第3章
モデリングとシステム仕様

"始まりは仕事の最も重要な部分である"-プラトン

システム仕様書」という用語は、ソフトウェアシステムの特性を導き出すプロセスと、その特性を記述した*文書の両方に使用される。*システムが開発されるにつれて、その特性はライフサイクルのさまざまな段階で変化するため、どの仕様を指しているのかが不明確になることがある。曖昧さを避けるため、共通の意味を採用する：システム仕様とは、システムが顧客に引き渡される時点で、システムに関して何が有効（真）であるべきかについて述べたものである。システムを仕様化するということは、達成したい*ことを*述べるということであり、*どのように*達成する予定なのか、あるいは中間段階で何が達成されたのかを述べる*ことではない。*本章の焦点は、システムの*機能を*記述することであり、*その形態ではない。*第5章では、どのようにシステムを構築するかという形態に焦点を当てる。

開発中のシステムを特定するには、以下のようないくつかの側面がある：

- 問題を理解し、何を規定する必要があるかを決定する。
- 仕様に使用する表記法の選択
- 仕様が要件を満たしているかどうかの検証

もちろん、これは直線的な一連の活動ではない。むしろ、問題に対する理解が深まるに

内容

つれて、別の表記法に切り替えたいと思うかもしれない。また、検証活動によって問題理解の弱点が明らかになり、手元の問題をさらに研究するきっかけになるかもしれない。

私たちはすでに、仕様のための一般的な記法、つまりUML標準に出会っています。私たちはUMLを使い続け、それについて、また他の記法についても学んでいきます。ほとんどの開発者は、自明でないシステムを仕様化するためには、単一のタイプのシステムモデルだけでは不十分であることに同意します。通常、さまざまな利害関係者のために、さまざまな「言語」で語られた、複数の異なるモデルが必要になります。エンドユーザーは、そのシステムに対して、自分の仕事がしやすくなるというような特定の要求を持っている。ビジネス・マネジャーは、ポリシーにもっと関心があるかもしれない、

ルール、そしてシステムがサポートするプロセス。エンジニアリング設計の詳細を気にする利害関係者もいれば、気にしない利害関係者もいる。したがって、異なる表記法を用いて、いくつかの異なる角度から見たシステム仕様を作成することが望ましい。

ここで私が最も懸念しているのは、開発者の視点である。私たちは、静止／平衡状態と予想される摂動を特定する必要がある。システムは平衡状態でどのように見えるのか? 摂動にどのように反応し、どのような一連のステップを経て新しい平衡状態に到達するのか? ユースケースがこのような問題を、非公式ではあるがある程度扱っていることはすでに見た。ここでは、より正確なアプローチについてレビューする。これは必ずしも形式的な方法を意味するものではない。特定のタイプの問題に適した表記法もある。私たちのゴールは、ある種の分析が可能なある程度の精度で作業することである。

システム仕様は要求事項から導き出されるべきである。仕様は、要求を満たすために必要なシステム動作を正確に記述しなければならない。ほとんどの開発者は、ソフトウェア・タスクの最も難しい部分は、完全で一貫性のある仕様に到達することであると主張するだろう。そして、プログラム構築の本質の多くは、実際にその仕様をデバッグすることである。開発者は顧客のニーズを誤解しているかもしれない。顧客は確信が持てないかもしれないし、最初の要件はあいまいだったり不完全だったりすることが多い。何度も強調しておくが、要件を書き、仕様を導き出すのは、厳密に連続したプロセスではない。むしろ、満足のいく解決策 が 見つかるまで、要求とシステム仕様を繰り返し検討しなければならない。その場合でも、設計や実装の過程で疑問が生じれば、両方を再検討し、再検討する必要があるかもしれない。

システム要件は最終的には顧客が決めるものだが、開発者は正しい質問の仕方や、顧客から集めた情報をどのように体系化するかを知っておく必要がある。しかし、どのような質問をすればよいのだろうか? 有用なアプローチは、現実世界のあらゆる問題で発生しがちな*単純な代表的問題の*カタログから始めることであろう。これらの基本的な構成要素は、"問題フレーム "と呼ばれる。それぞれが明確に定義されたフォーマットで記述でき、それぞれがよく知られた解決策を持ち、それぞれがよく知られた関連する問題の集合を持っている。我々はセクション2.3.1ですでに最初のステップを踏んだ。セクション3.3では、問題フレームを適用することで、複雑な問題がどのように管理しやすくなるかを見ていく。このように、問題フレームはシステム要求とシステム仕様の間のギャッ

プを埋めるのに役立つ。

3.1システムとは何か?

"すべてのモデルは間違っているが、いくつかは役に立つ"-ジョージ・E・P・ボックス

「あるものにとって絶対不可欠な性質はない。同じ性質であっても、あるときには事物の本質として現れるが、別のときにはきわめて本質的でない特徴になる。"-ウィリアム・ジェームズ

第2章では、ソフトウェア・エンジニア（またはエンジニアのチーム）が開発しようとするソフ トウェア製品として、*system-to-be*、より正確には*software-to-be*を紹介した。システムとは別に、それ以外の世界（「環境」）については、以下と相互作用する限りにおいてのみ関心が持たれてきた。

システムをアクターの集合として抽象化した。さまざまな相互作用のシナリオをユースケースの集合として記述することで、インクリメンタルな方法でソフトウェアシステムを開発することができた。

しかし、このアプローチにはいくつかの限界がある。第一に、システムが直接相互作用する「アクター」のみを考慮することで、ソフトウェアと直接相互作用しないが、問題とその解決にとって重要な環境の一部を除外してしまう可能性がある。例えば、株式市場のファンタジーリーグシステムと、それが動作するコンテクストを考えてみましょう（図1-32）。ここで、現実の株式市場取引所はソフトウェアと相互作用しないので、"アクター "とはみなされない。考えられることは、どのユースケースでも言及されないということです！プロジェクト全体が証券取引所について展開されているにもかかわらず、証券取引所はシステムの説明にまったく登場しない。

第二に、相互作用シナリオに焦点を当てることから始めることは、問題を記述する上で最も簡単なルートではないかもしれません。ユースケースは、ユーザー（アクター）とシステムとのインタラクションのシーケンスを記述します。ユースケースがオブジェクト指向ではなく手続き指向であることはすでに述べた。逐次的な手順に焦点を当てることは、そもそも難しくないかもしれませんが、"主な成功シナリオ "から分岐していないか常に監視する必要があります。意思決定（分岐点）を検出するのは難しいかもしれない。何がうまくいかないかを考えるのは難しいかもしれない-特に、問題構造の有用な表現に導かれていない場合は。

概念モデリングを開始する最良の方法は、ユーザーや顧客が自分たちの世界をどのように概念化することを好むかから始めることかもしれない。なぜなら、開発者は、問題を定義する時点で、顧客と多くの対話をする必要があるからです。

本章では、問題記述（すなわち要求と仕様）に対する代替アプローチをいくつか紹介する。これらは、より複雑かもしれないが、大規模で複雑な問題を解決するための容易なルートを提供すると考えられている。

3.1.1　世界の現象とその抽象化

問題を解決する鍵は、問題を理解することにある。問題は現実の世界にあるのだから、世界の現象をうまく抽象化する必要がある。優れた抽象化は、私たちが世界について収

集した知識（つまり、目下の問題に関連する「アプリケーション・ドメイン」）を正確に表現するのに役立ちます。オブジェクト指向のアプローチでは、重要な抽象化はオブジェクトとメッセージであり、第2章では問題を理解し、解決策を導き出すのに大いに役立った。私たちは今、これらを放棄するつもりはありません。むしろ、視野を広げ、おそらくは少し違った視点を持つことになるでしょう。

通常、私たちは世界をさまざまな部分（または領域、ドメイン）に分割し、さまざまな現象について考える。*現象とは*、世界あるいはその一部を観察したときに、存在する、あるいは存在すると思われる、あるいは認識される事実、あるいは物体、あるいは出来事である。私たちは世界現象をさまざまな基準で区別することができる。構造的には、個体と個体間の*関係という*2つの大まかな現象カテゴリーがある。論理的には、*因果的な*現象と*象徴的な*現象を区別することができる。行動面では、*決定論的な*現象と*確率論的な*現象を区別することができる。次に、それぞれの種類について簡単に説明する。

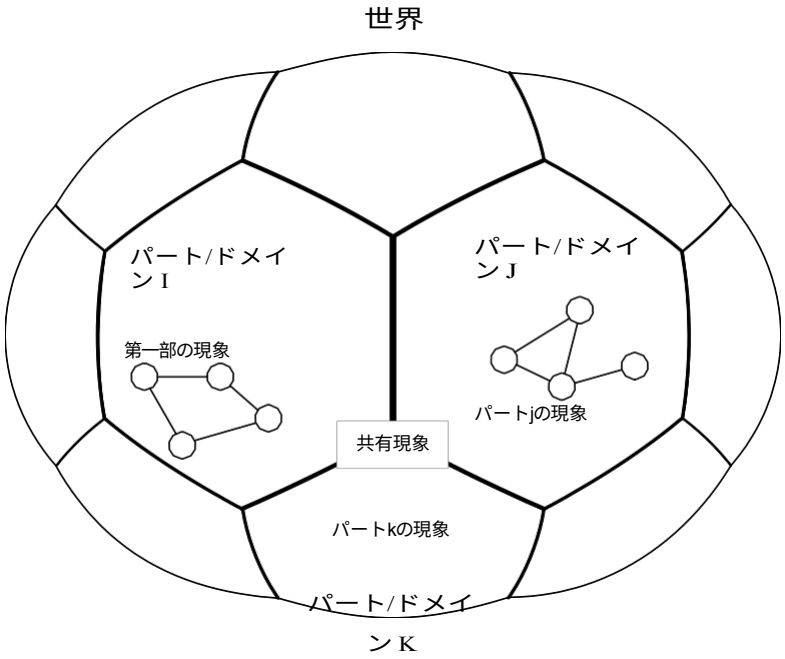


図3-1：ドメインとその現象に分割された世界。

強調しておきたいのは、これはソフトウェア工学に適していると思われる分類のひとつにすぎないということだ。さらに、世界現象の具体的な同定は、それを導き出すためにどれだけの労力を費やしたとしても、時間の経過とともに儚くなり、不具合を生じるに違いない。1.1.1節で熱力学第二法則の影響について述べた。世界の現象を特定する際、私たちは必然的に近似を行う。ある種の情報は重要視され、それ以外の情報は重要でないものとして扱われ、無視される。自然界や社会にはランダムな変動があるため、重要な情報と重要でない情報を分ける根拠となった現象の一部が混ざり合い、当初のモデルが無効になってしまう。それゆえ、我々のモデリング努力が達成できることの究極的な限界がある。

個人

個体とは、名前が付けられ、他の個体と確実に区別できるものである。ある現象を個体として扱うかどうかの判断は客観的なものではない。どの抽象度を選択するかは、観察者にとって相対的なものであることはもう明らかだろう。私たちは、問題の解決に役立ち、現実的に区別可能な個体だけを認識することにしている。イベント、エンティティ、バリューの3種類の個体を区別することにする。

◆ イベントとは、特定の時点で発生する個々の出来事のことである。各イベントは

つまり、出来事そのものは内部構造を持たず、発生するまでの時間もかからない。起こる。したがって、"イベントの前 "と "イベントの後 "については語ることができるが、"イベントの最中 "については語ることができない。イベントの例は、取引注文の発注であり、別の例は株式取引の実行であり、さらに別の例は株価の提示である。イベントの詳細については、セクション3.1.3を参照のこと。

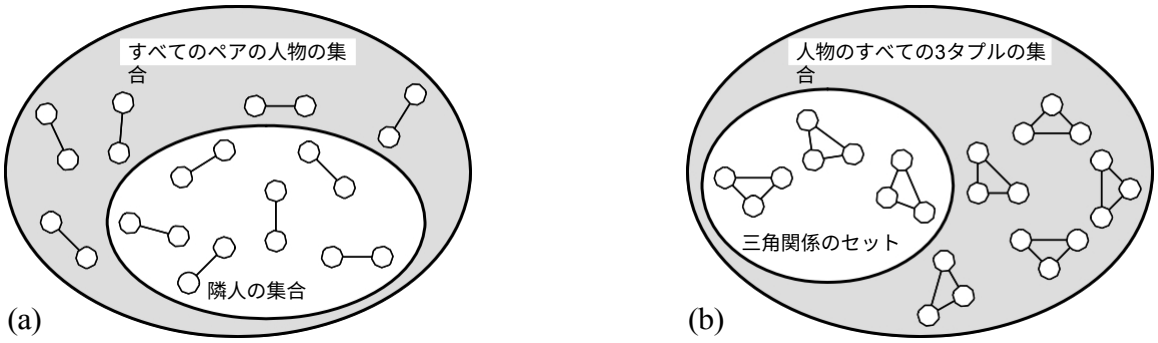


図3-2：関係の例：図3-2：関係例：*Neighbors*(Person_i, Person_j)と *InLoveTriangle*(Person_i, Person_j, Person_k)。

◆ **エンティティは、**質や関係とは対照的に、明確な存在を持つ個体である。エンティティは時間の経過とともに持続し、ある時点から別の時点へとその特性や状態を変化させることができる。

ある実体は出来事を起こすかもしれないし、ある実体は自らの状態に自発的な変化をもたらすかもしれないし、ある実体は受動的かもしれない。

第2章でモデル化したソフトウェア・オブジェクトや抽象概念はエンティティである。しかし、エンティティには現実世界のオブジェクトも含まれる。エンティティは、世界のどの部分をモデル化するかによって決まる。投資アシスタントの事例（セクション1.3.2）における*金融トレーダー*はエンティティであり、彼の*投資ポートフォリオ*もエンティティで*ある*。これらはそれぞれ、*トレーダー*、*ポートフォリオ*、*株式*というエンティティ・クラスに属します。

◆ **価値とは、**時間と空間の外に存在し、変化することのない無形の個体である。私たちが興味を持っている価値は、数字や文字などである。

記号である。例えば、値とは、ある量の数値的な尺度であったり、7キログラムのような従来の尺度での量を示す数値であったりする。

我々のケーススタディ（セクション1.3.2）では、特定の*株価は*、株式が値付けされる貨幣単位の数であり、したがって値である。値クラスの例としては、*整数*、*文字*、*文字列などがあります*。

関係

「私は関係を視覚化する能力が低いことで悪名高い。
そこから派生する主題は私には不可能である"-ジークムント・フロイト

個体がある特徴を共有している場合、その個体は*関係にあると言う。*関係を定義するには、一度に何人の個人を考慮するかも指定する必要がある。例えば、どのような2人組であっても、お互いの家が100メートル以内離れていれば、隣人であると判断することができる。Person_iとPerson_jの2人がこのテストに合格した場合、その関係は成立する（真である）。このテストに合格した人物の組はすべて、*Neighbors*(Person_i, Person_j)という関係にあるという。図3-2(a)に示すように、隣人である人物の組はすべての人物の組の部分集合を形成する。

関係は個体のペアだけで成立する必要はない。任意の数の個体を考慮し、それらが関係にあるかどうかを決定することができる。考慮する個体の数*n*は、*n*≧2の任意の正の整数であり、関係のテストごとに固定されなければならない。
タプルとする。ここでは関係を*RelationName*(*個人*$_1$, ..., *個人*$_n$)と表記する。のいずれかが

個体がその関係のすべてのテストにおいて一定である場合、その名前を関係の名前に含めることができる。例えば、メガネをかけているという特徴を考えてみよう。そして、Person_iが関係*Wearing*(Person_i, Glasses)に含まれるかどうかをテストすることができる。Glassesはすべてのテストにわたって一定なので、*WearingGlasses*(Person_i)、または単に*Bespectacled*(Person_i)と書くことができる。次に、いわゆる「三角関係」を$n = 3$.明らかに、この特性をテストするためには、一度にちょうど3人の人物を考慮しなければならない。そうすると、*InLoveTriangle*(Person_i, Person_j, Person_k)という関係式は、図3-2(b)に示すように、この特性が真である人物のすべての3つ組（3タプル）の集合を形成することになる。関係の正式な定義は、いくつかの表記法を示した後、3.2.1節で述べる。

ここでは、状態、真実、役割という3種類の関係を考える。

◆ **状態とは、**個々のエンティティや値の間の関係であり、時間とともに変化する可能性がある。状態についてはセクション3.1.2で説明し、今は省略する。

◆ **真理とは、**個人間の固定された関係であり、時間とともに変化することはありえない。時間とともに変化する状態とは異なり、真理は不変のままである。もう少し緩やかな定義は

は、関心のある時間スケールで不変の関係を考える。時間スケールの例としては、プロジェクト期間や予想される製品ライフサイクルがある。真理を述べるとき、個体は常に値であり、真理は*GreaterThan*(5, 3)や*StockTickerSymbol*("Google, Inc.", "GOOG")のように不変の事実を表現する。企業の株式シンボルは変わらないと考えても、それなりに安全である（ただし、合併や買収がこれに影響する可能性はある！）。

◆ **役割とは、**ある出来事と、それに特定の方法で参加する個人との関係である。各役割は、"引数"（または"引数"）の1つとして考えられるものを表現している。

"パラメータ")のイベントである。

因果的現象と象徴的現象

◆ **因果**現象とは、実体に関する出来事、役割、状態のことである。これらは、何らかの実体によって直接生み出され、あるいは支配されるため、因果的現象である。

他の現象が順番に起こる。

◆ **象徴的**現象とは、価値、そして価値だけに関係する真理や状態のことである。それらは、他の現象やそれらの間の関係を象徴するために使用されるため、象徴的と呼ばれる

例えば、ディスク・レコードのデータ内容など、価値を関係づける象徴的な状態は、外的な因果関係によって変化させることができるが、それ自体を変化させることも、他の場所に変化を引き起こすこともできないため、因果関係があるとは考えない。

決定論的現象と確率論的現象

◆ **決定論的**現象とは、発生の有無が確実に確定できる因果現象のことである。

◆ **確率的**現象とは、確率のランダムな分布に支配される因果現象のことである。

3.1.2 州と州の変数

状態とは、ある特定の時点において、世界で何が真実であるかを表すものである。個体の状態は、その個体の行動の累積結果を表す。デジタル・ビデオ・ディスク（DVD）プレーヤーのような装置を考えてみよう。入力コマンドに対してデバイスがどのように反応するかは、その入力だけでなく、デバイスが現在置かれている内部状態にも依存する。つまり、DVDプレーヤーの「PLAY」ボタンが押された場合、次に何が起こるかは、プレーヤーの電源が入っているかどうか、ディスクが入っているかどうか、すでに再生中かどうかなど、さまざまなことに左右される。これらの状態は、DVDプレーヤーのさまざまな状態を表しています。

このような選択肢を考えることで、DVDプレーヤーの全状態のリストが思いつくかもしれない：

状態1: *NotPowered* (プレイヤーはパワーアップしていない) 状態2: *Powered* (プレイヤーはパワーアップしている)
状態3: *Loaded* (ディスクがトレイに入っている) 状態4: *Playing(再生中)*

状態をより正確に定義すると、オブジェクトの集合に対する*関係*であり、単に集合の部分集合を選択するだけである。DVDプレーヤーの例では、表現したいのは "DVDプレーヤーの電源が切れている "ということである。*Is*(DVDplayer,NotPowered) または *IsNotPowered*(DVDplayer)と書くことができます。ここでは、この形式に落ち着くことにする：*NotPowered*(DVDplayer)。*NotPowered*(*x*)は、電源が入っていないDVDプレーヤー*x*の部分集合です。言い換えれば、*NotPowered*(*x*)は、*x*が現在電源オフの場合に真となる。そのようなプレーヤーの1つがリビングルームにあるもので、DVDinLivRmとラベル付けされていると仮定すると、*NotPowered*(DVDinLivRm)は、リビングルームにあるプレーヤーの電源が入っていない場合に真になります。

よくよく考えてみると、上記の状態は、電源が入っていないプレーヤーがディスクをトレイに入れていないことを意味している。もしDVDプレーヤー用のソフトウエアを開発することになったら、このことを明確にしなければならない。これは、電源オフボタンを押すとディスクが自動的に排出されるという意味ですか？もしそうでないか、あるい

はこの問題がまだ解決されていないのであれば、DVDプレーヤーの状態のリストを次のように作り直した方がいいかもしれません：

状態1：*NotPoweredEmpty*（プレーヤーの電源は入っておらず、ディスクも入っていない) 状態2：*NotPoweredLoaded*(プレーヤーの電源は入っていないが、トレイにディスクが入っている) 状態3：*PoweredEmpty*（プレーヤーの電源は入っているが、ディスクが入っていない) 状態4：*PoweredLoaded*（プレーヤーの電源は入っており、トレイにディスクが入っている) 状態5：*Playing(再生中)*

この時点で、総体的あるいは「グローバル」なシステム状態の代わりに、DVDプレーヤーのさまざまな部分（サブオブジェクト）を識別し、順番に各部分の状態を検討する方がよりエレガントであることに気づくかもしれない（図3-3）。 各パーツは次の表のように「ローカル」な状態を持つ。

システム部分（オブジェクト）	**状態関係**
電源ボタン	{オフ、オン}
ディスクトレイ	{空、ロード済み}。
再生ボタン	{オフ、オン}
...	...

関係*Off*(*b*)はボタンの集合上で定義されることに注意。すると、これらの関係は真になる可能性がある：

Off(PowerButton)と*Off*(PlayButton)。*On*(*b*)についても同様である。

個々の部品の状態が与えられたとき、システム全体の状態をどのように定義できるだろうか? 明らかに、システム全体の状態は、各部分の状態によって定義されると言える。例えば

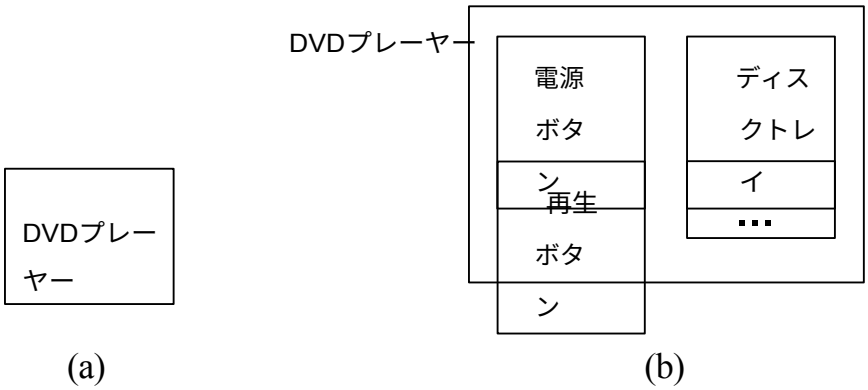


図3-3：異なる詳細レベルにおけるDVDプレーヤーの抽象化。(b) 複数のエンティティから構成されるプレーヤー。

DVDプレーヤーの状態は、{ *On*(PowerButton), *Empty*(), *Off*(PlayButton), ... }である。*Empty*()という関係が引数なしで残されていることに注意してください。この場合、括弧なしで*Empty*と書くこともできる。各関係がどの部分を参照しているかが明確であれば、この「状態タプル」における関係の配置は重要ではない。

ここで、部品の状態の組み合わせはすべて許されるのかという疑問が生じる。これらの部分は互いに独立しているのか、それともある部分の状態には、他の部分の現在の状態によって課される制約があるのか。複合領域の各部分の状態には、*相互に排他的な*ものがあるかもしれない。先ほどの問題に戻りますが、電源ボタンが「オフ」状態のときに、ディスクトレイが「ロード」状態になることがありますか? これらは同じシステムの一部分なので、*各部分の状態の相互依存性を*明示しなければならない。有効なシステム状態のタプルのリストが、構築可能なすべてのタプルを含んでいないことになるかもしれません。

システム・ステートの表現（単一の集約ステート vs. パーツのステートのタプル）はどちらも正しいが、どちらが適しているかは、システムについてどのような詳細を知りたいかによる。一般的に、システムを、ステート・タプルを定義するパーツの集合として考えることは、単一の集約ステートよりもすっきりした、よりモジュール化されたアプローチを示す。

ソフトウェア工学では、ソフトウェアシステムの目に見える側面を気にする。一般に、目に見える側面は、必ずしもシステムの「部分」に対応する必要はない。むしろ、システムの観察可能なあらゆる品質です。例えば、セクション2.5で特定されたドメインモデ

ル属性は、システムの観察可能な品質を表す。各観測可能な品質を**状態変数と呼ぶ**。最初の事例では、変数にはロックと電球が含まれます。もう一つの変数は、鍵を開けようとした回数のカウンターである。さらに別の変数は、オートロック機能をサポートするために、ロックが開いてからの経過時間をカウントダウンするタイマーの量である。我々のシステムの状態変数は、次の表 の ように要約できる。

	VariableState関係
ドアロック	{*非武装*、武装}
電球	{点灯、非点灯}
失敗した試行のカウンタ	{0 , 1, ..., maxNumOfAttempts}。
オートロックタイマー	{0 , 1, ..., autoLockInterval}.

複数のロックや電球がある場合は、上記のDVDプレーヤーのボタンの例と同様に、ロックや電球ごとに異なる状態変数を持つ。つまり、裏庭と玄関のドアのロックの状態関係は、*Disarmed*(Backyard)と*Disarmed*(Front)と定義できる。

数値関係の場合は少し厄介だ。カウンタが現在 "2 "の状態にあるという意味で2(Counter)と書くこともできるが、これは少し厄介である。そうではなく、便宜上*Equals*(Counter, 2)と書き、同様に*Equals*(Timer, 3)と書くことにする。

システム状態は、状態関係の有効な組み合わせを含む状態変数のタプルと定義される。状態とは、*システムの外から見たときに*気になるシステム特性の集約表現である。上記の例では、ステート・タプルの例は以下のとおりです:

{Disarmed(Front), *Lit*, *Armed*(Backyard), *Equals*(Counter, 0), *Equals*(Timer, 0) }.状

態を分類する1つの方法は、オブジェクトがある状態で何をしているかによっ

て分類することである:

- オブジェクトがただイベントの発生を待っている場合、状態は*受動的な性質を*持つ。DVDの場合前述したように、このような状態は "Powered "と "Loaded "である。
- 状態は、オブジェクトがアクティビティを実行している場合、*アクティブな品質である。*DVDプレーヤーが「再生」状態にあるとき、それはディスクをアクティブに再生している。

これらのオプションの組み合わせも可能である。つまり、オブジェクトはアクティビティを実行しながら、イベントも待つことができる。

状態間の移動は*遷移*と呼ばれ、多くの場合、イベント（セクション3.1.3で説明）によって引き起こされる。各状態遷移は2つの状態を接続する。通常、すべての状態のペアが遷移で結ばれるわけではなく、特定の遷移だけが許される。

例3.1証券取引所の状態を　識別する（最初の試み）

投資支援システムに関する2つ目のケーススタディ（セクション1.3.2）を考えてみよう。取引所の場所、建物の寸法、建設年月日など、取引所について言えることはたくさんある。しかし、私たちの問題に関連して、私たちが知りたい特性は何だろうか? 候補をいくつか挙げてみよう:

- 取引所の営業時間は何時から何時までで、現在は "オープン "か "クローズ "か?
- 現在上場している銘柄は?
- 各上場株式について、気配値（売買・売出・買付）と募集株式数は?
- 現在の全体的な取引量は?
- 現在の市場指数や平均値は?

状態変数は次のように要約できる:

	VariableState関係
動作状態（またはゲート状態）{	開、閉}。
i^{th} 株価	任意の正の実数
i^{th} 公募株数{0	, 1, 2, 3, ...}.
取引量{0	, 1, 2, 3, ...} (0 , 1, 2, 3, ...})
市場指数／平均	任意の正の実数

表中の＊印は、価格がある小数点以下の桁数まで引用されており、価格に妥当な上限があることを示している。言い換えれば、これは有限の値の有限の集合である。

明らかに、このシステムは非常に多くの可能な状態を持つが、それでも有限である。即興的な図式表現を図3-4に示す。(状態図のUML標準記号については、セクション3.2.2で後述します）。

状態タプルの例を以下に示す：{*Open*, *Equals*(Volume, 783014), *Equals*(Average, 1582), *Equals*(Price_1, 74.52), *Equals*(Shares_1, 10721), *Equals*(Price_2, 105.17), *Equals*(Shares_2, 51482), ... }。すべての上場銘柄について、価格と株数を指定しなければならないことに注意。

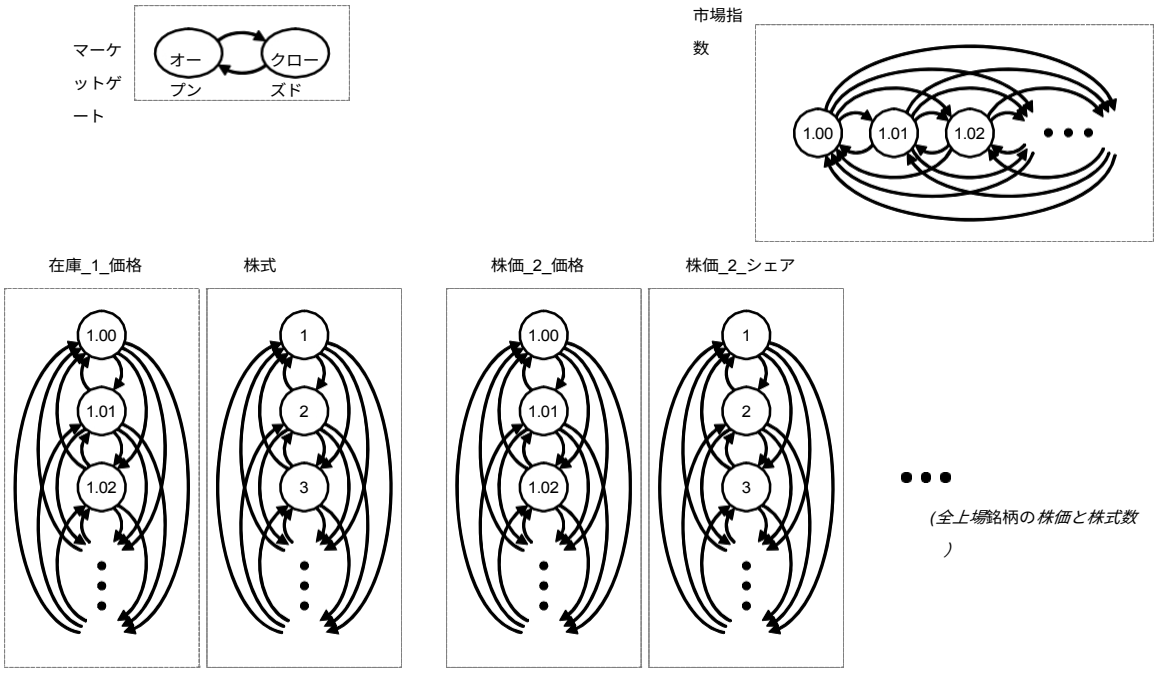
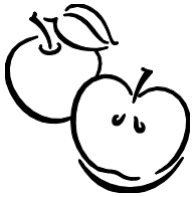


図3-4：例3.1の状態を図式化したもの。矢印は、異なる状態間を遷移するための許容される経路を示している。

もうお分かりのように、状態現象の選択は観察者と観察者の問題によって異なります。市場状態の代替的な特徴付けは、後の例3.2で紹介します。

観測変数と隠れ変数

観測可能な現象から定義される状態



状態とは抽象的なものであり、主観的なものである。客観的な状態」は存在せず、状態の分類はすべて観察者にとって相対的なものである。もちろん、同じ観察者であっても、異なる抽象度を導き出すことができる。
観測者はまた、観測可能な現象に基づいて新しい状態を*定義することも*できる。
が直接観察される。例えば、果物の状態を考えてみよう：「緑」「半熟」「完熟」「過熟」「腐敗」である。果物の "熟度 "の状態は、果皮の色や質感、大きさ、香り、触ったときの柔らかさなど、観察可能なパラメータに基づいて定義される。同様に、エレベーターの「動いている」状態は、その後の時間の経過とともにその位置を観察し、傾向を計算することによって定義される。

先に説明したオートロック・タイマーの場合、「CountingDown」と「Idle」という状態

を次のように定義できる：

CountingDown(Timer) $\triangleq$ τが時間と共に減少するとき、*Equals*(Timer, τ)の関係が成り立つ *Idle*(Timer) $\triangleq$ τが時間と共に一定であるとき、*Equals*(Timer, τ)の関係が成り立つ 記号$\triangleq$は、これが定義された状態であることを意味する。

例3.2証券取引所の状態を　識別する（2回目の試み）

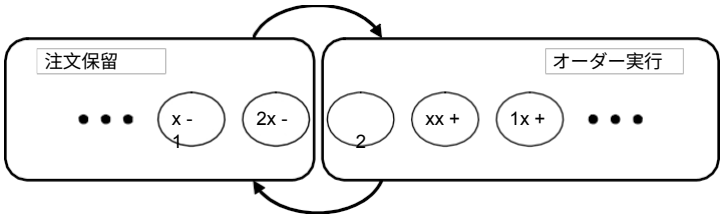


図3-5：例3.2の状態のグラフ表示。図3-4からオファー株数を表すミクロ状態が2つのマクロ状態に集約される。

例3.1をもう一度見てみよう。よく調べてみると、トレーダーはそこで特定された変数をあまり有用だとは思わないかもしれない、という結論に達するかもしれない。セクション1.3.2で、トレーダーが本当に気にしているのは、取引機会が発生したかどうかを知ることと、いったん取引注文を出したら、その注文の状況を追跡することだと推測しました。ここで、売買の判断は株価のトレンド方向に基づいて行われると仮定してみよう。また、Stock_iの株価が上昇トレンドにあるとき、買いの決断が下され、Stock_iの*x*株の成行注文が発注されると仮定する。要約すると、トレーダーは2つの状態を表したい：

- 「株価の取引可能状態（「買い」、「売り」、「ホールド」）は、ある銘柄の直近の株価の時間窓を考慮し、線を補間することに基づいて定義される。その線が上昇トレンドを示した場合

  株の状態は*買い*。売りと*ホールドの*状態も同様に決定される。より財務に精通したトレーダーは、単純な回帰線の代わりに、テクニカル分析指標（例えば、図1-23）のいくつかを使用することができます。

- 「注文の状態」（「保留」対「約定」）は、買い取引を実行するのに十分な株数が提供されているかどうかに基づいて定義されます。ここで注意しなければならないのは

  売り取引は、喜んで買う人がいる場合にのみ実行できる。つまり、買い注文と売り注文は同じ状態を持つが、定義が異なる。

そうすると、トレーダーは市場の状態を次のように定義することができる：

買い $\triangleq$ $t = {}_{tcurrent} - Window, \ldots, {}_{tcurrent} - 2$ の関係 $Equals(\text{Price_i}(t), p)$の回帰線、
${}_{tcurrent} - 1, {}_{tcurrent},$ は正の傾きを持つ。

売り $\triangleq$ $t = {}_{tcurrent} - Window, \ldots, {}_{tcurrent}$の関係$Equals(\text{Price_i}(t), p)$の回帰線は負の傾きを持つ。

保持 $\triangleq$ $t = {}_{tcurrent} - Window, \ldots, {}_{tcurrent}$の関係$Equals(\text{Price_i}(t), p)$の回帰直線の傾きがゼロである。

$SellOrderPending \triangleq Equals(\text{Shares_i}, y)$関係は、*x*より小さい*y*のすべての値に対して真となる。

売り注文の実行 $\triangleq$ 以上のすべての*y*の値に対して、$Equals(\text{Shares_i}, y)$という関係が成り立つ。
x

ここで行ったのは、基本的には、例3.1の多数の詳細な状態を、少数の集約的な状態にグループ化することである（図3-5参照）。これらのグループ化された状態は、トレーダーの作業を

簡素化するのに役立つ。

これらの各状態には、さらなるニュアンスの違いがある。例えば、トレーダーがより大きな損失を回避するために売るべきときと、市場の頂点で利益を得たいときとで、*売り*状態の2つのサブ状態を区別することができる。留意すべき最も重要な点は、トレーダーの目標とそれを達成するための戦略である。これは決して、トレーダーがマーケットを見る唯一の方法ではない。より熟練したトレーダーであれば、ロングとショートの取引ポジションで状態を定義することもできる（セクション1.3.2、図1-22参照）。状態の例としては、次のようなものがある：

GoLong - 指定された銘柄は現在ロングポジションを取るのに適している。

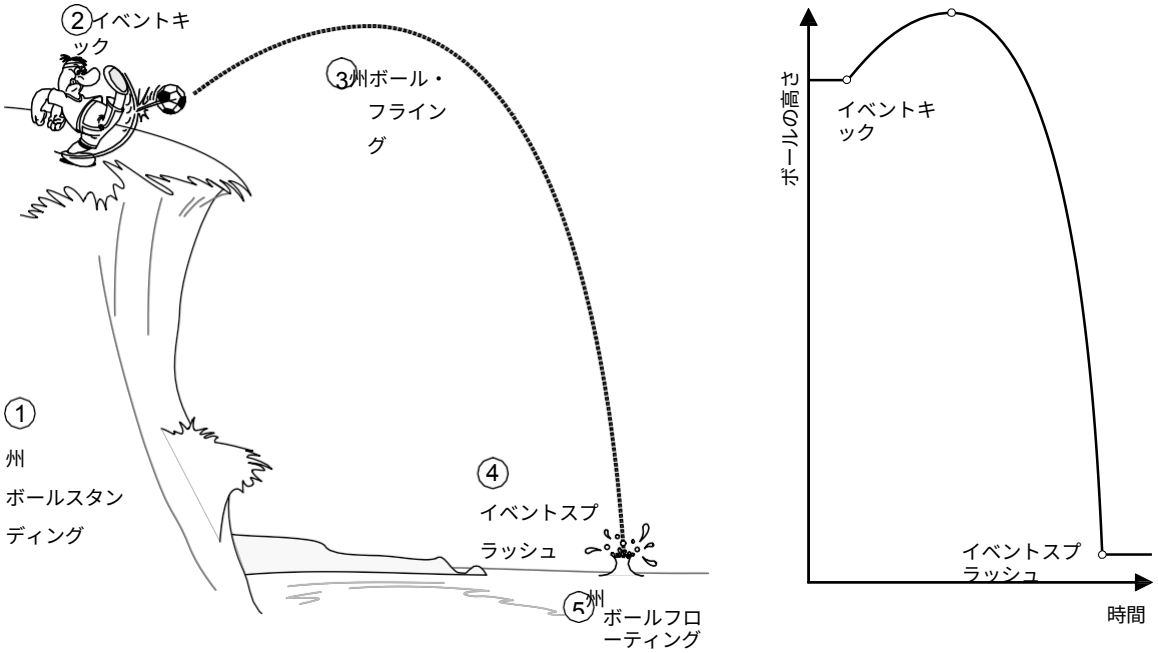


図3-6：状態は遷移する。

GoShort - 指定された銘柄は現在ロングポジションを取るのに適している。

GoNeutral - トレーダーは現時点で指定された銘柄をホールドするか避けるべきである。

所定の詳細レベル（粗視化）で直接観察可能な状態を**ミクロ状態と呼ぶ**。ミクロ状態のグループを**マクロ状態**（またはスーパー状態）と呼ぶ。例3.2で定義した状態はマクロ状態である。

抽象化によって、オブジェクトがあるステートで実行するアクティビティが、同時に（あるいは**同時に**）特定されることがある。例えば、DVDプレーヤーが「再生」状態にあるとき、ディスクの再生（ビデオ出力の生成）とタイムプログレス表示の更新を同時に行うことがある。

セクション3.2.2では、状態および状態間の遷移を表すための標準化されたグラフィカル表記法として、UMLステートマシンダイアグラムについて説明します。

3.1.3　イベント、シグナル、メッセージ

イベントの定義は、イベントが不可分であることを要求する。内部的な時間構造を持つハプニング（またはパフォーマンス、アクション）は、2つ以上のイベントとみなされなければならない。この制限の動機は、中間状態を持たないようにするためである。イ

ベントは、2つの異なる状態の間の境界を明確に表す。また、2つのイベントが同時に発生することはないと仮定する必要がある。すべての事象は連続的に起こり、連続する事象の間には何も起こらない時間間隔がある。イベントとインターバルは交互に起こる。各イベントはあるインターバルを終了し、別のインターバルを開始する。図3-6の例を考えてみよう。時間図を調べることによって、私たちは時間を区間（「状態」）に分割し、どの点（「出来事」）が2つの区間を区切るかを特定する。そして、図3-6に示すように、結果として生じる5つの現象に名前をつける。私たちは

これは単純に間違ったモデルであり、連続する出来事の間隔を特定するために時間スケールを改良する必要がある。

開発者は、何を一つのイベントとして扱うかを選択する必要があるかもしれない。ホームアクセス制御のケーススタディ（セクション1.3.1）を考えてみよう。テナントが識別キーを打ち込んでいるとき、これを1つのイベントとして扱うべきでしょうか、それともキーストロークの1つ1つを別のイベントと考えるべきでしょうか? 答えは、問題文がどちらかの方法で処理することを要求しているかどうかによります。異なるキー操作の間に発生する可能性のある、問題に関連する例外はありますか? もしそうなら、それぞれのキー入力をイベントとして扱う必要があります。

読者は、イベントとメッセージ、あるいはオブジェクト指向アプローチにおける操作の関係について疑問に思うかもしれない。上で定義したイベントの概念は、オブジェクト指向に限定されないため、より一般的なものである。*メッセージの*概念は、あるエンティティから別のエンティティに信号が送られることを意味します。メッセージとは異なり、*イベントは*起こるものであり、1つ以上の個体が含まれるかもしれないが、必ずしも1つの個体から別の個体へ*指示される*わけではない。イベントは、連続する状態間の遷移をマークするだけである。このビューの利点は、問題定義の初期段階で処理の詳細を特定することを避けられる ことである。ユースケース分析（セクション2.4.3）は、システム操作につながる逐次的な処理手順（「シナリオ」）を明示する必要があるという点で異なる。

もう1つの違いは、イベントは常に状態変化を意味することである。システムが同じ状態にとどまっている状況であっても、イベントと状態変化の明示的な記述がある。したがって、イベントは、対応するステートセットがどのように定義されているかに依存する。一方、メッセージは状態変化とは関係ないかもしれない。例えば、オブジェクトの属性値を取得するだけの操作（アクセサー操作と呼ばれる）は、オブジェクトの状態に影響を与えない。

イベント例:

listStock - このイベントは、初めて取引が可能になったことを示す。

splitStock - このイベントは、価格状態間の移行を示す。

submitOrder（注文の送信） - このイベントは、取引注文の状態間の遷移をマークします。また、価格の状態間の遷移もマークします（株価が更新されます）。

matchFound - このイベントは、一致する注文が見つかった場合に、取引注文の状態遷移を示す。

上記のイベントは、"取引量 "や "市場指数/平均 "の変化を示すこともある。読者は、イベント名が動詞句として形成されていることに気づいたかもしれない。これは、イベントをステートと区別するためである。これはオブジェクト指向におけるメッセージを彷彿とさせるが、すでに前述したように、イベントは必ずしもメッセージに対応するわけではない。

例3.3 証券取引所イベントの特定

例3.2では、「*買い*」、「*売り*」、「*ホールド*」*の状態が*、最近の値動きに基 づいて定義されている。これらの状態間の移行に直接つながるイベントは、他のトレーダーによる注文の発注である。遷移が起こるまで、さまざまな注文が発注されるかもしれない。

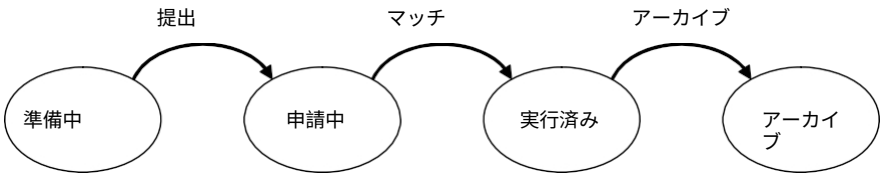


図3-7：取引注文の状態遷移を示すイベントのグラフ表示。

回帰直線の傾きが与えられたしきい値を超えた瞬間である。イベントは次のように要約できる：

イベント	内容
取引	*買い、売り、ホールドの*株式状態間の移行を引き起こす。
	取引注文の状態を遷移させる *準備中→ 順番待ち*
マッチ	トレーディング・オーダー間の状態遷移を引き起こす *OrderPending → OrderExecuted*
...	...
...	...

取引注文の遷移を示すイベントを図3-7に示す。その他の可能なイベントには、ビッドと*オファーがあり、これらは取引*注文の状態遷移につながる かもしれないし、つながらないかもしれない。これらについてはセクション3.2.2で検討する。

3.1.4 コンテクスト図とドメイン

さて、基本的な現象を定義したところで、問題領域の分析を、計画されたシステムを、それが機能する環境であるコンテキストの中に置くことから始めることができる。このために、私たちは*コンテキスト図を使います。コンテキスト*図は、本質的に、ありふれた "ブロック図 "より も少し上のものです。コンテキスト図はUMLの一部ではありません。1970年代の構造化分析にさかのぼる記法を基に、Michael Jackson [1995]によって導入されました。コンテキスト図は、開発者が解決しようとする*問題のコンテキストを表します。*図1-20（b）と図1-32で示したブロック図は、基本的にコンテキスト図です。図3-1の分割に基づき、異なるドメインを矩形の箱で示し、ある現象を共有していることを示す線で結んでいます。図3-8は、図1-20(b)をコンテクスト図として描き直し、いくつかの詳細を加えたものである。マシン」とラベル付けされたわれわれのシステムとなるべきものは、ブローカーの役割を包含し、図にはポートフォリオ、取引注文、i^{th} 株と

いった抽象的な概念も示されている。ジャクソンは、セクション2.4.2で説明した「システム」という言葉の曖昧さを避けるために、「マシン」という言葉を使っている。われわれは、"system-to-be"、"software-to-be"、"machine "という3つの用語をすべて使用している。

コンテキスト図は、私たちの問題に関連する世界の一部（図3-1）を示し、関連する部分のみを示します。コンテキスト・ダイアグラムの各ボックスは、異なるドメインを表します。**ドメインとは、**全体として考えるのが便利で、ある程度まで世界の他の部分と切り離して考えることができるため、区別することができる世界の部分である。それぞれの領域は、問題の記述に登場する異なる主題である。ドメインは、その中に存在する、あるいは発生する現象によって記述される。すべてのソフトウェア開発問題には、少なくとも2つのドメインがある：*アプリケーション・ドメイン*（または環境、または現実世界-与えられたもの）と

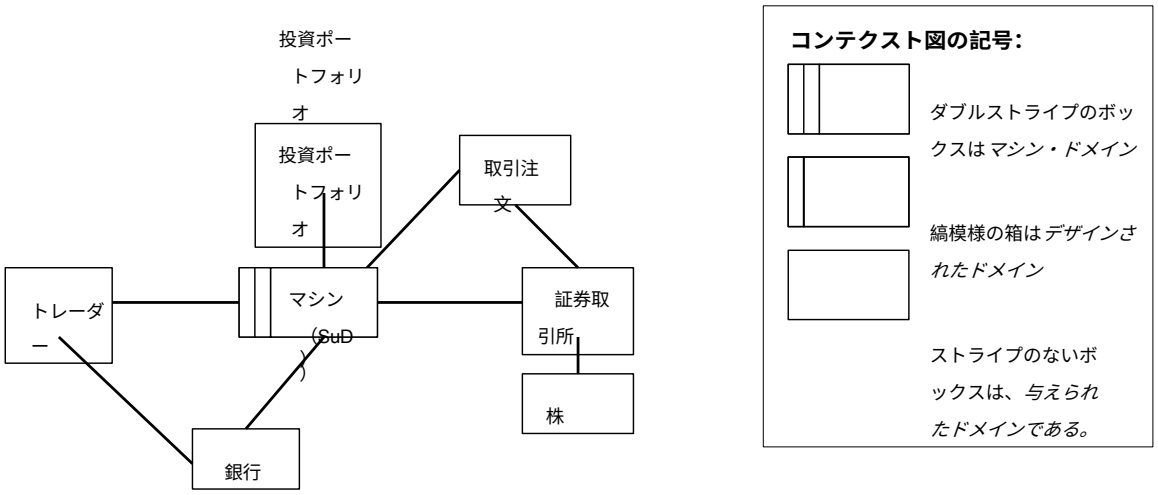


図3-8：ケーススタディ2「投資顧問システム」のコンテキスト図。

マシン（または構築されるシステム-ツー-ビー）。図3-8のドメインのいくつかは、第2章で「アクター」と呼んだものに対応する。しかし、"投資ポートフォリオ "のような他の主題もある。

単純化するために、コンテキスト図のドメインはすべて*物理ドメインとする*。図3-8では、他のドメインについては明らかかもしれないが、「投資ポ ートフォリオ」でさえ物理的ドメインであるべきである。対応するボックスは、トレーダーが所有する株式に関する情報の物理的表現を表すと仮定する。言い換えれば、これはコンピュータのメモリに保存された、あるいはスクリーンに表示された、あるいは紙に印刷された表現である。物理的ドメインと物理的相互作用を強調する理由は、ソフトウェア開発の要諦が、物理的世界と相互作用し、ユーザーの問題解決を支援するシステムを構築することにあるからである。

ドメインの種類

ドメインは与えられたものか、設計されるべきものかによって区別される。*与えられたドメインとは*、その特性が与えられた問題領域のことである。場合によっては、マシンは与えられたドメインの動作に影響を与えることができる。例えば、図3-8では、取引注文の実行が証券取引所（所与のドメイン）の動作に影響を与える。*設計されたドメインとは*、データ構造と、ある程度そのデータ内容が決定され、構築される必要がある問題領域のことである。例として、図 3-8 の "投資ポートフォリオ "ドメインを挙げることができる。

多くの場合、ある種の問題は異なる領域タイプによって区別される。このような区別は、かなりの程度、領域現象から自然に生じるものである。しかし、大きく3つのタイプに分類することも有用である。

◆ **因果領域とは、**その特性として、因果現象間の予測可能な因果関係が含まれる領域のことである。

因果関係ドメインは、他のドメインとのインターフェイスで共有される現象の一部または全部を制御することができる。

◆ **入札可能な領域は**通常、人から構成される。入札可能な領域の最も重要な特徴は、正の予測可能な内部因果関係を欠いていることである。つまり、ほとんどの状況では人にある出来事を強制的に起こさせることは不可能であり、できることといえば、従うべき指示を出すことくらいである。

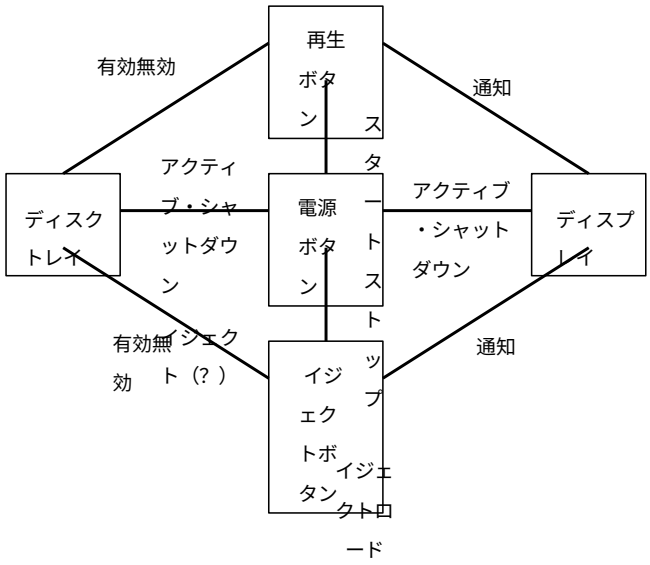


図3-9：DVDプレーヤーの制御問題におけるドメインと共有現象。

◆ レキシカル・ドメインとは、データの物理的表現、つまり記号的現象の表現である。

共有現象

ここまでは、世界の現象を特定の領域に属するものとして考えてきた。ある現象は共有される。異なるドメインから見た共有された現象は、ドメインの相互作用とコミュニケーションの本質である。ドメインは、同じ事象を異なる視点から見ていると考えることができる。

図3-9に示す。

3.1.5　システムとシステム説明

さて、ドメインを世界の区別可能な部分として定義したので、どのドメインもシステムとして考えることができる。**システムとは、**組織化された複雑な全体であり、協調して相互作用する物事や部 品 の 集合体である。すべてのシステムは、内部にあって組織の管理下にあるもの、あるいは外部にあって組織の管理下にないもの、いずれかの環境における出来事の影響を受ける。

摂動下の振る舞い：初期状態、他の平衡状態、状態遷移を定義する必要がある。

実世界のほとんどの問題では、時間と共に変化するプロセスを捉えるために力学的モデルが必要となる。アプリケーションによって、モデルは連続的か離散的か（微分方程式や差分方程式を用いる）、決定論的か確率的か、あるいはハイブリッドか、といった選択が可能である。力学系理論は、科学全般にわたって普及しているモデルの解の特性を記述する。動的システム理論は、状態空間を類似した漸近的振る舞いをする解の軌跡の領域に分割するフェーズ・ポートレートの幾何学的記述、アトラクターの統計的特性の特徴づけ、パラメータによって一般的なシステムの力学的振る舞いが質的に変化する分岐の分類などをもたらし、大きな成功を収めている。[Strogatz, 1994].

S.Strogatz, *Nonlinear Dynamics and Chaos：物理学、生物学、化学、工学への応用*.Perseus Books Group, 1994.

外的な摂動や刺激が与えられると、システムは平衡状態に落ち着くまで、一連の過渡状態を通過することによって反応する。平衡状態には、例えば体内時計によって駆動される動作のような、安定した振動が含まれることがある。

力学では、外力が物体に作用した場合、その挙動を一連の数式で記述する。ここでは、それを（離散的な）作用・反作用、あるいは刺激・反作用の一連の出来事として、わかりやすく表現する。

図3-*x*に状態遷移図を示す。アクション "turnLightOff "にはクエスチョンマークが付されているが、これはこのケースの受け入れ可能な解決策にまだ到達していないためである。状態[disarmed, unlit]が示されていないのは、ロックが解除された状態で長く留まることが想定されていないからである。

3.2システム仕様の表記法

「心理学的に、我々はすべての理論を頭の中に入れておかなければならない。理論物理学者で優秀な人は皆、まったく同じ物理学について6つか7つの理論的表現を知っている。彼は、それらがすべて等価であり、そのレベルではどれが正しいか誰も決められないことを知っている。
-リチャード・ファインマン*『物理法則の特徴*

3.2.1　仕様のための基本的な形式論

「ロジックなしですでに真理を発見している場合のみ、ロジックで真理を見つけることができる」。
-ギルバート・キース・チェスタートン*『オーソドックスだった男*

「論理: の限界と能力に厳密に従った思考と推論の技術。人間の誤解"-アンブローズ・ビアス*『悪魔の辞典*

このセクションでは、仕様書によく登場する基本的な離散数学について概説する。最初に、集合の表記法について簡単に説明する。*集合とは、メンバー*または*要素と呼ばれる*オブジェクトの、明確に定義されたコレクションである。集合はその要素によって完全に定義される。オブジェクトx
逆に、オブジェクトyが集合Aのメンバーでないことを宣言するには、$x \in A$と書きます。Aを$x \notin A$と書く。メンバを持たない集合は*空集合*であり、$\{\}$または$\varnothing$と表記される。
集合AとBが*等しい*（$A = B$と表記する）のは、両者がまったく同じメンバーを持ってい

る場合である。*B*のメンバがすべて*A*のメンバ*である*場合、集合*B*は集合*A*の*部分集合*であり、これを*B*⊆*A*と表す。Bが*A*の部分集合であり、*B*≠*A*である場合、集合*B*は*A*の適切な部分集合である、
これは*B*⊂*A*と表記される。

2つの集合Aと*B*の*和は*、そのメンバが*A*、*B*、またはその両方に属する集合であり、*A*∪*B*と*表記される*。2つの集合Aと*B*の交点は、そのメンバがAと*B*の両方に属する集合であり、*A*∩*B*と表記される。2つの集合Aと*B*は、その交点が空集合である場合、*不連続*である：A∩
B = ∅.*B* ⊆ *A*のとき、*差A* \ *B*の*集合は*、*A*のメンバーのうち、Bのメンバーでないものの集合である。*B*.

集合のメンバーはそれ自体が集合であることもある。特に興味深いのは、∅とA自身を含む、与えられた集合Aのすべての部分集合を含む集合である。この集合は集合Aの*冪集合*と呼ばれ、$\mathbb{P}(A)$、または$\mathbb{P}A$、または2^Aと表記される。

x,*y*〉はxを*第1目的語*、yを*第2目的語とする2つの*目的語の組である。2つの順序ペア〈x, *y*〉と〈a, *b*〉は、$x=a$、$y=b$のときのみ*等しい*。
x,*y*〉（$x\in A$、$y\in B$）*は*、A×A×...×Aと定義できる。
セクション3.1.1の個体間の関係の議論を思い出してほしい。A上のn-ary*関係R*は、$n>1$のとき、*n-fold* Cartesian product, $R \subseteq A \times A \times ... \times A$の部分集合として定義される。

ブール論理

論理のルールは文に正確な意味を与えるので、仕様書では重要な役割を果たす。もちろん、これらすべてを自然言語（英語など）で表現することもできるし、システム要件を記述するための独自の構文を考案することもできる。

p	q	$p \Rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

***p* ⇒ *q*の真理値表。**

の仕様に準拠している。しかし、これらの記述が標準的で予測可能な方法で表現されていれば、容易に理解できるだけでなく、そのような記述を理解する自動化ツールも開発できる。これにより、記述の自動チェックが可能になる。

命題は論理学の基本的な構成要素である。命題とは
宣言文（事実を宣言する文）であり、真か偽のどちらかであるが、両方はない。セクション1.3で、命題とは概念間の関係を記述したものであることを見た。
文は既知である。宣言文の例は、"犬は哺乳類である "と "1＋1＝3 "である。最初の命題は真であり、2番目の命題は偽である。これを書きなさい」という文は宣言文ではないので、命題ではない。また、「xは5より小さい」という文は、真でも偽でもない（xが*何*であるかに依存する）ので、命題ではない。命題を表すのに使われる文字は、p、q、r、s、...であり、これらは*命題変数*または*文変数*と呼ばれる。命題が真である場合、その真理値はTで示され、逆に偽の命題の真理値はFで示される。

多くの文は、論理演算子を使って1つ以上の命題を組み合わせ、複合命題を構成する。命題論理の演算子のいくつかを表3-1の上に示す。

*条件*文、あるいは単に*条件文*とは、2つの命題pとq*を組み合わせ*て、"pならばq "という複合命題にすることで得られる。$p \Rightarrow q$とも書き、"p implies q "とも読める。条件文$p \Rightarrow q$*において*、pは*前提*（または*先行詞*、または前置詞）と呼ばれる。

仮説）、qは*結論*（または*結果*）と呼ばれる。条件文$p \Rightarrow$ qは前提pが真で結論qが偽のときは偽、そうでないときは真。重要なのは

条件文は因果関係で解釈すべきではないことに注意。したがって、「もし*p*ならば、*q*」と言うとき、前提*p*が結論*q*を*引き起こす*という意味ではなく、*p*が真であるとき、qも真でなければならないという意味である。[1]

---

[1] これは、多くのプログラミング言語で使われるif-then構文とは異なる。ほとんどのプログラミング言語には、**if** *p* **then** Sのような文がある。

表3-1：命題論理と述語論理の演算子。

命題論理			
∧	れんごん	⇒	含意
∨	論理和	⇔	にじょうけんつき
¬	否定	≡	同値
述語論理（命題論理を2つの量化子で拡張したもの）			
∀	普遍量化（すべての*x*について、*P*(*x*) 存		
∃	在量化（*x*が存在する、*P*(*x*)		

これは、$p \Rightarrow q$、$q \Rightarrow p$を意味する。二条件文$p \Leftrightarrow q$は、pとqが同じ真理値を持つ場合に真となり、そうでない場合は偽となる。

これまで命題論理について考えてきたが、ここで述語論理を簡単に紹介しよう。xは5より小さい」という文は真でも偽でもないため、命題ではないことは前述した。この文には2つの部分がある。主語である変数xと、文の主語が持ちうる性質を指す***述語*** "is smaller than 5 "である。*P*は「5より小さい」という述語を表し、xは変数である。*P*(*x*)という文は、*x*における命題関数*P*の値であるとも言える。変数*x*に特定の値が代入されると、*P*(*x*)という文は命題となり、真理値を持つことになる。この例では、$x = 3$とすると*P*(*x*)は真となり、逆に$x = 7$とすると*P*(*x*)は偽となる[2]。

命題関数から命題を作るもう一つの方法は、*数量*化と呼ばれる。量化とは、*all*、*some*、*many*、*none*、*few*といった単語を使い、ある述語がある要素 の範囲においてどの程度真であるかを表現するものである。量化の最も一般的な種類は、表3-1の下に示す普遍量化と実存量化である。

P(*x*)の*普遍量*化とは、「*P*(*x*)は領域内のすべての*x*の値に対して真である」という命題であり、∀x *P*(*x*)と表記される。*P*(*x*)が偽である*x*の値は、P(x)の*反例*と呼ばれる。
∀*x P*(*x*)。*実存的数量*化とは、命題 "There exists a value of *x* in the domain
P(*x*)が真であるようなもの」をヨ*x P*(*x*)と表す。

有効な引数を構成する上で、重要な初歩的ステップは、ある文を同じ真理値を持つ別の文に置き換えることである。我々が特に興味を持つのは、表3-1に示すような論理演算子を用いて命題変数から形成される複合命題である。2種類の複合命題が特に興味深い。構成命題の真理値に関係なく常に真である複合命題は*トートロジー*と呼ばれる。単純な

*p*は真か偽のどちらかであるから、常に真である。一方、常に偽である複合命題は*矛盾*と呼ばれる。簡単な例は$p \wedge \neg$pで、*p*は真であると同時に偽でもあり得ないからである。当然ながら、同語反復の否定は

実行される1つ以上のステートメント。プログラムの実行中にこのようなif-then文に遭遇すると、*pが真の場合はSが*実行されるが、*pが*偽の場合は*Sは実行されない*。

[2]読者は、第3.1.2節でオブジェクトの状態関係が述語であることにすでに気づいたかもしれない。

逆もまた同様である。最後に、同語反復でも矛盾でもない複合命題を*偶発*性と呼ぶ。

$p \Leftrightarrow q$*が*トートロジーである場合、複合命題pと*q は論理的に等価であると*言われ、$p \equiv q$ *と*表記される。言い換えれば、pと*q が*、それらの構成変数のすべての可能な真理値に対して同じ真理値を持つ場合、$p \equiv q$ *である。*例えば、$r \Rightarrow$ sと¬r ∨ *s は*論理的に等価である。先に、ある条件文は偽であると述べた。

は、その前提が真で結論が偽のときのみ成り立ち、そうでないときは真となる。これを次のように書くことができる。

$r \Rightarrow s \equiv \neg\ (r \wedge \neg s)$

最初のド・モルガンの法則により、$\equiv \neg r \vee \neg(\neg s)$となる：$\neg p \wedge$ q） $\equiv \neg p \vee \neg q$
$\equiv s$

[念のため、ここで第二のド・モルガンの法則も述べておく：$(p \vee q) \equiv \neg(p \vee q)$
$\neg p \wedge \neg q$]。

自然言語の文を論理式に変換することは、システムを仕様化する上で不可欠な部分である。例えば、金融投資アシスタントに関する2つ目のケーススタディ（セクション1.3.2）における以下の要件を考えてみよう。

例3.4 要件を論理式に変換する

個人投資アシスタントに関する2つ目のケーススタディ（表2-2）の次の2つの要件を論理式に変換する：

REQ1.新規投資家登録は、本サイトの外部メールアドレスによる登録とす る 。 必要な情報には、ガイドラインに準拠した固有のログインIDとパスワード、および投資家の姓名やその他の属性情報が含まれるものとする。登録に成功すると、システムは投資家のために残高ゼロの口座を設定する。

REQ2.システムは、アクション（買い/売り）、取引銘柄、株数を指定した成行注文の発注をサポートすること。現在の気配値（売値・買値）がリアルタイムで表示・更新されること。システムは、投資家が取引の実行を望まない株価の上限／下限を指定することもできるものとする。買い注文の場合、システムは投資家の口座に十分な資金があることを確認する。成行注文が現在の市場価格と一致すると、システムは即座に取引を執行する。このチケットには、一意のチケット番号、投資家名、銘柄シンボル、株数、取引株価、新ポートフォリオの状態、投資家の新口座残高が含まれる。

まず、要求から抽出できる宣言文をすべて列挙する。REQ1からは以下の宣言文が得られる。これらが真理値を持っているかどうかはまだ分からないので、必ずしも命題ではないことに留意されたい。

ラベル	宣言文（命題とは限らない）

a 投資家はシステムに登録できる

b 投資家が入力した電子メールアドレスが現実世界に存在すること。

c 投資家が入力したEメールアドレスは、当ウェブサイトの外部アドレスです。

d 投資家が入力したログインIDは一意である

e 投資家が入力したパスワードは、ガイドラインに準拠しています。

f 投資家は姓名とその他の属性情報を入力する。

g 登録成功

h 残高ゼロの口座が投資家に設定される

次に、その真理値を確認する必要がある。仕様書には、システムが顧客に納品される時点で何が真であるかが記述されている。*aの真理値は*、システムが納品される前に開発者によって確立されなければならない。*b*、*c*、*d*、*e*の真理値は、投資家が何を入力するかに依存する。したがって、これらは投資家の入力における命題関数である。次の文を考えてみよう。

*b.email*が投資家の入力を示し、*B*が*b*の述語を示すと仮定すると、命題関数は*B*(*email*)となる。同様に、*c*は*C*(*email*)、dは*D*(*id*)、eは*E*(*pwd*)と書くことができる。システムは、投資家登録時の実行時にこれらの関数を評価することが可能であり、またそうすべきであるが、この仕様はシステムのデプロイ時に言及しているのであって、実行時に言及しているのではない。文*f*の真偽を確認するのは難しいので、システムはどのような入力値も認め、*f*を真とみなすと仮定する。

REQ1から導かれる命題は以下の通りである。

$$\frac{(\forall\ email)(\forall\ id)(\forall\ pwd)\ [B(email) \wedge C(email) \wedge D(id) \wedge E(pwd) \Rightarrow g]f}{g \Rightarrow h}$$

論理における条件文は、プログラミング言語におけるif-then構文とは異なることを読者は再認識すべきである。したがって、$g \Rightarrow h$は、「登録が成功したら、残高ゼロの口座を開設する」というよ う な因果関係のある一連の命令を記述するものではない。そうではなく
gが真であるとき、hも真でなければならない。

このシステムは、4つの命題をすべて真とする真理値の代入に関するREQ1を正しく実装している。上の2番目の命題の代わりに$(b \wedge c \wedge d \wedge e) \Rightarrow g$と書くのは間違いである。
入力パラメータ。

REQ2から宣言文を抽出するのは、REQ1の場合よりも少し複雑である。REQ2の最も複雑な2つの側面は、株式購入のための十分な資金を確保することと、現在の価格が範囲内（トレーダーが上限/下限を指定した場合）にある場合にのみ注文を実行することである。トレーダーが選択したティッカーシンボルをSYMとし、取引所での現在のアスク価格をIP（気配値）とします。REQ1の電子メールとパスワードとは異なり、ここでは許容できる選択肢のみを表示することで、ユーザに有効なティッカーシンボルを選択させることができることに注意してください。投資家が指定した取引の株数（数量）はVOLと表記されます。投資家が上限／下限を指定した場合、その値をそれぞれ UB、LB とする。最後に、投資家の現在の口座残高を BAL とする。

以下は、これら2つの制約を述べるために必要な命題の部分的な

リストである：ラベル命題　（部分リスト）

m	投資家が指定したアクションは "買い"
n	投資家は「買い」価格の上限を指定した。
o	投資家は「売り」価格の下限を指定した。

上の表が命題を含んでいるのは、その真理値がユーザーの選択とは無関係に成立するからで

ある。例えば、開発者はトレーディング・アクションを "買い "か "売り "の2択しか許さないはずなので、¬*m*は投資家が "売り "を選択したことを意味する。投資家が上限/下限を指定した場合
バウンズでは、システムは[$n \wedge m \wedge (IP \leq UB)$] ∨ [$o \wedge \neg m \wedge (LB \leq IP)$] の場合にのみトランザクションを実行する。
投資家の口座残高が現在の取引に十分であることを確認するために、システムは [¬n ∧ (VOL × IP ≤ BAL)] ∨ [$n \wedge (VOL \times UB \leq BAL)$] をチェックする必要があります。

REQ2から抽出された追加の宣言文は以下の通りである：

	ラベル命題（上記のリストを完成させる）
p	投資家が成行注文の発注を依頼する
q	投資家には空白のチケットが表示され、そこで取引を指定することができる（アクション、シンボルなど）。
r	直近に検索された気配値が、現在開いている注文チケットに表示されます。
s	投資家が指定したシンボルSYMは有効なティッカーシンボルである。
t	取引所から得られる現在の指標価格

u システムが取引を実行する
v システムはプレーヤーの口座の新しい残高を計算する。
w システムはトランザクションの結果について確認を発行する。
x システムはトランザクションをアーカイブする

REQ2は命題$p \Rightarrow q \Rightarrow r$の集合として表される。

s

$y = v \wedge \{\neg(n \vee o) \vee [(o \wedge p \vee \neg o \wedge q) \wedge (\exists\ \mathrm{IP})(\mathrm{LB} \leq \mathrm{IP} \leq \mathrm{UB})]\}.$
$z = \neg m \vee \{ [\neg n \wedge (\mathrm{VOL} \times \mathrm{IP} \leqq \mathrm{BAL})] \vee [n \wedge (\mathrm{VOL} \times \mathrm{UB} \leqq \mathrm{BAL})] \}$ 。
$y \wedge z \Rightarrow u$
$u \Rightarrow v \wedge w \wedge x$

繰り返しになるが、システムがREQ2を正しく実装するためには、上記の命題がすべて真と評価されなければならない。REQ1とは異なり、REQ2ではユーザーの選択を制限し、表現を単純化することに成功した。このようにすることで、単なる問題記述の枠を超え、問題解決にいくつかの選択肢を課したことは事実である。しかし、今回の場合、これらは非常に単純でわかりやすい選択だと思います。単純化が行き過ぎて解の選択肢を狭めてしまうかどうかを判断するには、開発者の判断と経験が必要だが、純粋さの追求が無用な余分な作業をもたらすだけということもある。

システム仕様は**一貫して**いなければならない。つまり、矛盾する要件を含んではならない。言い換えれば、要求が命題の集合として表現される場合、すべての要求命題を真にする命題変数への真理値の割り当てがあるべきである。

例...

セクション3.2.3では、オブジェクト制約言語（OCL）と呼ばれるUML標準の一部において、論理がどのような役割を果たすかを説明する。セクション3.2.4で説明する、ブール論理に基づいたもう1つの表記法がTLA+である。

有限状態マシン

複雑な物体やシステムの動作は、直近の入力だけでなく、過去の入力履歴にも依存する。*状態として*表現されるこの記憶特性により、そのようなシステムは時間とともに動作を変化させることができる。このようなシステムを記述するための単純だが重要な形式表記法は、*有限状態機械*（FSM）と呼ばれる。FSM はコンピューター・サイエンス

やデータ・ネットワーキングで広く使われており、UML 標準では FSM を UML ステートマシン図に拡張しています（セクション 3.2.2）。

有限状態機械を表現する方法はいろいろある。1つの方法は、各入力がマシンの状態にどのように影響するかを示す表を作ることである。以下は、ケーススタディの例で使用したドアロックの*状態表である。*

	現状

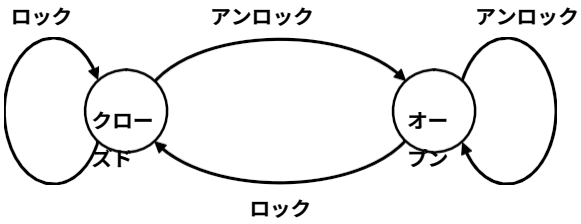
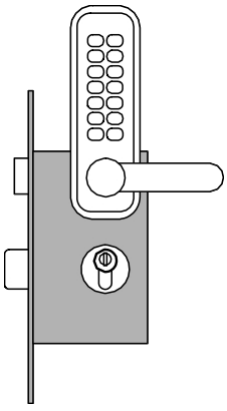


図 3-10：ドア・ロックの状態遷移図

		武装	武装解除
インプット	ロック	武装	武装
	アンロック	武装解除	武装解除

ここで、表の本文の項目は、現在の状態（列）と入力（行）に応じて、マシンが次に入る状態を示している。

また、ラベル付きの辺を持つ有向グラフである*遷移図を用い*て、マシンをグラフィカルに表現することもできる。この図では、各状態は円で表される。矢印は各遷移の入力を示す。例を図3-10に示す。ここでは、状態 "Disarmed "と"Armed "が円で示され、ラベル付き矢印は、マシンが各状態にあるときの各入力の効果を示す。



有限状態機械は、有限の状態集合S、有限の入力集合I、および$S \times I$をドメインとし、Sをコドメイン（または範囲）とする遷移関数から構成されると形式的に定義される。
関数fは部分関数とすることができる。
そのドメインのいくつかの値については未定義である。ある種のアプリケーションでは、*初期状態* s_0 と、マシンを終了させたい*最終状態* $S' \subset S$ のセットを指定することもできる。最終状態は、状態図では
二重同心円。M = maxNumOfAttemptsが最終状態である。この状態でマシンは停止し、外部から再起動する必要がある。

*文字*列は入力の有限列である。文字列$i\ i_{12}\ ...\ i_n$ と初期状態s_0 が与えられると、マシンは$s_1 = f(s_0, i_1)$、次に$s_2 = f(s_1, i_2)$、...と逐次計算し、最終的に状態s_n で終わる。 図3-11の例では、入力文字列*iiv*はFSMを次の状態へと遷移させる。

$s\ s\ s_{0120}$ s.$s_n \in$ S′、つまり受理状態であれば、その文字列は*受理*されたと言う。
は*拒否される*。図3-11において、M iの入力文字列（i^M）は、以下のようになることが容易にわかる。
を認識した。この機械はこの文字列を認識し、この意味で、この機械は

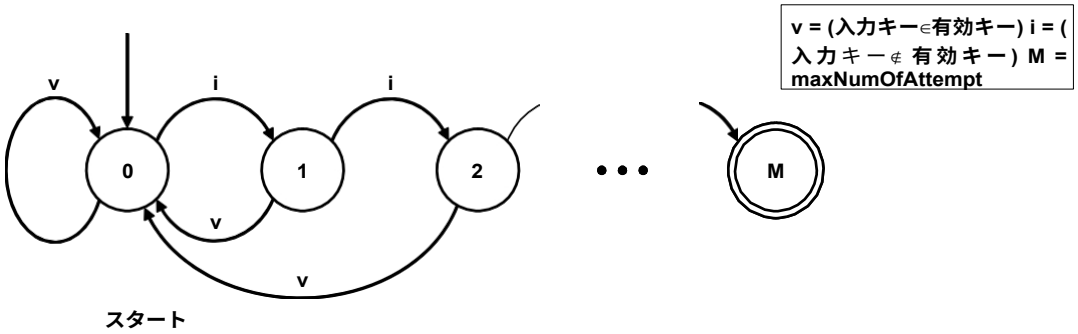


図 3-11：開錠失敗カウンタの状態遷移図。

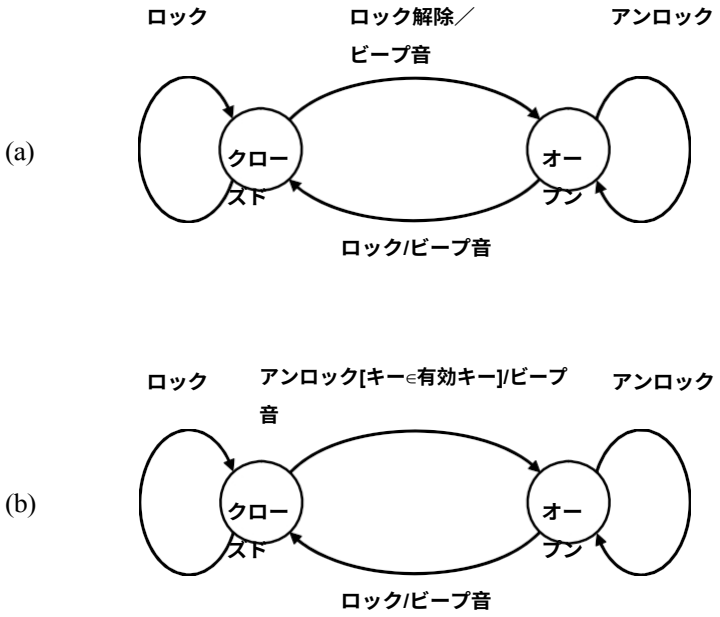


図3-12：図3-10の状態遷移図（出力ラベルを含むように修正(a）とガードラベル（b）。

侵入を試みた。

もう少し複雑な機械は、次の状態に遷移するときに出力を生成する FSM である。例えば、図 3-10 のドア・ロックが、使用者に武装または解除を知らせるために音声信号も生成するとする。図 3-12(a)に修正図を示す。各遷移矢印の入力と出力のラベルをスラッシュ記号で区切っています。(ここでは、マシンが重複入力を受信したときに出力を生成しないことを選択していることに注意)。

有限の状態集合S、有限の入力集合I、有限の出力集合Oと、各（状態、入力）対に新しい状態を割り当てる関数$f : S \times I \rightarrow S$、および各（状態、入力）対に次の状態を割り当てる別の関数$g : S \times I \rightarrow O$から構成される*出力付き有限状態機械*を定義する。
出力。

新しい機能を追加することで、元のFSMモデルを充実させることができる。図3-12(b)は遷移に*ガード*を追加する方法を示しています。 遷移記述の完全な表記法は次のようになります。

入力[guard]/出力]。ガードは次のようなブール命題である。
はトランジションを許可またはブロックする。ガードが存在する場合、移行は以下の場合に行われる。
は真と評価されますが、ガ ードが偽の場合、遷移はブロックされます。セクション3.2.2では、FSMモデルをUMLステートマシン図に拡張するために、UMLがどのように他の機能を追加してい るかについて説明します。

3.2.2 UML ステートマシン図

システムおよびソフトウェア仕様の文脈における（前節で説明した）オリジナルの有限状態機械モデルの主な弱点の1つは、*モジュール化*のメカニズムがないことである。セクション3.1.2における状態と状態変数の定義を考慮すると、FSMは個々の単純な状態（または微小状態）を表すのに適しています。UML ステートマシン図は、ステートマシンのための標準化された図式表記法を提供し、マクロ状態や並行動作などの拡張も組み込んでいます。

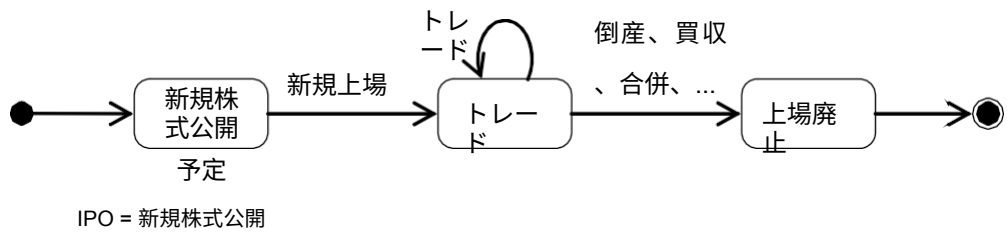


図 3-13: 在庫の状態と遷移を示す UML ステートマシン図。

基本的な表記

これは、塗りつぶされた円で示された特別なアイコンから、この状態へのラベルのない遷移を記述することで指定する。その例を図3-13に示す。停止状態を指定する必要がある場合もある。ほとんどの場合、オブジェクトやシステム全体に関連付けられているステートマシンが停止状態に達することはありません。*停止状態を*指定するには、この状態から特別なアイコンへのラベルのない状態遷移を描きます。[3]初期状態と最終状態は*擬状態*と呼ばれる。

状態のペア間の遷移は有向矢印で示される。状態間を移動することを*遷移の発射と呼ぶ*。状態はそれ自身に遷移することがあり、同じ状態から多くの異なる状態遷移があるのが普通である。すべての遷移は一意でなければならない。つまり、同じ状態から複数の遷移を引き起こすような状況は決して存在しない。

トランジションの発火を制御するにはさまざまな方法がある。注釈のないトランジションは*完了トランジション*と呼ばれる。これは単に、オブジェクトがソース状態でのアクティビティの実行を完了すると、トランジションが自動的に発火し、ターゲット状態に入ることを意味します。

また、トランジションが発火するためには、特定のイベントが発生しなければならない場合もある。そのようなイベントはトランジションに注釈が付けられる。(図3-13では、倒産や*買収の*現象はイベントではなく状態と考えるべきだという意見があるかもしれない。正しい答えは、オブザーバーにとって相対的なものである。私たちのトレーダーは、その会社がどれくらいの期間倒産状態にあるかなんて気にしていない。重要なのは、倒産が有効になった瞬間から、その会社の株式が取引できなくなることだけである。

FSMでは、遷移を制御するために*ガード条件を*指定することができることはすでに説明

した。これらの条件は、イベントが発生したときに、（条件が真であれば）遷移を許可し、（条件が偽であれば）遷移を許可しないように、ガードとして機能する。

州の活動エントリー、ドゥ、イグジット・アクティビティ

状態には受動的なものと能動的なものがあることはすでに述べた。特に、あるアクティビティは、ある状態に対してある時点で実行されるように指定されることがある：

---

[3]図3-13の*上場廃止*状態は、指定された取引所に関する停止状態です。投資家はその取引所では株式を取引できなくなりますが、他の市場では取引される可能性があります。

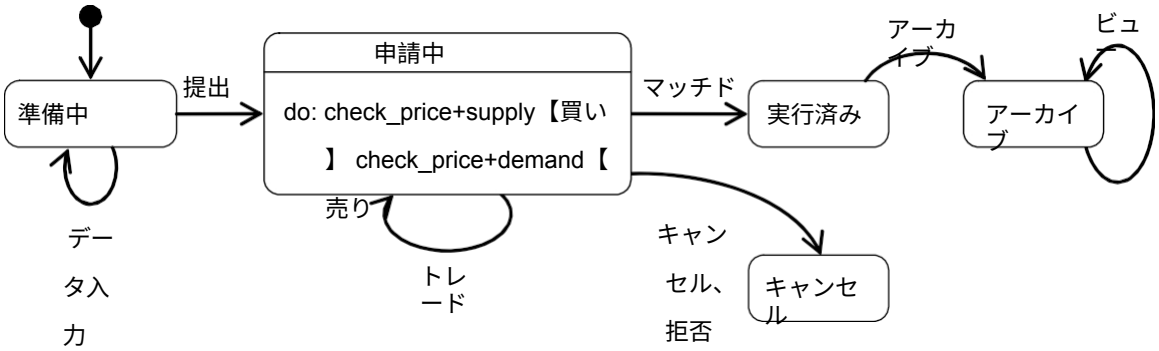


図3-14：取引注文の状態アクティビティ例。図 3-7 と比較する。

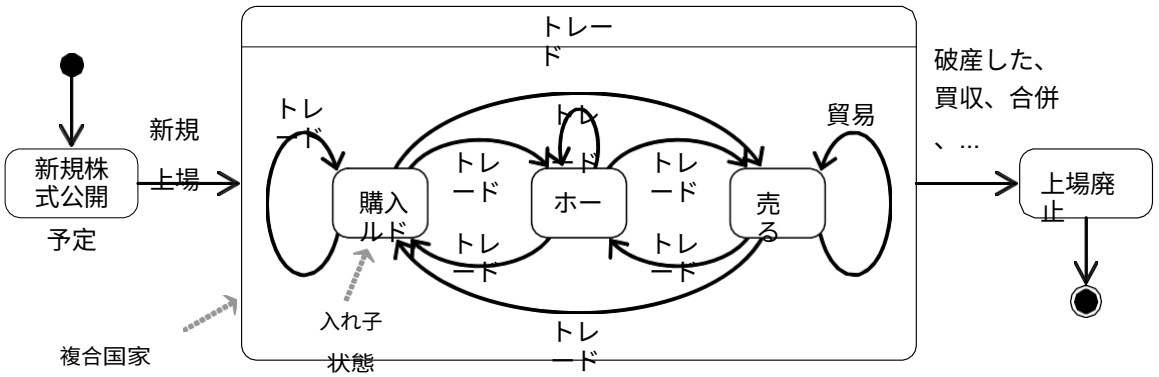


図3-15：株式の複合状態と入れ子状態の例。図 3-13 と比較する。

- 入国時に活動を行う
- 州滞在中にアクティビティを行う
- 状態の終了時にアクティビティを実

行する 例を図3-14に示す。

複合ステートと入れ子ステート

UML の状態図では、スーパー*状態*（またはマクロ状態）を定義します。スーパー状態とは、有限ステートマシンに分解することでさらに洗練された複雑な状態のことです。すでにセクション3.1.2で説明したように、スーパー状態は、初等状態の集約によっても得られます。

ここで、図3-13の図を拡張して、例3.2で定義した「*買い*」、「*売り*」、「*ホールド*」*の状態を表示したいとする。*これらの状態は、図 3-15 に示すように、それらが入れ子になっている*Traded* 状態を洗練したものである。この入れ子は、*リージョン*と呼ばれる境界で描かれ、その境界は*複合*状態と呼ばれる。3つのサブステートを持つ複合ステー

ト *"Traded "*が与えられた場合、ネスティングのセマンティクスは排他的論理和(XOR)の関係を意味する。株価が*Traded*状態(複合状態)にある場合、その株価は3つのサブステートのうちの1つにもなっていなければならない：*Buy*、*Hold*、または *Sell*。

ネスティングは任意の深さまで可能であり、その結果、サブステートは他の下位のサブステートと複合状態になる可能性がある。深さのある状態遷移図を簡単に描くために、特定の状態に対してズームインまたはズームアウトすることができる。ズーム・アウトすると、図3-13のように部分状態が隠され、図3-13のように

図3-16：状態における並行性の例。

図3-15のように、ズームインするとサブステート（部分状態）が表示されます。ズーム・アウト表示により、複雑なステート・マシン図の理解度が向上する可能性があります。

コンカレンシー

図3-16

アプリケーション

ステートマシン図は、通常、個々のオブジェクトの振る舞いを記述するために使われます。ただし、開発者が現在検討している抽象要素の振る舞いを記述するために使用することもできます。また、検討中のシステム全体のステートマシン図を提供することもあります。開発ライフサイクルの分析フェー ズ（セクション2.5で説明）では、システム全体のイベント順の振る舞いを検討します。設計フェーズ（セクション2.6で説明）では、個々のクラスやクラスのコラボレーションの動的な振る舞いを表現するために、 ステートマシン図を使用することがあります。

セクション3.3では、複雑な問題を理解し、基本的な問題に分解しようとするときに、ステートマシン図を使って問題領域を記述する。

3.2.3 UML オブジェクト制約言語（OCL）

「フランス語は話せるが、理解できない。-マーク・トウェイン

UML標準では、ブール論理に基づくオブジェクト制約言語（OCL）を定義しています。

OCLでは、演算子に数学記号を使用する代わりに（表3-1）、ASCII文字のみを使用します。また、OCL はところどころで少し言葉が多くなります。

OCL は独立した言語ではなく、UML の不可欠な一部分です。OCL 式は、UML モデルのコンテキスト内に配置する必要があります。UML 図では、OCL は主に、クラス図の制約や、状態図やアクティビティ図のガード条件を記述するために使用されます。*制約として*知られる OCL 式は、UML 図の要素に関する事実を表現するために追加されます。

表3-2：基本的な定義済みOCL型とそれに対する操作。

タイプ	価値観	オペレーション
ブーリアン	真、偽	and、or、xor、not、implies、if-then-else
整数	1, 48, -3, 84967, ...	*, +, -, /, abs()
リアル	0.5, 3.14159265, 1.e+5	*, +, -, /, floor()
ストリング	探検が増えればテキストも増える』。	concat(), size(), substring()

このような設計モデルから派生した実装は、各制約が常に真であることを保証しなければならない。

ソフトウェア・クラスには、明確に定義された始点と終点を持つという意味で指定すべき計算という概念がないことに留意すべきである。クラスはプログラムやサブルーチンではない。むしろ、オブジェクトの操作は、特定の順序によらず、任意のタイミングで呼び出すことができる。また、操作の入出力関係だけでなく、オブジェクトの状態がその動作の重要な要因となることもある。その状態によって、同じ操作でもオブジェクトの振る舞いが異なることがある。オブジェクトの状態に対する操作の影響を特定するには、オブジェクトに対して呼び出された以前の一連の操作の結果であるオブジェクトの現在の状態を記述できる必要がある。オブジェクトの状態は、その属性や他のオブジェクトとの関連で捕捉されるため、OCL 制約は通常、オブジェク トのこれらの特性に関係します。

OCL構文

OCL の構文は、C++ や Java などのオブジェクト指向言語に似ています。OCL 式は、モデル要素、制約、および演算子で構成されます。「Constraints」（制約）は、「制約の有効性」、「制約の有効性」、「制約の有効性」を表します。「Constraints」 なし 限定子を付けない名前で、クラス、制約、および演算子を表すことができます。OCL 式が評価されると、単に値が返されます。OCL 式の評価によって、システムの状態が変化することはありません。たとえ、OCL 式が後条件指定のよう に状態変化を*指定*するために使用されたとしてもです。

OCLには4つの組み込み型がある： Boolean, Integer, Real, String です。このような場合、OCL は、「Constraints（制約）」（「Constraints」）に従います。これらの定義済み値型は、どのオブジェクト・モデルにも依存せず、OCL の定義の一部です。

このセクションでは、OCL でのクラス定義について説明します。「Element Import」 なし 部分とコンポジットの間の破線の矢印。「Element Import」 なし 限定子を付けない名前で、要素の振る舞いを記述します。図 2-35（セクション 2.6）のクラス図の例を考えてみましょう。`Controller` クラスの `numOfAttempts_` 属性にアクセスするには、次のように記述します。

```
self.numOfAttempts_
```

カプセル化されているため、オブジェクトの属性にはアクセサー・メソッドでアクセスしなければならないことが多い。したがって、`self.getNumOfAttempts()`と書く必要があるかもしれない。

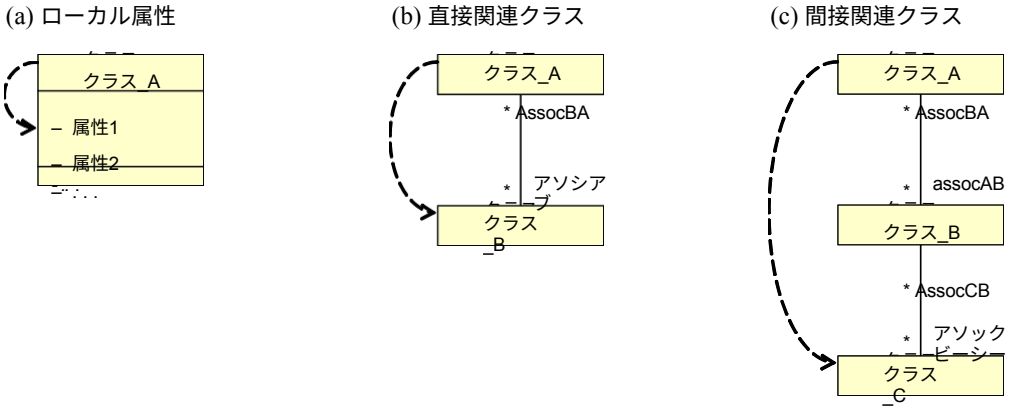


図3-17：UML クラス図におけるナビゲーションの 3 つの基本タイプ。(a)クラス A のインスタンスからアクセスされるクラス A の属性 (b)クラス A のインスタンスからクラス B のインスタンス・セットにアクセスする (c)クラス A のインスタンスからクラス C のインスタンス・セットにアクセスする

Together では、モデル要素やそのプロパティーを参照するために、クラス図上の*関連付けをナビゲートすることが*できます。図 3-17 に、3 つの基本的なナビゲーション・タイプを示します。`Class_A` のコンテキストで、そのローカル属性にアクセスするには、`self.attribute2` と記述します。同様に、直接関連付けられたクラスのインスタンスにアクセスするには、クラス図内の反対側の関連端の名前を使用します。つまり、図3-17(b)では、`Class_A`のコンテキストで`Class_B`のインスタンスの集合にアクセスするには、`self.assocAB`と記述します。最後に図3-17(c)では、`Class_A`のコンテキストで、間接的に関連付けられたクラス`Class_C`のインスタンスにアクセスするには、`self.assocAB.assocBC`と記述します。(このアプローチは、JavaやC#のようなオブジェクト・プログラミング言語に慣れている読者には驚きではないだろう)。

UMLのクラス図から、オブジェクトのアソシエーションは、個々のオブジェクト（アソシエーションの多重度は1に等しい）かコレクション（アソシエーションの多重度＞1）であることはすでに知っている。1対1の関連をナビゲートすると、直接オブジェクトが得られます。図2-35は、1つの`LockCtrl`（仮に
単一のロックデバイスがシステムによって制御されていることを意味する)。この関連付けがリスト2.2のように`lockCtrl_`という名前だとすると、ナビゲーション`self.lockCtrl_`は単一のオブジェクト`lockCtrl_ : LockCtrl`を生成する。しかし、コントローラーが複数のロック（例えば、フロントドアとバックヤードドア）に関連している場合、このナビゲーションは2つの`LockCtrl`オブジェクトのコレクションを

生成します。

OCLは3種類の*コレクション*を規定している：

- *OCL集合は*、一対多の多重度を持つ即時的な関連をナビゲートする結果を収集するために使用される。
- *OCL シーケンスは*、即座に順序付けられた関連をナビゲートするときに使われます。
- *OCL バッグは*、間接的に関連するオブジェクトをナビゲートするときに、オブジェクトを蓄積するた めに使用されます。この場合、同じオブジェクトがコレクションに複数回現れることがあります。

  異なるナビゲーション・パスを介してアクセスする。

図3-17(c)の例では、`self.assocAB.assocBC`式は、クラス_Aオブジェクトに関連付けられたクラス_Bオブジェクトのすべてのインスタンスに関連付けられたクラス_Cオブジェクトのすべてのインスタンスの集合として評価されることに注意してください。

表 3-3: コレクションにアクセスするための OCL 操作の概要。

OCL記法	意味
すべてのOCLコレクションに対する操作例	
c->size()	コレクション c の要素数を返します。
c->isEmpty()	c に要素がなければ真を、なければ偽を返す。
c1->includesAll(c2)	c2 のすべての要素が c1 で見つかった場合に真を返す。
c1->excludesAll(c2)	c1 に c2 の要素が見つからない場合に真を返す。
c->forAll(var | expr)	要素が評価されるとき、それは変数varにバインドされ、exprで使用することができます。この は普遍的な数量化∀を実装している。
c->forAll(var1, var2) | expr)	上記と同じ。ただし、exprは、以下の場合を含め、cから取り得るすべての要素の*組に対し*て評価される。 は同じ要素で構成される。
c->exists(var | expr)	が成り立つ要素が c に少なくとも1つ存在すれば真を返す。 を返す。 が真である。 この は を実装している。 定量化 ∃。
c->isUnique(var | expr)	expr が c の各要素に適用されたときに異なる値に評価された場合に真を返す。
c->select(expr)	の c の要素のみを含むコレクションを返す。 が真になる。
OCLセット特有の操作例	
s1->intersection(s2)	s1 と s2 で見つかった要素の集合を返す。
s1->union(s2)	s1 または s2 で見つかった要素の集合を返す。
s->excluding(x)	オブジェクトxを含まない集合sを返す。
OCLシーケンス特有の操作例	
seq->first()	シーケンスの最初の要素であるオブジェクトを返します。 seq。

クラスの属性とコレクションを区別するために、OCL では属性へのアクセスにドット記法を、コレクションへのアクセ スに矢印演算子 **->** を用います。コレクションのプロパティにアクセスするには、コレクションの名前に続いて矢印 -> を書き、その後にプロパティの名前を書きます。OCL は、コレクションにアクセスするための多くの定義済み操作を提供します。

定数は、あらかじめ定義された OCL 型（表 3-2）のいずれかを持つ不変の値です。

*演算*子は、モデル要素と定数を組み合わせて*式を*構成する。

OCLの制約と契約

コントラクトは、クラスのユーザー、実装者、および拡張者がクラスについて同じ前提を共有できるようにする、クラスに対する制約です。コントラクトは、常に有効でなければならない、または操作が呼び出される前や後などの特定の時に有効でなければならない、クラスの状態に対する制約を指定します。このコントラクトは、クラスの実装者との間で、何ができるかを約束するものです。

このような場合、「Constraints」（制約）は、そのクラスが使用される前に満たさなければならない義務について、クラス・ユーザーと期待される関係を表します。このような場合、「Constraints」（制約）は、「Constraints」（制約）は、「Constraints」（制約）が「Constraints」（制約） であることを意味します。

オブジェクトの状態の重要な特徴付けのひとつは、オブジェクトのライフタイムを通じて何が不変であるかを記述することである。これは、不変述語を使って記述することができます。**不変量は、**どのような操作がどのような順序で呼び出されたかに関係なく、クラスのすべてのインスタンスオブジェクトに対して常に真と評価されなければなりません。不変量は、クラス属性に適用されます。

さらに、各操作は前提条件と後条件を記述することで指定できる。**前提**条件は、操作が実行される前にチェックされる述語である。前提条件は特定の操作に適用される。前提条件は、操作への入力パラメータを検証するために頻繁に使用される。

後条件は、操作の実行後に真でなければならない述語である。後条件は特定の操作にも適用される。後条件は、操作によってオブジェクトの状態がどのように変化したかを記述するためによく使われる。

ドメインモデルの文脈では、すでにいくつかの前提条件と後件条件に遭遇した（セクション2.5.4）。その後、図 2-35 で、ドメイン属性を特定のクラスに割り当てた。ここでは、非公式なその場しのぎの記法を使いました。OCL は、制約を表現するための正式な記法を提供します。例えば、我々のケーススタディー・システムの制約の1つは、ロック解除の最大許容試行回数が正の整数であることです。この制約は常に真でなければならないので、不変量として記述します:

コンテキスト コントローラ **inv:**

```
self.getMaxNumOfAttempts() > 0
```

ここで、最初の行は、制約のタイプと同様に、コンテキスト、すなわち、制約が適用されるモデル要素を指定します。この場合、`inv`キーワードは、*不変*制約タイプを示します。ほとんどの場合、文脈が明確なので、キーワード*self*は省略できます。

その他の制約のタイプとしては、*前提条件 (*`pre` キーワードで指定) および *事後条件* (`post` キーワードで指定) があります。操作 `enterKey()` を実行するための前提条件は、失敗した試行回数が最大許容回数未満であることです:

```
context Controller::enterKey(k : Key) : boolean pre:
       self.getNumOfAttempts() < self.getMaxNumOfAttempts()
```

enterKey()の後条件は、(Poc1)失敗した試行が記録されること、および(Poc2)失敗した試行の回数が許容最大数に達した場合、システムがブロックされ、警報ベルが鳴ることである。最初の後条件(Poc1)は次のように言い換えることができる：

(Poc1′) 指定されたキーが有効なキーの集合の要素でない場合、enterKey()から抜けた後の試行失敗カウンタは、enterKey()に入る前の値よりも1つ大きくなければならない。

上記の2つの後条件（Poc1′）と（Poc2）は、OCLでは次のように表現できる：

```
context Controller::enterKey(k : キー) : ブール

   -- 後条件（Poc1′）：
   post: let allValidKeys : Set = self.checker.validKeys()
         もしallValidKeys.exists(vk | k = vk) ならば
```

```
                getNumOfAttempts()= getNumOfAttempts()@pre
```

その他

```
                getNumOfAttempts()= getNumOfAttempts()@pre + 1
```

```
-- ポストコンディション（Poc2）：
post: getNumOfAttempts() >= getMaxNumOfAttempts() implies
          self.isBlocked() and self.alarmCtrl.isOn()
```

このような制約の中で使用できる変数（この場合、Set 型の OCL コレクションの allValidKeys）を定義することができます。まず、`let` 式によって、制約で使用できる変数（この場合、OCL コレクション型 `Set` の`allValidKeys`）を定義できます。

第二に、`@pre`指示文は、操作の*前に*存在したオブジェクトの値を示す。したがって、`getNumOfAttempts()@pre`は、`getNumOfAttempts()`を呼び出す前の値を示す。を呼び出す前に　　を呼び出す前に　`getNumOfAttempts()` は、`enterKey()` を呼び出した後の同じ操作によって返される値を示します。

第三に、`if-then-else` 操作の中の `getNumOfAttempts()` に関する式は*代入ではありません*。OCL はプログラミング言語ではなく、OCL 式の評価は決してシステムの状態を変更しないことを思い出してください。OCL はプログラミング言語ではなく、OCL 式の評価は決してシステムの状態を変更しません。その結果はブール値の真か偽です。

サイドバー3.1：従属代議員のジレンマ

◆ クラス不変量はオブジェクト指向プログラミングの重要な概念であり、クラスとそのインスタンスについての推論に不可欠である。残念なことに、クラス不変量は自明でない限り

の例では、常に満たされるとは限りません。クライアント・オブジェクトがサーバー・オブジェクト上で呼び出したメソッド（「従属デリゲート」）の実行中に、一時的に不変量が破られることがあります。なぜなら、そのような中間状態では、サーバーオブジェクトは、クライアントが呼び出したメソッドの実行に忙しく、他の世界からは直接使用できないため、その状態が矛盾していても問題ないからです。重要なのは、メソッド呼び出しの実行前と実行後に不変性が保持されることです。

しかし、サーバーのメソッド実行中にサーバーがクライアントのメソッドをコールバックした場合、サーバーはクライアントオブジェクトを矛盾した状態でキャッチする可能性があります。これは*従属デリゲートのジレンマとして*知られており、扱いが難しい。詳しくは[Meyer, 2005]を参照されたい。

OCL標準はコントラクトだけを規定している。OCL標準の一部ではありませんが、ブール論理を使ってプログラムの振る舞いを指定することを妨げるものは何もありません。

例

3.2.4 TLA+表記

このセクションでは、レスリー・ランポートによって定義されたTLA+システム仕様言語を紹介する。TLA+の解説本はhttp://lamport.org/。他にも多くの仕様言語がありますが、TLA+は多くの点でZ（発音はゼット、ジーではありません）仕様を思い起こさせます。

```
1  ──────────── MODULE AccessController ────────────
2  CONSTANTS validKeys、                    有効なキーの集合。
3                  ValidateKey( _ )   ValidateKey(k) ステップは、k が有効なキーであるかどうかを
                                      チェックします。
4  ASSUME validKeys ⊂ STRING
5  ASSUME ∀ key ∈ STRING : ValidateKey(key) ∈ BOOLEAN
6  変数 status
7  TypeInvariant =^ status∈ [lock : {"disarmed", "armed"}, bulb : {"lit", "unlit"}]
8  ──────────────────────────────────────────────
9  イニシャル =^ ∧ TypeInvariant            初期述語。
10            ∧ status.lock = "armed"
11            ∧ status.bulb = "unlit"
12 Unlock(key) =^ ∧ ValidateKey(key)        ユーザーが有効なキーを入力した場合のみ
13            ∧ ステータス'.ロック = "解除"  ロックを解除し
14            ∧ status'.bulb = "点灯"       まだ点灯していない場合）点灯させる。
15 ロック =^ ∧ status'.lock = "武装"         誰でもドアをロックできる
16            ∧ UNCHANGED ステータス.バルブ  ただし、ライトを弄らないこと。
17 Next =^ Unlock(key)∨Lock次の状態のアクション。18
19 Spec =^ Init∧□[Next]status               仕様。20
21 ──────────────────────────────────────────────
   定理 Spec ⇒ □ TypeInvariant              仕様の型の正しさ。22
```

図3-18：事例研究システムのTLA+仕様。

の言語を使っている。私がTLA+を選んだ理由は、別の形式主義を発明するのではなく、数学の言語、特にブール代数の言語を使うからだ。

TLA+仕様は、以下の例（図3-18）のようにモジュールで構成される。これは、我々のホームアクセスケーススタディシステム（セクション1.3.1）を仕様化している。TLA+言語の予約語はSMALL CAPSで表示され、コメントは強調表示される。モジュールはいくつかのセクションから構成される

- *変数*の宣言。これは主に、外部のオブザーバーから見えるシステムの現れである。
- *ビヘイビア*の定義：*初期*状態と、それに続く（*次の*）すべての*状態*。
- 仕様に関する*定理*

変数にはシステムの内部、目に見えない側面も含まれ得るが、それらは主に外部システムの発現に対応する。ホームアクセスコントローラーのケーススタディでは、興味のある変数はロックと電球の状態を記述します。それらは、6行目と7行目の1つの*ステータス*レコードに集約されます。

区切り線8と20は純粋な飾りであり、省略可能である。これらとは異なり、モジュールの開始行と終了行、それぞれ1行目と22行目には意味上の意味があり、必ず表示されなければならない。

2行目と3行目ではモジュールの定数を宣言し、4行目と5行目ではこれらの定数に関する仮定を列挙している。例えば、有効なパスワードの集合は、STRINGで記号化されたすべての文字列の部分集合であると仮定する。5行目では、基本的に任意のキー*kに対して*、*ValidateKey*(*k*)はBOOLEAN値を返すと仮定しています。

7行目の*TypeInvariantは*、仕様を満たす動作においてシステム変数が取り得るすべての値を指定する。これは仕様の性質であり、仮定ではない。そのため、仕様の最後、21行目に定理として記述されている。

システムの初期状態の定義は9行目と10行目にある。

17行目で次の状態を定義する前に、システムに要求できる機能を定義する必要がある。この場合、ロックの解除とアームという重要な機能（それぞれDisarmと*Arm*）にのみ焦点を当て、その他は無視する（セクション2.2のすべてのユースケースを参照）。これらの機能を定義することは、おそらく仕様の最も重要な部分である。

アポストロフィ記号の付いた変数status′*は*、操作が行われた後の、次のステップの状態変数を表す。

3.3問題フレーム

「コンピューターは役に立たない。答えを与えるだけだ。"-パブロ・ピカソ "問題を解決するということは、単に解決策が透明になるように表現するということだ"

-ハーバート・サイモン*『人工の科学*

問題フレームは、マイケル・ジャクソン[1995; 2001]によって、問題を理解し体系的に記述するための方法として提案された。問題フレームは、元の複雑な問題を単純で既知の部分問題に分解する。各フレームは、標準的な方法で解決できるように十分に定型化され、明確に分離された関心事を提示できるように十分に単純化された問題クラスを捉えます。

データの取得と表示に関する問題は、テキスト編集の問題とは異なり、ソースコードをマシンコードに変換するコンパイラを書くこととは異なる、というのが私たちの直感的な感覚だ。問題によっては、このような単純な問題を多数組み合わせたものもある。問

題フレームの重要な考え方は、単純な問題のカテゴリーを特定し、複雑な問題を単純な問題で表現するための方法論を考案することである。

問題フレームの方法論を成功させるためには、解決すべきいくつかの課題がある。まず、フレームのカテゴリーを特定する必要がある。一例として、*情報フレームがあり*、*これは*主にデータの取得と表示に関する問題のクラスを表しています。フレームを記述／表現するために使用する*表記法を*定義する必要があります。次に、複雑な問題が与えられた場合、それをどのように問題フレームの集合に*分解*するかを決定する必要があります。そうすれば、個々のフレームは、他のフレームから独立して考え、解くことができる。フレームを解決するための重要なステップは、*フレームの懸念に*対処することです。これは、特定のタイプの問題を解決するために対処する必要がある、各問題タイプの一般的な側面です。

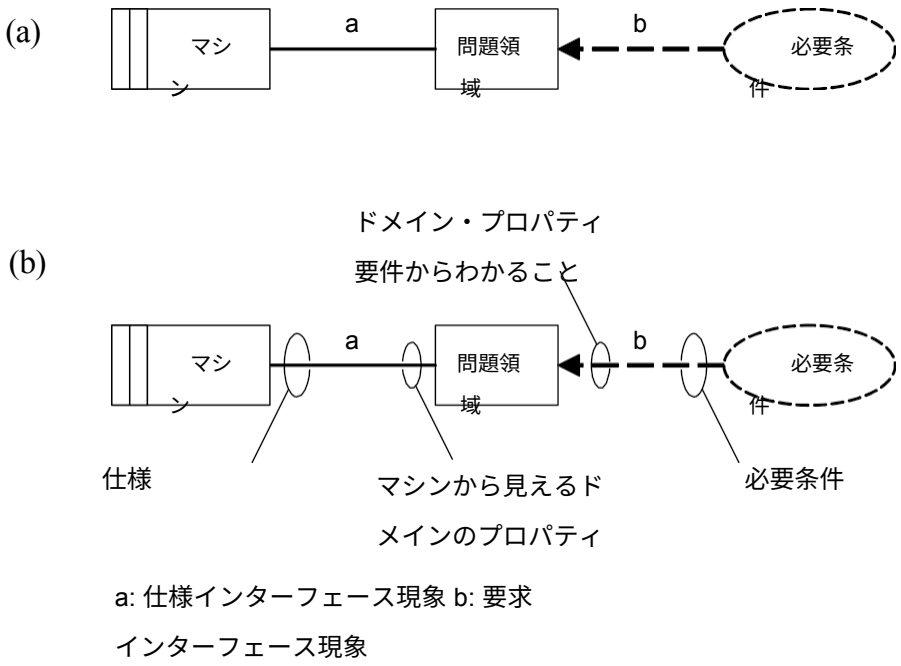


図 3-19: (a) マシンと問題領域。(b) 問題領域、要求、およびマシン間のインターフェース。

最後に、個々の解決策をどのように元の問題の全体的な解決策に*まとめるか*を決定する必要がある。個々のフレームが互いにどのように影響し合うかを判断する必要があり、それらの相互作用の潜在的な矛盾を解決する必要があるかもしれない。

3.3.1 問題フレームの表記

図3-19のように、開発するコンピュータ・システムと問題が存在する現実世界との関係をイメージすることができる。ソフトウェア開発の課題は、汎用コンピュータをプログラミングしてマシンを構築することである。マシンは、問題領域と共有される現象（典型的にはイベントと状態）の集合からなるインタフェース*a*を持つ。例となる現象は、コンピュータのキーボード上のキーストローク、コンピュータ画面に表示される文字や記号、コンピュータと計測器を接続する回線上の信号などである。

マシンの目的は要件によって記述されます。要件は、マシンが問題領域の現象間の何らかの関係を生成し、維持しなければならないことを指定します。例えば、正しいコードが提示されたときにロック装置を解除することや、レストランの小切手に印刷された数字が利用者の消費量を正しく反映することを保証することなどです。

問題領域と機械が共有する現象*aを仕様現象と呼ぶ*。逆に、現象*bは*要求を明確化するものであり、*要求現象と呼ばれる*。a と*bは*重複する場合もあるが、一般的には区別される。要求現象は顧客の要求の主題であり、仕様現象は機械が問題領域を監視し制御できるインタフェースを記述する。

図3-19のような問題図は、問題分析の基礎となるものである。なぜならば、問題分析が何を問題にしているのか、問題を完全に分析するためには何を記述し、推論しなければならないのかを示しているからである。記述の主なトピックは次のとおりです：

- 機械が何をしなければならないかを示す*要件*。要件とは、顧客が問題領域で達成したいこと。その記述は*光学的*である。顧客が選択した*オプション*）。すでに要件を正確に記述している場合もあれば、そうでない場合もあります。たとえば、表2-2の要件REQ1には、ユーザーがシステムに登録する方法が正確に記述されています。
- 各問題領域の関連する特性を記述する*領域特性*。これらの記述は、マシンの行動を理解する必要がある。例えば、セクション1.3.2では、投資アドバイスを提供する有用なシステムを実装するために理解しなければならない金融市場の機能について説明している。
- マシン*仕様*。要件と同様に、これは*選択的*記述である。問題領域とのインターフェースにおけるマシンの望ましい動作を記述する。

金融市場の仕組みを明確に理解しなければ、有用な投資支援システムを開発することはできないだろう。

3.3.2　問題のフレームへの分解

問題解析は、問題領域の種類と領域特性に基づく*問題分解戦略に*依存している。その結果生じる*部分問題は*、他の部分問題から独立して扱われ、これが効果的な関心事の分離の基礎となる。各サブ問題は、それ自身のマシン（仕様）、問題領域、および要求を持つ。それぞれの部分問題は、印刷における色分解のように、完全な問題の*投影である。*

Jackson[2001]は、問題分解の基本単位となる5つのプリミティブな*問題*フレームを特定している。これらは、(*i*)要求動作、(*ii*)命令動作、(*iii*)情報表示、(*iv*)単純作業、(*v*)変換である。これらは以下の点で異なる。

要求事項、ドメインの特性、ドメインの関与（ドメインが制御されているか、アクティブか、不活性か、など）、フレームの懸念事項。これらの問題フレームは、先に2.3.1節で特定した問題タイプに対応する（図2-11参照）。

各フレームには特定の懸念事項があり、それはフレームを解決する際に解決しなければならない一般的な問題の集合である：

(a) *要求される動作フレーム*：(1)制御されるドメインが現在どのように振る舞うか、(2)要件で述べられているドメインの望ましい振る舞い、(3)システムとなるべきマシンで使用されるセンサーによって、ドメインの状態についてマシン（ソフトウェアとなるべきもの）が何を観察できるかを正確に記述すること。

(b) *コマンド動作フレームの懸念事項*：(1)想定されるシステムで可能なすべてのコマンドを特定すること。

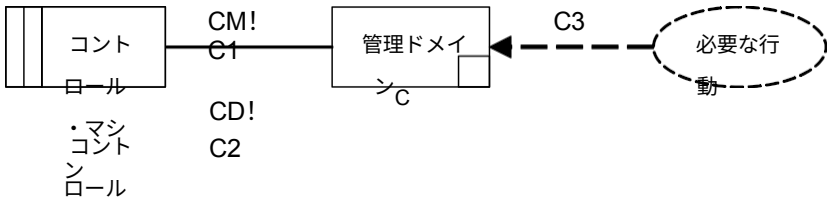


図3-20：*要求動作フレームの*問題フレーム図

そして、(3) ユーザーが現在のシナリオではサポートされていない／許可されていないコマンドを実行しようとした場合、どうすべきか。

(c) *情報表示フレームの問題*：(1)システム・トゥー・ビーで使用されるセンサーによって、マシンが問題領域から観察できる情報、(2)要求によって示される、表示される必要のある情報、(3)表示可能な情報を得るために、観察された生の情報を処理するために必要な変換。

(d) *単純なワークピースのフレームに関すること*：(1)ワークのデータ構造、(2)想定されるシステムで可能なすべてのコマンド；
(3)異なるシナリオの下でサポートまたは許可されるコマンド、(4)現在のシナリオの下でサポート/許可されていないコマンドをユーザーが実行しようとした場合に起こるべきこと。

(e) *変換フレームの関心事*：入力データと出力データのデータ構造を正確に記述すること、
(3) 出力データ構造の対応する要素を得るために、入力データ構造の各要素がどのように変換されるか。

ドメイン特性のより細かい分類を反映した*フレームフレーバー*の識別と分析

フレームの関心事は、要求、仕様、ドメインの記述を作成し、それらを構築されるマシンの正しさの議論に適合させることである。フレームの懸念には、初期化、オーバーラン、信頼性、同一性、完全性が含まれる。初期化に関する*懸念は*、マシンが起動したときに現実世界と適切に同期していることを保証することである。

... 通常は問題フレームにドメインが追加される*フレームのバリエーション*

基本フレームタイプ1：必須動作

このシナリオでは、与えられた条件に従って物理世界の一部の動作を制御するマシンを作る必要がある。

図3-20は、必要な動作フレームのフレーム図である。制御機械は、構築される機械（システム）である。被制御領域は、制御される世界の一部である。被制御領域の振る舞いが満たすべき条件を与える要件を要求振る舞いと呼ぶ。

制御される領域は、そのボックスの右下隅のCで示されるように、因果的な領域である。マシンとのインターフェースは2組の因果現象からなる：C1はマシンによって制御され、C2は被制御領域によって制御される。マシンによって制御されるC1と、制御される領域によって制御されるC2である。マシンは、現象C1によって制御される領域に振る舞いを課し、現象C2はフィードバックを提供する。

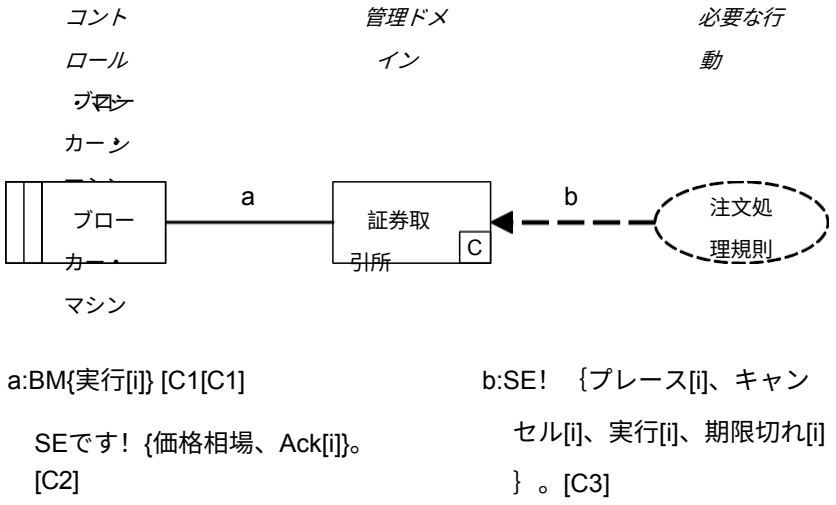


図 3-21：Required Behavior 基本フレームの例：取引注文の処理。

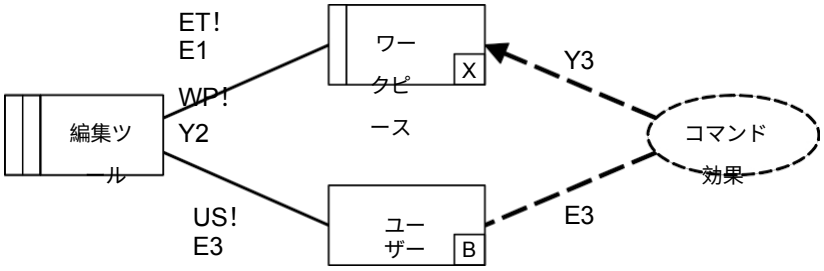


図3-22：*単純ワークフレームの*問題フレーム図。

図3-21に、ある証券会社のシステムが売買注文をどのように処理するかの例を示す。ユーザーが注文を出すと、その注文はブローカーのマシンに記録され、以後マシンは相場価格を監視して、注文の執行条件がいつ満たされるかを決定する。条件が満たされると、例えば価格が指定された値に達すると、管理されたドメイン（証券取引所）に注文の執行が要求される。管理されたドメインは注文を執行し、"注文チケット "として知られる肯定応答を返す。

基本フレームタイプ2：命令行動

このシナリオでは、オペレーターがコマンドを発行して物理世界の一部の動作を制御できるマシンを作る必要がある。

基本フレームタイプ3：情報表示

このシナリオでは、物理世界の一部に関する情報を取得し、それを所定の場所に所定の

形で提示するマシンを作る必要がある。

基本フレームタイプ4：単純なワーク

このシナリオでは、ユーザーがテキストやグラフィックなどのデータ表現を作成、編集、保存できるマシンを構築する必要がある。編集される語彙領域は、メモを取るためのテキスト文書など、設計が比較的単純なものかもしれない。また、ソーシャル・ネットワーキング・ウェブサイトの「ソーシャル・グラフ」の作成と維持のように、非常に複雑な場合もある。ビデオゲームもまた、非常に複雑なデジタル（語彙）ドメインの一例であり、ユーザーがプレイし、さまざまなコマンドを発行することで編集される。

図 3-22 は単純ワークフレームのフレーム図である。

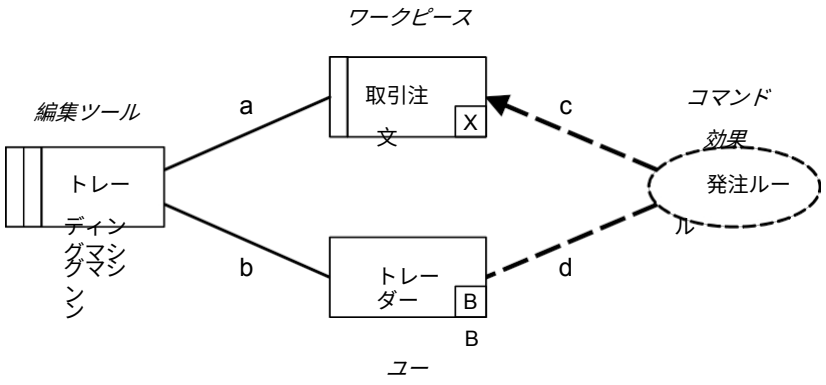


図3-23：シンプルなワークピースの基本フレームの例：取引注文を出す。

例を図3-23に示す。

基本フレームタイプ5：トランスフォーメーション

このシナリオでは、入力文書を受け取り、一定の規則に従って出力文書を生成するマシンを構築する必要がある。例えば、リレーショナル・データベースから検索されたレコードが与えられた場合、タスクはそれらをウェブ・ブラウザで表示するためのHTML文書にレンダリングすることである。

変換フレーム問題の重要な関心事は、入力データと出力データのデータ構造を走査し、それらの要素にアクセスする順序を定義することである。例えば、入力データ構造がバイナリーツリーの場合、それはプレオーダー、インオーダー、ポストオーダーの方法でトラバースされる。

図 3-24 はフレーム分解の背後にある重要な考え方を示しています。複雑なアプリケーション・ドメインに関連する複雑な要件セットとして表される問題が与えられた場合、私たちの目標は、基本的な問題フレームのセットを使用して問題を表現することです。

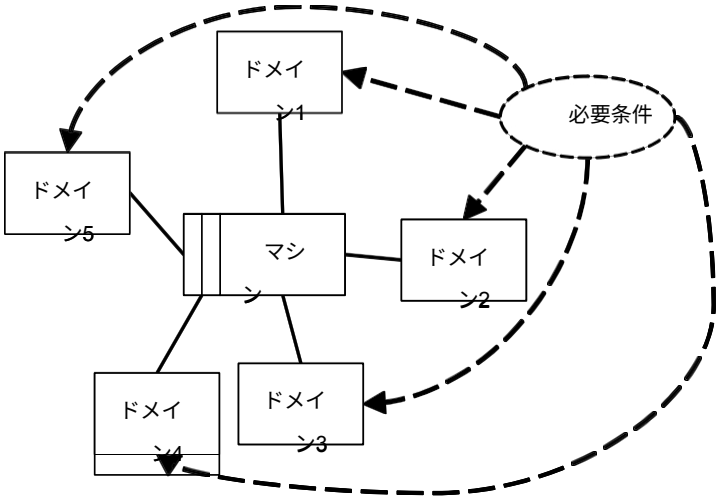
3.3.3　問題フレームの構成

実世界の問題は、ほとんどの場合、単純な問題フレームの組み合わせで構成されている

。問題フレームは、単純な下位問題の理解を達成し、これらの問題フレームに対する解決策（機械）を導き出すのに役立つ。一旦、専門化されたフレームの局所的なレベルで解決策に到達すると、首尾一貫した全体を作るために、統合（または合成）または専門化された理解が必要となる。

2つ以上の単純な問題フレームを組み合わせた標準的な*複合フレームもある。*

(a)



(b)

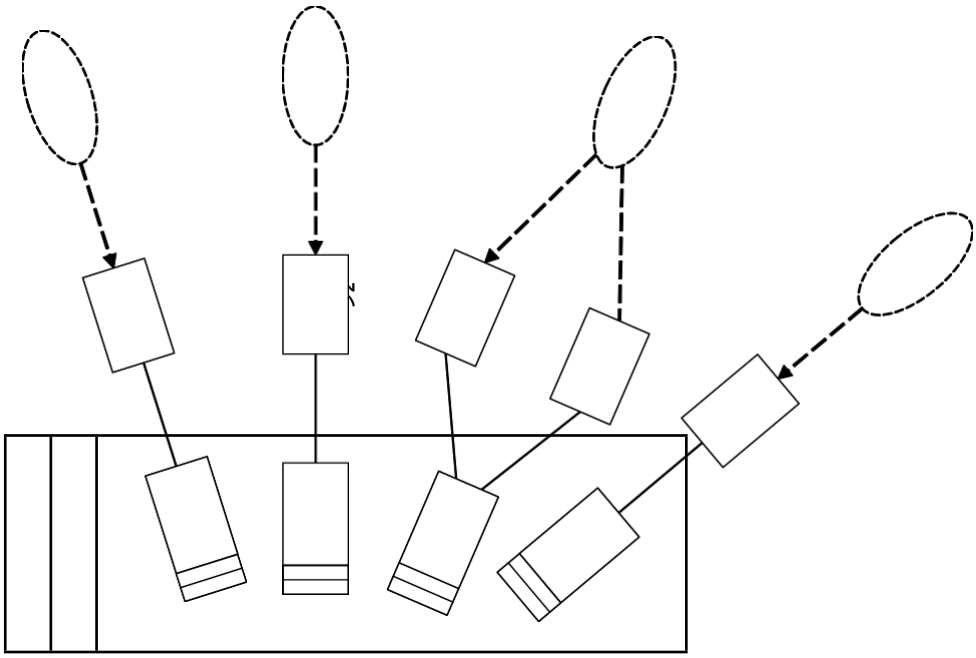


図3-24：フレーム分解の目的は、複雑な問題（a）を基本的な問題フレーム（b）の集合として表現することである。

3.3.4 モデル

ソフトウェア・システムは世界を表現することができるが、これは常に理想化されたものである。例えば、私たちのロック・システムでは、（ランタイムに感知/獲得されるのとは対照的に）内蔵された知識は以下の通りです：有効なキーが入力され、暗さを感知したら、ライトをつける。

3.4目標の特定

"正しさ "こそが最も重要な品質であることは明らかだ。もしシステムがやるべきことをやらなかったら、その時点ですべてが台無しになってしまう。
それ以外のことはほとんど重要ではない"-ベルトラン・メイヤー

ゴール指向の要求工学の基本的な考え方は、システム全体の総体的なゴールから出発し、それを段階的にゴール階層に*洗練していくことである。*

AND-ORリファイン ...

問題フレームはゴールと関連づけることができる。ゴール指向アプローチは、問題フレームアプローチが異なる問題クラスを区別するように、異なる種類のゴールを区別する。問題を基本フレームに分解した場合、これをゴール階層のAND-refinementと言い換えることができる：システム要求ゴールを満たすためには、（各基本フレームの）個々のサブゴールを満たす必要がある。

プログラムされているとき、プログラムはその目標を*明示的にではなく暗黙的に*「知っている」ので、それを他のコンポーネントに伝えることはできない。セクション9.3で見るように、目標を他者に伝えるこの能力は、オートノミック・コンピューティングにおいて重要である。

A=武装状態、B=消灯状態、C=ユーザー識別状態、
D=昼光
(ゴールは摂動後に到達すべき平衡状態である。) 初期状態：
A∧B、ゴール状態：¬A∧¬B。
可能なアクション： α-setArmed; α^{-1} -setDisarmed; β-setLit; β^{-1} -setUnlit αの前
提条件$^{-1}$： C; βの場合：D
A∧¬Bを達成するために、許可されたアクションを適用する計画を立てる必要がある。

プログラムゴール、マルチフィデリティアルゴリズム、MFAの「ファジー」ゴールも参照

[Satyanarayanan & Narayanan, 2001]。http://www.cs.yale.edu/homes/elkin/ (Michael Elkin)

サーベイ「分散近似」、Michael Elkin著。*ACM SIGACT News*, vol. 36, no. 1, (Whole Number 134), March 2005 .http://theory.lcs.mit.edu/~rajsbaum/sigactNewsDC.html

この形式的表現の目的は、プログラムを自動的に構築することではなく、むしろプログラムがその仕様を満たしていることを立証できるようにすることである。