付 録C　モデル・パターン カタログ

C.1　モデル・パターンの分類

本カタログは、Boxed Economy Foundation Modelに基づくモデルにおいて繰り返し登場するモデル・パターンを記述したものである。取り上げるモデル・パターンは、大きく分けて次のような分類ができる。

- エレメンタリーなモデル・パターン
- コミュニケーションのモデル・パターン
- 行動変化のモデル・パターン
- アクティベーションのモデル・パターン

各分類に属するモデル・パターンの一覧は次のようになる。

モデル・パターンの分類	モデル・パターン名
エレメンタリーなモデル・パターン	Agent Creation (p. 236) Relation Creation (p. 238) Related Agent Creation (p. 240) Agent Destruction (p. 242) Goods Creation (p. 244) Information Creation (p. 246)
コミュニケーションのモデル・パターン	Information Sending (p. 248) Blank Information Sending (p. 252) Internal Information Sending (p. 256) Immediate Reply (p. 260) Collect Immediate Replies (p. 264) Appointed Destination Reply (p. 268) Super BehaviorType Calling (p. 272)
行動変化のモデル・パターン	Behavior Creation (p. 276) Behavior Destruction (p. 278) Behavior Switching (p. 280) Temporary Behavior Creation (p. 282) Requested Behavior Attachment (p. 284) Forced Behavior Attachment (p. 288)
アクティベーションのモデル・パターン	TimeEvent Distributer Agent (p. 290) TimeEvent Filtering (p. 294) TimeEvent Distributer Behavior (p. 296) Time-Consuming Behavior (p. 298)

C.2　パターンにおけるクラス名・オブジェクト名について

基本動作、設計、サンプルコードに記載されているエージェント名などは、ここでの説明用の名称がつけられている。モデルにおいて現れるときには、そのコンテクストにあった名前をつける必要がある。

C.3　設計におけるオブジェクト図について

本カタログの「設計」の部分は、オブジェクト図を変形した図で記述している。この図は、モデルにおいて登場するすべてのモデル要素を記述しているため、正しいオブジェクト図とはいえない。なぜなら、本来オブジェクト図は、あるシステムのある特定の時間における状態のスナップショットを記述するものだからである。Type オブジェクトによるモデル要素の表現の関係から、本カタログでは、このような変形版オブジェクト図を用いる。

状態遷移図では、そのパターンに関する処理を行う action については、その目的をわかりやすくするための名前をつけている (そうでない場合には、nextAction という名前をつけてある)。これらの action 名も、モデルのコンテクストに合った名前に付けなおすことが望ましい。

C.4　サンプルコードについて

World クラスを継承した「〜World」は、initializeWorld メソッドと initializeAgents メソッドをオーバーライドする。本カタログに示すサンプルはすべて、以下のような initializeWorld メソッドを想定している。

【〜World クラス】

```
...
  public void initializeWorld() {
     super.initializeWorld();

     //時計に StepClock を設定
     this.setClock(new StepClock());
  }
...
```

initializeAgents メソッドは、モデルによってその処理の内容が異なるため、カタログごとにサンプルコードを掲載している。「〜World」の全体像は、次のようになる (AgentCreationWorld クラスの例)。

【AgentCreationWorld クラス】

```
package org.boxed_economy.agentcreation.model;

import org.boxed_economy.besp.container.BESP;
import org.boxed_economy.besp.model.fmfw.*;
import org.boxed_economy.components.stepclock.StepClock;


public class AgentCreationWorld extends World {

   //World の初期化

   public void initializeWorld() {
      super.initializeWorld();

      //時計に StepClock を設定
      this.setClock(new StepClock());
   }


   //Agent の初期化

   public void initializeAgents() {

      //AgentCreator エージェントの生成
      Agent agentCreator = createAgent(AgentCreationModel.AGENTTYPE_AgentCreator);

      //そのエージェントへの CreateAgentBehavior の追加
      agentCreator.addBehavior(AgentCreationModel.BEHAVIORTYPE_CreateAgent);
   }
}
```

C.5　バリエーションについて

ふつう、同じ目的を満たすようなモデルは、いくつか存在する。モデル・パターンでは典型的なサンプルモデルをあげているが、常にこれが最良であるというわけではない。「バリエーション」の項では、代替的な案について、若干補足している。これらを参考に、適用する文脈に適した形で変形して利用することが期待される。

なお、サンプルモデルでは、登場するエージェントが、それぞれ異なる AgentType をもつというモデル化をしているが、多くの場合、そのようにする必要はない。説明の便宜上のものだと考えてほしい。

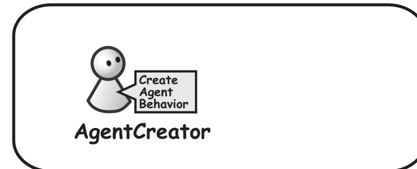
Agent Creation

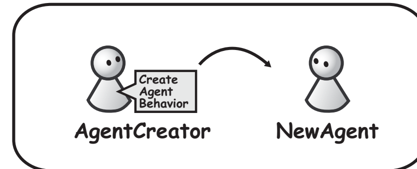
■ 目的

新しいエージェントを生成する。

■ 動機

人口が増減するモデルや、組織が形成・解体されるモデルでは、シミュレーション実行中に、新しいエージェントを生成し、世界に追加する必要がある。このエージェント生成処理を内生化したい場合には、モデル内のいずれかのエージェントが、生成処理を行う必要がある。



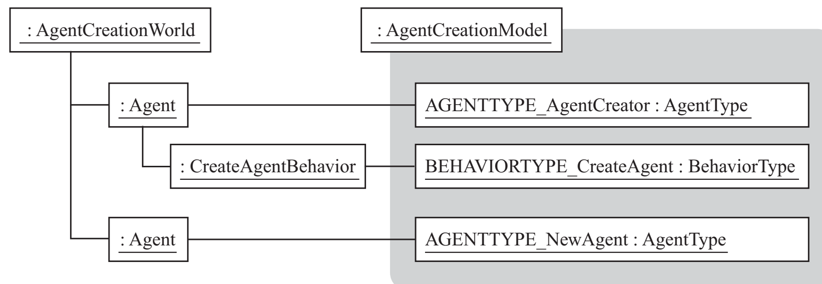


■ 基本動作

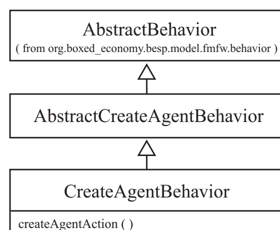
AgentCreator エージェントは CreateAgentBehavior を持っており、この CreateAgentBehavior によって、NewAgent エージェントを生成する。

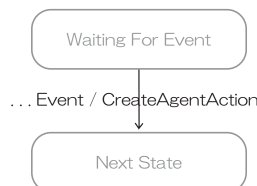
■ 設計

【全体像】



【CreateAgentBehavior】





■ サンプルコード

【AgentCreationWorld クラス】

```
…
public void initializeAgents() {
    //AgentCreator エージェントの生成
    Agent agentCreator = createAgent(AgentCreationModel.AGENTTYPE_AgentCreator);

    //そのエージェントへの CreateAgentBehavior の追加
    agentCreator.addBehavior(AgentCreationModel.BEHAVIORTYPE_CreateAgent);
}
…
```

【CreateAgentBehavior クラス】

```
…
protected void createAgentAction() {
    //新しいエージェントの生成
    Agent createAgent = this.getWorld().createAgent(AgentCreationModel.AGENTTYPE_NewAgent);
}
…
```

■ バリエーション

ここでのサンプルでは、生成したエージェントに行動をもたせていないため、このエージェントは何もしない。何らかの行動をさせたい場合には、このエージェントに行動を付加する必要がある (→ Forced Behavior Attachment 参照)。

■ 関連するパターン

Related Agent Creation: 新しくエージェントを生成して、そのエージェントに関係を結ぶ。

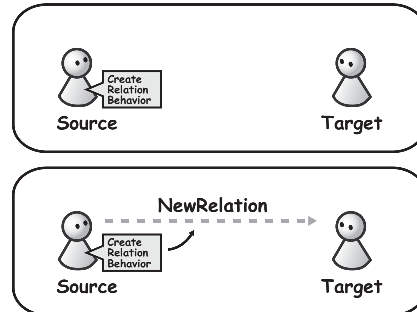
Relation Creation

■ 目的

他のエージェントとの関係を生成する。

■ 動機

エージェント間の関係が変化したり、新しい関係性が生じるモデルでは、シミュレーション実行中に、エージェント間の関係を生成して結ぶ必要がある。この関係生成処理を内生化したい場合には、モデル内のいずれかのエージェントが、この生成処理を行う必要がある。

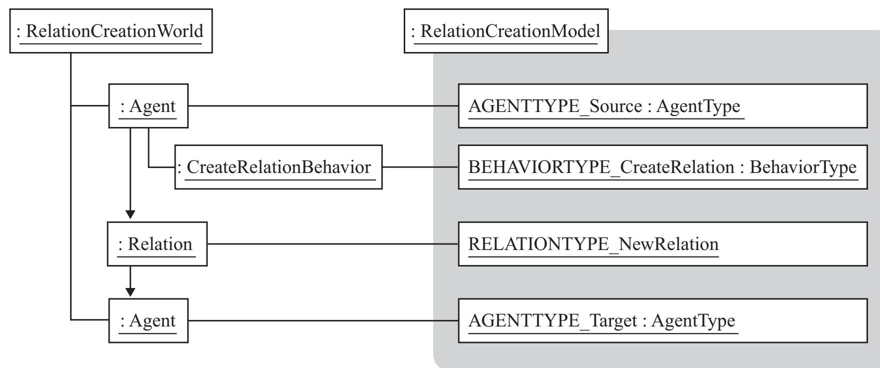


■ 基本動作

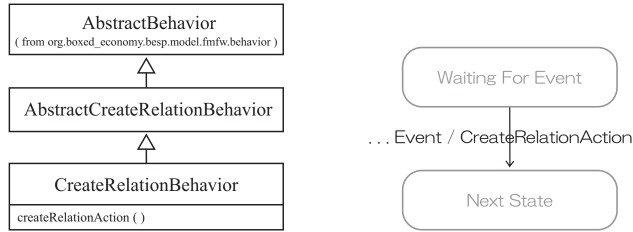
Source エージェントと Target エージェントが登場する。Source エージェントは CreateRelationBehavior を持っており、この CreateRelationBehavior によって NewRelation を生成し、Target エージェントと関係を結ぶ。

■ 設計

【全体像】



【CreateAgentBehavior】



■ サンプルコード

【RelationCreationWorld クラス】

```
…
public void initializeAgents() {
   //Source エージェントと Target エージェントの生成
   Agent source = createAgent(RelationCreationModel.AGENTTYPE_Source);
   Agent target = createAgent(RelationCreationModel.AGENTTYPE_Target);

   //Source エージェントへの行動の追加
   source.addBehavior(RelationCreationModel.BEHAVIORTYPE_CreateRelation);
}
…
```

【CreateRelationBehavior クラス】

```
…
protected void createRelationAction() {
   //関係を結ぶ相手の特定
   Agent target = this.getWorld().getAgent(RelationCreationModel.AGENTTYPE_Target);
   //相手と関係を結ぶ
   this.getAgent().addRelation(RelationCreationModel.RELATIONTYPE_NewRelation, target);
}
…
```

■ バリエーション

ここでのサンプルでは、関係を結ぶエージェントの特定化に、AgentType を用いている。このほかの代替案としては、(1) エージェントを特定化するための情報を受取り、それを用いて指定する、(2) 世界に存在するエージェントを調べて、その中から選択して指定する、という方法が考えられる。

■ 関連するパターン

Related Agent Creation: 新しくエージェントを生成して、そのエージェントに関係を結ぶ。

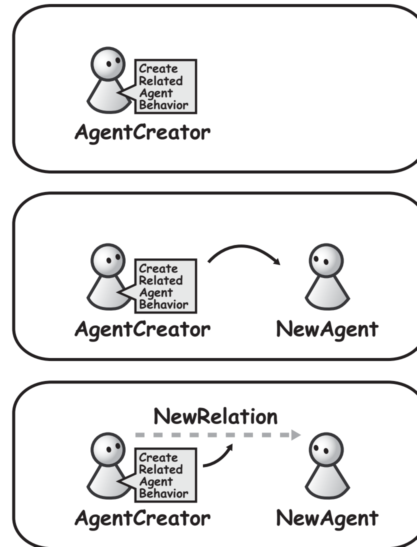
Related Agent Creation

■ 目的

新しいエージェントを生成し、関係を結ぶ。

■ 動機

エージェントの生成を伴うモデルでは、生成処理を行うエージェントと、新しく生成されたエージェントの間に、なんらかの関係を持たせたいということがしばしばある。例えば、個人エージェントが子供を生む場合は、それらの間に親子関係を結ぶことになるだろう。また、個人もしくは組織エージェントが、新しい組織を形成したり、内部組織を分化させる場合にも、生成処理を行ったエージェントとなんらかの関係を結ぶことが想定される。

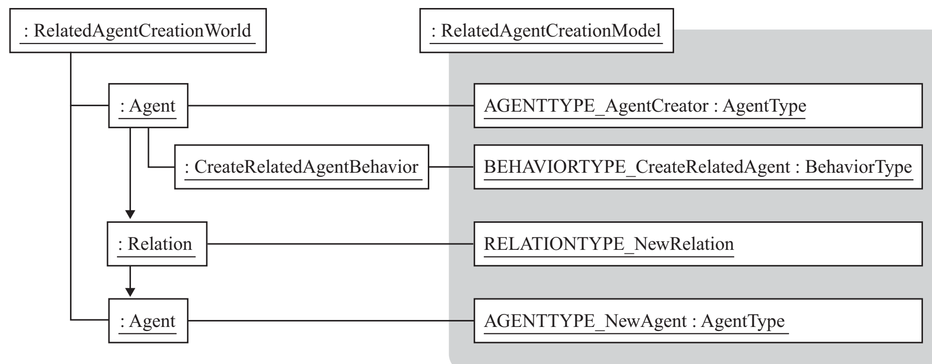


■ 基本動作

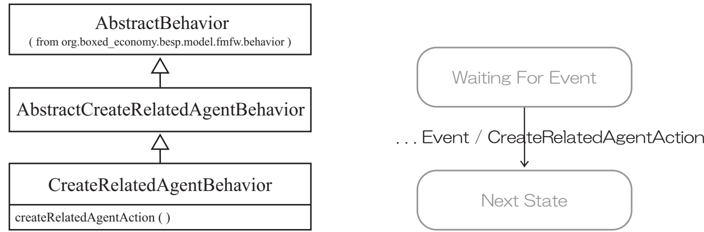
AgentCreator エージェントは CreateRelatedAgentBehavior を持っている。この CreateRelatedAgentBehavior によって、NewAgent エージェントを生成した後、NewRelation を結ぶ。

■ 設計

【全体像】



【CreateRelatedAgentBehavior】



■ サンプルコード

【RelatedAgentCreationWorld クラス】

```
 …
public void initializeAgents() {
   //AgentCreator エージェントの生成
   Agent agentCreator = createAgent(AgentCreationModel.AGENTTYPE_AgentCreator);

   //そのエージェントへの CreateAgentBehavior の追加
   agentCreator.addBehavior(RelatedAgentCreationModel.BEHAVIORTYPE_CreateRelatedAgent);
}
 …
```

【CreateRelatedAgentBehavior クラス】

```
 …
protected void createRelatedAgentAction() {
   //新しいエージェントの生成
   Agent createAgent =
      this.getWorld().createAgent(RelatedAgentCreationModel.AGENTTYPE_NewAgent);

   //自分から新しいエージェントへの関係の設定
   this.getAgent().
      addRelation(RelatedAgentCreationModel.RELATIONTYPE_NewRelation, createdAgent);
}
 …
```

■ バリエーション

ここでのサンプルでは、生成したエージェントに行動をもたせていないため、このエージェントは何もしない。何らかの行動をさせたい場合には、このエージェントに行動を付加する必要がある (→ Forced Behavior Attachment 参照)。

■ 関連するパターン

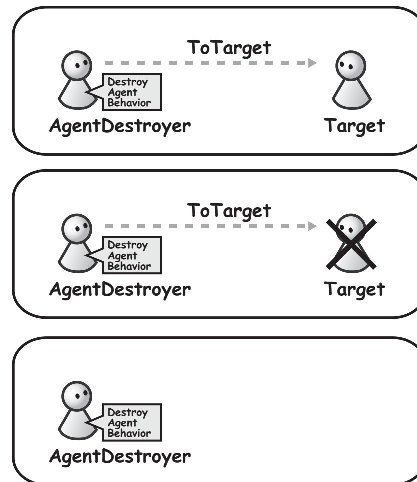
Agent Creation: 関係をもたないエージェントを生成する。
Relation Creation: すでに存在するエージェントと関係を結ぶ。

Agent Destruction

■ 目的

他のエージェントを消滅させる。

■ 動機



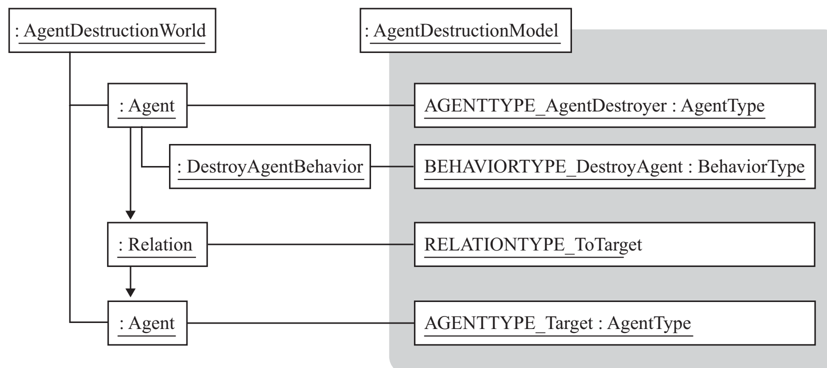
人口が増減するモデルや、組織が形成・解体されるモデルでは、シミュレーション実行中に、エージェントを消滅させる必要がでてくる。例えば、個人エージェントの死亡や、組織エージェントの解散などが、これにあたる。このエージェント消滅処理を内生化したい場合には、モデル内のいずれかのエージェントが、消滅処理を行う必要がある。

■ 基本動作

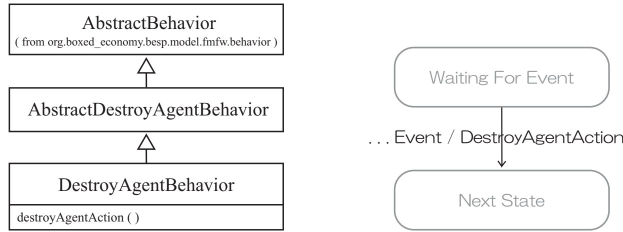
AgentDestroyer エージェントは DestroyAgent-Behavior を持っている。この DestroyAgentBehavior によって、Target エージェントを消滅させる (このとき、関係は自動的に消滅する)。

■ 設計

【全体像】



【DestroyAgentBehavior】



■ サンプルコード

【AgentDestructionWorld クラス】

```
…
public void initializeAgents() {

   //AgentDestroyer エージェントの生成
   Agent agentDestroyer = createAgent(AgentDestructionModel.AGENTTYPE_AgentDestroyer);

   //そのエージェントへの DestroyAgentBehavior の追加
   agentDestroyer.addBehavior(AgentDestructionModel.BEHAVIORTYPE_DestroyAgent);

   //消滅させる Target エージェントの生成
   Agent target = createAgent(AgentDestructionModel.AGENTTYPE_Target);

   //AgentDestroyer エージェントから Target エージェントへの関係を結ぶ
   agentDestroyer.addRelation(AgentDestructionModel.RELATIONTYPE_ToTarget, target);
}
 …
```

【DestroyAgentBehavior クラス】

```
…
protected void destroyAgentAction() {
   //削除するエージェントの特定
   Agent target = this.getAgent()
      .getRelation(AgentDestructionModel.RELATIONTYPE_ToTarget).getTarget();

   //target のエージェントの削除
   this.getWorld().destroyAgent(target);
}
 …
```

■ バリエーション

ここでのサンプルでは、消滅させるエージェントの特定化に、AgentType を用いている。このほかの代替案としては、(1) エージェントを特定化するための情報を受取り、それを用いて指定する、(2) 世界に存在するエージェントを調べて、その中から選択して指定する、という方法が考えられる。

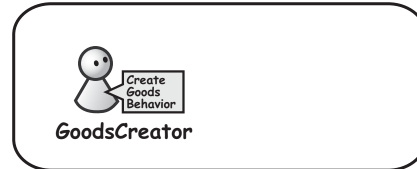
Goods Creation

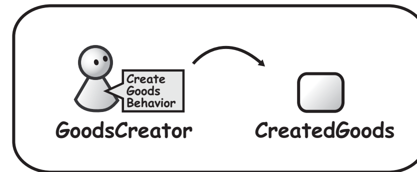
■ 目的

財を生成する。

■ 動機

商品を生産したり調達したりするモデルでは、シミュレーション実行中に、財を生成する必要がある。この財生成処理を内生化したい場合には、モデル内のいずれかのエージェントが、生成処理を行う必要がある。



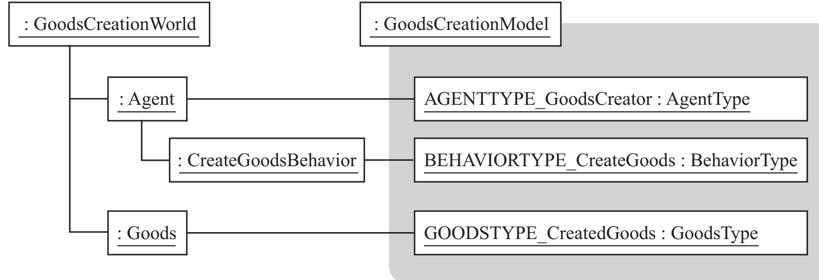


■ 基本動作

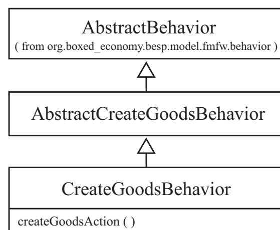
GoodsCreator エージェントは CreateGoodsBehavior を持っている。この CreateGoodsBehavior によって、CreatedGoods を生成する。

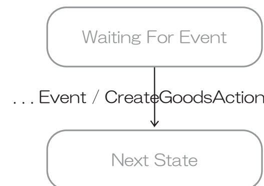
■ 設計

【全体像】



【CreateGoodsBehavior】





■ サンプルコード

【GoodsCreationWorld クラス】

```
…
public void initializeAgents() {
   //GoodsCreator エージェントの生成
   Agent goodsCreator = createAgent(GoodsCreationModel.AGENTTYPE_GoodsCreator);

   //そのエージェントへの CreateGoodsBehavior の追加
   goodsCreator.addBehavior(GoodsCreationModel.BEHAVIORTYPE_CreateGoods);
}
…
```

【CreateGoodsBehavior クラス】

```
…
protected void createGoodsAction() {
   //Goods の作成
   Goods createdGoods =
      this.getWorld().createGoods(GoodsCreationModel.GOODSTYPE_CreatedGoods, 1.0);
}
…
```

■ バリエーション

このサンプルで行っているのは、財を生成することのみである。この後に行う動作として考えられるのは、(1) 財を他のエージェントに送るという動作であり、それは、Behavior の sendGoods( ) で行う。あるいは、(2) 自分の所有財として保管するという動作も考えられ、それは、Agent の addGoods( ) で行う。

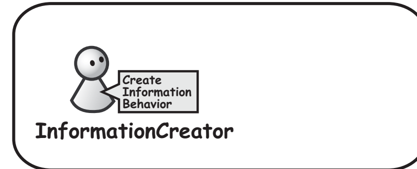
Information Creation

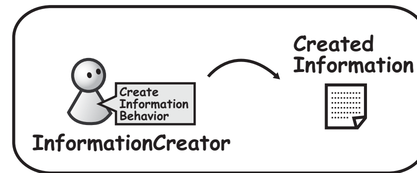
■ 目的

情報を作成する。

■ 動機

エージェント間の相互作用を行う場合や、何らかのデータを記憶したい場合には、シミュレーション実行中に、エージェントが情報を作成する必要がある。



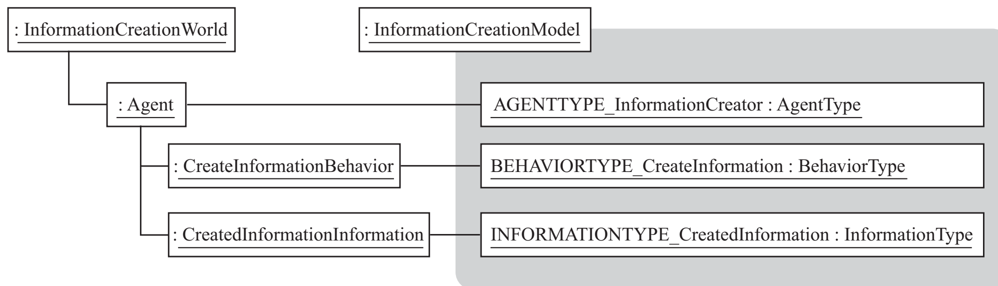


■ 基本動作

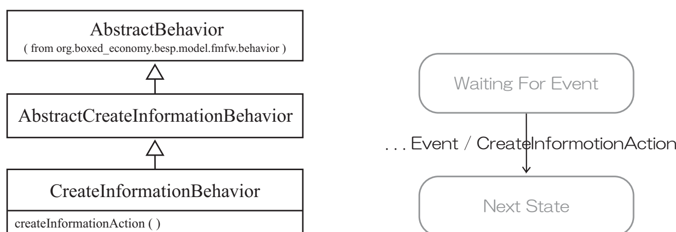
InformationCreator エージェントは、CreateInformationBehavior もっており、この CreateInformationBehavior によって CreatedInformation を作成する。

■ 設計

【全体像】



【CreateInformationBehavior】



■ サンプルコード

【InformationCreationWorld クラス】

```
…
public void initializeAgents() {
   //InformationCreator エージェントの生成
   Agent informationCreator =
      createAgent(InformationCreationModel.AGENTTYPE_InformationCreator);

   //そのエージェントへの CreateInformationBehavior の追加
   informationCreator.addBehavior(
      InformationCreationModel.BEHAVIORTYPE_CreateInformation);
}
…
```

【CreateInformationBehavior クラス】

```
…
protected void createInformationAction() {
   //情報の生成
   CreatedInformation createdInformation = new CreatedInformation();

   //情報の追加
   this.getAgent().putInformation(createdInformation);
}
…
```

【CreatedInformation クラス】

```
…
   public class CreatedInformation implements Information {
      //ここに情報の形式、および設定・取得のためのメソッド等を書く
      …
   }
…
```

■ 関連するパターン

Information Sending: 情報を送る。

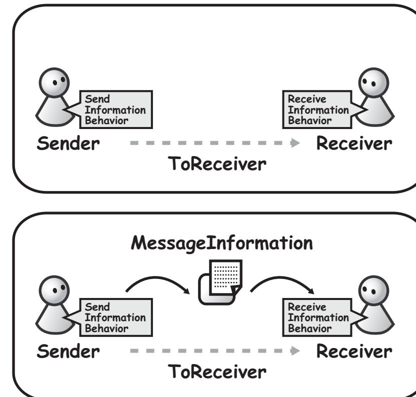
Information Sending

■ 目的

他のエージェントに、Information を送信する。

■ 動機

エージェントが、他のエージェントに何らかのメッセージを送ったり、質問をしたりするようにしたい。

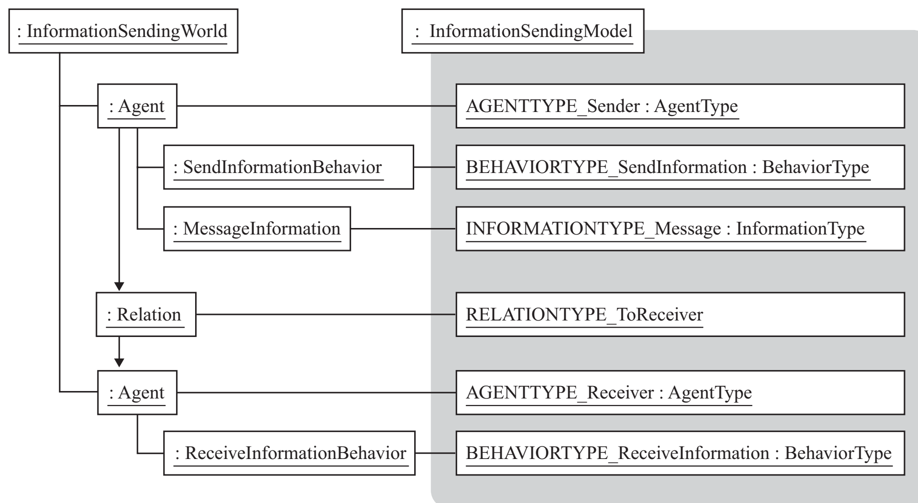


■ 基本動作

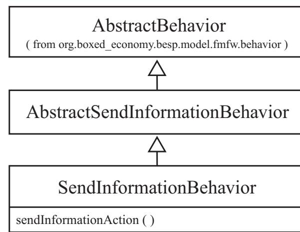
Sender エージェントと Receiver エージェントが登場する。Sender エージェントは、SendInformation-Behavior をもっている。SendInformationBehavior は、(ここでは) MessageInformation を生成し、Receiver エージェントに送信する。Receiver エージェントは、ReceiveInformationBehavior でそれを受ける。

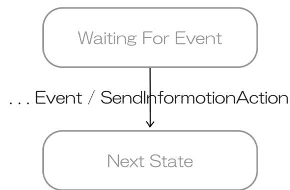
■ 設計

【全体像】

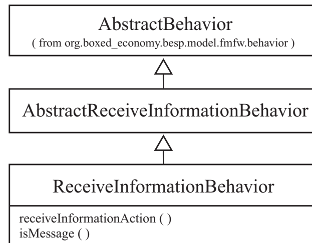


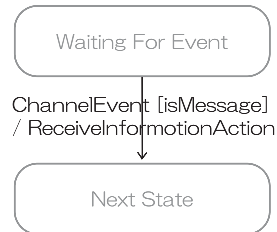
【SendInformationBehavior】



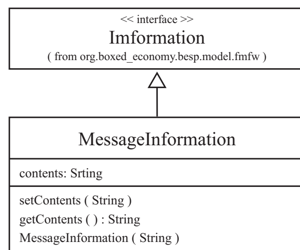


【ReceiveInformationBehavior】





【MessageInformation】



■ サンプルコード

【InformationSendingWorld クラス】

```
…
public void initializeAgents() {
   //Sender エージェントの生成
   Agent sender = createAgent(InformationSendingModel.AGENTTYPE_Sender);
   //Sender エージェントへの SendInformationBehavior の追加
   sender.addBehavior(InformationSendingModel.BEHAVIORTYPE_SendInformation);

   //Receiver エージェントの生成
   Agent receiver = createAgent(InformationSendingModel.AGENTTYPE_Receiver);
   //Receiver エージェントへの ReceiveInformationBehavior の追加
   receiver.addBehavior(InformationSendingModel.BEHAVIORTYPE_ReceiveInformation);

   //Sender エージェントから Receiver エージェントへの関係の追加
   sender.addRelation(InformationSendingModel.RELATIONTYPE_ToReceiver, receiver);
}
…
```

【SendInformationBehavior クラス】

```
…
protected void sendInformationAction() {
   //情報の生成 (ここでは、メッセージの内容は文字列)
   MessageInformation message = new MessageInformation("Hello!");

   //生成した情報の送信
   this.sendInformation(InformationSendingModel.RELATIONTYPE_ToReceiver,
      InformationSendingModel.BEHAVIORTYPE_ReceiveInformation, message);
}
…
```

【ReceiveInformationBehavior クラス】

```
…
protected void receiveInformationAction() {
   //受信した情報を取得
   Information receivedInformation = getReceivedInformation();
   //メッセージの内容を取得 (ここでは文字列)
   String messageContents =
      ((MessageInformation)receivedInformation).getContents();
}

protected boolean isMessage(Event e) {
   //送られてきた情報が、MessageInformation であれば true を返す。
   return this.getWorld().getInformationType(getReceivedInformation())
      == InformationSendingModel.INFORMATIONTYPE_Message;
}
…
```

【MessageInformation クラス】

```
…
public class MessageInformation implements Information {
   //ここでは、内容は文字列
   String contents;

   //コンストラクタ (引数 = 文字列)
   public MessageInformation(String contents) {
      this.contents = contents;
   }

   //内容を返す
   public String getContents() {
      return contents;
   }

   //内容を設定する
   public void setContents(String contents) {
      this.contents = contents;
   }
}
…
```

■ バリエーション

このサンプルでは、文字列の内容を記録する MessageInformation という情報を送っているが、それ以外の情報形式でも構わない。文脈に応じて自由に設計することができる。

また、このサンプルでは、どのような情報が送られてきたのかを、InformationType で判断させているが、情報の内容で判別させることもできる。

■ 関連するパターン

Information Creation: Information を生成する。
Blank Information Sending: 内容を伴わないシグナルを送る。

Blank Information Sending

■ 目的

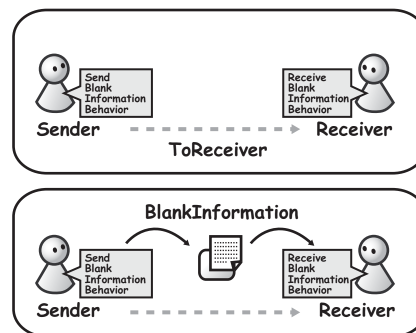
他のエージェントに、内容が空の Information を送信する。

■ 動機

単に合図や質問を送りたいときには、相手に情報の種類が伝わればよいということがしばしばある。このような場合には、内容を伴わない Information を送るだけで済ましたい。

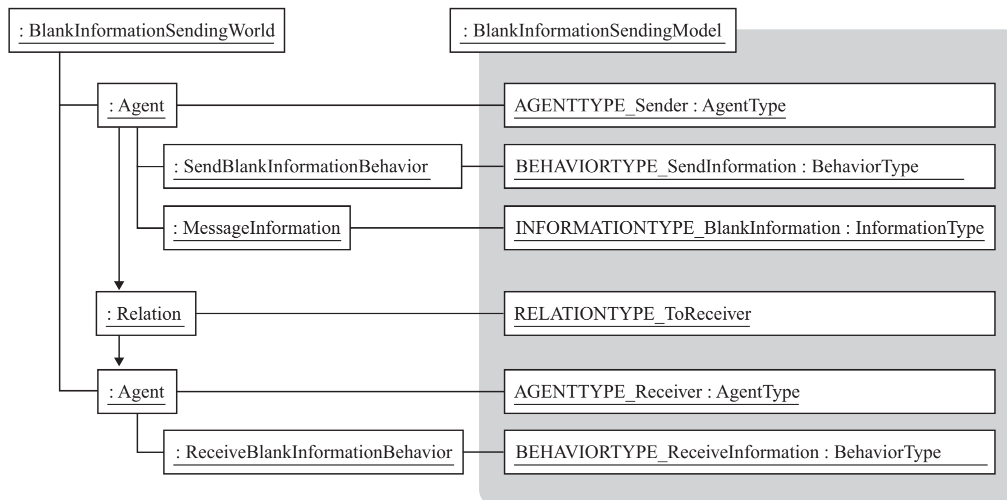
■ 基本動作

Sender エージェントと Receiver エージェントが登場する。Sender エージェントは、SendBlankInformationBehavior をもっている。SendBlankInformationBehavior では、内容を伴わない BlankInformation を生成し、Receiver エージェントに送信する。Receiver エージェントは、ReceiveBlankInformationBehavior でそれを受ける。どのような情報が送られてきたのかの識別は、BlankInformation の InformationType で判断する。

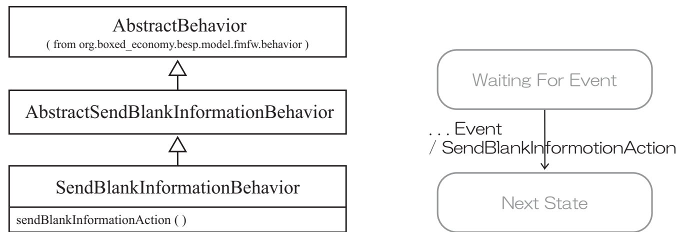


■ 設計

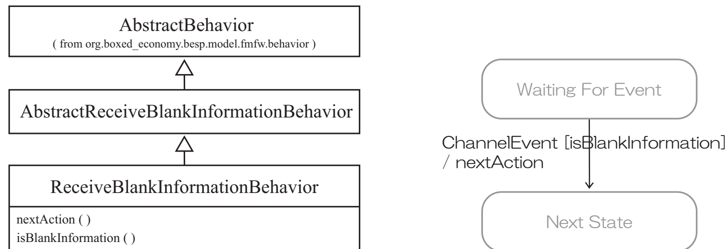
【全体像】



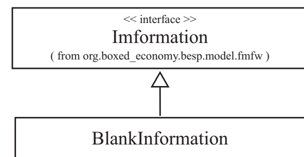
【SendBlankInformationBehavior】



【ReceiveBlankInformationBehavior】



【BlankInformation】



■ サンプルコード

【BlankInformationSendingWorldクラス】

```
…
public void initializeAgents() {
   //Sender エージェントの生成
   Agent sender = createAgent(BlankInformationSendingModel.AGENTTYPE_Sender);
   //Sender エージェントへの SendInformationBehavior の追加
   sender.addBehavior(BlankInformationSendingModel.BEHAVIORTYPE_SendBlankInformation);

   //Receiver エージェントの生成
   Agent receiver = createAgent(BlankInformationSendingModel.AGENTTYPE_Receiver);
   //Receiver エージェントへの ReceiveInformationBehavior の追加
   receiver.addBehavior(
      BlankInformationSendingModel.BEHAVIORTYPE_ReceiveBlankInformation);

   //Sender エージェントから Receiver エージェントへの関係の追加
   sender.addRelation(BlankInformationSendingModel.RELATIONTYPE_ToReceiver, receiver);
}
…
```

【SendBlankInformationBehavior クラス】

```
…
protected void sendBlankInformationAction() {
   //空の情報の生成
   BlankInformation blankInformation = new BlankInformation();

   //その情報の送信
   this.sendInformation(BlankInformationSendingModel.RELATIONTYPE_ToReceiver,
      BlankInformationSendingModel.BEHAVIORTYPE_ReceiveBlankInformation,
       blankInformation);
}
…
```

【ReceiveBlankInformationBehavior クラス】

```
…
protected void nextAction() {

   //BlankInformation を受け取ったら行うアクション

}

protected boolean isBlankInformation(Event e) {
   //送られてきた情報が、BlankInformation であれば true を返す。
   return this.getWorld().getInformationType(getReceivedInformation())
      == BlankInformationSendingModel.INFORMATIONTYPE_BlankInformation;
}
…
```

【BlankInformation クラス】

```
…
public class BlankInformation implements Information {
   //内容は空でよい
}
```

■ 関連するパターン

Information Creation: Information を生成する。
Information Sending: 内容を伴う Information を送信する。

Model Patterns

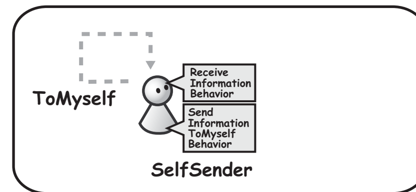
Internal Information Sending

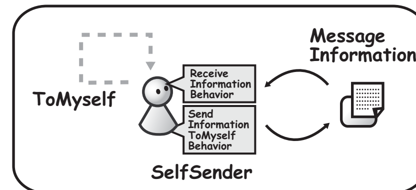
■ 目的

自分自身 (エージェント) に対して、Information を送信する。

■ 動機

エージェントが複数の行動をもっている場合、それらの行動を連携させたいことがある。例えば、他のエージェントとのコミュニケーションを行う行動が集めた情報を、意思決定・戦略行動が利用するという場合などである。行動間の単なる情報共有であれば、情報を記憶し、それを取り出すことで共有できるが、行動をアクティベートして何らかの処理をさせたい場合には、直接情報を送る方がわかりやすい。



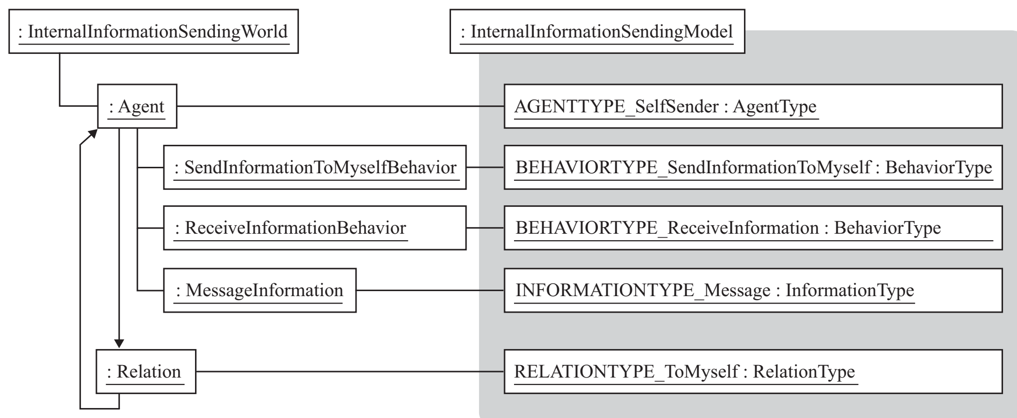


■ 基本動作

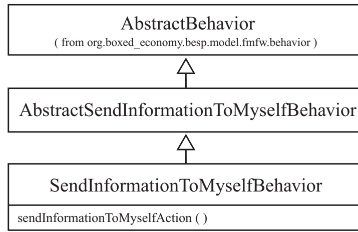
SelfSender エージェントは、SendInformationBehavior と ReceiveInformationBehavior をもっている。SendInformationBehavior では、MessageInformation を生成し、自分自身に送信する。SelfSender エージェントは、ReceiveInformationBehavior でそれを受け取る。

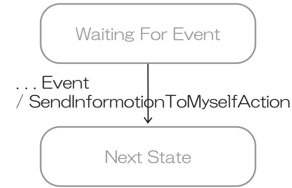
■ 設計

【全体像】

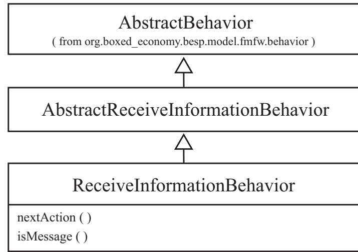


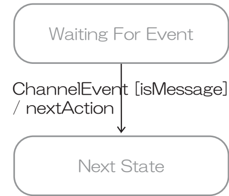
【SendInformationBehavior】



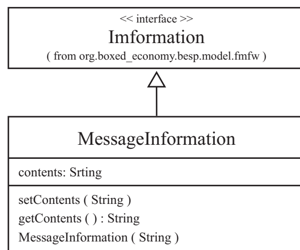


【ReceiveInformationBehavior】





【MessageInformation】



■ サンプルコード

【InternalInformationSendingWorld クラス】

```
…
public void initializeAgents() {
   //SelfSender エージェントの生成
   Agent selfSender = this.createAgent(
      InternalInformationSendingModel.AGENTTYPE_SelfSender);

   //SelfSender エージェントに、SendInformationBehavior、
   //および ReceiveInformationBehavior を追加する
   selfSender.addBehavior(
      InternalInformationSendingModel.BEHAVIORTYPE_SendInformation);
   selfSender.addBehavior(
      InternalInformationSendingModel.BEHAVIORTYPE_ReceiveInformation);

   //SelfSender の自分自身への Relation の追加
   selfSender.addRelation(
      InternalInformationSendingModel.RELATIONTYPE_ToMyself,
      selfSender);
}
…
```

【SendInformationBehavior クラス】

```
…
protected void sendInformationToMyselfAction() {
   //送信する情報の作成
   MessageInformation message = new MessageInformation("Fight!");

   //作成した情報を自分に送信する
   sendInformation(
      InternalInformationSendingModel.RELATIONTYPE_ToMyself,
      InternalInformationSendingModel.BEHAVIORTYPE_ReceiveInformation,
      message);
}
…
```

【ReceiveInformationBehavior クラス】

```
…
protected void receiveInformationAction() {
   //受信した情報を取得
   Information receivedInformation = getReceivedInformation();
   //メッセージの内容を取得 (ここでは文字列)
   String messageContents =
      ((MessageInformation)receivedInformation).getContents();
}

protected boolean isMessage(Event e) {
   //送られてきた情報が、MessageInformation であれば true を返す。
   return this.getWorld().getInformationType(getReceivedInformation())
      == InformationSendingModel.INFORMATIONTYPE_Message;
}
…
```

【MessageInformation クラス】

```
…
public class MessageInformation implements Information {
   //ここでは、内容は文字列
   String contents;

   //コンストラクタ (引数 = 文字列)
   public MessageInformation(String contents) {
      this.contents = contents;
   }

   //内容を返す
   public String getContents() {
      return contents;
   }

   //内容を設定する
   public void setContents(String contents) {
      this.contents = contents;
   }
}
…
```

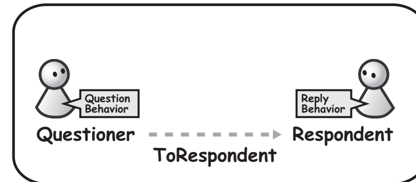
■ 関連するパターン

Information Creation: Information を生成する。
Information Sending: 他のエージェントに情報を送る。
Blank Information Sending: 内容を伴わないシグナルを送る。

Immediate Reply

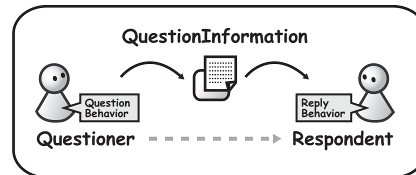
■ 目的

他のエージェントに質問し、直ちに返答を受ける。



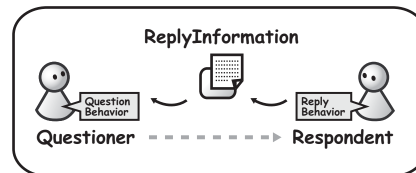
■ 動機

他のエージェントの属性等について知りたい場合に、質問をして問い合わせることがある。



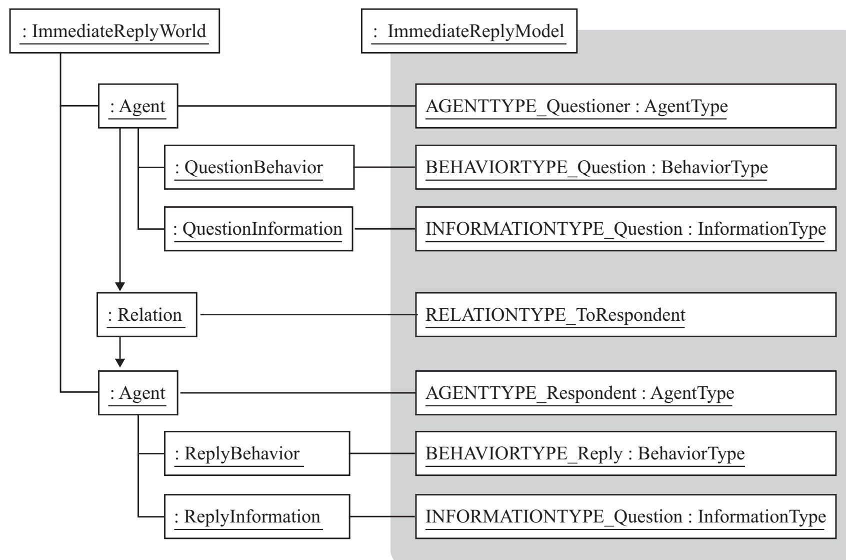
■ 基本動作

Questioner エージェントと Respondent エージェントが登場する。Questioner エージェントは、QuestionBehavior をもっており、これによって QuestionInformation を生成し (ここでは内容は空とする)、Respondent エージェントに送信する。Respondent エージェントは、ReplyBehavior でそれを受けて、直ちに ReplyInformation を送り返す (ここでは文字列の内容をもつとする)。

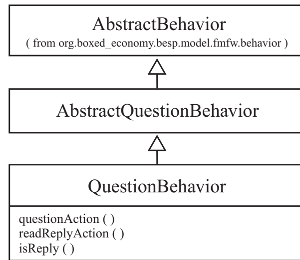


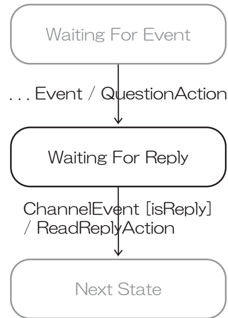
■ 設計

【全体像】

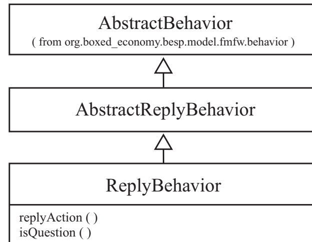


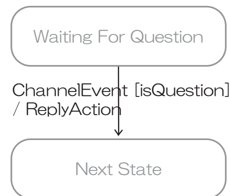
【QuestionBehavior】



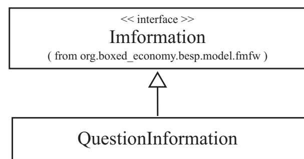


【ReplyBehavior】

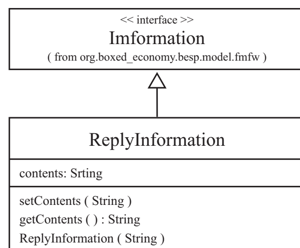




【QustionInformation】



【ReplyInformation】



■ サンプルコード

【ImmediateReplyWorld クラス】

```
…
public void initializeAgents() {
   //Questioner エージェントの生成
   Agent questioner = createAgent(ImmediateReplyModel.AGENTTYPE_Questioner);
   //Questioner エージェントへの QuestionBehavior の追加
   questioner.addBehavior(ImmediateReplyModel.BEHAVIORTYPE_Question);

   //Respondent エージェントの生成
   Agent respondent = createAgent(ImmediateReplyModel.AGENTTYPE_Respondent);
   //Respondent エージェントへの ReplyBehavior の追加
   respondent.addBehavior(ImmediateReplyModel.BEHAVIORTYPE_Reply);

   //Questioner エージェントから Reply エージェントへの関係の追加
   questioner.addRelation(ImmediateReplyModel.RELATIONTYPE_ToRespondent, respondent);
}
…
```

【QuestionBehavior クラス】

```
…
protected void questionAction() {
   //質問 (空の情報) の生成
   QuestionInformation questionInformation = new QuestionInformation();

   //質問の送信
   this.sendInformation(ImmediateReplyModel.RELATIONTYPE_ToRespondent,
      ImmediateReplyModel.BEHAVIORTYPE_Reply, questionInformation);
}

protected void readReplyAction() {
   //送られてきた返信内容の取得
   Information receivedInformation = getReceivedInformation();
   String replyContents = ((ReplyInformation)receivedInformation).getContents();
}

protected boolean isReply(Event e) {
   //送られてきた情報が ReplyInformation であれば true を返す
   return this.getWorld().getInformationType(getReceivedInformation())
      == ImmediateReplyModel.INFORMATIONTYPE_Reply;
}
…
```

【ReplyBehavior クラス】

```
…
protected void replyAction() {
   //返信情報の作成
   ReplyInformation replyInformation = new ReplyInformation("My answer is Yes.");

   //その返信情報の送信 (現在開いている Channel へ送信)
   this.sendInformation(replyInformation);
}

protected boolean isQuestion(Event e) {
```

```
    //送られてきた情報が QuestionInformation であれば true を返す
    return this.getWorld().getInformationType(getReceivedInformation())
        == ImmediateReplyModel.INFORMATIONTYPE_Question;
}
...
```

【QuestionInformation クラス】

```
...
public class QuestionInformation implements Information {
    //ここでは、質問の内容は空
}
```

【ReplyInformation クラス】

```
...
public class ReplyInformation implements Information {
    //ここでは、内容は文字列
    String contents;

    //コンストラクタ (引数 = 文字列)
    public ReplyInformation(String contents) {
        this.contents = contents;
    }

    //内容を返す
    public String getContents() {
        return contents;
    }

    //内容を設定する
    public void setContents(String contents) {
        this.contents = contents;
    }
}
...
```

■ バリエーション

このサンプルでは、質問を送る行動 (QuestionBehavior) が返答を受け取ったが、これらを分離することもできる。例えば、ReceiveReplyBehavior のように、受取り専用の行動をつくることができる。その場合には、ReplyBehavior は、返答情報の送り先として、明示的に ReceiveReplyBehavior を指定する必要がある。もし事前に ReplyBehavior が返信先を知っていたくない場合 (行動コンポーネントのの独立性のため) には、QuestionBehavior が返信先を指定することもできる (Appointed Destination Reply パターン)。

■ 関連するパターン

Information Creation: Information を生成する部分で使用。
Blank Information Sending: 空のメッセージを送信する場合。
Collect Immediate Replies: 複数のエージェントから、直ちに返答を集める場合。この Immediate Reply パターンを内包している。

Collect Immediate Replies

■ 目的

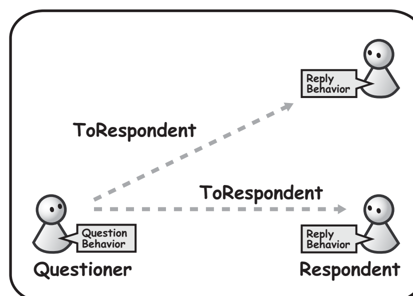
複数のエージェントに質問し、直ちに返された返答を集める。

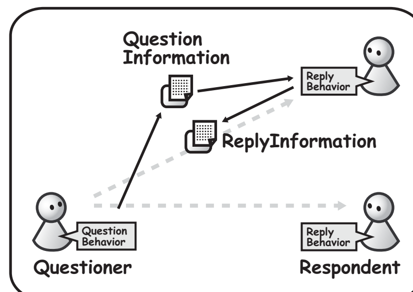
■ 動機

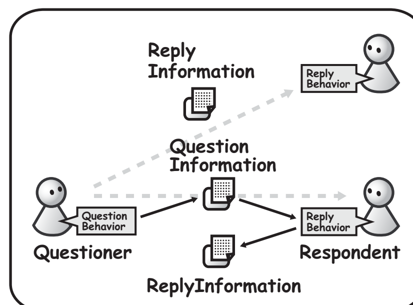
複数のエージェントの属性等を知りたい場合に、質問をして問い合わせせることがある。例えば、注文を集める場合、データの合計・平均を調査したい場合などがこれにあたる。

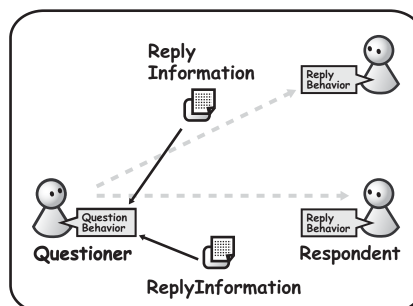
■ 基本動作

Questioner エージェントと複数の Respondent エージェントが登場する。Questioner エージェントは、QuestionBehavior をもっており、これによって QuestionInformation を生成し (内容は空でもよい)、連続して Respondent エージェントに送信していく。Respondent エージェントは、ReplyBehavior でそれを受けて、直ちに ReplyInformation を送り返す。ここで、送り返された ReplyInformation は、一度、Questioner のスタックにたまり、次の Respondent エージェントの ReplyBehavior が実行されて、その返信も、ためられる。すべての Respondent エージェントの返答が終った時点で、Questioner エージェントは、スタックにたまったものをまとめて処理する。



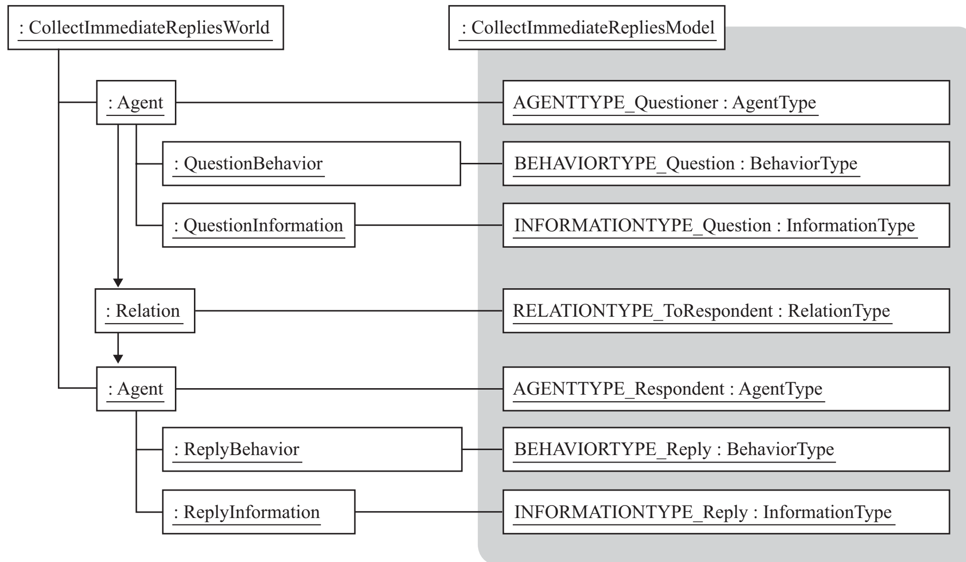




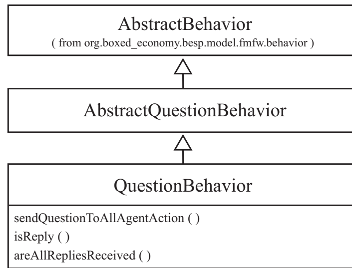


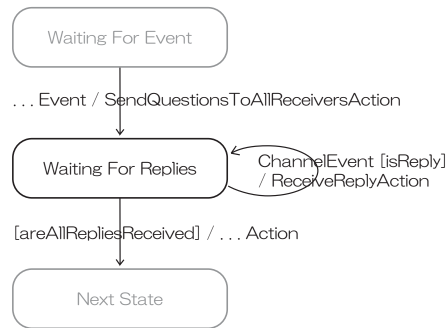
■ 設計

【全体像】

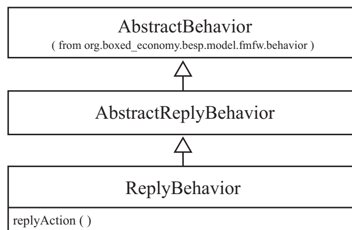


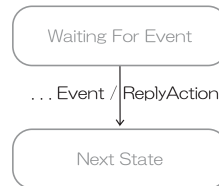
【QuestionBehavior】





【ReplyBehavior】





■ サンプルコード

【CollectImmediateRepliesWorld クラス】

```
…
public void initializeAgents() {

   //Questioner エージェントの生成
   Agent questioner =
      this.createAgent(CollectImmediateRepliesModel.AGENTTYPE_Questioner);

   //Questioner エージェントへの QuestionBehavior の追加
   questioner.addBehavior(CollectImmediateRepliesModel.BEHAVIORTYPE_Question);

   //Respondent エージェントの生成（複数人）
   for (int i = 0; i < 10; i++) {
      //Respondent エージェントの生成
      Agent respondent =
         this.createAgent(CollectImmediateRepliesModel.AGENTTYPE_Respondent);

      //Respondent エージェントへの ReplyBehavior の追加
      respondent.addBehavior(
         CollectImmediateRepliesModel.BEHAVIORTYPE_Reply);

      //Questioner エージェントの Respondent エージェントへの Relation の追加
      questioner.addRelation(
         CollectImmediateRepliesModel.RELATIONTYPE_ToRespondent,respondent);
   }
}
…
```

【QuestionBehavior クラス】

```
…
   //返信の残り数のカウンタ（クラス変数）
   private int receiveCount = 0;
…

protected void sendQuestionToAllAgentAction() {
   // ToRespondent 関係のエージェント全員に対して、
   // QuestionInformation を送り、Respondent の人数を receiveCount に記録する
   this.receiveCount = this.sendInformation(
      CollectImmediateRepliesModel.RELATIONTYPE_ToRespondent,
      CollectImmediateRepliesModel.BEHAVIORTYPE_Reply,
      new QuestionInformation());
}

protected void receiveReplyAction() {
   private List replies = new ArrayList();

   //受け取った ReplyInformation を記録する
   replies.add(this.getReceivedInformation());

   //返信の残り数を減らす
   this.receiveCount--;
}

protected boolean isReply(Event e) {
   //ReplyInformation を受け取った場合に true を返す
```

```
   return this.getWorld().getInformationType(this.getReceivedInformation())
      == CollectImmediateRepliesModel.INFORMATIONTYPE_Reply;
}

protected boolean areAllRepliesReceived(Event e) {
   //QuestionInformation を送った数だけ、返信を受け取った場合に true を返す。
   return this.receiveCount == 0;
}
...
```

【ReplyBehavior クラス】

```
...
protected void replyAction() {
   //ReplyInformation を送り返す
   this.sendInformation(new ReplyInformation());
}

protected boolean isQuestion(Event e) {
   //QuestionInformation を受け取った場合、true を返す
   return this.getWorld().getInformationType(this.getReceivedInformation())
      == CollectImmediateRepliesModel.INFORMATIONTYPE_Question;
}
...
```

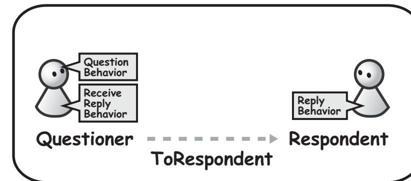
■ 関連するパターン

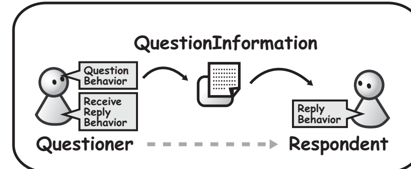
Information Creation: Information を生成する部分で使用。
Blank Information Sending: 空のメッセージを送信する場合。
Immediate Reply: 一人のエージェントに対して。Collect Immediate Replies パターンは、この Immediate Reply パターンを内包している。

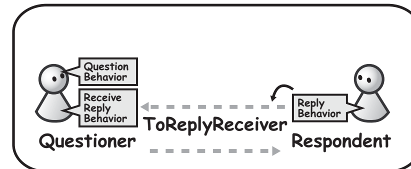
Appointed Destination Reply

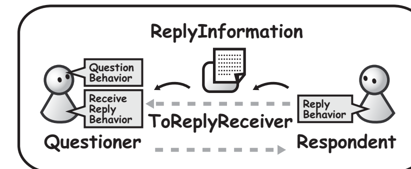
■ 目的

質問を送ったエージェントに、指定した宛先 (Appointed Destination) に返答してもらう。









■ 動機

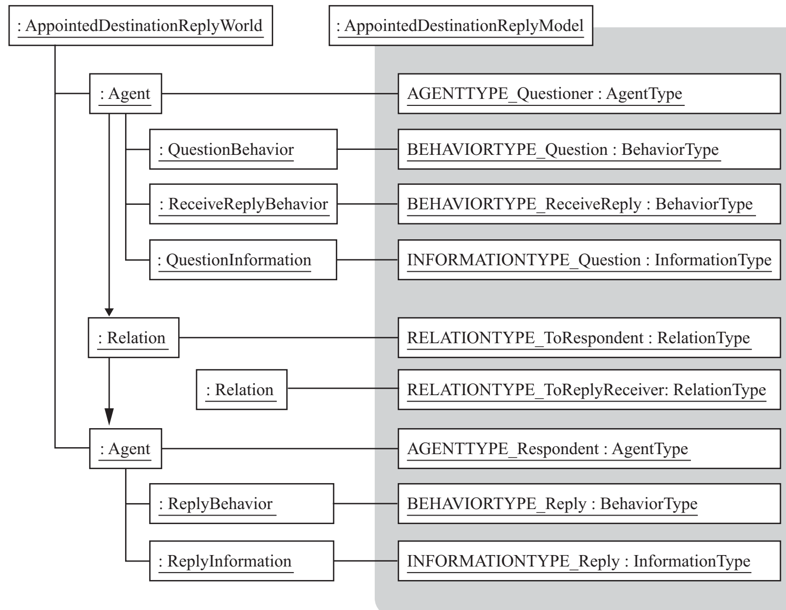
他のエージェントに質問して返答を求める際に、質問した行動以外の行動に返信を求めたいことがある。例えば、質問を受けてから返答するまでの間に時間を要する場合や、返信内容によって返信先が異なる場合などがある。

■ 基本動作

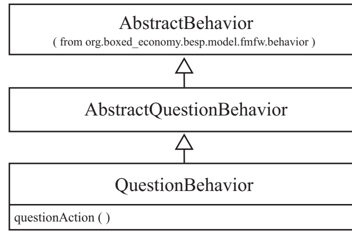
Questioner エージェントと Respondent エージェントが登場する。Questioner エージェントは、QuestionBehavior をもっており、これによって、返答先の Agent と BehaviorType を含む QuestionInformation を生成し、Respondent エージェントに送信する。Respondent エージェントは、ReplyBehavior でそれを受けて、その返答先に ReplyInformation を送る。ReplyBehavior は、ReceiveReplyBehavior のことを事前に知っている必要はない。

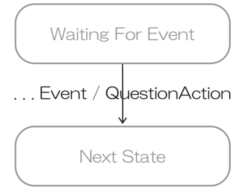
■ 設計

【全体像】

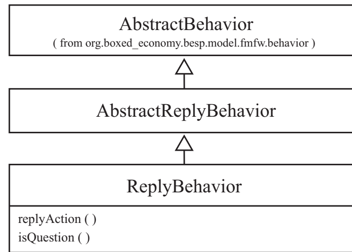


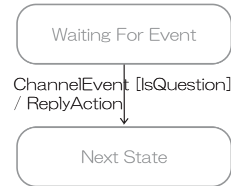
【QuestionBehavior】



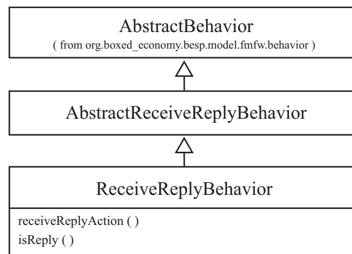


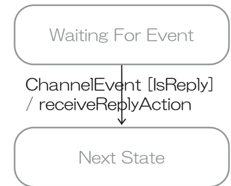
【ReplyBehavior】



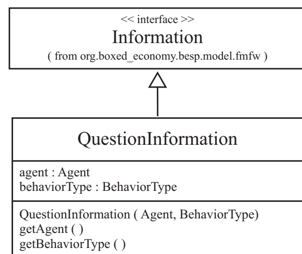


【ReceiveReplyBehavior】





【QuestionInformation】



■ サンプルコード

【AppointedDistinationReplyWorld クラス】

```
…
public void initializeAgents() {
   //Questioner エージェントの生成
   Agent questioner = createAgent(AppointedDestinationReplyModel.AGENTTYPE_Questioner);
   //Questioner エージェントへの QuestionBehavior と ReceiveReplyBehavior の追加
   questioner.addBehavior(AppointedDestinationReplyModel.BEHAVIORTYPE_Question);
   questioner.addBehavior(AppointedDestinationReplyModel.BEHAVIORTYPE_ReceiveReply);

   //Respondent エージェントの生成
   Agent respondent = createAgent(AppointedDestinationReplyModel.AGENTTYPE_Respondent);
   //Respondent エージェントへの ReplyBehavior の追加
   respondent.addBehavior(AppointedDestinationReplyModel.BEHAVIORTYPE_Reply);

   //Questioner エージェントの Respondent エージェントへの Relation の追加
   questioner.addRelation(
      AppointedDestinationReplyModel.RELATIONTYPE_ToRespondent, respondent);
}
…
```

【QuestionBehavior クラス】

```
…
   protected void questionAction() {
   //質問情報の作成
   QuestionInformation question = new QuestionInformation(this.getAgent(),
      AppointedDestinationReplyModel.BEHAVIORTYPE_ReceiveReply);

   //質問情報を送る
   this.sendInformation(AppointedDestinationReplyModel.RELATIONTYPE_ToRespondent,
      AppointedDestinationReplyModel.BEHAVIORTYPE_Reply, question);
}
…
```

【ReplyBehavior クラス】

```
…
protected void replyAction() {
   //受け取った情報を取得する
   QuestionInformation question = (QuestionInformation) this.getReceivedInformation();

   //質問情報から返信先のエージェントと行動を調べる
   Agent appointedAgent = question.getAgent();
   BehaviorType appointedBehaviorType = question.getBehaviorType();

   //指定された返信先エージェントに関係を結ぶ
   this.getAgent().addRelation(
      AppointedDestinationReplyModel.RELATIONTYPE_ToReplyReceiver, appointedAgent);

   //指定された返信先エージェントの指定された行動に返信を送る
   this.sendInformation(AppointedDestinationReplyModel.RELATIONTYPE_ToReplyReceiver,
      appointedBehaviorType, new ReplyInformation());
}

protected boolean isQuestion(Event e) {
```

```
    //QuestionInformation を受け取った場合に true を返す
    return this.getWorld().getInformationType(this.getReceivedInformation())
        == CollectImmediateRepliesModel.INFORMATIONTYPE_Question;
}
...
```

【ReceiveReplyBehavior クラス】

```
...
protected void receiveReplyAction() {
    //受信した情報を取得
    Information receivedInformation = getReceivedInformation();
}

protected boolean isReply(Event e) {
    //送られてきた情報が ReplyInformation であれば true を返す
    return this.getWorld().getInformationType(getReceivedInformation())
        == InformationSendingModel.INFORMATIONTYPE_Reply;
}
...
```

【QuetionInformation クラス】

```
...
public class QuestionInformation implements Information {
    private Agent agent;
    private BehaviorType behaviorType;

    //コンストラクタ（引数 = Agent, BehaviorType）
    public QuestionInformation(Agent agent, BehaviorType behaviorType) {
        this.agent = agent;
        this.behaviorType = behaviorType;
    }

    public Agent getAgent() {
        return this.agent;
    }

    public BehaviorType getBehaviorType() {
        return this.behaviorType;
    }
}
...
```

■ 関連するパターン

Information Creation: Information を生成する。
Collect Immediate Replies: 複数のエージェントから直ちに返答を集める。

Super BehaviorType Calling

■ 目的

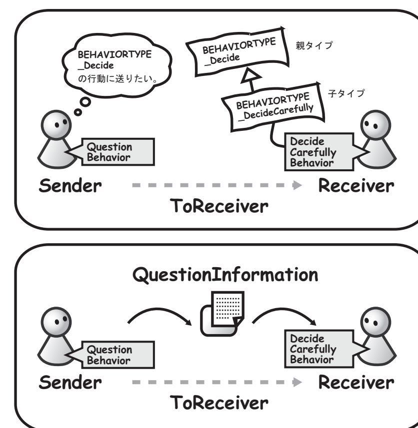
具体的行動ではなく、その行動の親 BehaviorType で指定したい。

■ 動機

戦略行動のように、同種の行動であるが内容が異なるという行動を扱いたいことがある。これらの行動は、シミュレーションの実行中に、状況に応じて切り替えることが多い。そのため、他の行動が、これらの行動と連携・コミュニケーションをはかりたい場合には、指定の仕方に工夫が必要となる。

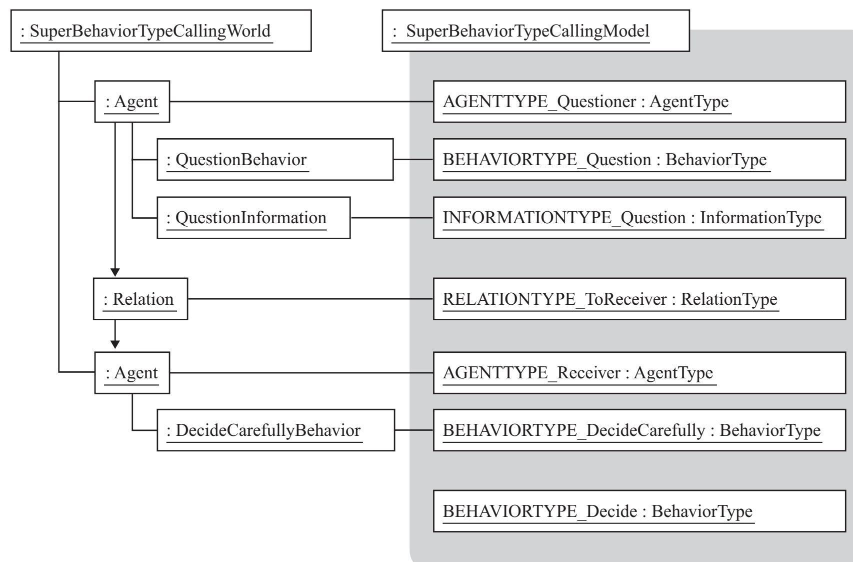
■ 基本動作

Questioner エージェントと Receiver エージェントが登場する。Questioner エージェントは、QuestionBehavior をもっており、それによって QuestionInformation を作成し、Receiver エージェントに送信する。そのとき、送信先は、Receiver エージェントの「DecideBehaviorType」の行動になっている。Receiver エージェントは、この DecideBehaviorType の子タイプ「DecideCarefullyBehaviorType」の行動 (DecideCarefullyBehavior) をもっており、この DecideCarefullyBehavior で、QuestionInformation を受け取る。

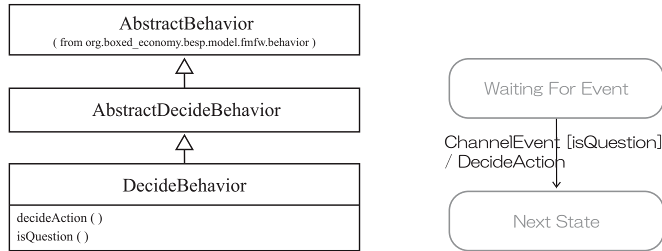


■ 設計

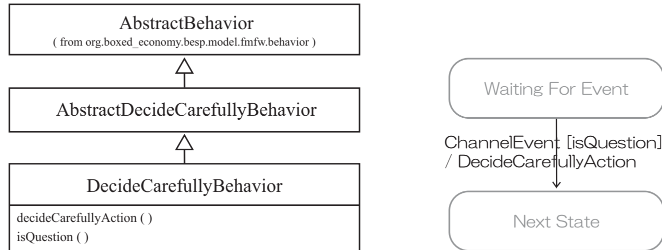
【全体像】



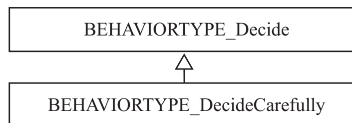
【DecideBehavior】



【DecideCarefullyBehavior】



【タイプの親子関係】



■ サンプルコード

【SuperBehaviorTypeCallingModel クラス】

```
…
private static void initializeTypes(BESPContainer container) {
   //BEHAVIORTYPE_DecideCarefully を、BEHAVIORTYPE_Decide の子タイプに設定
   BEHAVIORTYPE_Decide.addChild(BEHAVIORTYPE_DecideCarefully);
}
…
```

【SuperBehaviorTypeCallingWorld クラス】

```
…
public void initializeAgents() {
   //Questioner エージェントの生成
   Agent questioner = this.createAgent(
      SuperBehaviorTypeCallingModel.AGENTTYPE_Questioner);
   //Questioner エージェントへの QuestionBehavior の追加
   questioner.addBehavior(SuperBehaviorTypeCallingModel.BEHAVIORTYPE_Question);

   //Receiver エージェントの生成
```

```
   Agent receiver = this.createAgent(
      SuperBehaviorTypeCallingModel.AGENTTYPE_Receiver);
   //Receiver エージェントへの DecideCarefullyBehavior の追加
   receiver.addBehavior(
      SuperBehaviorTypeCallingModel.BEHAVIORTYPE_DecideCarefully);

   //Questioner エージェントの Receiver エージェントへの Relation の追加
   questioner.addRelation(
      SuperBehaviorTypeCallingModel.RELATIONTYPE_ToReceiver, receiver);
}
...
```

【QuestionBehavior クラス】

```
...
protected void questionAcrion() {

   //Receiver エージェントに、QuestionInformation を送る。
   this.sendInformation(
      SuperBehaviorTypeCallingModel.RELATIONTYPE_ToReceiver,
      SuperBehaviorTypeCallingModel.BEHAVIORTYPE_Decide,
      new QuestionInformation());
}
...
```

【DecideCarefullyBehavior クラス】

```
...
protected void decideCarefullyAction() {
   //慎重に意思決定する
}

protected boolean isQuestion(Event e) {
   //QuestionInformation を受け取った時に true を返す
   return this.getWorld().getInformationType(this.getReceivedInformation())
      == SuperBehaviorTypeCallingModel.INFORMATIONTYPE_Question;
}
...
```

■ バリエーション

このサンプルでは、DecideCarefullyBehavior だけを用意し、DecideBehavior は用意しなかった。もし複数の Decide〜Behavior があり、それらのプログラムに共通部分があるならば、これらの Behavior の親クラスとして、DecideBehavior を定義してもよい。この場合には、BehaviorType の親子関係とは別に、プログラム的な汎化関係を用いるということである。

■ 関連するパターン

Information Sending: 他のエージェントに情報を送る。
Internal Information Sending: 自分の持っている他の行動に情報を送る。

Model Patterns

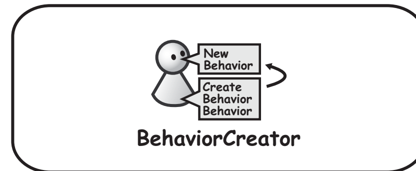
Behavior Creation

■ 目的

自分自身に新しい行動を生成・追加する。

■ 動機

エージェントの役割が動的に変化するモデルや、状況に応じた振舞いをするモデルでは、シミュレーション実行中に、そのエージェントがいままで持っていなかった行動を生成し追加する必要がある。

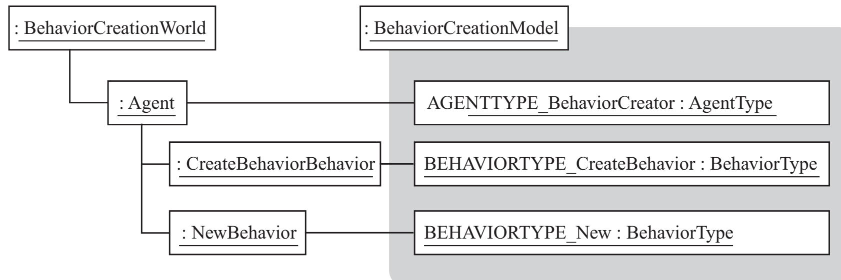


■ 基本動作

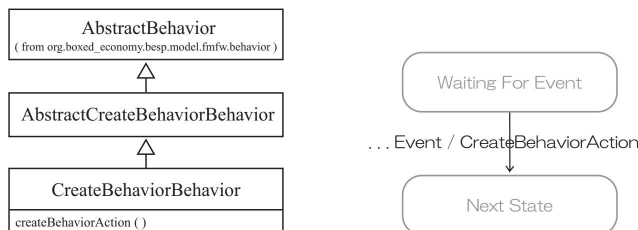
BehaviorCreator エージェントは CreateBehaviorBehavior をもっている。この CreateBehaviorBehavior は、NewBehavior を生成する。その結果、BehaviorCreator エージェントは、新たに NewBehavior を持つことになる。

■ 設計

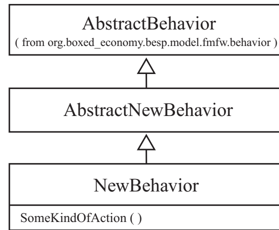
【全体像】



【CreateBehaviorBehavior】



【NewBehavior】



※この行動の内容は、文脈に合わせて作成する。

■ サンプルコード

【BehaviorCreationWorld クラス】

```
…
public void initializeAgents() {
   //BehaviorCreator エージェントの生成
   Agent behaviorCreator =
      createAgent(BehaviorCreationModel.AGENTTYPE_BehaviorCreator);

   //そのエージェントへの CreateBehaviorBehavior の追加
   behaviorCreator.addBehavior(BehaviorCreationModel.BEHAVIORTYPE_CreateBehavior);
}
…
```

【CreateBehaviorBehavior クラス】

```
…
protected void createBehaviorAction() {
   //NewBehavior の追加
   this.getAgent().addBehavior(BehaviorCreationModel.BEHAVIORTYPE_New);
}
…
```

■ 関連するパターン

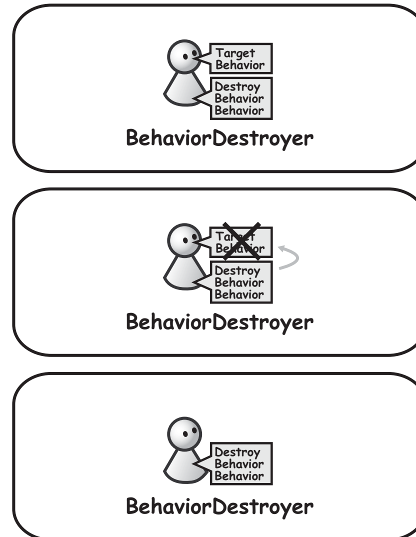
Behavior Switching: 新しいエージェントが持っている行動を削除し、新しい行動を追加する。
Temporary Behavior Creation: 生成・追加する行動が、一時的に処理を行って自動消滅する場合。

Behavior Destruction

■ 目的

エージェントが持っている行動を削除する。

■ 動機



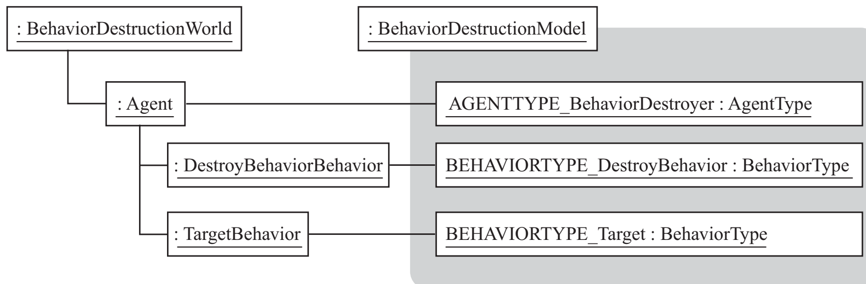
エージェントの役割が動的に変化するモデルや、状況に応じた振舞いをするモデルでは、シミュレーション実行中に、そのエージェントが持っている行動を削除したいことがある。現在もっている行動を、強制的に終了したり、同種の別の行動に置き換えたりするためである。

■ 基本動作

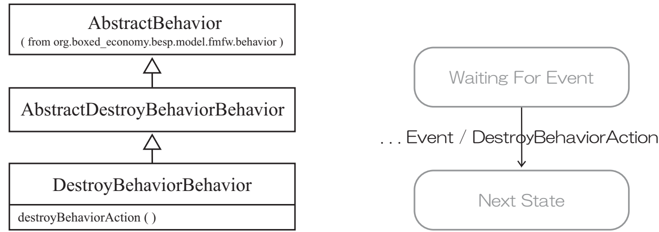
BehaviorDestroyer エージェントは TargetBehavior と DestroyBehaviorBehavior を持っている。DestroyBehaviorBehavior によって、TargetBehavior を削除する (TargetBehavior の状態にかかわらず、外部から強制的に削除する)。

■ 設計

【全体像】



【DestroyBehaviorBehavior】



■ サンプルコード

【BehaviorDestructionWorld クラス】

```
…
public void initializeAgents() {
   //BehaviorDestroyer エージェントの生成
   Agent behaviorDestroyer =
      createAgent(BehaviorDestructionModel.AGENTTYPE_BehaviorDestroyer);

   //そのエージェントへの DestroyBehaviorBehavior の追加
   behaviorDestroyer.addBehavior(
      BehaviorDestructionModel.BEHAVIORTYPE_DestroyBehavior);

   //そのエージェントへの削除対象動の追加
   behaviorDestroyer.addBehavior(
      BehaviorDestructionModel.BEHAVIORTYPE_Target);
}
…
```

【DestroyBehaviorBehavior クラス】

```
…
protected void destroyBehaviorAction() {
   //Behavior の削除
   this.getAgent().removeBehavior(this.getAgent()
      .getBehavior(BehaviorDestructionModel.BEHAVIORTYPE_Target));
}
…
```

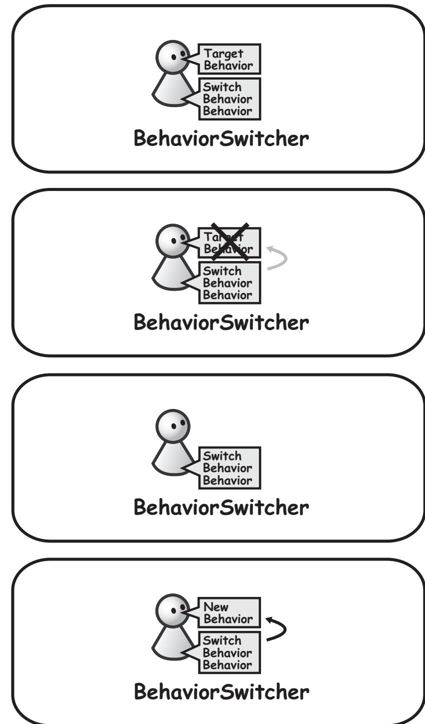
■ 関連するパターン

Behavior Switching: ある行動を他の行動に切り替える場合 (現在持っている行動を削除し、新しい行動を追加する)。

Behavior Switching

■ 目的

エージェントが現在持っている行動を、新しい他の行動に切り替える。



■ 動機

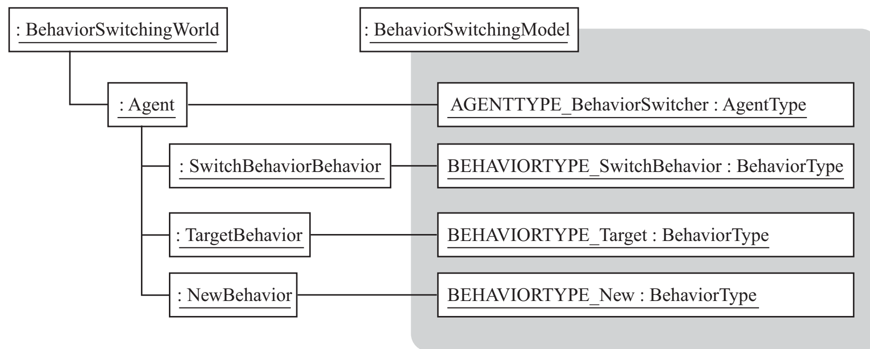
エージェントの役割が動的に変化するモデルや、状況に応じた振舞いをするモデルでは、シミュレーション実行中に、そのエージェントが持っている行動を切り替える必要がある。特に典型的な例としては、戦略 (行動) の切り替えがある。

■ 基本動作

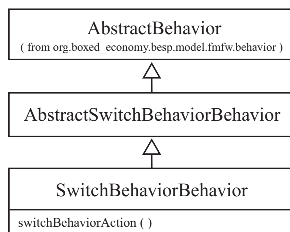
BehaviorSwitcher エージェントは TargetBehavior と SwitchBehaviorBehavior を持っている。SwitchBehaviorBehavior によって、TargetBehavior を削除し、SwitchBehaviorBehavior によって、NewBehavior を追加する。

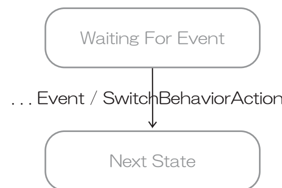
■ 設計

【全体像】



【SwitchBehaviorBehavior】





■ サンプルコード

【BehaviorSwitchingWorld クラス】

```
…
public void initializeAgents() {
   //エージェントの生成
   Agent behaviorSwitcher =createAgent(
      BehaviorSwitchingModel.AGENTTYPE_BehaviorSwitcher);

   //そのエージェントへの SwitchBehavior 行動と、切り替え前行動の追加
   behaviorSwitcher.addBehavior(
      BehaviorSwitchingModel.BEHAVIORTYPE_SwitchBehavior);
   behaviorSwitcher.addBehavior(BehaviorSwitchingModel.BEHAVIORTYPE_Target);
}
…
```

【SwitchBehaviorBehavior クラス】

```
…
protected void switchBehaviorAction() {
   //切り替え前の行動の削除
   this.getAgent().removeBehavior(
      this.getAgent().getBehavior(BehaviorSwitchingModel.BEHAVIORTYPE_Target));
   //切り替え後の行動の追加
   this.getAgent().addBehavior(BehaviorSwitchingModel.BEHAVIORTYPE_New);
}
…
```

■ バリエーション

このサンプルでは、ソースコード中に明示的に切り替え後の BehaviorType を指定しているが、BehaviorType を情報として入手し、それに応じて切り替え後の行動を決めるということもできる。

なお、戦略行動のように、同種の行動であるが内容が異なるという行動を切り替えることがある (その場合には、Super BehaviorType Calling パターンを使って行動のアクティベーションが行われていると思われる)。このような場合には、切り替え前の行動の削除の際に、親 BehaviorType を指定して削除することができる。これにより、切り替え前の行動が具体的に何であるかを意識することなく、削除することができる。

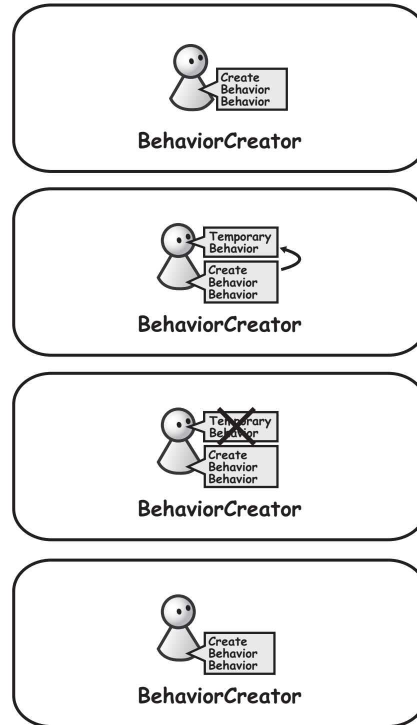
■ 関連するパターン

Behavior Destruction: 行動を削除する (切り替え前の行動を削除する際に用いる)。
Behavior Creation: 行動を生成する (切り替え後の行動を生成する際に用いる)。
Temporary Behavior Creation: 一時的に行われる行動を生成する (切り替え後の行動が一時的な行動である場合は、これを用いる)。
Super BehaviorType Calling: 具体的な BehaviorType ではなく、親 BehaviorType で指定する (行動を切り替えても、動作するために必要となる)。

Temporary Behavior Creation

■ 目的

一時的に行う行動を追加する。



■ 動機

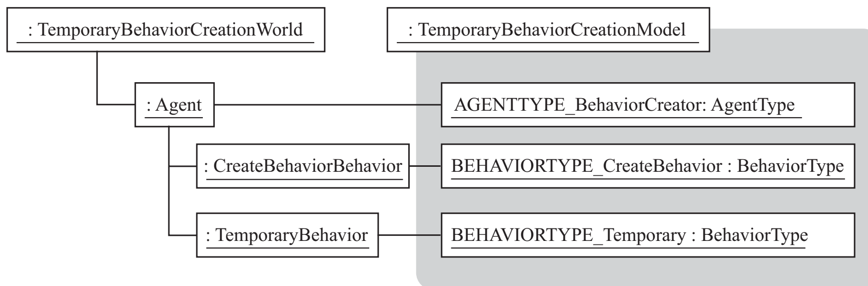
付加的な処理や例外的な処理などのように、エージェントが一時的に行う行動を追加したいことがある。そのような行動は、通常の行動とは別に動き、処理が終了したときには自働的に消滅させたい。

■ 基本動作

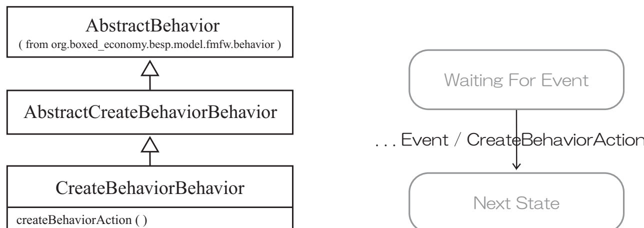
BehaviorCreator エージェントは CreateBehaviorBehavior をもっている。CreateBehaviorBehavior は TemporaryBehavior を生成・追加する。TemporaryBehavior は、処理を完了すると、終了状態に遷移して自ら消滅する。

■ 設計

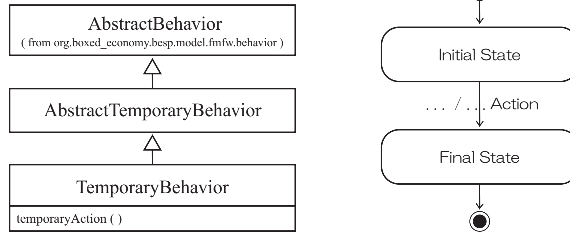
【全体像】



【CreateBehaviorBehavior】



【TemporaryBehavior】



■ サンプルコード

【TemporaryBehaviorCreationWorld クラス】

```
…
public void initializeAgents() {
   //BehaviorCreator エージェントの生成
   Agent behaviorCreator = createAgent(
      TemporaryBehaviorCreationModel.AGENTTYPE_BehaviorCreator);

   //そのエージェントへの CreateBehavior 行動の追加
   behaviorCreator.addBehavior(
      TemporaryBehaviorCreationModel.BEHAVIORTYPE_CreateBehavior);
}
…
```

【CreateBehaviorBehavior クラス】

```
…
protected void createBehaviorAction() {
   //TemporaryBehavior の追加
   this.getAgent().addBehavior(
      TemporaryBehaviorCreationModel.BEHAVIORTYPE_Temporary);
}
…
```

■ バリエーション

TemporaryBehavior が「モデル上の時間」を要する場合には、TimeEvent を必要な数だけ受けた後に終了するという拡張を行う。

■ 関連するパターン

Behavior Creation: 新しい行動を生成・追加する。
Behavior Request Attachment: 何の行動を追加するのかを、他のエージェントから指定される場合。
Time-Consuming Behavior: 遂行するのに一定の時間がかかる Behavior を表現する。

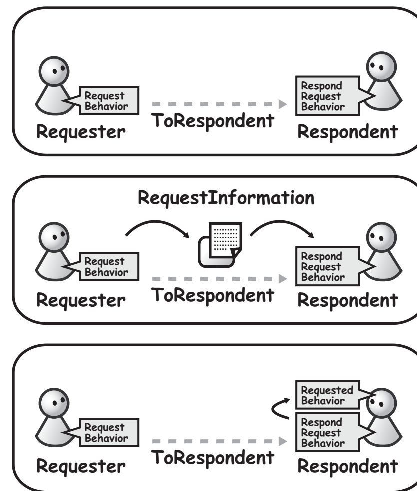
Requested Behavior Attachment

■ 目的

こちらが指定した BehaviorType の行動を、他のエージェントに生成・追加してもらう。

■ 動機

相手にコミュニケーションの手順を合わせてほしい場合がある。例えば、独特の手順で買い物をする店では、来店した顧客にその手順を知らせて、それに合せて行動してもらう必要がある。また、新しい役割を委譲する場合などにも、こちらが指定した行動を、相手にもってもらう必要がある。そのほか、環境エージェントが、対象となるエージェントの行動をアフォードさせたいことがある。

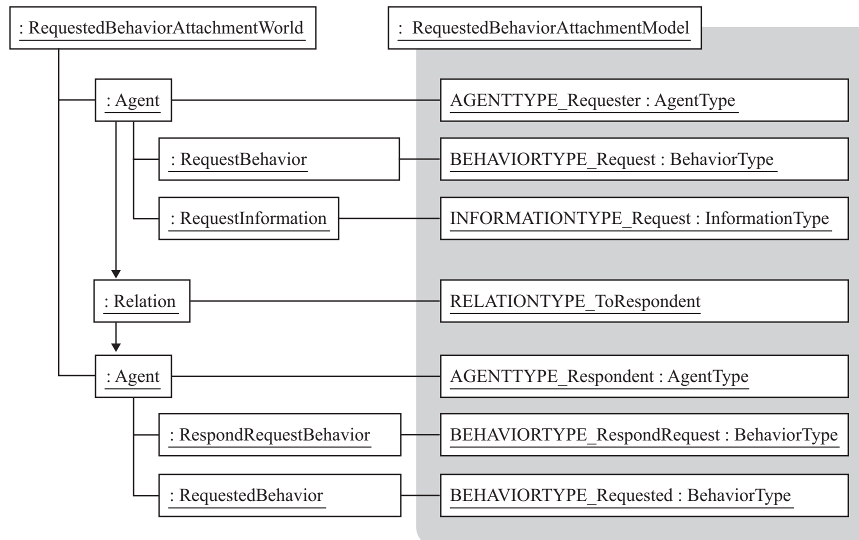


■ 基本動作

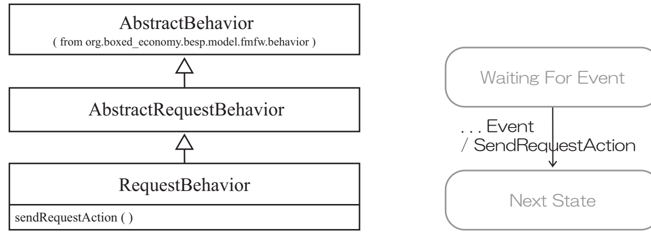
Requester エージェントと Respondent エージェントが登場する。Requester エージェントは、RequestBehavior によって、相手にもってほしい BehaviorType を相手に送信する (Type は Information の一種なので、そのまま送信することができる)。Respondent エージェントは、RespondRequestBehavior をもっており、RequestInformation を受けて RequestedBehavior を生成する。

■ 設計

【全体像】



【RequestBehavior】



【RespondRequestBehavior】

■ サンプルコード

【RequestedBehaviorAttachmentWorld クラス】

```
…
public void initializeAgents() {
   //Requester エージェントを生成する
   Agent requester = createAgent(
      RequestedBehaviorAttachmentModel.AGENTTYPE_Requester);

   //Requester エージェントに、RequestBehavior を追加する
   requester.addBehavior(
      RequestedBehaviorAttachmentModel.BEHAVIORTYPE_Request);

   //Respondent エージェントを生成する
   Agent respondent = createAgent(
      RequestedBehaviorAttachmentModel.AGENTTYPE_Respondant);

   //Respondent エージェントに、RespondRequestBehavior を追加する
   respondent.addBehavior(
      RequestedBehaviorAttachmentModel.BEHAVIORTYPE_RespondRequest);

   //Requester エージェントに、Respondent エージェントへの Relation を追加する
   requester.addRelation(
      RequestedBehaviorAttachmentModel.RELATIONTYPE_ToRespondent, respondent);
}
…
```

【RequestBehavior クラス】

```
…
protected void sendRequestAction() {
   //Respondent エージェントへ追加したい BehaviorType を送る
   sendInformation(
      RequestedBehaviorAttachmentModel.RELATIONTYPE_ToRespondent,
      RequestedBehaviorAttachmentModel.BEHAVIORTYPE_RespondRequest,
      RequestedBehaviorAttachmentModel.BEHAVIORTYPE_Requested);
}
…
```

【RespondRequestBehavior クラス】

```
…
protected void addRequestedBehaviorAction() {
   //送られてきた BehaviorType を取得する
   BehaviorType sentBehaviorType = (BehaviorType) this.getReceivedInformation();

   //送られてきた BehaviorType を自分自身に追加する
   this.getAgent().addBehavior(sentBehaviorType);
}

protected boolean isBehaviorType(Event e) {
   //送られてきた情報が BehaviorType であれば true を返す
   return getReceivedInformation() instanceof BehaviorType;
}

…
```

■ バリエーション

このサンプルでは、持ってほしいといわれた BehaviorType の行動を無条件に追加するが、受け手側 (Respondent エージェント) の判断に基づいて、追加するかどうかを決定することもできる。

また、送られてくる BehaviorType は、具体的な BehaviorType である必要はなく、その子 BehaviorType の行動を選択して追加させることもできる。例えば、BEHAVIORTYPE_Strategy という戦略行動をもつように言われた場合、その子 BehaviorType である BEHAVIORTYPE_StrategyA や BEHAVIORTYPE_StrategyB の行動を追加することができる (その場合、Super BehaviorType Calling パターンを使って、行動のアクティベーションが行われるだろう)。

■ 関連するパターン

Behavior Creation: 行動を生成・追加する。
Information Sending: 情報を送る。
Forced Behavior Attachment: 行動を付加するように依頼するのではなく、外部から強制的に行動を付加する (この場合には、相手の意思を介在させる余地はないが、受け手側が RespondRequestBehavior を持っている必要はない)。

Model Patterns

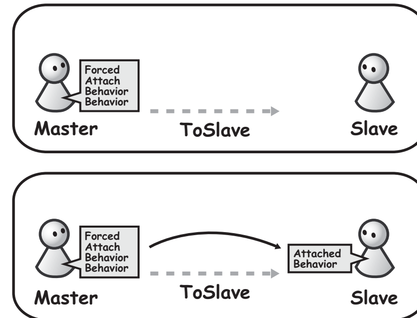
Forced Behavior Attachment

■ 目的

こちらが指定した BehaviorType の行動を、他のエージェントに強制的に追加する。

■ 動機

相手にコミュニケーションの手順を合わせてほしい場合がある。例えば、独特の手順で買い物をする店では、来店した顧客にその手順を知らせて、それに合せて行動してもらう必要がある。また、新しい役割を委譲する場合などにも、こちらが指定した行動を、相手にもってもらう必要がある。そのほか、環境エージェントが、対象となるエージェントの行動をアフォード させたいことがある。

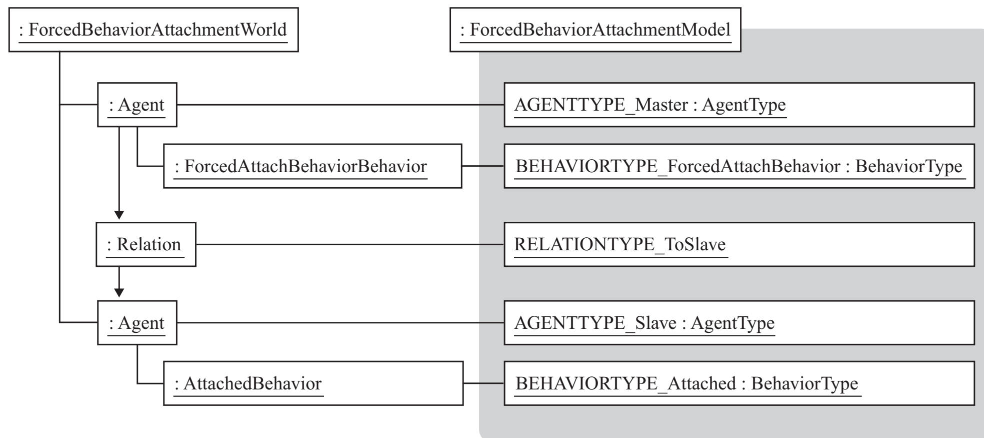


■ 基本動作

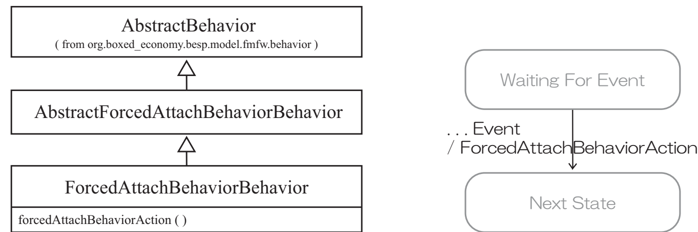
Master エージェントと Slave エージェントが登場する。Master エージェントは ForcedAttatchBehaviorBehavior をもっており、Slave エージェントに強制的に AttachedBehavior を付加する。

■ 設計

【全体像】



【AttachBehaviorBehavior】



■ サンプルコード

【ForcedBehaviorAttachmentWorld クラス】

```
…
public void initializeAgents() {
   //Master エージェントを生成する
   Agent master = createAgent(ForcedBehaviorAttachmentModel.AGENTTYPE_Master);

   //Master エージェントに、ForcedAttachBehavior を追加する
   master.addBehavior(ForcedBehaviorAttachmentModel.BEHAVIORTYPE_ForcedAttachBehavior);

   //Slave エージェントを生成する
   Agent slave = createAgent(ForcedBehaviorAttachmentModel.AGENTTYPE_Slave);

   //Master エージェントに、Slave エージェントへの Relation を追加する
   master.addRelation(ForcedBehaviorAttachmentModel.RELATIONTYPE_ToSlave, slave);
}
…
```

【ForcedAttachBehaviorBehavior クラス】

```
…
protected void forcedAttachBehaviorAction() {
   //Slave エージェントへの Relation を取得する
   Relation toSlave = this.getAgent().getRelation(
      ForcedBehaviorAttachmentModel.RELATIONTYPE_ToSlave);

   //Relation から相手を取得し、AttachedBehavior を追加する
   toSlave.getTarget().addBehavior(
      ForcedBehaviorAttachmentModel.BEHAVIORTYPE_Attached);
}
…
```

■ 関連するパターン

Behavior Creation: 行動を生成・追加する。
Requested Behavior Attachment: 外部から強制的に行動を付加するのではなく、行動を付加するように依頼する (この場合には、相手の意思を介在させる余地がある)。

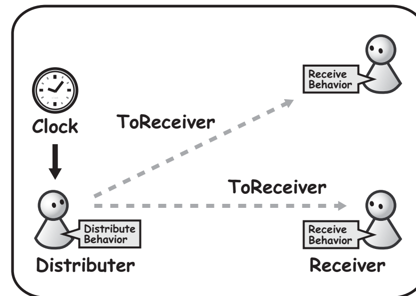
TimeEvent Distributer Agent

■ 目的

TimeEvent を一部のエージェントにだけ送りたい。

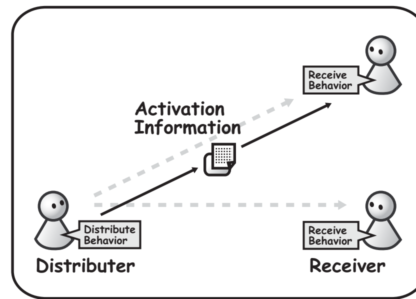
■ 動機

モデルのすべてのエージェントではなく、一部のエージェントだけを活性化させたいときがある。例えば、1 ステップに 1 人だけが動作するような場合である。配信先の決定は、ランダムに選ぶ場合もあれば、リストに従って順番に選んでいく場合もあるだろう。



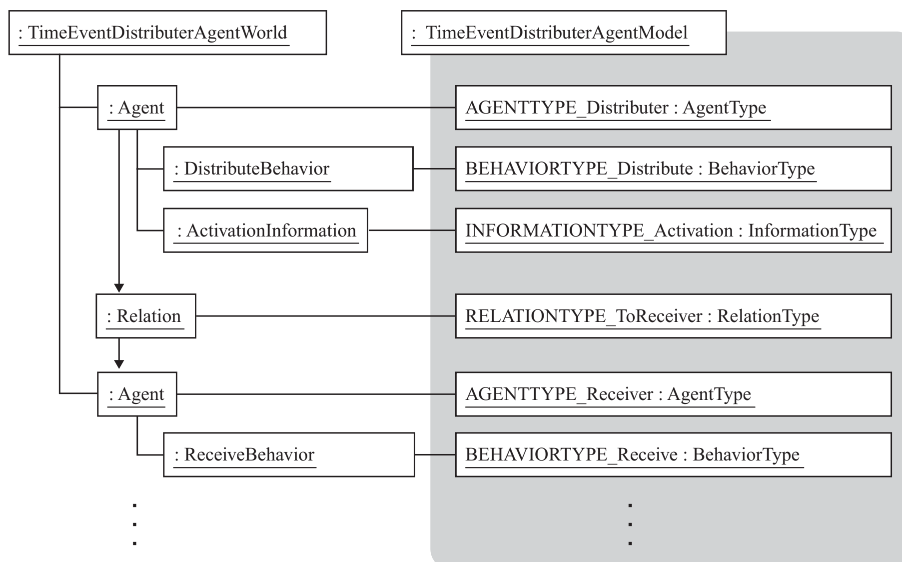
■ 基本動作

Distributer エージェントを用意する。Distributer エージェントは、TimeEvent を代表して受け取り、ActivationInformation を作成して、対象となるエージェントに配信する。活性化されるエージェントは、TimeEvent ではなく ChannelEvent で状態遷移するように記述する。

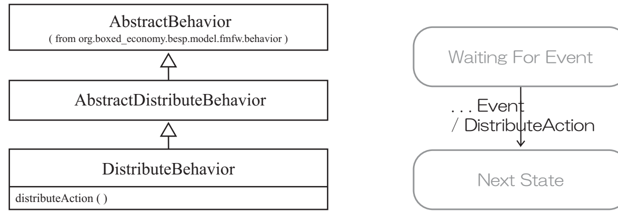


■ 設計

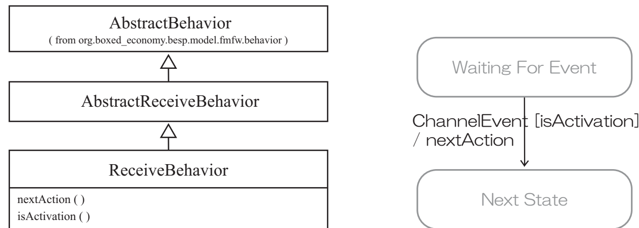
【全体像】



【DistributeBehavior】



【ReceiveBehavior】



■ サンプルコード

【TimeEventDistributerAgentWorld クラス】

```
…
public void initializeAgents() {
   //Distributer エージェントの生成
   Agent distributer = this.createAgent(
      TimeEventDistributerModel.AGENTTYPE_Distributer);
   //Distributer エージェントへの DistributeBehavior の追加
   distributer.addBehavior(TimeEventDistributerModel.BEHAVIORTYPE_Distribute);

   //Receiver エージェントの生成（複数人）
   for (int i = 0; i < 10; i++) {
      //Receiver エージェントの生成
      Agent receiver = this.createAgent(
         TimeEventDistributerModel.AGENTTYPE_Receiver);
      //Receiver エージェントへの ReceiveBehavior の追加
      receiver.addBehavior(TimeEventDistributerModel.BEHAVIORTYPE_Receive);
      //Distributer エージェントの Receiver エージェントへの Relation の追加
      distributer.addRelation(
         TimeEventDistributerModel.RELATIONTYPE_ToReceiver, receiver);
   }
}
…
```

【DistributeBehavior クラス】(ランダムに 1 人ずつに送る場合)

```
…
protected void distributeAction() {
   //Receiver エージェントへの Relation のリストを取得する
   List toReceivers = (List) this.getAgent().getRelations(
      TimeEventDistributerModel.RELATIONTYPE_ToReceiver);
   //乱数ジェネレータを用いて、リストからランダムにひとつの Relation を選び出す
   int targetIndex = this.getWorld().getRandomNumberGenerator().generate(
```

```
        toReceivers.size());
    Relation toReceiver = (Relation) toReceivers.get(targetIndex);

    //ActivationInformation を送る
    this.sendInformation(toReceiver, TimeEventDistributerModel.BEHAVIORTYPE_Receive,
        new ActivationInformation());
}
...
```

【DistributeBehavior クラス】(順番にひとりずつに送る場合)

```
...
private int targetIndex = 0;

protected void distributeAction() {
    //Receiver への Relation のリストを取得する
    List toReceivers = (List) this.getAgent().getRelations(
        TimeEventDistributerModel.RELATIONTYPE_ToReceiver);

    //リストから順番にひとつの Relation を選び出す
    if (toReceivers.size() <= targetIndex) {
        throw new ModelException("OutOfIndex : " + this);
    }
    Relation toReceiver = (Relation) toReceivers.get(targetIndex);

    //targetIndex を進める（終点の場合、始点に戻る）
    if (toReceivers.size() - 1 == targetIndex)
        targetIndex = 0;
    else
        targetIndex++;

    //ActivationInformation を送る
    this.sendInformation(toReceiver, TimeEventDistributerModel.BEHAVIORTYPE_Receive,
        new ActivationInformation());
}
...
```

【ReceiveBehavior クラス】

```
...
protected boolean isActivation(Event e) {
    //ActivationInformation を受け取った時に true を返す
    return this.getWorld().getInformationType(this.getReceivedInformation())
    == TimeEventDistributerModel.INFORMATIONTYPE_Activation;
}
...
```

■ 関連するパターン

TimeEvent Filtering: このパターンとの違いはわずかだが、決定的な差異を生み出す可能性があることに注意が必要である。例えば、全体の 20%の数のエージェントが活性化される (TimeEvent Distributer パターン) のと、各エージェントが 20%の確率で活性化する (TimeEvent Filtering パターン) というのとでは、結果が異なる場合がある。前者は、必ず全体の 20%のエージェントが活性化されるのに対し、後者は、平均すると全体の 20%のエージェントが活性化する。つまり、後者の場合は、各エージェント間の活性化確率は独立であるため、全体としてみると、活性化する人数が多いときや少ないときがある。

Model Patterns

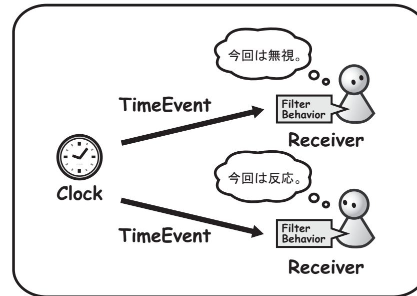
TimeEvent Filtering

■ 目的

TimeEvent を確率的に受信するようにしたい。

■ 動機

特定のエージェントを、毎回ではなく、ある確率で活性化させたいときがある。例えば、数回に 1 回だけ活性化するような行動をモデル化する場合である。

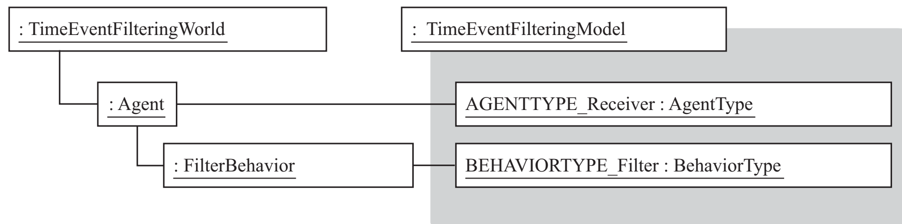


■ 基本動作

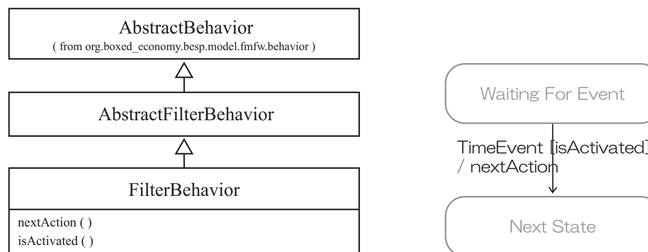
Receiver エージェントは、FilterBehavior をもっており、ある確率で TimeEvent を受け取る。

■ 設計

【全体像】



【FilterBehavior】



■ サンプルコード

【TimeEventFilteringWorld クラス】

```
…
public void initializeAgents() {
   //Receiver エージェントを生成する（複数人）
   for(int i = 0; i<10; i++){
      //Receiver エージェントの生成
      Agent receiver = createAgent(TimeEventFilteringModel.AGENTTYPE_Receiver);
      //Receiver エージェントへの FilterBehavior の追加
      receiver.addBehavior(TimeEventFilteringModel.BEHAVIORTYPE_Filter);
   }
}
…
```

【FilterBehavior クラス】

```
…
protected boolean isActivated(Event e) {
   //20 %の確率で true を返す
   return this.getWorld().getRandomNumberGenerator().generate() < 0.2;
}
…
```

■ 関連するパターン

TimeEvent Distributer Agent: このパターンとの違いはわずかだが、決定的な差異を生み出す可能性があることに注意が必要である。例えば、全体の 20%の数のエージェントが活性化される (TimeEvent Distributer パターン) のと、各エージェントが 20%の確率で活性化する (TimeEvent Filtering パターン) というのとでは、結果が異なる場合がある。前者は、必ず全体の 20%のエージェントが活性化されるのに対し、後者は、平均すると全体の 20%のエージェントが活性化する。つまり、後者の場合は、各エージェント間の活性化確率は独立であるため、全体としてみると、活性化する人数が多いときや少ないときがある。

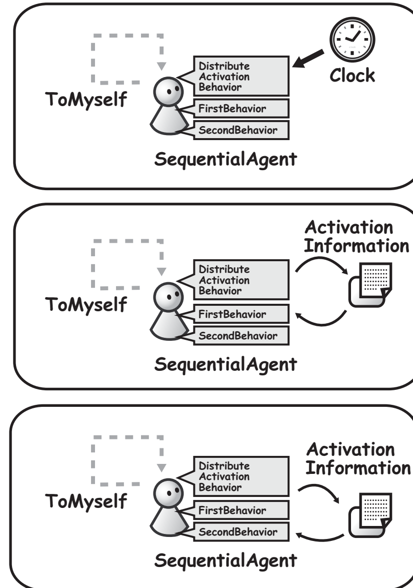
TimeEvent Distributer Behavior

■ 目的

エージェント内の Behavior に送る TimeEvent の送信順序 (プライオリティ) を制御したい。

■ 動機

エージェントが行う複数の行動を決められた順番で活性化したい場合がある。しかし、TimeEvent の送信順序 (プライオリティ) はエージェントごとに設定できるが、Behavior ごとには設定できない。そこで、Behavior の活性化を制御するための工夫が必要である。

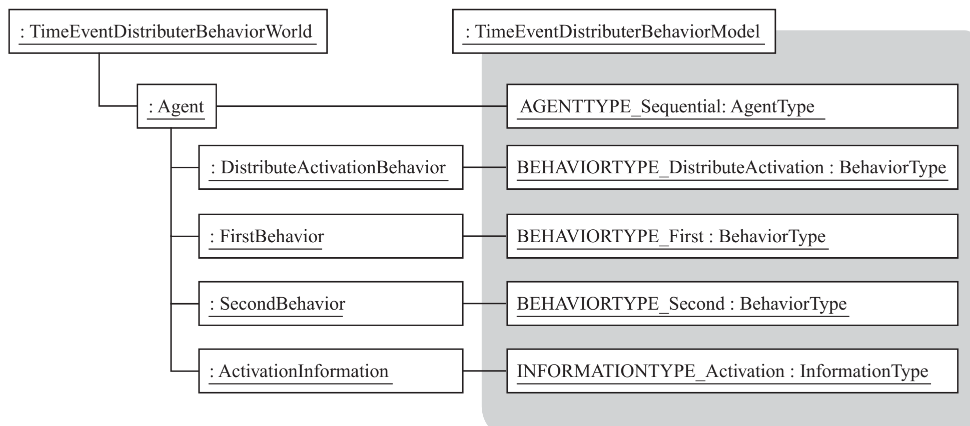


■ 基本動作

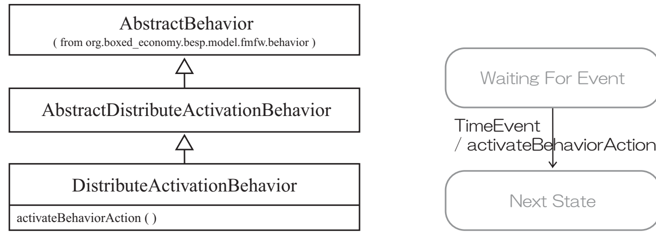
SequentialAgent は、DistributeSignalBehavior と、最初に活性化したい FirstBehavior、および 2 番目に活性化したい SecondBehavior をもっている。DistributeActivationBehavior が、TimeEvent を受け取ると、あらかじめ決められた順番で、自分自身の Behavior に ActivationInformation を配信していく。

■ 設計

【全体像】



【DistributeBehavior】



■ サンプルコード

【TimeEventDistributerBehaviorWorld クラス】

```
…
public void initializeAgents() {
   //SequentialAgent の生成
   Agent sequentialAgent = this.createAgent(
      TimeEventDistributerBehaviorModel.AGENTTYPE_Sequential);

   //SequentialAgent への DistributeSignalBehavior、FirstBehavior、
   //SecondBehavior の追加
   sequentialAgent.addBehavior(
      TimeEventDistributerBehaviorModel.BEHAVIORTYPE_DistributeActivation);
   sequentialAgent.addBehavior(TimeEventDistributerBehaviorModel.BEHAVIORTYPE_First);
   sequentialAgent.addBehavior(TimeEventDistributerBehaviorModel.BEHAVIORTYPE_Second);

   //SequentialAgent の自身への Relation の追加
   sequentialAgent.addRelation(TimeEventDistributerBehaviorModel.RELATIONTYPE_ToMyself,
      sequentialAgent);
}
…
```

【DistributeActivationBehavior クラス】

```
…
protected void activateBehaviorAction() {

   //自分の FirstBehavior に ActivationInformation を送る
   this.sendInformation(
      TimeEventDistributerBehaviorModel.RELATIONTYPE_ToMyself,
      TimeEventDistributerBehaviorModel.BEHAVIORTYPE_First,
      new ActivationInformation());

   //自分の SecondBehavior に SignalInformation を送る
   this.sendInformation(
      TimeEventDistributerBehaviorModel.RELATIONTYPE_ToMyself,
      TimeEventDistributerBehaviorModel.BEHAVIORTYPE_Second,
      new ActivationInformation());
}
…
```

■ 関連するパターン

Internal Information Sending: 自分の他の行動に情報を送る。

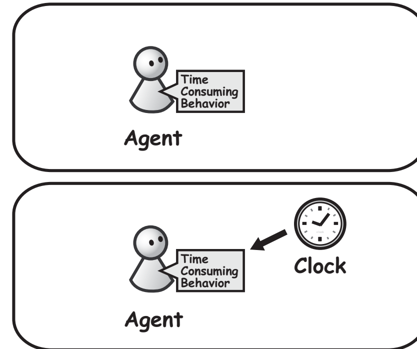
Time-Consuming Behavior

■ 目的

遂行するのに一定の時間がかかる Behavior を表現する。

■ 動機

ある行動を開始して、しばらくたってから次の動作を行わせたい場合がある。例えば、数時間ステップかかる思考や行動をモデル化する場合などである。

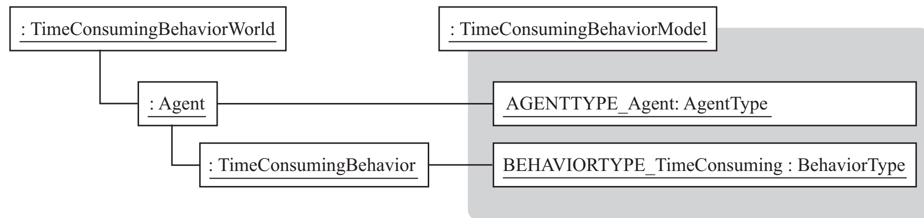


■ 基本動作

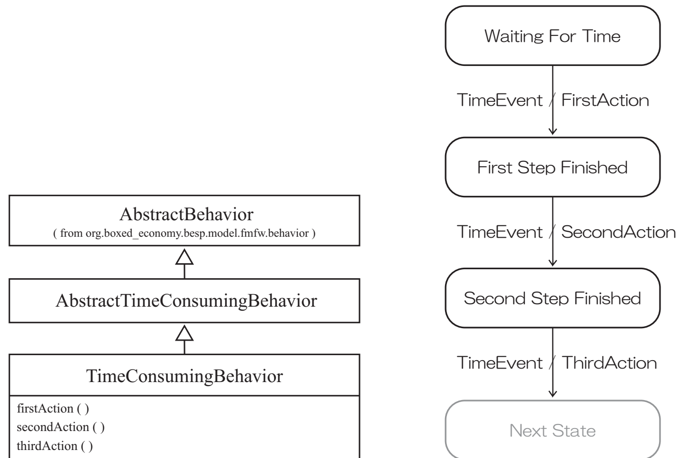
Agent エージェントは、TimeConsumingBehavior をもっている。TimeConsumingBehavior では、その一連の動作を終了するまでに、何度か TimeEvent を受ける。ここでは、Event を受けるごとに、FirstAction、SecondAction、ThirdAction を行う。

■ 設計

【全体像】



【TimeConsumingBehavior】



■ サンプルコード

【TimeConsumingBehaviorWorld クラス】

```
…
public void initializeAgents() {
   //Agent エージェントの生成
   Agent agent = this.createAgent(TimeConsumingModel.AGENTTYPE_Agent);

   //Agent エージェントに、TimeConsumingBehavior を追加する
   agent.addBehavior(TimeConsumingModel.BEHAVIORTYPE_TimeConsuming);
}
…
```

【TimeConsumingBehavior クラス】

```
…
protected void firstAction() {
   //ここで、作業の第 1 ステップを行う
}

protected void secondAction() {
   //ここで、作業の第 2 ステップを行う
}

protected void thirdAction() {
   //ここで、作業の第 3 ステップを行う
}
…
```

■ 関連するパターン

Temporary Behavior Creation: 一時的に実行する行動を追加する。