsi :

```
public interface OrderPolicy extends java.io.Serializable {
   public Order build(Signal.Direction direction, OrderBook m,
                      Agent a,int externalOrderID);
}
```

A chaque fois qu'on appelle la méthode `decide` d'un `StrategicAgent`, celui-ci utilise un ou plusieurs signaux et une stratégie pour produire une direction d. Il appelle ensuite la méthode `build` de sa politique, avec d comme premier paramètre. L'ordre retourné par la méthode `build` de la politique est à son tour retourné comme résultat de la méthode `decide` de l'agent.

Pour créer une nouvelle politique, il suffit de créer une classe implémentant l'interface `OrderPolicy`, en implémentant la méthode `build` et en s'assurant que la classe est `Serializable`. Plusieurs politiques sont proposées sur ATOM.

4.1 La politique `BestBidAskPolicy`

Cette politique raisonne sur les meilleurs ordres contenus dans le carnet pour fixer le prochain prix.

4.1.1 Paramètres et usage

- un entier $defaultPrice$, 14500 par défaut (en cents)
- un entier $quantity$, 75 par défaut
- un pourcentage $rate$ (1.0 signifie 1%), 1.0 par défaut
- un booléen $strictlyBest$, vrai par défaut

```
public BestBidAskPolicy();

public BestBidAskPolicy(long defaultPrice, int quantity,
                        double rate, boolean strictlyBest);
```

4.1.2 Fonctionnement

On utilise comme prix de référence $refPrice$ le prix du meilleur ordre du carnet $bestPrice$ (prix du meilleur ordre d'achat si on souhaite acheter, prix du meilleur ordre de vente si on souhaite vendre). S'il n'y a pas d'ordre de ce type dans le carnet, $refPrice \leftarrow lastPrice$ (dernier prix fixé). Si aucun prix n'a été fixé, $refPrice \leftarrow defaultPrice$.

rate designe le taux d'augmentation ou de diminution du prix par rapport à $refPrice$, en pourcents. Si *strictlyBest* est vrai, on doit fixer un prix strictement meilleur que $refPrice$. Sinon on s'autorise à fixer un prix égal à $refPrice$.
$step \leftarrow \frac{refPrice \times rate}{100}$
Si $step = 0$ et *strictlyBest* alors $step \leftarrow 0.01$.

On peut maintenant calculer le prix :
- achat : $price \leftarrow refPrice + step$
- vente : $price \leftarrow max(refPrice - step, 0.01)$

On retourne un ordre de la direction demandée avec la quantité *quantity* et au prix *price*.

4.1.3 Exemple

Extrait de carnet d'ordre :

Direction	Ordre	Quantité	Prix
	...	...	...
Ask	o1	3	**11.0**
	bid-ask spread = 0.2 ↕		
Bid	o2	2	10.8
	...	...	...

Pour un ordre de vente, avec $rate = 1.0$:
$refPrice = 11.0$
$step = 11.0 * 1.0/100 = 0.11$
$price = refPrice - step = 10.89$
On envoie donc un ordre de vente de prix 10.89.

Pour un ordre de vente, avec $rate = 0.01$ et $strictlyBest = false$:
$refPrice = 11.0$
$step = 11.0 * 0.01/100 = 0.00$ (les sommes sont arrondies au cent près)
$price = refPrice - step = 11.00$
On envoie donc un ordre de vente de prix 11.00.

Pour un ordre de vente, avec $rate = 0.01$ et $strictlyBest = true$:
$refPrice = 11.0$
$step = 11.0 * 0.01/100 = 0.00$ (les sommes sont arrondies au cent près)
$step \leftarrow 0.01$ car $strictlyBest = true$.
$price = refPrice - step = 10.99$
On envoie donc un ordre de vente de prix 10.99.

4.2 La politique `LastPricePolicy`

Cette politique raisonne sur le dernier prix fixé pour fixer le prochain prix.

4.2.1 Paramètres et usage

- un entier $defaultPrice$, 14500 par défaut (en cents)
- un entier $quantity$, 75 par défaut
- un pourcentage $rate$ (1.0 signifie 1%), 1.0 par défaut
- un booléen $strictlyBest$, vrai par défaut

```
public LastPricePolicy();

public LastPricePolicy(long defaultPrice, int quantity, double rate,
                       boolean strictlyBest);
```

4.2.2 Fonctionnement

On utilise comme prix de référence $refPrice$ le dernier prix fixé $lastPrice$, et $defaultPrice$ tant qu'aucun prix n'a été fixé.

$rate$ designe le taux d'augmentation ou de diminution du prix par rapport à $refPrice$, en pourcents. Si $strictlyBest$ est vrai, on doit fixer un prix strictement meilleur que $refPrice$, sinon on s'autorise à fixer un prix égal à $refPrice$.
$step \leftarrow \frac{refPrice \times rate}{100}$
Si $step = 0$ et $strictlyBest$ alors $step \leftarrow 1$.

On peut maintenant calculer le prix :
- achat : $price = refPrice + step$
- vente : $price = max(refPrice - step, 1)$

On retourne un ordre de la direction demandée avec la quantité $quantity$ et au prix $price$.

4.2.3 Exemple

Dernier prix fixé : 11.85

Pour un ordre d'achat, avec $rate = 1.0$:
$refPrice = 11.85$
$step = 11.0 * 1.0/100 = 0.12$ (les sommes sont arrondies au cent près)
$price = refPrice + step = 11.97$
On envoie donc un ordre de vente de prix 10.97.

Pour un ordre d'achat, avec $rate = 0.01$ et $strictlyBest = false$:
$refPrice = 11.85$
$step = 11.85 * 0.01/100 = 0.00$ (les sommes sont arrondies au cent près)
$price = refPrice + step = 11.85$
On envoie donc un ordre d'achat de prix 11.85.

Pour un ordre d'achat, avec $rate = 0.01$ et $strictlyBest = true$:
$refPrice = 11.85$
$step = 11.85 * 0.01/100 = 0.00$ (les sommes sont arrondies au cent près)
$step \leftarrow 0.01$ car $strictlyBest = true$.
$price = refPrice + step = 11.86$

On envoie donc un ordre d'achat de prix 11.86.

4.3 La politique `MarketOrderPolicy`

Cette politique fixe les prix à la manière d'un ordre au marché (acheter ou vendre autant de titres que possible, à tout prix) ou d'un ordre à la meilleur limite (prix du meilleur ordre opposé).

4.3.1 Paramètres et usage

- un entier $defaultPrice$, 14500 par défaut (en cents)
- un entier *quantity*, 75 par défaut
- un booleen $bestLimit$, faux par défaut

```
public MarketOrderPolicy();

public MarketOrderPolicy(long defaultPrice,int quantity,
                         boolean marketPrice);
```

4.3.2 Fonctionnement avec $bestLimit = false$

Si $bestLimit = false$, on fixe le prix de manière à obtenir le comportement d'un *market order*. On utilise comme prix *price* le prix du pire ordre opposé du carnet$worstPrice$. Cela permet d'échanger autant d'ordres que possible à tout prix. Si on cherche à échanger plus d'ordres que n'en contient le carnet, la quantité restante est ajoutée au carnet au prix $worstPrice$. On évite ainsi les prix infinis ou nuls, qui sont irréalistes.
Si $bestLimit = false$, $price \leftarrow worstPrice$
S'il n'y a pas d'ordre du type opposé dans le carnet, $price \leftarrow lastPrice$ (dernier prix fixé).
Si aucun prix n'a été fixé, $price \leftarrow defaultPrice$
On retourne un ordre de la direction demandée avec la quantité *quantity* et au prix *price*.

4.3.3 Exemple avec $bestLimit = false$

Voici un exemple de carnet d'ordres :

Direction	Ordre	Quantité	Prix
	o1	8	**111.5**
	o2	10	111.1
Ask	o3	3	111.0
		bid-ask spread = 0.2 ↕	
Bid	o4	2	110.8
	o5	6	110.6

Pour un ordre d'achat, le prix fixé par cette politique avec $bestLimit = false$ sera celui du pire ordre de vente (111.5). Avec $quantite = 15$, on retourne un ordre d'achat de prix 111.5 de quantité 15. Le carnet passe alors dans l'état suivant, ce qui correspond bien au comportement d'un market order :

Direction	Ordre	Quantité	Prix
Ask	o1	6	**111.5**
			bid-ask spread = 0.7 ↕
Bid	o4	2	110.8
	o5	6	110.6

4.3.4 Fonctionnement avec $bestLimit = true$

Si $bestLimit = true$, on fixe le prix de manière à obtenir le comportement d'un best limit order. On utilise comme prix *price* le prix du meilleur ordre opposé du carnet $bestPrice$. Cela permet d'échanger autant d'ordres que possible au prix $bestPrice$, puis d'ajouter les éventuels ordres restants au carnet au prix $bestPrice$.
Si $bestLimit = true$, $price \leftarrow bestPrice$
S'il n'y a pas d'ordre du type opposé dans le carnet, $price \leftarrow lastPrice$ (dernier prix fixé).
Si aucun prix n'a été fixé, $price \leftarrow defaultPrice$.
On retourne un ordre de la direction demandée avec la quantité *quantity* et au prix *price*.

4.3.5 Exemple avec $bestLimit = true$

Voici un exemple de carnet d'ordres :

Direction	Ordre	Quantité	Prix
	o1	8	111.5
	o2	10	111.1
Ask	o3	3	**111.0**
			bid-ask spread = 0.2 ↕
Bid	o4	2	110.8
	o5	6	110.6

Pour un ordre d'achat, le prix fixé par cette politique avec $bestLimit = true$ sera celui du meilleur ordre de vente (111.0). Avec $quantite = 15$, on retourne un ordre d'achat de prix 111.0 et de quantité 15. Le carnet passe alors dans l'état suivant, ce qui correspond bien au comportement d'un best limit order :

Direction	Ordre	Quantité	Prix
	o1	8	111.5
Ask	o2	10	111.1
			bid-ask spread = 0.1 ↕
Bid	o6	12	**111.0**
	o4	2	110.8
	o5	6	110.6

4.4 La politique `RandomPolicy`

Cette politique fixe aléatoirement le prix et la quantité.

4.4.1 Paramètres et usage

- un entier $minQuant$, 0 par défaut
- un entier $maxQuant$, 150 par défaut
- un entier $minPrice$, 14000 par défaut (en cents)
- un entier $maxPrice$, 15000 par défaut (en cents)

```
public RandomPolicy();

public RandomPolicy(int minQuant, int maxQuant, long minPrice, long maxPrice);
```

4.4.2 Fonctionnement

On fixe une quantité *quantity* aléatoire entre $minQuant$ et $maxQuant$. On fixe un prix *price* aléatoire entre $minPrice$ et $maxPrice$.

On retourne un ordre de la direction demandée avec la quantité *quantity* et au prix *price*.

5 Conclusion

Le paquetage `chartists` permet de créer toute une variété de comportements d'agents traders en exploitant les notions de signal, de stratégie et de politique de fixation de prix et quantités. Il est ainsi facile de peupler les simulations réalisées sous ATOM, comme l'illustre la classe `StrategicAgent_Test`.