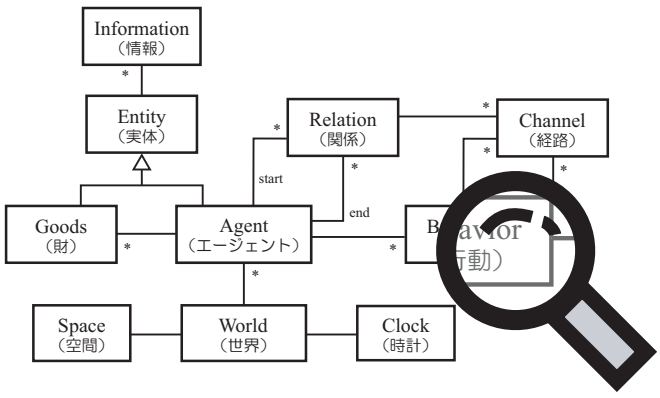
Designers' Guide to Social Simulations, No.3

モデル作成リファレンスガイド



PlatBox Project

目次

第 I 部

PlatBox 基礎モデル

第 1 章

PlatBox 基礎モデルとは

1.1 エージェントベースモデルのための枠組み

エージェントベースによるアプローチでは、社会を多数のエージェント (自律的主体) のミクロ的な相互作用からなるシステムとしてモデル化します。モデル化というのは、その対象を体系的なまとまりとして写し取るということを意味していますが、通常その写し取り方には個人差があります。このことは、モデルの粒度や要素の捉え方がさまざまに異なってしまい、モデルの融合やモデル部品の再利用ができなくなってしまうという問題を引き起こします。そのため、どのようにモデル化すればよいかのガイドラインを示すためにも、モデルの融合やモデル部品の再利用を行うためにも、なんらかの指針が求められます。このような問題意識から、私たちは「PlatBox 基礎モデル」(以下、「基礎モデル」) を提案しています。

基礎モデルは、現実の社会をオブジェクト指向分析によって抽象化して作成されたもので、エージェントベースによる社会モデルの基本デザインを提供するものです。基礎モデルは、社会の認識・分析の指針となるだけでなく、そのモデル要素を基本語句として用いてモデルを記述できるようになります。また、基礎モデルに基づいて作成されたモデルは PlatBox Simulator 上でシミュレートしてその振る舞いを観察することができます。さらに、基礎モデルに基づいて作成されたモデルは、その部品の共用・再利用も可能となるため、複雑なモデルの共同開発や累積的発展を支援します。さらに、基礎モデルのモデル要素の名称を、研究者同士が共有可能な語彙として用いることで、シミュレーションにおける実装形態にとらわれないコミュニケーションが可能になります。

1.2 オブジェクト指向と基礎モデル

基礎モデルは現実世界を体系化して整理する方法としてオブジェクト指向を採用しています。オブジェクト指向とは、物理的あるいは概念的なもの＝「オブジェクト」がそれぞれ役割を分担しながら相互作用することで世界が動いていると捉える考え方です。オブジェクトは「属性」と「振る舞い」をひとまとめにカプセル化したものであり、個別の具

体的なものを表わしています。

オブジェクト指向では、概念的なまとまりを「クラス」として表現することができます。クラスという型によってオブジェクトを分類し、効率的に記述できるのです。オブジェクトをクラスに分類するという仕組みは、実は、人間の認知プロセスにおける「概念化」と同じメカニズムです。例えば、私たちは、ベンチや折りたたみの椅子、車椅子、背もたれのない横長の椅子などを見て、それぞれ形が異なっているにもかかわらず、これらをひとまとめに『椅子』として認識します。認知における「具体的なもの」と、オブジェクト指向におけるオブジェクトは、それぞれ概念とクラスの「インスタンス」(実例) と呼ばれます (図 1.1)。

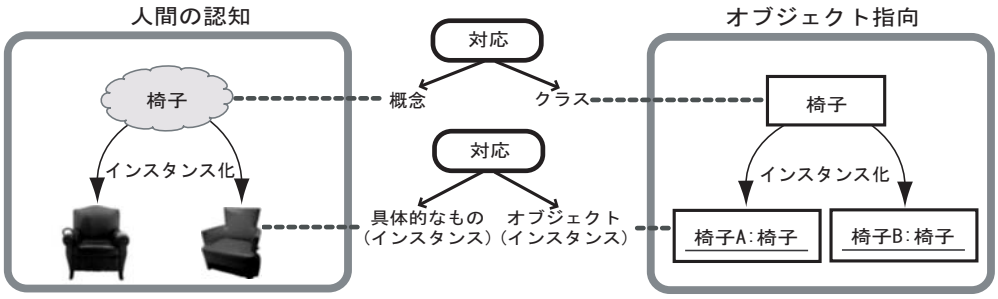


図 1.1: 人間の認知の仕組みとオブジェクト指向

このように、オブジェクト指向によるモデル化の過程は、人間の認知の仕組みに近く、より直感的なモデル化ができるのです。個別のものをクラス (概念) で分類するということは、世界の複雑さに対処するためのひとつの方法だといえるでしょう。また、シミュレーションにオブジェクト指向の考え方を導入することで、作成したモデルを作成者以外の人と共有することが容易になります。というのは、人間の認知の仕組みに近いため、他の人が作ったモデルを理解するのも容易になるためです。

基礎モデルは、社会を抽象化したものですから、個別具体的なオブジェクトではなく、クラスを用いて記述されています。

1.3　基礎モデルの二つの捉え方

基礎モデルを用いてモデルを記述すれば、モデルの要素が実際にどのようなメカニズムで動作するのかという詳細は意識しなくて済むことになります。例えば、ユーザは「Goods」や「Relation」というような基礎モデルの要素を語彙としてモデルを記述するだけで、シミュレーションを行うことが可能になります。しかし、シミュレーションをコンピュータ・ソフトウェアとして見た場合、これだけでは充分ではありません。基礎モデルに基づくモデルがプログラムとして実装されている必要があります。このため、基礎モデルは、抽象度の違いによって「概念モデル・レベル」と、ソフトウェアフレームワークとしての「シミュレーションモデル・レベル」という 2 種類のレベルで捉えることができ

ます。

概念モデル・レベル

概念モデル・レベルは、対象についての概念モデルを作成する際に、共通して登場する要素と構造を定義したものです。モデル作成者は、モデル化しようとしている対象が「どのようなものであるか」（What）を洗い出し、記述する際に、この概念モデル・レベルの語彙を用いることができます。加えて、概念モデル・レベルがシミュレーションモデル・レベルと一貫性を有することから、シミュレーションモデルへの移行をシームレスに行うことができることも、基礎モデルの利点です。

シミュレーションモデル・レベル

シミュレーションモデル・レベルは、得られた概念モデルを、シミュレーションモデルとして「どのように実現するか」（How）を規定するものです。シミュレーションモデル・レベルは、ソフトウェア・フレームワークとして実行環境の一部となることで、シミュレーションを行うことができます。

第 2 章

基礎モデルの概念モデル・レベル

ここでは、基礎モデルの概念モデル・レベルについて説明します。この概念モデルを詳細化したものがソフトウェアフレームワークとして実装されているので、まずは、この概念モデルのレベルで理解することが不可欠です。

2.1 全体像

基礎モデルの中心となる部分は、World(世界)、Space(空間)、Clock(時計)、Entity(実体)、Agent(主体)、Goods(財)、Information(情報)、Behavior(行動)、Relation(関係)、Channel(経路) というクラスで構成されています。

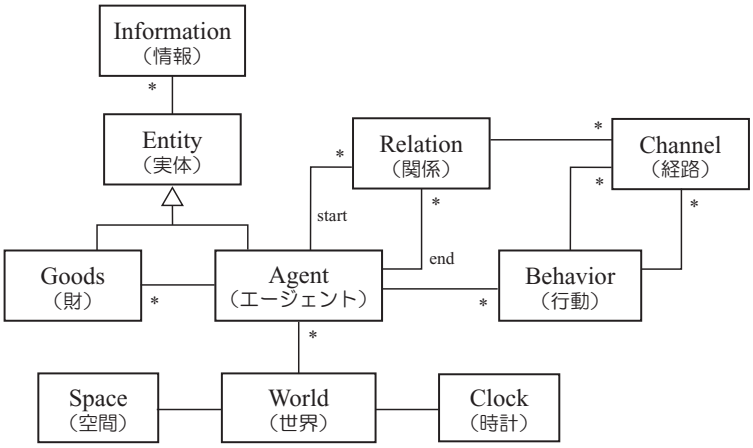


図 2.1: 基礎モデルの概念モデル・レベル

2.2　役割を外部化したエージェントの設計

基礎モデルは、エージェントのもつ行動・関係・財・情報を外部化し、Agent オブジェクトに付加するという方法を採用しています[*1]。これらの外部化された要素を組み合わせることによって、エージェントを設定していくのです。そのため、Agent 自体はそれらを束ねる役割を果たしているにすぎないということになります。例えば、Agent をインスタンス化すると、単なる Agent オブジェクトが得られますが、そこに PurchaseBehavior を加えると、購買行動を行う「消費者エージェント」となります。このようなエージェントの設計は、新しい行動・関係・財・情報の追加や削除、組み換えなどを簡単に行えるという柔軟性があります。こういった設計が、基礎モデルの最大の特徴といえるでしょう。

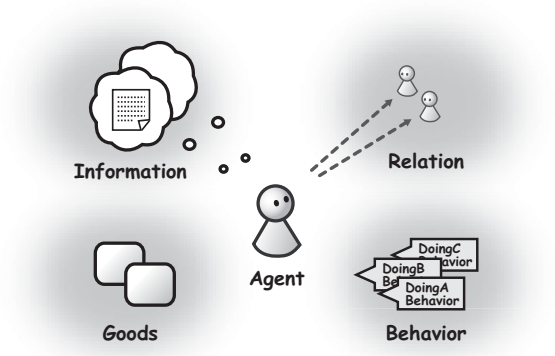


図 2.2: Agent は Behavior・Relation・Goods・Information をもつ

2.3　クラスの解説

2.3.1　Agent

Agent とは、社会・経済においてさまざまな活動を行う個人や社会集団 (企業・政府・家族・学校・地域社会・国) のことです。それぞれの Agent には、個性をもたせ、多様な振舞いや状態をとらせることができます。Agent は、世界に存在する Entity(実体) の一種です。

[*1] このような設計を、オブジェクト指向の分野では、「オブジェクトコンポジション」といいます。オブジェクトコンポジションとは、役割を外部化するためのオブジェクトを用意して振舞いを委譲し、そのオブジェクトを実行時に関連づける設計のことです。

2.3.2 Goods

Goods は、Agent に所有し交換される有形/無形の「もの」のことです。ここでいう「Goods」とは、人間の欲求を充足する性質をもつという経済学における狭義の意味ではなく、世界におけるさまざまなものを示す広義の概念として用いられています。例えば、自動車、石油、トウモロコシ、株、土地の権利、広告、書籍、水、声、騒音、ごみ、貨幣などは、どれも Goods オブジェクトとして表されます。

Goods は Information と違い、「量」の概念を持っています。また、Goods は、世界に存在する Entity(実体) の一種であり、Information を保持することができます。

2.3.3 Information

Entity(実体) である Agent と Goods は、Information を持つことができます。Goods に付随して取引される情報や、エージェントの記憶した情報などは、Information として定義されます。Information は単独では存在せず、必ず Entity(すなわち、Goods または Agent) によって保持されます。例えば、新聞を「紙」という Goods に新聞記事の Information が付随したものとして捉え、会話を「声」という無形で瞬間的な Goods に Information が付随したものとして捉えるということです。Agent が保持する Information は、主体の内部に貯蔵されているものであり、「記憶」や「遺伝子」などを指しています。

2.3.4 Behavior

Agent の行動は、「Behavior」として表現します。例えば、企業における生産行動や販売行動、個人における購買行動や労働行動などはどれも Behavior です。これらの Behavior によって、Goods や Information の作成や処理を行います。Agent は複数の Behavior をもつことができ、それらを並列的に実行することができます。Agent 間のやりとりは、Behavior が Goods，Information を送受信することで実現されます。

このような設計はモデルの表現力の点からみると、いくつかの利点があります。まず第一に、ひとつのエージェントが複数の社会的役割を担っているということを自然な形で表現できることにあります。例えば、あるエージェントが消費者かつ労働者であるといったようなモデルを作成できるようになります。また、新しい行動の追加や削除、行動の手続きの変更などが可能になり、これにより、エージェントの行動、役割が変化するということを扱うことができるようになります。

2.3.5　Relation と Channel

基礎モデルでは、あるエージェントが他のエージェントを知っているという状態を Relation（関係）によって表現します。これによって、友達関係や家族関係，雇用関係，行きつけの店などの関係を方向性をつけて表現することができます。基礎モデルでは Relation は 2 つの Agent を、方向を含めて接続するオブジェクトとして定義します。実際にコミュニケーションを行う際には、この Relation に基づいて開設されるコミュニケーション・パスを Channel（経路）を利用します。この Channel を通じて商品や会話内容，貨幣などの Goods，Information の送受信を行うことができます。

Relation は方向性をもっていますが、Channel には方向性はありません。そのため、一方向の Relation しかもたない Agent ともコミュニケーションをはかることができます。例えば、ある Agent が別の Agent に対して一方向の Relation を持っているとします。その Agent が Relation を通じて他方の Agent に Goods の送信をした場合、両者間には Channnel が開設されます。このとき、Goods を送られた方の Agent は、相手への Relation を持っていないのですが、開設されている Channel を通じて Goods や Information を送り返すことができるわけです。

Channel は、Agent 間を接続するものではなく、Behavior 間を接続するものです。また、同じ Agent に配置された別の Behavior とのやり取りも Channel を用いて行われます。このように、エージェント間のコミュニケーションにおいても、エージェント内部の行動の連携においても、Behavior 同士のやり取りはすべて Channel を通じた取引として抽象化されます[*2]。

2.3.6　World

対象世界を表現する土台が「World」です。World は、その世界に固有の空間と時間によって規定されています。World には、Agent が配置されます。

2.3.7　Space

「Space」は、その世界の地理的な構造を実現するものです。

2.3.8　Clock

「Clock」は不可逆的な時間の流れを表します。この時間が進むことで、シミュレーション上の時間が経過します。このとき、後に説明する「TimeEvent」が発生します。

[*2] このような抽象化により、モデルにおける統一性を保つだけでなく、Behavior のモデル部品やプログラム部品の独立性を高めることにもつながっています。

第 3 章

基礎モデルのシミュレーションモデル・レベル

基礎モデルのシミュレーションモデルの枠組みそのものは、概念モデルの枠組みと同じです。この章以降では基礎モデルのシミュレーションモデル・レベル全体からいくつかの重要なトピックを区切って述べていきます (詳細は API を参照してください)。

3.1 基礎モデルのシミュレーションモデル・レベルでの実現

この章以降では基礎モデルのシミュレーション・モデルレベルについて解説しますが、「モデルを作成すること」に焦点をあて、Java ソースコードとしての詳細な構造は解説しません。そこで、基礎モデルに基づくモデルの実装方法について整理します。

表 3.1: 各モデル要素の作成方法

基礎モデル要素	作成方法
Agent	AgentType を作成
Goods	GoodsType を作成
Relation	RelationType を作成
Information	サブクラスを作成（多数の Information が用意されている）
Behavior	Component Builder を使ってサブクラスを作成
World	Component Builder を使ってサブクラスを作成
Clock	サブクラスを作成（StepClock、RealClock を利用可能）
Space	サブクラスを作成（CellSpace を利用可能）

3.2　シミュレーションモデル・レベルで新たに登場する概念

3.2.1　Type

Type とは Agent、Goods、Relation、Information、Behavior のそれぞれに付属し、その型を表すクラスです。Type はモデルに存在するオブジェクトの要素を抽象化・分類して取得や指定するときに識別するための道具として導入されました。Type を用いることで、モデルの語彙を表現することが可能になります。これにより、同じ振る舞いをする Agent でも Type が異なれば違う Agent として扱い、また逆に違う振る舞いをする Agent でも Type が同じであれば、同じ Agent として扱うことができます。これにより、柔軟な検索・取得・識別を可能になっています。

また、Type はそれぞれが親子関係を持つことができます。これによって上位概念・下位概念を表現することができ、Type 間の包含・継承・依存などの関係を表現することもできます。Type は、多重継承も可能なので、親子関係で関連付けられた Type 群を意味空間として捉えると、実装に依存してしまうクラスよりも柔軟な要素の体系化を行うことが出来ます[*1]。

Type はモデルに付属するものではなく、コンテナによって管理されるため、複数のモデルで意味空間を共有することができます。

このような Type の仕組みを実装したのが以下のクラス図です。

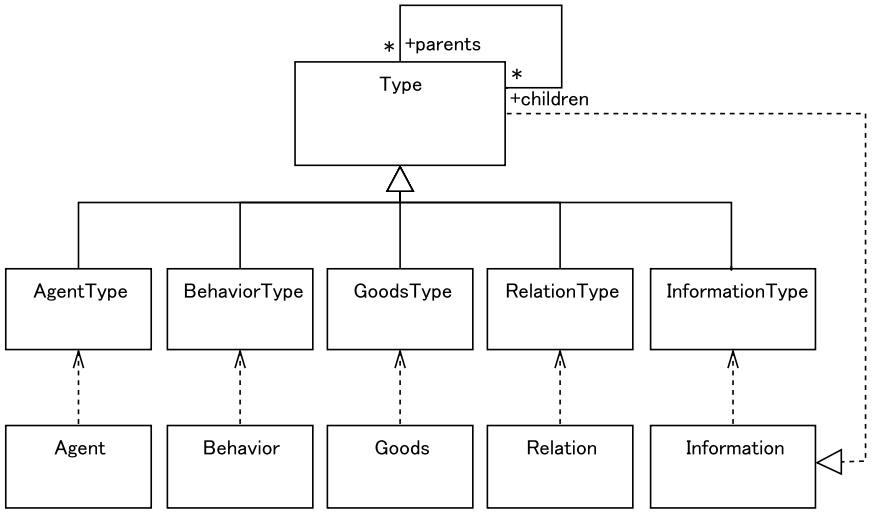


図 3.1: Type クラス図

親である抽象 Type クラスは 2 つの特徴を持っています。1 つは Type の識別子としても用いられる名前を持っているということ、もう 1 つは、複数個の親・子 Type を持つことができるということです。

[*1] クラスによる要素の分類も可能ですが、Java の言語規定によりクラスの多重継承は禁止されています。

抽象 Type クラスを継承して、AgentType、BehaviorType、GoodsType、Information-Type、RelationType の５つの Type が存在し、それぞれ対応するオブジェクトは必ず Type を持っています。これら５つの Type は別々に管理されます（つまり、GoodsType と AgentType で同じ名前の Type が存在しているとしても、それらは別 Type として扱われます)。

Type は Information を継承しているため、Information と同じように扱うことができます（例えば、Goods にそのまま付随させて他のエージェント送ることができます)。

それぞれの Type は、Model Designer でクラス図を書くことによって定義できます。また、定義された Type をシミュレーションモデル内で用いたいときは、World から取得します。

3.2.2 Priority

Agent への TimeEvent 発信は、通常ランダムに行います。しかし、Priority を設定することで、AgentType ごとに配信順番を制御することができます。Priority の値の大きさは、優先度の高さを表しています。つまり、１ステップの中で、Priority の値が大きい順に TimeEvent が発信されます。なお、設定は Model Designer 上で行います。

3.2.3 RandomNumberGenerator

RandomNumberGenerator は、乱数を利用するときに用いるクラスです。現在私たちが使用しているコンピュータは、その構造上完全にランダムな数字を生成することはできませんが、複雑なアルゴリズムによって事実上乱数と捉えて差し支えない数字（擬似乱数）を生成することができます。乱数生成にはいろいろな方法があるため、モデルやモデル作成者によって利用したい方法も異なると考えられます。このようなことを考慮し、RandomNumberGenerator は、乱数として数字を生成するアルゴリズムを内包し、値を取得するためのメソッドを公開するクラスとして定義されています。

3.3 World

Agent が配置される存在として World（世界）が定義されています。World と関連のクラスの構成は以下のようになっています[*2]。

World は複数の Agent オブジェクトを持ちます。また、RandomNumberGenerator を同様に複数持ちます。そして、時間・空間を表す Clock、Space クラスを持ちます。World が提供するインターフェースは以下の通りです。

[*2] 定義上、基礎モデルのシミュレーションモデル・レベルには含まれない役割なのですが、PlatBox Simulator から見た場合、World はモデルそのものを表すクラスとして扱われます。これにより、PlatBox Simulator は直接的には World クラスを扱い、World クラスには PlatBox Simulator のコンテナ部分へアクセスするためのメソッドを持っています。

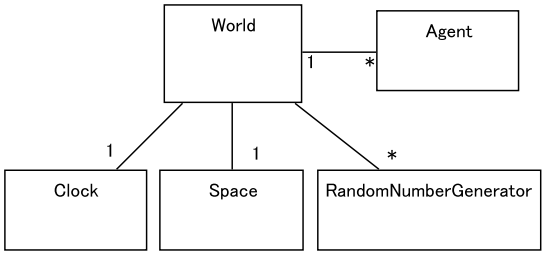


図 3.2: World クラス図

Agent の追加・削除・取得

World は Agent を配置してそれらを管理します。それらの追加、削除、生成、取得を行うことができます。取得は AgentType を利用して行われます。

Goods の生産・消費

Goods の生産・消費を行います。初期設定を除いては、Goods の生産や消費を決定し、実行しようとするのは、Agent です。しかし、Goods の生産や消費とは、世界におけるモノの生成・消滅でもあります。そのため、最終的には自然現象として World が行うということになっています。

Clock、Space の管理

Clock、Space の取得・設定を行います。モデルにおける時間・空間を定義するためにそれぞれただ 1 つの Clock、Space を持ちます。

Clock と Space はインターフェースだけ用意することで、時間・空間の柔軟な実装を行うことができるようにしてあります。ただし、Clock に関しては StepClock(整数で時刻を表した時計)、RealClock(実時間で時刻を表した時計)、Space に関しては CellSpace(2 次元のセル格子で空間を表す) のコンポーネントが用意されています。これらのコンポーネントをそのまま利用すれば、新たに Space、Clock を実装する必要はありません。

これら Clock、Space の設定、取得はいつでも可能ですが、モデル実行中にこれらの設定を変えてもモデルが正しく動作するかは保証しません。

RandomNumberGenerator（乱数生成アルゴリズム）の管理

RandomNumberGenerator は、World 上に同時に複数存在できます。その場合、RandomNumberGenerator は名前によって識別されます。Behavior などから名前で取得されて、利用されます。

3.4 Agent

Agent は、モデルにおいて World に配置されて主体性を持って動くオブジェクトとして設計されています。また、有限個の Goods、Information、Behavior、Relation を保持します。

以下に Agent と関連クラスのクラス図を示し、インターフェースの解説を行います (ただし、Information に関する機能は Entity の項で述べるため割愛します)。

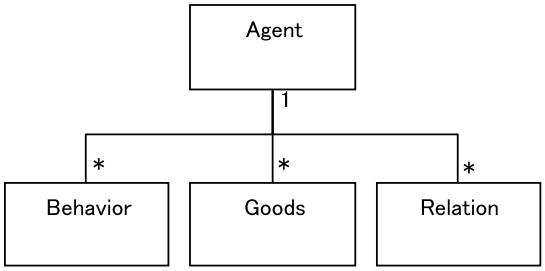


図 3.3: Agent クラス図

Agent の持つ機能は以下のとおりです。

TimeEvent の受信

時刻の経過を示す TimeEvent を受信して所持する Behavior に配布します。これによって Behavior が稼動するため Agent が行動すると言えます。

Behavior の追加・削除・取得

BehaviorType を指定して Behavior を追加します。追加した Behavior は自動的に開始状態になります。また、同じように Type を元に Behavior を取得、削除するメソッドも存在しています。

OpenChannelEvent の受信

経路の開設を求める OpenChannelEvent を受信して経路開設を試みます。成功すれば、適切な Behavior に対して開設された Channel を返します。

Goods の追加・削除・調査

Goods を追加する AddGoods、GoodsType を元に Goods を取り出す removeGoods、GoodsType に該当する Goods の量（Quantity）を調べる getQuantity メソッドがあります。多重参照をなるべく抑制するため、getGoods メソッドは用意されていません。

Relation の追加・削除・取得

Relation の追加及び Type による検索と削除を行うことが出来ます。

3.5 Goods

Goods には様々な事柄が記述された Information（情報）を付随させることができます。例えば、新聞は紙という Goods に新聞記事の Information が付随したものであり、会話は声という無形で瞬間的な Goods に Information が付随したものとなります。

Goods のクラス図は以下のとおりです

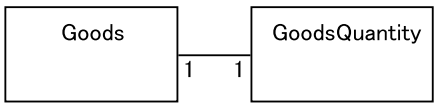


図 3.4: Goods クラス図

Goods はその量として GoodsQuantity を持ちます。注意すべき点として、Goods の Quantity を直接変更することはできません。分割や結合は Agent のメソッドを利用するようにしてください。

便利な機能として Goods には attachment として Information を保持させることができます。Goods は複数の Informaiton を管理することができますが（詳しくは Entity の項参照）、よりシンプルに 1 つだけ Information を扱いたい場合に attachment を利用することができます。

3.6 Entity

Entity は Agent、Goods の親クラスです。これらの 2 つのクラスの共通の性質である Information を管理するインターフェースを持ちます。

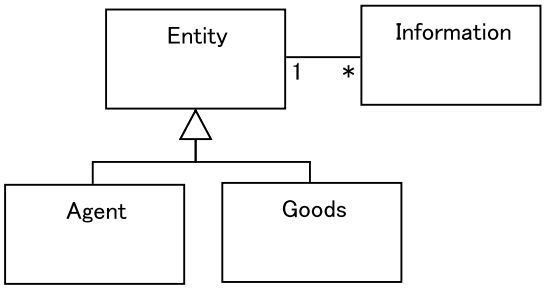


図 3.5: Entity クラス図

Information の管理はハッシュテーブルによって行われます。ハッシュテーブルにおけるキーは Information です。これにより、Information をキーとして Information を格納し、取り出すことができます。ただ、取り扱いを簡単にするために、自動的に取り扱いたい Information の InformationType をキーとして操作するメソッドが追加されています。

3.7 Information

Information は、モデルにおいて用いられる情報全般を表すために用いられます。

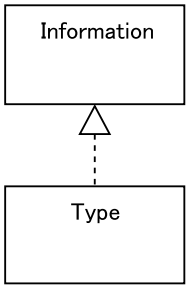


図 3.6: Information クラス図

Information の実装は Information インターフェースを実装して新たに作成できます。以下の Information は、あらかじめ用意されているので、自分で実装する必要はありません。

表 3.2: あらかじめ用意されている Information

Information 名	説明
AgentInformation	Agent への参照を記憶するための Information
BooleanInformation	boolean を記憶するための Information
ChannelInformation	Channel への参照を記憶するための Information
CollectionInformation	Information の集合を記憶するための Information
DoubleInformation	double を記憶するための Information
GoodsInformation	Goods への参照を記憶するための Information
IntegerInformation	int を記憶するための Information
ListInformation	Information の List を記憶するための Information
MapInformation	Informaiotn の Map を記憶するための Information
RelationInformation	Relation への参照を記憶するための Information
SetInformation	Information の Set を記憶するための Information
StringInformation	String を記憶するための Information
TableInformation	Information の Table を記憶するための Information

Information インターフェースは空インターフェースで、何もメソッドは存在しません。このため、他のオブジェクトと異なり、Type を持っていない（あるいは取得できない）Information を実装することができます。ただし、全ての Information はモデルコンテナ

起動時にクラス名を名前とする InformationType が登録されるので、World に問い合わせることによって、その Information の InformationType を取得することができます。

3.8 Relation と Channel

基礎モデルでは、ある Agent から他の Agent 間への関連性は「Relation」によって表されます。コミュニケーションの際には、この Relation に基づいて開設されるコミュニケーション・パスである「Channel」を通じて商品や会話、貨幣などのやりとりをします。

Relation 及び Channel のクラス図は以下のとおりです。

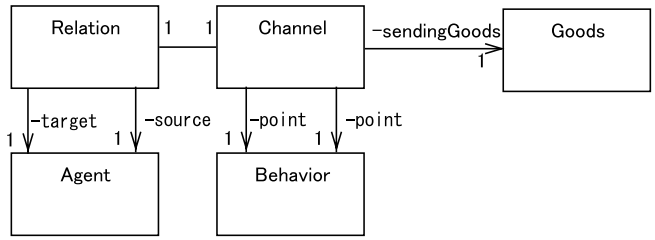


図 3.7: Relation 及び Channel クラス図

Channel の持つパラメータとして keep パラメータがあります。これは財送信後その Channel が継続して存在するかを設定するものです。false にしていると、財送信・受信処理が完了すると Channel は自動的に close が呼ばれます。true にしていると、財送信・受信処理後も Channel オブジェクトは存続します（Behavior から取得することができます)。

3.9 Behavior

エージェントの行動は、Behavior として定義されています。例えば、企業における生産行動や販売行動、個人における購買行動や労働行動などはどれも、個別に Behavior として作成します。

3.9.1 状態遷移による Behavior の実現

基礎モデルでは、エージェントの持つ Behavior を状態機械として定義しています。ある行動の手順や結果による分岐を状態として定義し、複数の状態が順に遷移し、その過程で処理を行うことで行動の実行を実現します。エージェントベースモデルでは、エージェントの内部状態が動的に変化していくということがしばしば強調されますが、基礎モデルでは、この内部状態の一部を各行動にそれぞれもたせ、より高度な内部状態の組み合わせを実現することができるのです。

注意すべき点として、遷移中には Event は受信されずに一時的に蓄積され、遷移先の状

態で受信されます。遷移中に送った Event の返事が即座に返ってきてもその受信は遷移先の状態で行われます。

3.9.2 Behavior の構造

Behavior のクラス図は以下の通りです。

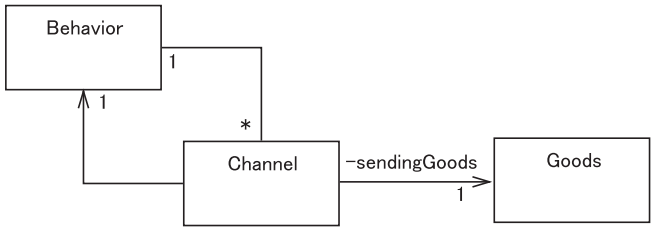


図 3.8: Behavior クラス図

Behavior を継承して実装しますが、Component Builder によって状態遷移を含む枠組みを作成することができます。またアクションは、Action Designer を用いて記述することができます。Behavior の多くの詳細な部分はユーザーには見えない形で隠蔽されています。特に、状態機械としての Behavior の動作の部分の実装はユーザーはほとんど意識する必要がありません。

3.9.3 Behavior に送られる Event

Behavior が受け取る Event は以下の 3 種類が存在しています。

TimeEvent

Agent から (その Agent に配置されている)Behavior に送られる Event です。モデル外部から時刻が経過したことを知らせるために Agent に送られて、Agent が Behavior に転送します。

ChannelEvent

Behavior で開設されている Channel から送られる Event です。Channel に対し別の Behavior から Goods が送られたときに発せられる Event です。

OpenChannelEvent

Relation から Agent を通して受け取る Event です。明示的に書かなくても受け取って Channel を開設しますが、あえて明示的にこの Event を受け取るように記述することもできます。

第 II 部

Action Block Language (ABL)

第 4 章

Action Block Language（ABL）とは

4.1 社会シミュレーションのモデル作成に特化した言語

これまでシミュレーションのモデル作成には、支援ツールを使う使わないに限らず、最終的には C 言語や Java 言語などのプログラミング言語による実装が必要でした。その結果、作成されたモデルが特定のプログラミング言語に依存し、そのプログラミング言語の知識がない人には、モデルを作成したり、他人が作ったモデルを理解したりすることができませんでした。

このような問題を解決するため、私たちは社会シミュレーションのモデル作成に特化したアクション記述言語「Action Block Language」(ABL) を提案しています。ABL は、基礎モデルで定義されている Agent，Relation，Information などの概念を基にして作成されています。また、ABL には「この Agent が持つ Goods を全て取り出す」というような、シミュレーションのモデル作成において頻出する処理の集まりが、語彙として定義されています。

ABL を使うことでモデル作成者は、より抽象度の高い記述でアクションのモデル化を行うことができます。例えば、これまでモデル作成者はアクションを実装する際、「this.getAgent()．removeAllGoods(GoodsType type)」というメソッドを呼ぶ、というプログラムを考える必要がありました (図 4.1 [a])。汎用的なアクション記述言語を使っても、特定のプログラミング言語に依存しないというだけで、考えなければならないことはあまり変わりません (図 4.1 [b])。これに対し、ABL を用いると、「この Agent が持つ Goods を全て取り出す」というように、シミュレーションの中で行いたい処理の目的を考え、それに対応する語句を ABL に定義されている語彙の中から選択することでアクションを記述することができます (図 4.1 [c])。

4.2 Action Block Language(ABL) の文法

ABL の文法には、制御構造と、論理的な構造を扱える仕組みが必要です。モデル作成者はときとして複雑な処理をアクションとして記述します。そのような場合、モデル作成

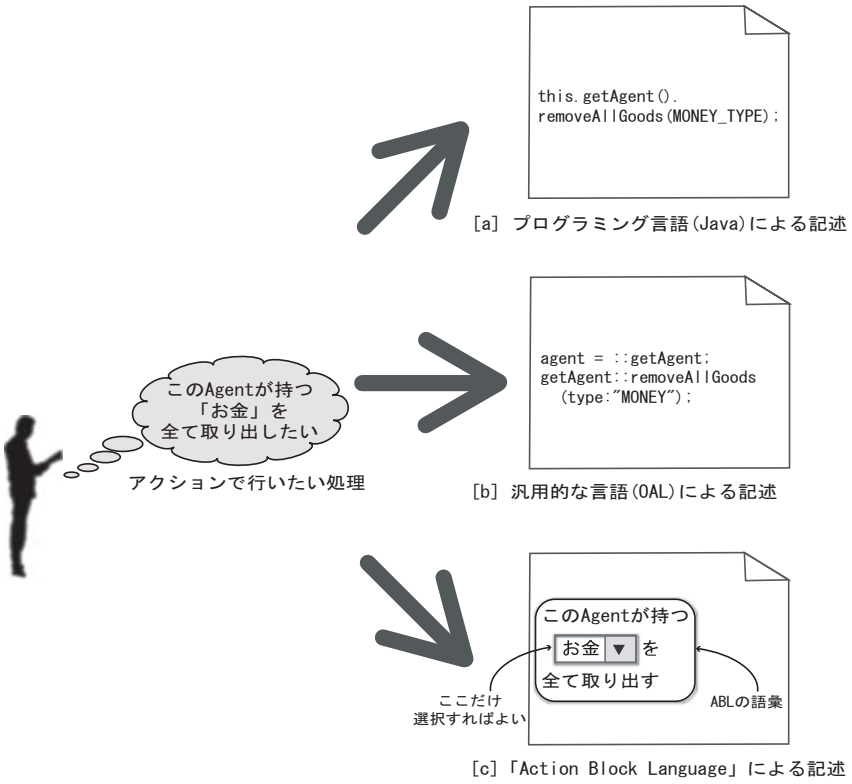


図 4.1: アクション記述の比較

者はアクションで行いたい処理をただ並べて書くのではなく、制御構造や論理的な構造を使い処理を階層化して記述する必要があります。

このために、ABL には「構造を記述するための要素」と「文を記述するための要素」の 2 種類の要素が文法として用意されています。「文を記述するための要素」はプログラミング言語における式の呼び出しなどの一般的な文や、制御文を記述するための要素です。「構造を記述するための要素」は制御構造と論理的な構造を扱うための要素です。「構造を記述するための要素」は下位の要素として、「文を記述するための要素」と他の「構造を記述するための要素」を持つことができます。

ABL には、制御構造を扱うための文法が定義されています。制御構造を表す要素の下位の要素として具体的な処理や、他の制御構造を定義することで、下位の要素を「繰り返す」もしくは「条件が真ならば実行する」という記述をすることができます。

また、ABL には制御構造以外に論理的な構造を記述するための文法が定義されています。論理的な構造はプログラムとしては特に意味を持たないが、これを使うことで処理のまとまりを階層化して記述することができます。モデル作成者は、まず論理的な処理の階層構造を設計し、次に設計した構造を実現するための手段として処理を記述していきます。

複雑なアクションを構造化して記述するために、アクション記述言語には 6 種類の「構造を記述するための要素」と、8 種類の「文を記述するための要素」が定義されています。

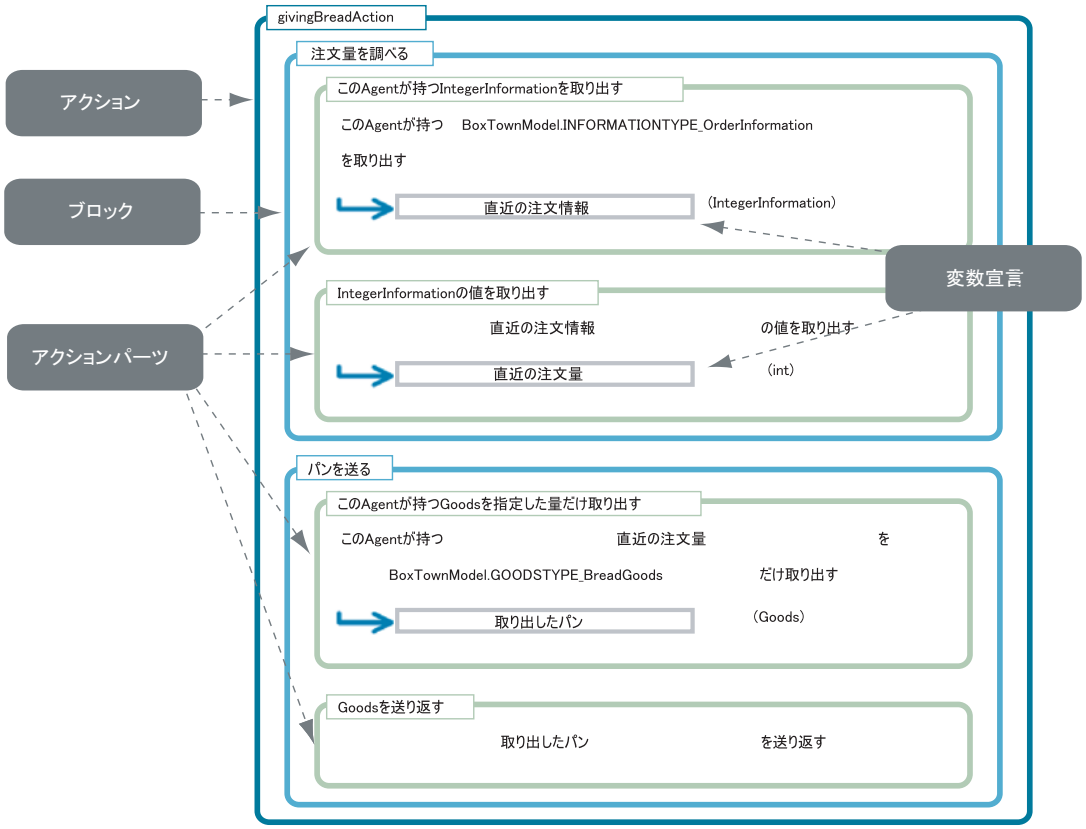


図 4.2: アクションの記述画面

以下にそれぞれの詳細を述べます。

構造を記述するための要素

(1) **メソッド**

メソッドは、プログラミング言語におけるメソッドと同じ役割を持ちます。複雑なアクションを構造化したり、アクションの中で重複する記述をサブルーチンとして定義したりするために使用します。

(2) **ブロック**

ブロックは、処理の論理的なまとまりを構造として記述するために利用します。プログラミング言語における一般的なブロックの定義とは違い、変数のスコープには関係しません。ブロックには名前をつけることができ、ブロックの中に定義された下位の要素の目的を記述することができます。

(3) **条件分岐**

条件分岐は、プログラミング言語における if 文に相当する構造です。条件分岐には、条件式として真偽値の変数を設定することができます。

(4) **集合操作**

集合操作は、集合のそれぞれの要素に対して何らかの処理を行いたいときに利用することができる構造です。集合操作には、操作を行いたい対象の集合と、集合の要素をどのような型のオブジェクトとして扱うかを指定することができます。

(5) **回数繰り返し**

回数繰り返しは、何らかの処理を数回繰り返して行いたい場合に利用することができる構造です。回数繰り返しには、下位の処理を何回繰り返すかを変数や値を使って設定することができます。

(6) **条件付繰り返し**

条件付繰り返しは、何らかの処理を一定の条件を満たすまで繰り返して行いたい場合に利用することができる構造です。条件付繰り返しには、繰り返しの継続条件を設定することができます。

文を記述するための要素

(1) **アクションパーツ**

アクションパーツには、それぞれ「この Agent が持つ Information を取り出す」というような、基礎モデルの用語で記述された名前がつけられています。また、アクションパーツには、それぞれ引数が定義されています。アクションパーツに引数を渡すことで、アクションパーツで操作する対象のオブジェクトを動的に変更したり、設定する値を動的に変更したりすることができます。どのようなアクションパーツが語彙として提供されているのかについては、次節で詳しく解説します。

(2) **操作の呼び出し**

操作の呼び出しは、アクションパーツとして用意されていない操作を行うための文です。メソッドを選択して呼び出すことができます。アクション記述の中では、モデル作成者が自ら定義したクラスやメソッドを利用するために用います。

(3) **変数宣言/代入**

変数宣言/代入は、アクションパーツとして用意されていない操作を行うための文です。呼び出したいメソッドを選択し、結果を格納する変数を定義することができます。

(4) **四則演算**

四則演算は、実数[*1]の値を式として設定し、その結果を変数として定義するための文です。四則演算の中では、値だけでなく、定義済みの他の変数を使うこともできます。

(5) **論理演算**

論理演算は、論理式を作成し、その結果を変数として定義するための文です。論理演算の中では、真偽値だけでなく、定義済みの他の変数を使うこともできます。

(6) **文字列**

アクション記述の中で使用する文字列を変数として定義するための文です。変数には、新たな文字列を値として格納するだけでなく、新たな文字列と他の変数に格納された文字列をつなげた結果を格納することもできます。

(7) return **文**

return 文は、プログラミング言語における return 文と同じく、メソッドが返す値を決定してメソッドの処理を中途脱出するための文です。

(8) continue **文**

continue 文は、プログラミング言語における continue 文と同じく、繰り返される処理を途中で止め、繰り返しの最初に戻って処理を行うための文です。そのため、continue 文は集合操作、回数繰り返し、条件付繰り返しの中でしか使用することができません。

(9) break **文**

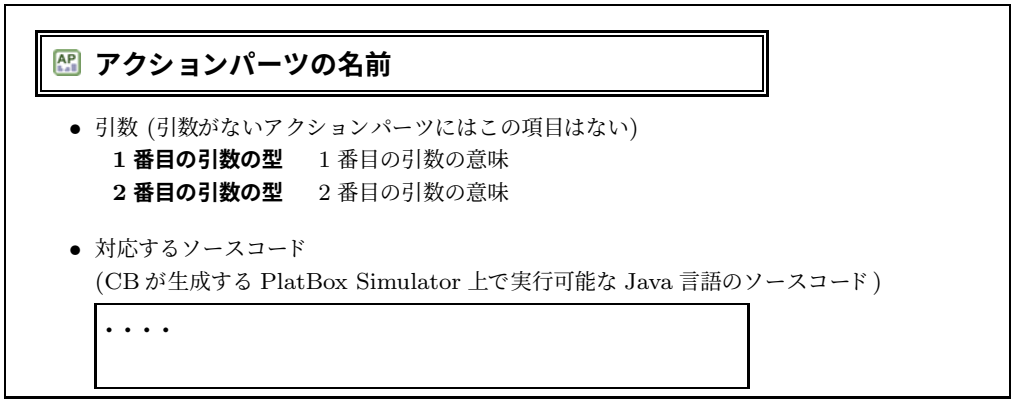
break 文は、プログラミング言語における break 文と同じく、繰り返される処理を途中で止め、繰り返しを中途脱出するための文です。そのため、break 文は集合操作、回数繰り返し、条件付繰り返しの中でしか使用することができません。

[*1] 正確には、浮動小数点数が扱えます。

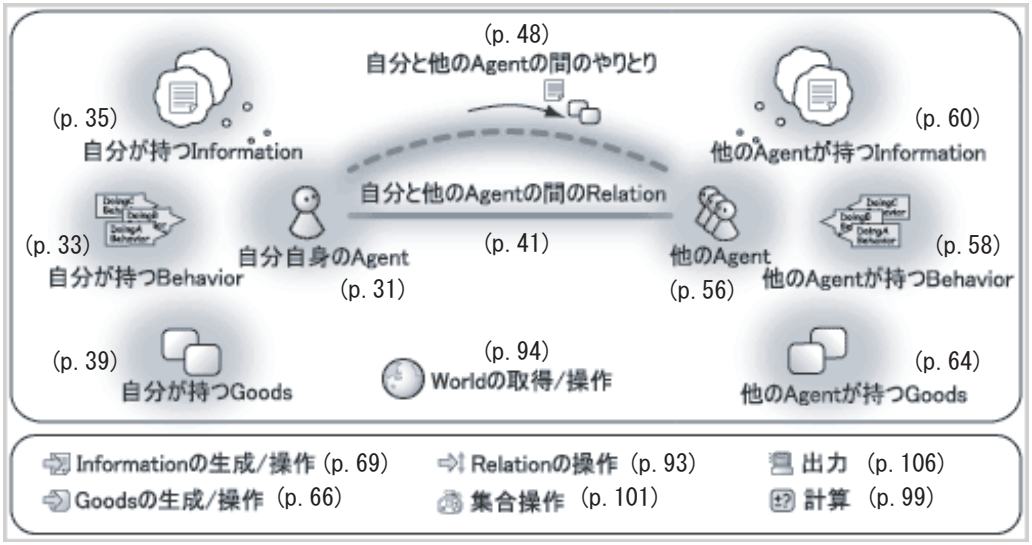
第 5 章

カテゴリ別のアクションパーツ一覧

ここでは Action Block Language(ABL) に語彙として用意されているアクションパーツの詳細な定義を紹介します。アクションパーツの定義は全て以下のような形式で記述されています。



【各カテゴリの解説ページ】



5.1 「自分自身の Agent」カテゴリ

この Agent が指定した Type の Behavior を持っているか調べる

- 引数

  BehaviorType 調べる対象となる BehaviorType
- 対応するソースコード

```
boolean 指定した Type の Behavior を持っているか
    = getAgent().getBehaviors([BehaviorType]).isEmpty();
```

この Agent が指定した Type の Goods を持っているか調べる

- 引数

  GoodsType 調べる対象となる GoodsType
- 対応するソースコード

```
boolean 指定した Type の Goods を持っているか
    = getAgent().hasGoods([GoodsType]);
```

この Agent が指定した Type の Information を持っているか調べる

- 引数

  InformationType 調べる対象となる InformationType
- 対応するソースコード

```
boolean 指定した Type の Information を持っているか
    = getAgent().hasInformation([InformationType]);
```

この Agent が指定した Type の Relation を持っているか調べる

- 引数

  RelationType 調べる対象となる RelationType
- 対応するソースコード

```
boolean 指定した Type の Relation を持っているか
    = getAgent().hasRelation([RelationType]);
```

この Agent の Type を取得する

- 対応するソースコード

```
AgentType 取得した AgentType = getAgent().getType();
```

この Agent を消滅させる

- 対応するソースコード

```
getAgent().destroy();
```

自分自身の Agent を取得する

- 対応するソースコード

```
Agent 自分自身の Agent = getAgent();
```

5.2 「自分が持つ Behavior」カテゴリ

この Agent が持つ Behavior を削除する

- 引数

  Behavior　削除する Behavior
- 対応するソースコード

```
getAgent().removeBehavior([Behavior]);
```

この Agent が持つ Behavior を取得する

- 引数

  BehaviorType　取得する Behavior の Type
- 対応するソースコード

```
Behavior 取得した Behavior = getAgent().getBehavior([BehaviorType]);
```

この Agent が持つ Behavior を親 Type を指定して再帰的に取得する

- 引数

  BehaviorType　取得する Behavior の Type
- 対応するソースコード

```
Collection 取得した Behavior の集合 = getAgent().getBehaviorsRecursively([BehaviorType]);
```

この Agent が持つ Behavior を全て取得する

- 対応するソースコード

```
Collection 取得した Behavior の集合 = getAgent().getAllBehaviors();
```

この Agent が持つ指定した Type の Behavior を全て取得する

- 引数

  BehaviorType　取得する Behavior の Type
- 対応するソースコード

```
Collection 取得した Behavior の集合 = getAgent().getBehaviors([BehaviorType]);
```

この Agent に Behavior を追加する

- 引数

  BehaviorType　追加する Behavior の Type

- 対応するソースコード

```
getAgent().addBehavior([BehaviorType]);
```

この Behavior の Type を取得する

- 対応するソースコード

```
BehaviorType 取得した BehaviorType = getType();
```

この Behavior の現在の状態名を取得する

- 対応するソースコード

```
String 現在の状態の名前 = getState().getName();
```

5.3 「自分が持つ Information」カテゴリ

この Agent が持つ DoubleInformation の値を減らす

- 引数
  - **InformationType** 値を減らす Information の Type
  - **double** 減らす量
- 対応するソースコード

```
InformationType 更新する Information の Type = [InformationType];
DoubleInformation 更新前の DoubleInformation
  = (DoubleInformation)getAgent().getInformation(更新する Information の Type);

//DoubleInformation の値を減らす
DoubleInformation 更新された DoubleInformation
  = new DoubleInformation(更新前の DoubleInformation.getValue() - [double]);
getAgent().putInformation(更新する Information の Type, 更新された DoubleInformation);
```

この Agent が持つ DoubleInformation の値を設定する

- 引数
  - **InformationType** 値を設定する Information の Type
  - **double** 設定する値
- 対応するソースコード

```
DoubleInformation 更新された DoubleInformation = new DoubleInformation([double]);
getAgent().putInformation([InformationType], 更新された DoubleInformation);
```

この Agent が持つ DoubleInformation の値を増やす

- 引数
  - **InformationType** 値を増やす Information の Type
  - **double** 増やす量
- 対応するソースコード

```
InformationType 更新する Information の Type = [InformationType];
DoubleInformation 更新前の DoubleInformation
  = (DoubleInformation)getAgent().getInformation(更新する Information の Type);

//DoubleInformation の値を増やす
DoubleInformation 更新された DoubleInformation
  = new DoubleInformation(更新前の DoubleInformation.getValue() + [double]);
getAgent().putInformation(更新する Information の Type, 更新された DoubleInformation);
```

この Agent が持つ DoubleInformation を取得する

- 引数
  InformationType　取得する Information の Type
- 対応するソースコード

```
DoubleInformation 取得した DoubleInformation
  = (DoubleInformation)getAgent().getInformation([InformationType]);
```

この Agent が持つ Information を削除する

- 引数
  InformationType　削除する Information の Type
- 対応するソースコード

```
Information 削除した Information = getAgent().removeInformation([InformationType]);
```

この Agent が持つ Information を取得する

- 引数
  InformationType　取得する Information の Type
- 対応するソースコード

```
Information 取得した Information = getAgent().getInformation([InformationType]);
```

この Agent が持つ Information を全て取得する

- 対応するソースコード

```
Map 取得した Information の Map = getAgent().getInformations();
```

この Agent が持つ IntegerInformation の値を減らす

- 引数
  InformationType　値を減らす Information の Type
  int　減らす量
- 対応するソースコード

```
InformationType 更新する Information の Type = [InformationType];
IntegerInformation 更新前の IntegerInformation
  = (IntegerInformation)getAgent().getInformation(更新する Information の Type);

//IntegerInformation の値を減らす
IntegerInformation 更新された IntegerInformation
  = new IntegerInformation(更新前の IntegerInformation.getValue() - [double]);
getAgent().putInformation(更新する Information の Type, 更新された IntegerInformation);
```

この Agent が持つ IntegerInformation の値を設定する

- 引数
  - **InformationType** 値を設定する Information の Type
  - **int** 設定する値
- 対応するソースコード

```
IntegerInformation 更新された IntegerInformation = new IntegerInformation([int]);
getAgent().putInformation([InformationType], 更新された IntegerInformation);
```

この Agent が持つ IntegerInformation の値を増やす

- 引数
  - **InformationType** 値を増やす Information の Type
  - **int** 増やす量
- 対応するソースコード

```
InformationType 更新する Information の Type = [InformationType];
IntegerInformation 更新前の IntegerInformation
  = (IntegerInformation)getAgent().getInformation(更新する Information の Type);

//IntegerInformation の値を増やす
IntegerInformation 更新された IntegerInformation
  = new IntegerInformation(更新前の IntegerInformation.getValue() + [double]);
getAgent().putInformation(更新する Information の Type, 更新された IntegerInformation);
```

この Agent が持つ IntegerInformation を取得する

- 引数
  - **InformationType** 取得する Information の Type
- 対応するソースコード

```
IntegerInformation 取得した IntegerInformation
   = (IntegerInformation)getAgent().getInformation([InformationType]);
```

この Agent に Information を記憶させる

- 引数
  - **InformationType** 記憶のキーにする InformationType
  - **Information** 記憶する Information
- 対応するソースコード

```
getAgent().putInformation([InformationType],[Information]);
```

5.4 「自分が持つ Goods」カテゴリ

この Agent が持つ Goods の Type を全て取得する

- 対応するソースコード

```
Collection この Agent が持つ Goods の Type の集合 = getAgent().getGoodsTypes();
```

この Agent が持つ Goods の量の合計を親 Type を指定して取得する

- 引数
  GoodsType　量を取得する Goods の親 Type
- 対応するソースコード

```
GoodsQuantity Goods の量 = getAgent().getQuantityRecursively([GoodsType]);
```

この Agent が持つ Goods の量を Type を指定して取得する

- 引数
  GoodsType　量を取得する Goods の Type
- 対応するソースコード

```
GoodsQuantity Goods の量 = getAgent().getQuantity([GoodsType]);
```

この Agent が持つ Goods を指定した量だけ取り出す

- 引数
  GoodsType　取り出す Goods の Type
  double　取り出す量
- 対応するソースコード

```
Goods 取り出した Goods = getAgent().removeGoods([GoodsType],[double]);
```

この Agent が持つ Goods を親 Type と量を指定して取り出す

- 引数
  GoodsType　取り出す Goods の親 Type
  double　取り出す量
- 対応するソースコード

```
Collection 取り出した Goods の集合
  = getAgent().removeGoodsRecursively([GoodsType],[double]);
```

この Agent が持つ Goods を親 Type を指定して全て取り出す

- 引数

  GoodsType　取り出す Goods の親 Type
- 対応するソースコード

```
Collection 取り出した Goods の集合 = getAgent().removeAllGoodsRecursively([GoodsType]);
```

この Agent が持つ指定した Type の Goods を全て取り出す

- 引数

  GoodsType　取り出す Goods の Type
- 対応するソースコード

```
Goods 取り出した Goods = getAgent().removeAllGoods([GoodsType]);
```

この Agent に Goods を持たせる

- 引数

  Goods　この Agent に持たせる Goods
- 対応するソースコード

```
getAgent().addGoods([Goods]);
```

5.5 「自分と他の Agent の間の Relation」カテゴリ

この Agent から他のエージェントへ Relation をむすぶ

- 引数
  RelationType　結ぶ Relation の Type
  Agent　結ぶ相手の Agent
- 対応するソースコード

```
getAgent().addRelation([RelationType],[Agent]);
```

この Agent が持つ Relation の Type を全て取得する

- 対応するソースコード

```
Collection 取得した RelationType の集合 = getAgent().getRelationTypes();
```

この Agent が持つ Relation を Type を指定して全て取得する

- 引数
  RelationType　取得する Relation の Type
- 対応するソースコード

```
Collection 取得した Relation の集合 = getAgent().getRelations([RelationType]);
```

この Agent が持つ Relation を削除する

- 引数
  Relation　削除する Relation
- 対応するソースコード

```
getAgent().removeRelation([Relation]);
```

この Agent が持つ Relation を取得する

- 引数
  RelationType　取得する Relation の Type
- 対応するソースコード

```
Relation 取得した Relation = getAgent().getRelation([RelationType]);
```

この Agent が持つ Relation を親 Type を指定して全て削除する

- 引数
  RelationType 削除する Relation の親 Type
- 対応するソースコード

```
getAgent().removeRelationsRecursively([RelationType]);
```

この Agent が持つ Relation を親 Type を指定して全て取得する

- 引数
  RelationType 取得する Relation の親 Type
- 対応するソースコード

```
Collection 取得した Relation の集合 = getAgent().getRelationsRecursively([RelationType]);
```

この Agent が持つ Relation を相手の Agent を指定して取得する

- 引数
  RelationType 取得する Relation の Type
  Agent 相手の Agent
- 対応するソースコード

```
Relation 取得した Relation = getAgent().getRelation([RelationType],[Agent]);
```

この Agent が持つ指定した Type の Relation を全て削除する

- 引数
  RelationType 削除する Relation の Type
- 対応するソースコード

```
getAgent().removeRelations([RelationType]);
```

この Agent と他の Agent の間で双方向の Relation をむすぶ

- 引数
  RelationType 結ぶ Relation の Type
  Agent 結ぶ相手の Agent
- 対応するソースコード

```
結んだ Relation の Type = [RelationType];
関係先の Agent = [Agent];

//双方向に結ぶ
getAgent().addRelation(結んだ Relation の Type, 関係先の Agent);
関係先の Agent.addRelation(結んだ Relation の Type,getAgent());
```

この Agent と他人の間の双方向の Relation を削除する

- 引数
  - **Agent**　Relation を削除する相手の Agent
  - **RelationType**　削除する Relation の Type
- 対応するソースコード

```
相手の Agent = [Agent];
削除した Relation の Type = [RelationType];

//自分からの Relation を削除する
自分からの Relation = getAgent().getRelation(削除した Relation の Type,target);
getAgent().removeRelation(自分からの Relation);

//相手からの Relation を削除する
相手からの Relation = target.getRelation(削除した Relation の Type,getAgent());
target.removeRelation(相手からの Relation);
```

この Agent の Relation を全て取得する

- 対応するソースコード

```
Collection 取得した Relation の集合 = getAgent().getAllRelations();
```

現在アクティブな Channel を開いている関係を取得する

- 対応するソースコード

```
Relation 現在アクティブな Channel を開いている関係 = getActiveChannel().getParentRelation();
```

他の Agent からこの Agent へ Relation をむすぶ

- 引数
  - **Agent**　結ぶ相手の Agent
  - **RelationType**　結ぶ Relation の Type
- 対応するソースコード

```
[Agent].addRelation([RelationType],getAgent());
```

他の Agent から他の Agent へ Relation をむすぶ

- 引数
  - **AgentA**　関係元の Agent
  - **RelationType**　結ぶ Relation の Type
  - **AgentB**　関係先の Agent
- 対応するソースコード

```
[AgentA].addRelation([RelationType],[AgentB]);
```

他の Agent が持つ Relation の Type を全て取得する

- 引数
  Agent 対象となる Agent
- 対応するソースコード

```
Collection 取得した RelationType の集合 = [Agent].getRelationTypes();
```

他の Agent が持つ Relation を Type を指定して全て取得する

- 引数
  Agent 対象となる Agent
  RelationType 取得する Relation の Type
- 対応するソースコード

```
Collection 取得した Relation の集合 = [Agent].getRelations([RelationType]);
```

他の Agent が持つ Relation を削除する

- 引数
  Agent 対象となる Agent
  Relation 削除する Relation
- 対応するソースコード

```
[Agent].removeRelation([Relation]);
```

他の Agent が持つ Relation を取得する

- 引数
  Agent 対象となる Agent
  RelationType 取得する Relation の Type
- 対応するソースコード

```
Collection 取得した Relation = [Agent].getRelation([RelationType]);
```

他の Agent が持つ Relation を親 Type を指定して全て削除する

- 引数
  Agent 対象となる Agent
  RelationType 削除する Relation の Type
- 対応するソースコード

```
[Agent].removeRelationsRecursively([RelationType]);
```

他の Agent が持つ Relation を親 Type を指定して全て取得する

- 引数
  - **Agent**　対象となる Agent
  - **RelationType**　取得する Relation の親 Type
- 対応するソースコード

```
Collection 取得した Relation の集合 = [Agent].getRelationsRecursively([RelationType]);
```

他の Agent が持つ Relation を相手の Agent を指定して取得する

- 引数
  - **AgentA**　対象となる Agent
  - **RelationType**　取得する Relation の Type
  - **AgentB**　相手の Agent
- 対応するソースコード

```
Relation 取得した Relation = [AgentA].getRelation([RelationType],[AgentB]);
```

他の Agent が持つ指定した Type の Relation を全て削除する

- 引数
  - **Agent**　対象となる Agent
  - **RelationType**　削除する Relation の Type
- 対応するソースコード

```
[Agent].removeRelations([RelationType]);
```

他の Agent の Relation を全て取得する

- 引数
  - **Agent**　対象となる Agent
- 対応するソースコード

```
Collection 取得した Relation の集合 = [Agent].getAllRelations();
```

他人と他人の間で双方向の Relation をむすぶ

- 引数
  - **RelationType** 結ぶ Relation の Type
  - **AgentA** Relation を結ぶ一方の Agent
  - **AgentB** Relation を結ぶ他方の Agent
- 対応するソースコード

```
結んだ Relation の Type = [RelationType];
一方の Agent = [AgentA];
他方の Agent = [AgentB];

//Relation を双方向に結ぶ
一方の Agent.addRelation(結んだ Relation の Type, 他方の Agent);
他方の Agent.addRelation(結んだ Relation の Type, 一方の Agent);
```

他人と他人の間の双方向の Relation を削除する

- 引数
  - **AgentA** Relation を持つ一方の Agent
  - **AgentB** Relation を持つ他方の Agent
  - **RelationType** 削除する Relation の Type
- 対応するソースコード

```
一方の Agent = [AgentA];
他方の Agent = [AgentB];
削除した Relation の Type = [RelationType];

//一方からの Relation を削除する
一方からの Relation = 一方の Agent.getRelation(削除した Relation の Type, 他方の Agent);
一方の Agent.removeRelation(一方からの Relation);

//他方からの Relation を削除する
他方からの Relation = 他方の Agent.getRelation(削除した Relation の Type, 一方の Agent);
他方の Agent.removeRelation(他方からの Relation);
```

5.6 「自分と他の Agent の間のやりとり」カテゴリ

Goods や Information を送ってきた相手にメッセージを送る

- 引数
  String　メッセージの内容
- 対応するソースコード

```
返事の Information = new MessageInformation([String]);
sendInformation(返事の Information);
```

Goods を送り返す

- 引数
  Goods　送り返す Goods
- 対応するソースコード

```
sendGoods([Goods]);
```

Information を送り返す

- 引数
  InformationType　送り返すときにキーとなる InformationType
  Information　送り返す Information
- 対応するソースコード

```
sendInformation([InformationType],[Information]);
```

RelationType と一人一人に送る量を指定して全員に Goods を送る

- 引数
  RelationType　送るために使う Relation の Type
  BehaviorType　送り先の Behavior の Type
  GoodsType　送りたい Goods の Type
  double　送りたい Goods の量
- 対応するソースコード

```
sendGoods([RelationType],[BehaviorType],[GoodsType],[double],false);
```

RelationType と一人一人に送る量を指定して全員に Goods を送る (Channel はキープする)

- 引数
  - **RelationType** 送るために使う Relation の Type
  - **BehaviorType** 送り先の Behavior の Type
  - **GoodsType** 送りたい Goods の Type
  - **double** 送りたい Goods の量
- 対応するソースコード

```
sendGoods([RelationType],[BehaviorType],[GoodsType],[double],false,true);
```

RelationType を指定してメッセージを送る

- 引数
  - **String** メッセージの内容
  - **RelationType** 送るために使う Relation の Type
  - **BehaviorType** 送り先の Behavior の Type
- 対応するソースコード

```
メッセージの Information = new MessageInformation([String]);
sendInformation([RelationType],[BehaviorType], メッセージの Information);
```

Relation を指定して Goods の量が N 以上だったら Goods を N だけ送る

- 引数
  - **GoodsType** 送りたい Goods の Type
  - **double** 送りたい Goods の量
  - **Relation** 送るために使う Relation
  - **BehaviorType** 送り先の Behavior の Type
- 対応するソースコード

```
送信する Goods の Type = [GoodsType];
送信する量 = [double];

//Goods を送る
if(getAgent.getQuantity(送信する Goods の Type).getValueAsDouble >= 送信する量){

  送信する Goods = getAgent().removeGoods(送信する Goods の Type, 送信する量);
   sendGoods([Relation],[BehaviorType], 送信する Goods,false);
}
```

Relation を指定して Goods の量が N 以上だったら Goods を N だけ送る（Channel はキープする）

- 引数
  - **GoodsType**　送りたい Goods の Type
  - **double**　送りたい Goods の量
  - **Relation**　送るために使う Relation
  - **BehaviorType**　送り先の Behavior の Type
- 対応するソースコード

```
送信する Goods の Type = [GoodsType];
送信する量 = [double];

//Goods を送る
if(getAgent.getQuantity(送信する Goods の Type).getValueAsDouble >= 送信する量){

  送信する Goods = getAgent().removeGoods(送信する Goods の Type, 送信する量);
   sendGoods([Relation],[BehaviorType], 送信する Goods,true);
}
```

Relation を指定して Goods を送る

- 引数
  - **Relation**　送るために使う Relation
  - **BehaviorType**　送り先の Behavior の Type
  - **Goods**　送る Goods
- 対応するソースコード

```
sendGoods([Relation],[BehaviorType],[Goods],false);
```

Relation を指定して Goods を送る（Channel はキープする）

- 引数
  - **Relation**　送るために使う Relation
  - **BehaviorType**　送り先の Behavior の Type
  - **Goods**　送る Goods
- 対応するソースコード

```
sendGoods([Relation],[BehaviorType],[Goods],true);
```

Relation を指定して一人に Information を送る

- 引数
  - **Relation**　送るために使う Relation
  - **BehaviorType**　送り先の Behavior の Type
  - **InformationType**　送るときのキーとする InformationType
  - **Information**　送る Information
- 対応するソースコード

```
sendInformation([Relation],[BehaviorType],[InformationType],[Information],false);
```

Relation を指定して一人に Information を送る（Channel はキープする）

- 引数
  - **Relation** 送るために使う Relation
  - **BehaviorType** 送り先の Behavior の Type
  - **InformationType** 送るときのキーとする InformationType
  - **Information** 送る Information
- 対応するソースコード

```
sendInformation([Relation],[BehaviorType],[InformationType],[Information],true);
```

現在アクティブな Channel をつないだ元の Behavior を取得する

- 対応するソースコード

```
Behavior Channel をつないだ元の Behavior = getActiveChannel().getBehaviorA();
```

現在アクティブな Channel をつないだ先の Behavior を取得する

- 対応するソースコード

```
Behavior Channel をつないだ先の Behavior = getActiveChannel().getBehaviorB();
```

現在アクティブな Channel を取得する

- 対応するソースコード

```
Channel アクティブな Channel = getActiveChannel();
```

現在アクティブな Channel を閉じる

- 対応するソースコード

```
getActiveChannel().close();
```

最後に受け取った Goods が指定された Type のものか調べる

- 引数
  - **GoodsType** 判定対象の GoodsType
- 対応するソースコード

```
boolean 受け取った Goods が指定された Type のものかどうか = receivedGoodsEquals([GoodsType]);
```

最後に受け取った Goods を持つ

- 対応するソースコード

```
Goods 受け取った Goods = getReceivedGoods();
getAgent().addGoods(受け取った Goods);
```

最後に受け取った Goods を取得する

- 対応するソースコード

```
Goods 最後に受け取った Goods = getReceivedGoods();
```

最後に受け取った Information が指定された Type のものか調べる

- 引数

  InformationType　判定対象の InformationType

- 対応するソースコード

```
boolean 受け取った Information が指定された Type のものかどうか
  = receivedInformationEquals([InformationType]);
```

最後に受け取った Information を DoubleInformation として取得する

- 対応するソースコード

```
DoubleInformation 最後に受け取った Information
  = (DoubleInformation)getReceivedInformation();
```

最後に受け取った Information を IntegerInformation として取得する

- 対応するソースコード

```
IntegerInformation 最後に受け取った Information
  = (IntegerInformation)getReceivedInformation();
```

最後に受け取った Information を記憶する

- 引数

  InformationType　記憶するときにキーとする InformationType

- 対応するソースコード

```
Information 受け取った Information = getReceivedInformation();
getAgent().putInformation([InformationType], 受け取った Information);
```

最後に受け取った Information を取得する

- 対応するソースコード

```
Information 最後に受け取った Information = getReceivedInformation();
```

指定した RelationType の Relation を持つ Agent 全員に Information を送る

- 引数
  - **RelationType** 送るために使う Relation の Type
  - **BehaviorType** 送り先の Behavior の Type
  - **InformationType** 送るときのキーとする InformationType
  - **Information** 送る Information
- 対応するソースコード

```
情報を送った人数
 = sendInformation(
    [RelationType],[BehaviorType],[InformationType],[Information],false);
```

指定した RelationType の Relation を持つ Agent 全員に Information を送る (Channel はキープする)

- 引数
  - **RelationType** 送るために使う Relation の Type
  - **BehaviorType** 送り先の Behavior の Type
  - **InformationType** 送るときのキーとする InformationType
  - **Information** 送る Information
- 対応するソースコード

```
情報を送った人数
 = sendInformation(
    [RelationType],[BehaviorType],[InformationType],[Information],true);
```

指定した RelationType の Relation 一つを取り出し Goods を送る

- 引数
  - **RelationType** 送るために使う Relation の Type
  - **BehaviorType** 送り先の Behavior の Type
  - **Goods** 送る Goods
- 対応するソースコード

```
Relation Goods の送信に使った Relation = getAgent().getRelation([RelationType]);
sendGoods(relation,[BehaviorType],[Goods],false);
```

指定した RelationType の Relation 一つを取り出し Information を送る

- 引数

RelationType	送るために使う Relation の Type
BehaviorType	送り先の Behavior の Type
InformationType	送るときのキーとする InformationType
Information	送る Information

- 対応するソースコード

```
Relation Information の送信に使った Relation = getAgent().getRelation([RelationType]);
sendInformation(relation,[BehaviorType],[InformationType],[Information],false);
```

指定した RelationType の Relation 一つを取り出し Information を送る（Channel はキープする）

- 引数

RelationType	送るために使う Relation の Type
BehaviorType	送り先の Behavior の Type
InformationType	送るときのキーとする InformationType
Information	送る Information

- 対応するソースコード

```
Relation Information の送信に使った Relation = getAgent().getRelation([RelationType]);
sendInformation(relation,[BehaviorType],[InformationType],[Information],true);
```

全ての Channel を取得する

- 対応するソースコード

```
List アクティブな Channel のリスト = getAllChannels();
```

5.7 「他の Agent」カテゴリ

新しい Agent を作る

- 引数
  AgentType 新たに生成する Agent の Type
- 対応するソースコード

```
Agent 新しい Agent = getWorld().createAgent([AgentType]);
```

他の Agent が指定した Type の Behavior を持っているか調べる

- 引数
  Agent 対象となる Agent
  BehaviorType 対象となる BehaviorType
- 対応するソースコード

```
boolean 指定した Type の Behavior を持っているか
  = [Agent].getBehaviors([BehaviorType]).isEmpty();
```

他の Agent が指定した Type の Goods を持っているか調べる

- 引数
  Agent 対象となる Agent
  GoodsType 対象となる GoodsType
- 対応するソースコード

```
boolean 指定した Type の Goods を持っているか = [Agent].hasGoods([GoodsType]);
```

他の Agent が指定した Type の Information を持っているか調べる

- 引数
  Agent 対象となる Agent
  InformationType 対象となる InformationType
- 対応するソースコード

```
boolean 指定した Type の Information を持っているか
  = [Agent].hasInformation([InformationType]);
```

他の Agent が指定した Type の Relation を持っているか調べる

- 引数
  Agent　　対象となる Agent
  RelationType　対象となる RelationType
- 対応するソースコード

```
boolean 指定した Type の Relation を持っているか
  = [Agent].hasRelation([RelationType]);
```

他の Agent の Type を取得する

- 引数
  Agent　対象となる Agent
- 対応するソースコード

```
AgentType 取得した AgentType = [Agent].getType();
```

他の Agent を消滅させる

- 引数
  Agent　対象となる Agent
- 対応するソースコード

```
[Agent].destroy();
```

5.8 「他の Agent が持つ Behavior」カテゴリ

他の Agent が持つ Behavior を削除する

- 引数
  Agent 対象となる Agent
  Behavior 削除する Behavior
- 対応するソースコード

```
[Agent].removeBehavior([Behavior]);
```

他の Agent が持つ Behavior を取得する

- 引数
  Agent 対象となる Agent
  BehaviorType 取得する Behavior の Type
- 対応するソースコード

```
Behavior 取得した Behavior = [Agent].getBehavior([BehaviorType]);
```

他の Agent が持つ Behavior を親 Type を指定して再帰的に取得する

- 引数
  Agent 対象となる Agent
  BehaviorType 取得する Behavior の親 Type
- 対応するソースコード

```
Collection 取得した Behavior の集合 = [Agent].getBehaviorsRecursively([BehaviorType]);
```

他の Agent が持つ Behavior を全て取得する

- 引数
  Agent 対象となる Agent
- 対応するソースコード

```
Collection 取得した Behavior の集合 = [Agent].getAllBehaviors();
```

他の Agent が持つ指定した Type の Behavior を全て取得する

- 引数
  Agent 対象となる Agent
  BehaviorType 取得する Behavior の Type
- 対応するソースコード

```
Collection 取得した Behavior の集合 = [Agent].getBehaviors([BehaviorType]);
```

他の Agent に Behavior を追加する

- 引数
  - **Agent**　対象となる Agent
  - **BehaviorType**　追加する Behavior の Type
- 対応するソースコード

```
[Agent].addBehavior([BehaviorType]);
```

5.9 「他の Agent が持つ Information」カテゴリ

他の Agent が持つ DoubleInformation の値を減らす

- 引数
  - **InformationType** 減らす Information の Type
  - **Agent** 対象となる Agent
  - **double** 減らす量
- 対応するソースコード

```
Information 更新する Information の Type = [InformationType];
DoubleInformation 更新前の DoubleInformation
  = (DoubleInformation) [Agent].getInformation(更新する Information の Type);

//DoubleInformation の値を減らす
DoubleInformation 更新された DoubleInformation
  = new DoubleInformation (更新前の DoubleInformation.getValue() - [double]);
[Agent].putInformation(更新する Information の Type, 更新する Information の Type);
```

他の Agent が持つ DoubleInformation の値を設定する

- 引数
  - **double** 設定する値
  - **Agent** 対象となる Agent
  - **InformationType** 設定する Information の Type
- 対応するソースコード

```
更新された DoubleInformation = new DoubleInformation([double]);
[Agent].putInformation([InformationType], 更新された DoubleInformation);
```

他の Agent が持つ DoubleInformation の値を増やす

- 引数
  - **InformationType** 増やす Information の Type
  - **Agent** 対象となる Agent
  - **double** 増やす量
- 対応するソースコード

```
更新する Information の Type = [InformationType];
DoubleInformation 更新前の DoubleInformation
  = (DoubleInformation) [Agent].getInformation(更新する Information の Type);

//DoubleInformation の値を増やす
DoubleInformation 更新された DoubleInformation
  = new DoubleInformation(更新前の DoubleInformation.getValue() + [double]);
[Agent].putInformation(更新する Information の Type, 更新された DoubleInformation);
```

他の Agent が持つ DoubleInformation を取得する

- 引数
  - **Agent**　対象となる Agent
  - **InformationType**　取得する Information の Type
- 対応するソースコード

```
DoubleInformation 取得した DoubleInformation
  = (DoubleInformation)[Agent].getInformation([InformationType]);
```

他の Agent が持つ Information を削除する

- 引数
  - **Agent**　対象となる Agent
  - **InformationType**　削除する Information の Type
- 対応するソースコード

```
Information 削除した Information = [Agent].removeInformation([InformationType]);
```

他の Agent が持つ Information を取得する

- 引数
  - **Agent**　対象となる Agent
  - **InformationType**　取得する Information の Type
- 対応するソースコード

```
Information 取得した Information = [Agent].getInformation([InformationType]);
```

他の Agent が持つ Information を全て取得する

- 引数
  - **Agent**　対象となる Agent
- 対応するソースコード

```
Map 取得した Information の Map = [Agent].getInformations();
```

他の Agent が持つ IntegerInformation の値を減らす

- 引数
  - **InformationType** 減らす Information の Type
  - **Agent** 対象となる Agent
  - **int** 減らす量
- 対応するソースコード

```
InformationType 更新する Information の Type = [InformationType];
IntegerInformation 更新前の IntegerInformation
 = (IntegerInformation)[Agent].getInformation(更新する Information の Type);

IntegerInformation 更新された IntegerInformation
 = new IntegerInformation(更新前の IntegerInformation.getValue() - [int]);
[Agent].putInformation(更新する Information の Type, 更新された IntegerInformation);
```

他の Agent が持つ IntegerInformation の値を設定する

- 引数
  - **int** 設定する値
  - **Agent** 対象となる Agent
  - **InformationType** 設定する Information の Type
- 対応するソースコード

```
IntegerInformation 更新された IntegerInformation = new IntegerInformation([int]);
[Agent].putInformation([InformationType], 更新された IntegerInformation);
```

他の Agent が持つ IntegerInformation の値を増やす

- 引数
  - **InformationType** 増やす Information の Type
  - **Agent** 対象となる Agent
  - **int** 増やす量
- 対応するソースコード

```
InformationType 更新する Information の Type = [InformationType];
IntegerInformation 更新前の IntegerInformation
 = [Agent].getInformation(更新する Information の Type);

IntegerInformation 更新された IntegerInformation
 = new IntegerInformation(更新前の IntegerInformation.getValue() + [int]);
[Agent].putInformation(更新する Information の Type, 更新された IntegerInformation);
```

他の Agent が持つ IntegerInformation を取得する

- 引数
  - **Agent** 対象となる Agent
  - **InformationType** 取得する Information の Type
- 対応するソースコード

```
IntegerInformation 取得した IntegerInformation
 = (IntegerInformation)[Agent].getInformation([InformationType]);
```

他の Agent に Information を記憶させる

- 引数

Agent	対象となる Agent
InformationType	記憶するときのキーとする InformationType
Information	記憶する Information

- 対応するソースコード

```
[Agent].putInformation([InformationType],[Information]);
```

5.10 「他の Agent が持つ Goods」カテゴリ

他の Agent が持つ Goods の Type を全て取得する

- 引数
  - **Agent** 対象となる Agent
- 対応するソースコード

```
Collection 他の Agent が持つ Goods の Type の集合 = [Agent].getGoodsTypes();
```

他の Agent が持つ Goods の量の合計を親 Type を指定して取得する

- 引数
  - **Agent** 対象となる Agent
  - **GoodsType** 量を取得する Goods の親 Type
- 対応するソースコード

```
GoodsQuantitiy Goods の量 = [Agent].getQuantityRecursively([GoodsType]);
```

他の Agent が持つ Goods の量を Type を指定して取得する

- 引数
  - **Agent** 対象となる Agent
  - **GoodsType** 量を取得する Goods の Type
- 対応するソースコード

```
GoodsQuantity Goods の量 = [Agent].getQuantity([GoodsType]);
```

他の Agent が持つ Goods を指定した量だけ取り出す

- 引数
  - **Agent** 対象となる Agent
  - **GoodsType** 取り出す Goods の Type
  - **double** 取り出す量
- 対応するソースコード

```
Goods 取り出した Goods = [Agent].removeGoods([GoodsType],[double]);
```

他の Agent が持つ Goods を親 Type と量を指定して取り出す

- 引数
  - **Agent**　対象となる Agent
  - **GoodsType**　取り出す Goods の親 Type
  - **double**　取り出す量
- 対応するソースコード

```
Collection 取り出した Goods の集合 = [Agent].removeGoodsRecursively([GoodsType],[double]);
```

他の Agent が持つ Goods を親 Type を指定して全て取り出す

- 引数
  - **Agent**　対象となる Agent
  - **GoodsType**　取り出す Goods の親 Type
- 対応するソースコード

```
Collection 取り出した Goods の集合 = [Agent].removeAllGoodsRecursively([GoodsType]);
```

他の Agent が持つ指定した Type の Goods を全て取り出す

- 引数
  - **Agent**　対象となる Agent
  - **GoodsType**　取り出す Goods の Type
- 対応するソースコード

```
Goods 取り出した Goods = [Agent].removeAllGoods([GoodsType]);
```

他の Agent に Goods を持たせる

- 引数
  - **Agent**　対象となる Agent
  - **Goods**　持たせる Goods
- 対応するソースコード

```
[Agent].addGoods([Goods]);
```

5.11 「Goods の生成/操作」カテゴリ

Goods に Information を付与する

- 引数
  - **Goods** 対象となる Goods
  - **InformationType** 付与するときのキーとなる InformationType
  - **Information** 付与する Information
- 対応するソースコード

```
[Goods].putInformation([InformationType],[Information]);
```

Goods に指定された Type の Information が付随しているか調べる

- 引数
  - **Goods** 対象となる Goods
  - **InformationType** 対象となる InformationType
- 対応するソースコード

```
boolean Goods に指定された Type の Information が付随しているか
  = [Goods].hasInformation([InformationType]);
```

Goods に付随する Information を取得する

- 引数
  - **Goods** 対象となる Goods
  - **InformationType** 取得する Information の Type
- 対応するソースコード

```
Information 取得した Information = [Goods].getInformation([InformationType]);
```

Goods に付随する Information を全て取得する

- 引数
  - **Goods** 対象となる Goods
- 対応するソースコード

```
Map 取得した Information のマップ = [Goods].getInformations();
```

Goods に付属する Information を削除する

- 引数
  Goods　対象となる Goods
  InformationType　削除する Information の Type
- 対応するソースコード

```
Information 削除した Information = [Goods].removeInformation([InformationType]);
```

Goods の Type を取得する

- 引数
  Goods　対象となる Goods
- 対応するソースコード

```
GoodsType 取得した Goods の Type = [Goods].getType();
```

Goods の量を取得する

- 引数
  Goods　対象となる Goods
- 対応するソースコード

```
GoodsQuantity 取得した Goods の量 = [Goods].getGoodsQuantity();
```

Goods の量を小数のオブジェクトとして取得する

- 引数
  Goods　対象となる Goods
- 対応するソースコード

```
Double 取得した Goods の量
  = [Goods].getGoodsQuantity().getValueAsDoubleObject();
```

Goods の量を小数の値として取得する

- 引数
  Goods　対象となる Goods
- 対応するソースコード

```
double 取得した Goods の量
  = [Goods].getGoodsQuantity().getValueAsDouble();
```

Goods の量を整数のオブジェクトとして取得する

- 引数
  Goods 対象となる Goods
- 対応するソースコード

```
Integer 取得した Goods の量
 = [Goods].getGoodsQuantity().getValueAsIntegerObject();
```

Goods の量を整数の値として取得する

- 引数
  Goods 対象となる Goods
- 対応するソースコード

```
int 取得した Goods の量
 = [Goods].getGoodsQuantity().getValueAsInt();
```

Goods を消費する

- 引数
  Goods 対象となる Goods
- 対応するソースコード

```
getWorld().consumeGoods([Goods]);
```

新しい Goods を作る

- 引数
  GoodsType 新しく生成する Goods の Type
  double 生成する量
- 対応するソースコード

```
Goods 新しい Goods = getWorld().createGoods([GoodsType],[double]);
```

5.12 「Information の生成/操作」カテゴリ

AgentInformation に記憶されている Agent の Type を取得する

- 引数
  AgentInformation　対象となる AgentInformation
- 対応するソースコード

```
AgentType 記憶されている Agent の Type = [AgentInformation].getAgentType();
```

AgentInformation に記憶されている Agent を取得する

- 引数
  AgentInformation　対象となる AgentInformation
- 対応するソースコード

```
Agent 記憶されている Agent = [AgentInformation].getAgent();
```

AgentInformation をつくる

- 引数
  AgentInformation　対象となる AgentInformation
- 対応するソースコード

```
AgentInformation 新しい AgentInformation = new AgentInformation([Agent]);
```

BooleanInformation の値を取得する

- 引数
  BooleanInformation　対象となる BooleanInformation
- 対応するソースコード

```
boolean booleanInformation の値 = [BooleanInformation].getValue();
```

BooleanInformation をつくる

- 引数
  BooleanInformation　新しい BooleanInformation
- 対応するソースコード

```
BooleanInformation 新しい BooleanInformation = new BooleanInformation(true);
```

CollectionInformation から CollectionInformation の要素を削除する

- 引数
  - **CollectionInformation** 対象となる CollectionInformation
  - **CollectionInformation** 削除する CollectionInformation
- 対応するソースコード

```
[CollectionInformation].removeAll([CollectionInformation]);
```

CollectionInformation から Information の集合の要素を削除する

- 引数
  - **CollectionInformation** 対象となる CollectionInformation
  - **Collection** 削除する集合
- 対応するソースコード

```
[CollectionInformation].removeAll([Collection]);
```

CollectionInformation から Information を削除する

- 引数
  - **CollectionInformation** 対象となる CollectionInformation
  - **Information** 削除する Information
- 対応するソースコード

```
[CollectionInformation].remove([Information]);
```

CollectionInformation が空かどうかを調べる

- 引数
  - **CollectionInformation** 対象となる CollecttionInformation
- 対応するソースコード

```
boolean 空かどうか = [CollectionInformation].isEmpty();
```

CollectionInformation に CollectionInformation の要素が全て含まれているかどうかを調べる

- 引数
  - **CollectionInformation** 対象となる CollectionInformation
  - **CollectionInformation** 判定対象の CollectionInformation
- 対応するソースコード

```
boolean 含まれているかどうか = [CollectionInformation].containsAll(CollectionInformation);
```

CollectionInformation に CollectionInformation の要素を全て追加する

- 引数
  - **CollectionInformation**　対象となる CollectionInformation
  - **CollectionInformation**　追加する CollectionInformation
- 対応するソースコード

```
[CollectionInformation].addAll([CollectionInformation]);
```

CollectionInformation に Information が含まれているかどうかを調べる

- 引数
  - **CollectionInformation**　対象となる CollectionInformation
  - **Information**　判定対象の Information
- 対応するソースコード

```
boolean 含まれているかどうか = [CollectionInformation].contains([Information]);
```

CollectionInformation に Information の集合の要素が全て含まれているかどうかを調べる

- 引数
  - **CollectionInformation**　対象となる CollectionInformation
  - **Collection**　判定対象の集合
- 対応するソースコード

```
boolean 含まれているかどうか = [CollectionInformation].containsAll([Collection]);
```

CollectionInformation に Information の集合の要素を全て追加する

- 引数
  - **CollectionInformation**　対象となる CollectionInformation
  - **Collection**　追加する集合
- 対応するソースコード

```
[CollectionInformation].addAll([Collection]);
```

CollectionInformation に Information を追加する

- 引数
  - **CollectionInformation**　対象となる CollectionInformation
  - **Information**　追加する Information
- 対応するソースコード

```
[CollectionInformation].add([Information]);
```

CollectionInformation の内容を Information の集合として取得する

- 引数
  CollectionInformation 対象となる CollectionInformation
- 対応するソースコード

```
Collection 取得した Information の集合 = [CollectionInformation].getInformations();
```

CollectionInformation の要素数を取得する

- 引数
  CollectionInformation 対象となる CollectionInformation
- 対応するソースコード

```
int 要素数 = [CollectionInformation].size();
```

CollectionInformation を Information の配列に変換する

- 引数
  CollectionInformation 対象となる CollectionInformation
- 対応するソースコード

```
org.platbox.simulator.model.fmfw.Information[] information の配列
= [CollectionInformation].toArray();
```

CollectionInformation を空にする

- 引数
  CollectionInformation 対象となる CollectionInformation
- 対応するソースコード

```
[CollectionInformation].clear();
```

DoubleInformation の値を取得する

- 引数
  DoubleInformation 対象となる DoubleInformation
- 対応するソースコード

```
double 取り出した少数値 = [DoubleInformation].getValue();
```

DoubleInformation をつくる

- 引数
  double　新しい DoubleInformation の値
- 対応するソースコード

```
DoubleInformation 新しい DoubleInformation = new DoubleInformation([double]);
```

GoodsInformation に記憶されている Goods の Type を取得する

- 引数
  GoodsInformation　対象となる GoodsInformation
- 対応するソースコード

```
GoodsType 記憶されている Goods の Type = [GoodsInformation].getGoodsType();
```

GoodsInformation に記憶されている Goods を取得する

- 引数
  GoodsInformation　対象となる GoodsInformation
- 対応するソースコード

```
Goods 記憶されている Goods = [GoodsInformation].getGoods();
```

GoodsInformation をつくる

- 引数
  Goods　新しい GoodsInformation の値
- 対応するソースコード

```
GoodsInformation 新しい GoodsInformation = new GoodsInformation([Goods]);
```

Information のマップから新しい MapInformation をつくる

- 引数
  Map 対象となるマップ
- 対応するソースコード

```
MapInformation 新しい MapInformation = new MapInformation([Map]);
```

Information の内容を文字列で取得する

- 引数
  Information　対象となる Information
- 対応するソースコード

```
String information の内容 = [Information].toString();
```

Information の集合から新しい ListInformation をつくる

- 引数
  Collection　対象となる集合
- 対応するソースコード

```
ListInformation 新しい ListInformation = new ListInformation([Collection]);
```

Information の集合から新しい SetInformation をつくる

- 引数
  Collection　対象となる集合
- 対応するソースコード

```
SetInformation 新しい SetInformation = new SetInformation([Collection]);
```

IntegerInformation の値を取得する

- 引数
  IntegerInformation　対象となる IntegerInformation
- 対応するソースコード

```
int 取り出した整数値 = [IntegerInformation].getValue();
```

IntegerInformation をつくる

- 引数
  int　新しい IntegerInformation の値
- 対応するソースコード

```
IntegerInformation 新しい IntegerInformation = new IntegerInformation([int]);
```

ListInformation の N 番目に CollectionInformation の要素を追加する

- 引数
  - **ListInformation**　対象となる ListInformation
  - **int**　何番目に追加するか
  - **CollectionInformation**　追加する要素
- 対応するソースコード

```
[ListInformation].addAll([int],[CollectionInformation]);
```

ListInformation の N 番目に Information の集合の要素を追加する

- 引数
  - **ListInformation**　対象となる ListInformation
  - **int**　何番目に追加するか
  - **Collection**　追加する集合の要素
- 対応するソースコード

```
[ListInformation].addAll([int],[Collection]);
```

ListInformation の N 番目に Information を追加する

- 引数
  - **ListInformation**　対象となる ListInformation
  - **int**　何番目に追加するか
  - **Information**　追加する Information
- 対応するソースコード

```
[ListInformation].add([int],[Information]);
```

ListInformation の N 番目の Information を上書きする

- 引数
  - **ListInformation**　対象となる ListInformation
  - **int**　何番目に上書きするか
  - **Information**　上書きする Information
- 対応するソースコード

```
[ListInformation].set([int},[Information]);
```

ListInformation の N 番目の Information を削除する

- 引数
  - **ListInformation**　対象となる ListInformation
  - **int**　何番目を削除するか
- 対応するソースコード

```
Information 削除した Information=[ListInformation].remove([int]);
```

ListInformation の N 番目の Information を取得する

- 引数
  - **ListInformation** 対象となる ListInformation
  - **int** 何番目を取得するか
- 対応するソースコード

```
Information 取得した Information=[ListInformation].get([int]);
```

ListInformation の一部を切り出す

- 引数
  - **ListInformation** 対象となる ListInformation
  - **int** 何番目から切り出すか
  - **int** 何番目まで切り出すか
- 対応するソースコード

```
ListInformation 切り出された ListInformation=[ListInformation].subList([int],[int]);
```

ListInformation の中で指定された Information が最後に登場するのは何番目かを調べる

- 引数
  - **ListInformation** 対象となる ListInformation
  - **Information** 調べる Information
- 対応するソースコード

```
何番目に最後に登場するか=[ListInformation].lastIndexOf([Information]);
```

ListInformation の内容をリストとして取得する

- 引数
  - **ListInformation** 対象となる ListInformation
- 対応するソースコード

```
List 取得した Information のリスト=[ListInformation].getInformationsAsList();
```

ListInformation をシャッフルする

- 引数
  - **ListInformation** 対象となる ListInformation
- 対応するソースコード

```
[ListInformation].shuffle();
```

ListInformation をソーティングアルゴリズムを指定してソートする

- 引数
  ListInformation　対象となる ListInformation
  Comparator　ソーティングアルゴリズム
- 対応するソースコード

```
[ListInformation].sort([Comparator]);
```

ListInformation をソートする

- 引数
  ListInformation　対象となる ListInformation
- 対応するソースコード

```
[ListInformation].sort();
```

ListInformation を乱数シードを指定してシャッフルする

- 引数
  ListInformation　対象となる ListInformation
  long　乱数シード
- 対応するソースコード

```
[ListInformation].shuffle([long]);
```

ListInformation を逆順にする

- 引数
  ListInformation　対象となる ListInformation
- 対応するソースコード

```
[ListInformation].reverse();
```

MapInformation から Information を削除する

- 引数
  Information　キーとなる Information
- 対応するソースコード

```
Information 削除した Information = 新しい MapInformation.remove([Information]);
```

MapInformation から Information を取得する

- 引数
  MapInformation 対象となる MapInformation
  Information キーとなる Information
- 対応するソースコード

```
Information 取得した Information = [MapInformation].get([Information]);
```

MapInformation が空かどうかを調べる

- 引数
  MapInformation 対象となる MapInformation
- 対応するソースコード

```
boolean 空かどうか = [MapInformation].isEmpty();
```

MapInformation に Information を追加する

- 引数
  MapInformation 対象となる MapInformation
  Information キーとなる Information
  Information 追加する Information
- 対応するソースコード

```
[MapInformation].put([Information], [Information]);
```

MapInformation に他の MapInformation の要素を全て追加する

- 引数
  MapInformation 対象となる MapInformation
  MapInformation 追加する MapInformation
- 対応するソースコード

```
[MapInformation].putAll([MapInformation]);
```

MapInformation に指定された Information がキーとして含まれているかどうかを調べる

- 引数
  MapInformation 対象となる MapInformation
  Information 調べる対象となる Information
- 対応するソースコード

```
boolean 含まれているかどうか = [MapInformation].containsKey([Information]);
```

MapInformation に指定された Information が値として含まれているかどうかを調べる

- 引数
  MapInformation　対象となる MapInformation
  Information　調べる対象となる Information
- 対応するソースコード

```
含まれているかどうか = [MapInformation].containsValue([Information]);
```

MapInformation のキーを全て SetInformation として取得する

- 引数
  MapInformation　対象となる MapInformation
- 対応するソースコード

```
SetInformation 取得したキーの SetInformation = [MapInformation].keySet();
```

MapInformation の値を全て ListInformation として取得する

- 引数
  MapInformation　対象となる MapInformation
- 対応するソースコード

```
ListInformation 取得した値の ListInformation = [MapInformation].values();
```

MapInformation の要素数を取得する

- 引数
  MapInformation　対象となる MapInformation
- 対応するソースコード

```
int 要素数 = [MapInformation].size();
```

MapInformation を空にする

- 引数
  MapInformation　対象となる MapInformation
- 対応するソースコード

```
[MapInformation].clear();
```

RelationInformation に記憶されている Relation の Type を取得する

- 引数

  RelationInformation 対象となる RelationInformation
- 対応するソースコード

```
RelationType 記憶されている Relation の Type = [RelationInformation].getRelationType();
```

RelationInformation に記憶されている Relation を取得する

- 引数

  RelationInformation 対象となる RelationInformation
- 対応するソースコード

```
Relation 記憶されている Relation = [RelationInformation].getRelation();
```

RelationInformation をつくる

- 引数

  Relation 対象となる Relation
- 対応するソースコード

```
RelationInformation 新しい RelationInformation = new RelationInformation([Relation]);
```

SetInformation の内容をセットとして取得する

- 引数

  SetInformation 取得する SetInformation
- 対応するソースコード

```
Set 取得した Information のセット = [SetInformation].getInformationsAsSet();
```

StringInformation の値を取得する

- 引数

  StringInformation 取得する StringInformation
- 対応するソースコード

```
String stringInformation の値 = [StringInformation].getValue();
```

StringInformation をつくる

- 引数
  String　新しい StringInformation の値
- 対応するソースコード

```
StringInformation 新しい StringInformation = new StringInformation([String]);
```

TableInformation が空かどうかを調べる

- 引数
  TableInformation　対象となる TableInformation
- 対応するソースコード

```
boolean 空かどうか = [TableInformation].isEmpty();
```

TableInformation に 1 行分の Information を挿入する

- 引数
  TableInformation　対象となる TableInformation
  int　何行目に挿入するか
  ListInformation　挿入する ListInformation
- 対応するソースコード

```
[TableInformation].insertRow([int], [ListInformation]);
```

TableInformation に 1 行分の Information を追加する

- 引数
  TableInformation　対象となる TableInformation
  ListInformation　追加する ListInformation
- 対応するソースコード

```
[TableInformation].addRow([ListInformation]);
```

TableInformation に Information を追加する

- 引数
  TableInformation　対象となる TableInformation
  Information　追加する列のヘッダ
  Information　追加する Information
- 対応するソースコード

```
[TableInformation].add([Information], [Information]);
```

TableInformation に含まれる 1 列分の Information を上書きする

- 引数
  - **TableInformation** 対象となる TableInformation
  - **int** 何行目を上書きするか
  - **ListInformation** 上書きする Information
- 対応するソースコード

```
[TableInformation].setRow([int], [ListInformation]);
```

TableInformation に含まれる 1 列分の Information を取得する

- 引数
  - **TableInformation** 対象となる TableInformation
  - **Information** 取得する列のヘッダ
- 対応するソースコード

```
ListInformation 1 列分の Information = [TableInformation].getValues([Information]);
```

TableInformation に含まれる 1 行分の Information を削除する

- 引数
  - **TableInformation** 対象となる TableInformation
  - **int** 削除する行
- 対応するソースコード

```
ListInformation 削除した行の Information = [TableInformation].removeRow([int]);
```

TableInformation に含まれる 1 行分の Information を取得する

- 引数
  - **TableInformation** 対象となる TableInformation
  - **int** 取得する行
- 対応するソースコード

```
ListInformation 取得した行分の Information = [TableInformation].getRow([int]);
```

TableInformation に含まれる Information を上書きする

- 引数
  - **TableInformation** 対象となる TableInformation
  - **int** 上書きする行
  - **Information** 上書きする列のヘッダ
  - **Information** 上書きする Information
- 対応するソースコード

```
[TableInformation].set([int], [Information], [Information]);
```

TableInformation に含まれる Information を削除する

- 引数
  - **TableInformation**　対象となる TableInformation
  - **int**　削除する行
  - **Information**　削除する列のヘッダ
- 対応するソースコード

```
Information 削除した Information = [TableInformation].remove([int],[Information]);
```

TableInformation に含まれる Information を取得する

- 引数
  - **TableInformation**　対象となる TableInformation
  - **int**　取得する行
  - **Information**　取得する列のヘッダ
- 対応するソースコード

```
Information 取得した Information = [TableInformation].get([int],[Information]);
```

TableInformation のセルを検索して結果を ListInformation として取得する

- 引数
  - **TableInformation**　対象となる TableInformation
  - **Information**　検索する列のヘッダ
  - **Information**　検索する Information
  - **Information**　検索した行の取得する列のヘッダ
- 対応するソースコード

```
ListInformation 全ての検索結果の Information
= [TableInformation].searchInformations([Information], [Information], [Information]);
```

TableInformation のヘッダを取得する

- 引数
  - **TableInformation**　対象となる TableInformation
- 対応するソースコード

```
ListInformation ヘッダ = [TableInformation].getHeaders();
```

TableInformation のヘッダを設定する

- 引数
  - **TableInformation**　対象となる TableInformation
  - **ListInformation**　ヘッダとなる ListInformation
- 対応するソースコード

```
[TableInformation].setHeaders([ListInformation]);
```

TableInformation の一部を新たな TableInformation として切り出す

- 引数
  - **TableInformation** 対象となる TableInformation
  - **int** 何行目から切り出すか
  - **int** 何行目まで切り出すか
- 対応するソースコード

```
TableInformation 切り出した TableInformation
= [TableInformation].getSubTable([int], [int]);
```

TableInformation の全ての行を取得する

- 引数
  - **TableInformation** 対象となる TableInformation
- 対応するソースコード

```
ListInformation 全ての行 = [TableInformation].getAllRow();
```

TableInformation の列数を取得する

- 引数
  - **TableInformation** 対象となる TableInformation
- 対応するソースコード

```
int 列数 = [TableInformation].getRowCount();
```

TableInformation の行を検索して結果の行を取得する

- 引数
  - **TableInformation** 対象となる TableInformation
  - **Information** 検索する列のヘッダ
  - **Information** 取得する行の Information
- 対応するソースコード

```
ListInformation 検索結果の行 = [TableInformation].searchRow([Information],[Information]);
```

TableInformation の行を検索して結果を ListInformation として取得する

- 引数
  - **TableInformation** 対象となる TableInformation
  - **Information** 検索する列のヘッダ
  - **Information** 取得する行の Information
- 対応するソースコード

```
ListInformation 全ての検索結果の行
= [TableInformation].searchRows([Information], [Information]);
```

TableInformation の行数を取得する

- 引数
  TableInformation　対象となる TableInformation
- 対応するソースコード

```
int 行数 = [TableInformation].getRowCount();
```

TableInformation をシャッフルする

- 引数
  TableInformation　対象となる TableInformation
- 対応するソースコード

```
[TableInformation].shuffle();
```

TableInformation をソーティングアルゴリズムを指定してソートする

- 引数
  TableInformation　対象となる TableInformation
  Comparator　ソーティングアルゴリズム
- 対応するソースコード

```
[TableInformation].sort([Information], [Comparator]);
```

TableInformation をソートする

- 引数
  TableInformation　対象となる TableInformation
  Information　ソートするキーとなる列のヘッダ
- 対応するソースコード

```
[TableInformation].sort([Information]);
```

TableInformation を乱数シードを指定してシャッフルする

- 引数
  TableInformation　対象となる TableInformation
  long　乱数シード
- 対応するソースコード

```
[TableInformation].shuffle([long]);
```

TableInformation を検索して結果の Information を取得する

- 引数
  - **TableInformation** 対象となる TableInformation
  - **Information** 検索する列のヘッダ
  - **Information** 検索する行の Information
  - **Information** 検索した行の中で取得する列のヘッダ
- 対応するソースコード

```
Information 検索結果の Information
= [TableInformation].searchInformation([Information],[Information],[Information]);
```

TableInformation を空にする

- 引数
  - **TableInformation** 対象となる TableInformation
- 対応するソースコード

```
[TableInformation].clear();
```

TableInformation を逆順にする

- 引数
  - **TableInformation** 対象となる TableInformation
- 対応するソースコード

```
[TableInformation].reverse();
```

他の CollectionInformation から新しい ListInformation をつくる

- 引数
  - **CollectionInformation** 新しい ListInformation の値
- 対応するソースコード

```
ListInformation 新しい ListInformation = new ListInformation([CollectionInformation]);
```

他の CollectionInformation から新しい SetInformation をつくる

- 引数
  - **CollectionInformation** 新しい SetInformation の値
- 対応するソースコード

```
SetInformation 新しい SetInformation = new SetInformation([CollectionInformation]);
```

他の MapInformation から新しい MapInformation をつくる

- 引数
  MapInformation　新しい MapInformation の値
- 対応するソースコード

```
MapInformation 新しい MapInformation = new MapInformation([MapInformation]);
```

同じ Information かどうかを調べる

- 引数
  Information　対象となる Information
  Information　比較する Information
- 対応するソースコード

```
boolean 同じ Information かどうか = [Information].equals([Information]);
```

指定された Information が ListInformation の何番目にあるかを調べる

- 引数
  ListInformation　対象とする ListInformation
  Information　調べたい Information
- 対応するソースコード

```
int 何番目にあるか = [ListInformation].indexOf([Information]);
```

空の ListInformation をつくる

- 引数

- 対応するソースコード

```
新しい ListInformation = new ListInformation();
```

空の MapInformation をつくる

- 引数

- 対応するソースコード

```
新しい MapInformation = new MapInformation();
```

空の SetInformation をつくる

- 引数
- 対応するソースコード

```
新しい SetInformation = new SetInformation();
```

空の TableInformation をつくる

- 引数
- 対応するソースコード

```
TableInformation 新しい TableInformation = new TableInformation();
```

表のヘッダとなる Information のリストを指定して TableInformation をつくる

- 引数
  List　新しい TableInformation のヘッダ
- 対応するソースコード

```
新しい TableInformation = new TableInformation([List]);
```

表のヘッダとなる ListInformation を指定して TableInformation をつくる

- 引数
  ListInformation　新しい TableInformation のヘッダ
- 対応するソースコード

```
新しい TableInformation = new TableInformation([ListInformation]);
```

5.13 「Relation の操作」カテゴリ

Relation の Type を取得する

- 引数
  Relation　対象となる Relation
- 対応するソースコード

```
RelationType Relation の Type = [Relation].getType();
```

Relation の関係元を取得する

- 引数
  Relation　対象となる Relation
- 対応するソースコード

```
Agent 関係元の Agent = [Relation].getSource();
```

Relation の関係先を取得する

- 引数
  Relation　対象となる Relation
- 対応するソースコード

```
Agent 関係先の Agent = [Relation].getTarget();
```

5.14 「World の取得/操作」カテゴリ

World が持つ Clock を取得する

- 対応するソースコード

```
Clock 取得した Clock = getWorld().getClock();
```

World が持つ Space を取得する

- 対応するソースコード

```
Space 取得した Space = getWorld().getSpace();
```

World に存在する全ての Agent を取得する

- 対応するソースコード

```
Collection 取得した Agent の集合 = getWorld().getAllAgents();
```

World に定義された AgentType を取得する

- 引数

  String　取得する AgentType の名前
- 対応するソースコード

```
AgentType 取得した AgentType = getWorld().getAgentType([String]);
```

World に定義された BehaviorType を取得する

- 引数

  String　取得する BehaviorType の名前
- 対応するソースコード

```
BehaviorType 取得した BehaviorType = getWorld().getBehaviorType([String]);
```

World に定義された GoodsType を取得する

- 引数
  String　取得する GoodsType の名前
- 対応するソースコード

```
GoodsType 取得した GoodsType = getWorld().getGoodsType([String]);
```

World に定義された InformationType を取得する

- 引数
  String　取得する InformationType の名前
- 対応するソースコード

```
InformationType 取得した InformationType = getWorld().getInformationType([String]);
```

World に定義された RelationType を取得する

- 引数
  String　取得する RelationType の名前
- 対応するソースコード

```
RelationType 取得した RelationType = getWorld().getRelationType([String]);
```

World の説明を取得する

- 対応するソースコード

```
String 取得した World の説明 = getWorld().getDescription();
```

World の名前を取得する

- 対応するソースコード

```
String 取得した世界の名前 = getWorld().getName();
```

World を取得する

- 対応するソースコード

```
World 取得した World = getWorld();
```

現在のステップ数を取得する

- 対応するソースコード

```
StepClock 時計 = (StepClock)getWorld().getClock();
long 現在のステップ数 = clock.getStep();
```

指定した AgentType に対応する TimeEvent の優先度を取得する

- 引数

  AgentType　優先度を取得する AgentType

- 対応するソースコード

```
int 優先度の数値 = getWorld().getPriority([AgentType]);
```

世界に存在する Agent の中から指定した Type の Agent を一人取得する

- 引数

  AgentType　取得する Agent の Type

- 対応するソースコード

```
Agent 取得した Agent = getWorld().getAgent([AgentType]);
```

世界に存在する指定した Type の Agent を親 Type を指定して全て取得する

- 引数

  AgentType　取得する Agent の親 Type

- 対応するソースコード

```
Collection 取得した Agent の集合 = getWorld().getAgentsRecursively([AgentType]);
```

世界に存在する指定した Type の Agent を全て取得する

- 引数

  AgentType　取得する Agent の Type

- 対応するソースコード

```
Collection 取得した Agent の集合 = getWorld().getAgents([AgentType]);
```

世界に定義されている整数型 (int) のパラメータの値を取得する

- 引数

  String　パラメータの名前

- 対応するソースコード

```
//値を取り出す準備
World にアクセスするためのオブジェクト
  = (ObjectObject)WrapperObject.getInstance(getWorld());

//値を取り出す
FieldObject 値を表すオブジェクト = object.getValue((Integer)object.getField([String]));
int 取り出した整数（int）の値 = value.intValue();
```

世界に定義されている少数型 (double) のパラメータの値を取得する

- 引数

  String　パラメータの名前

- 対応するソースコード

```
//値を取り出す準備
ObjectObject world にアクセスするためのオブジェクト
  = (ObjectObject) WrapperObject.getInstance(this.getWorld());

//値を取り出す
Double 値を表すオブジェクト = (Double) world にアクセスするためのオブジェクト
.getValue(world にアクセスするためのオブジェクト .getField([String]));
double 取り出した整数_double_の値 = 値を表すオブジェクト .doubleValue();
```

乱数ジェネレータを取得する

- 対応するソースコード

```
RandomNumberGenerator 乱数ジェネレータを取得する = getWorld().getRandomNumberGenerator();
```

乱数ジェネレータを名前を指定して取得する

- 引数

  String　乱数ジェネレータの名前

- 対応するソースコード

```
RandomNumberGenerator 乱数ジェネレータを取得する
  = getWorld().getRandomNumberGenerator([String]);
```

5.15 「計算」カテゴリ

0 以上 1 未満の実数の乱数を生成する

- 対応するソースコード

```
double 生成した乱数 = getWorld().getRandomNumberGenerator().generate();
```

実数配列の合計と平均を計算する

- 引数

  double[]　合計と平均を計算するための配列

- 対応するソースコード

```
double 合計 = 0;
double[] 実数配列 = [double[]];

//合計を計算する
for(double i = 0; i < 実数配列.length; i++){
合計 = 合計 + 整数配列[i];
}

//平均を計算する
double 平均 = 合計 / 実数配列.length;
```

整数の乱数を生成する

- 引数

  int　乱数の最大値

- 対応するソースコード

```
int 生成した乱数 = getWorld().getRandomNumberGenerator().generate([int]);
```

整数配列の合計と平均を計算する

- 引数

  int[]　合計と平均を計算するための配列

- 対応するソースコード

```
int 合計 = 0;
int[] 整数配列 = [int[]];

//合計を計算する
for(int i = 0; i < 整数配列.length; i++){
合計 = 合計 + 整数配列[i];
}

//平均を計算する
double 平均 = 合計 / 整数配列.length;
```

5.16 「集合操作」カテゴリ

Agent の集合からランダムで N 人の Agent を取り出す

- 引数
  Collection　抽出元の集合
  int　取り出す Agent の数
- 対応するソースコード

```
List 抽出元の Agent のリスト = new ArrayList([Collection]);
int 抽出する要素の数 = [int];
List 選ばれた Agent のリスト = new ArrayList();

//ランダムで要素を選ぶ
for(int i = 0; i < 抽出する要素の数; i++){
  int 生成した乱数
   = getWorld().getRandomNumberGenerator().generate(抽出元の Agent のリスト.size());

  Agent 取り出した Agent = (Agent) 抽出元の Agent のリスト.remove(生成した乱数);
 選ばれた Agent のリスト.add(取り出した Agent);
}
```

Agent の集合からランダムで一人を選ぶ

- 引数
  Collection　抽出元の集合
- 対応するソースコード

```
List Agent のリスト = new ArrayList([Collection]);

//ランダムで一人選ぶ
int インデックス番号
 = getWorld().getRandomNumberGenerator().generate(Agent のリスト.size());
Agent ランダムで選ばれた Agent = (Agent)Agent のリスト.get(インデックス番号);
```

Behavior の集合からランダムで一つを選ぶ

- 引数
  Collection　抽出元の集合
- 対応するソースコード

```
List Behavior のリスト = new ArrayList([Collection]);

//ランダムで一つ選ぶ
int インデックス番号
 = getWorld().getRandomNumberGenerator().generate(Behavior のリスト.size());
Behavior ランダムで選ばれた Behavior = (Behavior)Behavior のリスト.get(インデックス番号);
```

Goods の集合からランダムで一人を選ぶ

- 引数
  Collection 抽出元の集合
- 対応するソースコード

```
List Goods のリスト = new ArrayList([Collection]);

//ランダムで一つ選ぶ
int インデックス番号
  = getWorld().getRandomNumberGenerator().generate(Goods のリスト .size());
Goods ランダムで選ばれた Goods = (Goods)Goods のリスト .get(インデックス番号);
```

Information の集合からランダムで一人を選ぶ

- 引数
  Collection 抽出元の集合
- 対応するソースコード

```
List Information のリスト = new ArrayList([Collection]);

//ランダムで一つ選ぶ
int インデックス番号
  = getWorld().getRandomNumberGenerator().generate(Information のリスト .size());
Information ランダムで選ばれた Information
  = (Information)Information のリスト .get(インデックス番号);
```

Relation の集合からランダムで N 個の要素を取り出す

- 引数
  Collection 抽出元の集合
  int 取り出す Relation の数
- 対応するソースコード

```
List 抽出元の Relation のリスト = new ArrayList([Collection]);
int 抽出する要素の数 = [int];
List 選ばれた Relation のリスト = new ArrayList();

//ランダムで要素を選ぶ
for(int i = 0; i < 抽出する要素の数; i++){
  int 生成した乱数
    = getWorld().getRandomNumberGenerator().generate(抽出元の Relation のリスト .size());

  Relation 取り出した Relation
    = (Relation) 抽出元の Relation のリスト .remove(生成した乱数);
  選ばれた Relation のリスト .add(取り出した Relation);
}
```

Relation の集合からランダムで一人を選ぶ

- 引数
  Collection　抽出元の集合
- 対応するソースコード

```
List Relation のリスト = new ArrayList([Collection]);

//ランダムで一つ選ぶ
int インデックス番号
  = getWorld().getRandomNumberGenerator().generate(Relation のリスト.size());
Relation ランダムで選ばれた Relation
  = (Relation)Relation のリスト.get(インデックス番号);
```

マップに要素を追加する

- 引数
  Map　対象となるマップ
  Object　キーとなるオブジェクト
  Object　追加する要素
- 対応するソースコード

```
[Map].put([Object],[Object]);
```

マップのキーを取得する

- 引数
  Map　対象となるマップ
- 対応するソースコード

```
Set キーのセット = [Map].keySet();
```

マップの要素を取得する

- 引数
  Map　対象となるマップ
  Object　要素を取得するキー
- 対応するソースコード

```
Object 取得した要素 = [Map].get([Object]);
```

マップの要素を集合として取得する

- 引数
  Map　対象となるマップ
- 対応するソースコード

```
Collection マップの要素の集合 = [Map].values();
```

リストの N 番目を削除する

- 引数
  List 対象となるリスト
  int 何番目を削除するか
- 対応するソースコード

```
Object 削除した要素 = [List].remove([int]);
```

リストの N 番目を取得する

- 引数
  List 対象となるリスト
  int 何番目を取得するか
- 対応するソースコード

```
Object 取得した要素 = [List].get([int]);
```

集合からリストを作る

- 引数
  Collection 対象となる集合
- 対応するソースコード

```
List 新しいリスト = new ArrayList([Collection]);
```

集合から要素を削除する

- 引数
  Collection 対象となる集合
  Object 削除する要素
- 対応するソースコード

```
[Collection].remove([Object]);
```

集合に要素を追加する

- 引数
  Collection 対象となる集合
  Object 追加する要素
- 対応するソースコード

```
[Collection].add([Object]);
```

集合の要素数を取得する

- 引数

  Collection　対象となる集合

- 対応するソースコード

```
int 要素数 = [Collection].size();
```

5.17 「出力」カテゴリ

デバッグとしてログを出力する

- 引数
  Logger ログの出力に使う Logger
  String ログの内容
- 対応するソースコード

```
[Logger].debug([String]);
```

警告としてログを出力する

- 引数
  Logger ログの出力に使う Logger
  String ログの内容
- 対応するソースコード

```
[Logger].warn([String]);
```

情報としてログを出力する

- 引数
  Logger ログの出力に使う Logger
  String ログの内容
- 対応するソースコード

```
[Logger].info([String]);
```

標準出力に出力する

- 引数
  String 出力する内容
- 対応するソースコード

```
System.out.println([String]);
```

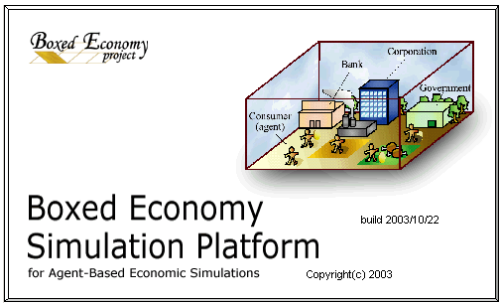
PlatBox
http://www.platbox.org/

PlatBox Project では、シミュレーション・プラットフォーム「PlatBox Simulator」、およびシミュレーション作成支援ツール「Component Builder」を開発・提供しています。 PlatBox Simulator は、シミュレーションを「動かす」ことによって、社会を理解するための道具です。これに加えて、私たちは、シミュレーションを「つくる」過程も社会を理解する助けになると考えています。「Component Builder」は、そのような理解を助けるモデル作成支援ツールです。この「動かすことでわかる」と「つくることでわかる」という二つのアプローチによって複雑な社会を理解する――――そのための「新しい思考の道具」をつくることが、私たちの目指すゴールです。

PlatBox Simulator は、Boxed Economy Project によって開発されてきた「Boxed Economy Simulation Platform」(BESP) の発展版です。Boxed Economy Project は、1999 年から、社会・経済のモデル化を支援する道具立てについて考えてきました。その後、私たちが提案するモデル化の枠組みは、経済分野に限定されたものではないと考え、2005 年にプロジェクト名を「PlatBox Project」に、ソフトウェアの名前を「PlatBox Simulator」に変更しました。

【Boxed Economy Project 時代の主なメンバー】

- 青山 希 （2002 年～現在）
- 浅加 浩太郎（2001 年）
- 井庭 崇（1999 年～現在）※プロジェクトリーダー
- 海保 研（2000 年～2002 年）
- 上橋 賢一（2000 年～2002 年）
- 北野 里美（2001 年～2002 年）
- 高部 陽平 （1999 年～2001 年）
- 武田 林太郎 （2003 年～2005 年）
- 田中 潤一郎 （2000 年～2002 年）
- 中鉢 欣秀（2000 年～2001 年）
- 津屋 隆之介（2000 年～2005 年）
- 広兼 賢治（1999 年～2001 年）
- 松澤 芳昭 （2000 年～2004 年）
- 森久保 晴美（2001 年～2002 年）
- 山田 悠 （2001 年～2005 年）



「モデル作成リファレンスガイド」
執筆・編集者

PlatBox Project (編著)

青山 希
鈴木 祐太
井庭 崇

Designers' Guide to Social Simulation, No.3
モデル作成リファレンスガイド

2005 年 4 月 1 日　初版発行
2005 年 8 月 27 日　第 2 版発行

編著: PlatBox Project
〒252-8520 神奈川県藤沢市遠藤 5322 慶應義塾大学 井庭崇研究室
E-Mail: platbox@sfc.keio.ac.jp, もしくは iba@sfc.keio.ac.jp
Web: http://www.platbox.org/


PlatBox
http://www.platbox.org/