第1章

知識をかみ砕く

数年前、私はプリント基板（PCB: Printed-Circuit Board）設計用の専門ソフトウェアツールを設計することになった。しかし、1つ問題があった。私は電子機器について何も知らなかったのだ。もちろん、PCB設計者から話は聞けたが、彼らの話にはものの3分で頭がクラクラしてくるのが常だった。このソフトウェアを作成するのに十分なくらい電子機器について理解するには、どうすればよいだろう？ 納期が来る前に電子技術者になることなど、できるはずもないのだ！

ソフトウェアがすべきことを、PCB設計者に正確に説明してもらおうとしたが、これがまずかった。彼らは素晴らしい回路設計者なのだが、彼らの考えているソフトウェアはたいてい、ASCIIファイルを読み込み、それをソートした上で注釈をつけ、そして帳票を作成する、というものだった。これでは、彼らの求める生産性の飛躍的向上につながらないことは明白だった。

最初の数回の打ち合わせには失望させられたが、彼らが必要としていた帳票にかすかな希望を託すことができた。帳票にはいつでも「ネット」と、それに関するさまざまな詳細を出力することになっていた。このドメインにおけるネットとは、要するに、PCB上で任意の数のコンポーネントを接続することができる導体のことで、接続されたものすべてに電気信号を伝えるのだ。我々はドメインモデルの最初の要素を手に入れたのである。

図1.1

ソフトウェアに対する彼らの要求について一緒に話し合いながら、私は図を描き始めた。シナリオのウォークスルーには、オブジェクト相互作用図のようなものを使用した。

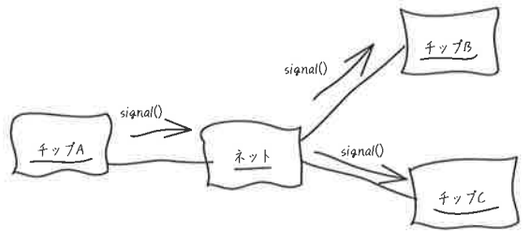

図1.2

PCBエキスパート1：コンポーネントはチップでなくてもよいのです。
開発者（私）：では、単に「コンポーネント」と呼んだ方がよいですか？
エキスパート1：我々は「コンポーネントインスタンス」と呼んでいます。同じコンポーネントが多数ある場合もありますから。
エキスパート2：「ネット」を表しているこの四角が、コンポーネントインスタンスとそっくりですね。
エキスパート1：我々の表記法を使っているわけではありませんからね。何でも四角で表すのではないですか？
開発者：残念ながら、そうなのです。この表記法については、もう少し説明した方がよさそうですね。

彼らはしきりに私の言うことを訂正し、それによって私にも知識が身につき始めた。用語法にあった不一致やあいまいさ、あるいは、技術的な見解の相違を一緒になって解決し、その過程で彼らも学んだ。その結果、彼らは物事をより厳密に矛盾なく説明してくれるようになったのだ。こうして、我々は一緒になってモデルの開発を始めたのである。

エキスパート1：信号（signal）がレフデスに届くと言うだけでは不十分で、ピンもわからないといけません。
開発者：レフデスとは何ですか？
エキスパート2：コンポーネントインスタンスと同じものです。レフデスというのは、我々が使っている特別なツールでの呼び名です。
エキスパート1：とにかく、ネットは、あるインスタンスの特定のピンを別のインスタンスの特定のピンに接続します。

開発者：つまり、ピンは1つのコンポーネントインスタンスだけに属し、1つのネットにしか接続されないということですか？
エキスパート1：ええ、その通りです。
エキスパート2：また、すべてのネットにはトポロジーがあります。トポロジーとは、ネットの要素がどう接続されるかを決定する配置の仕方です。
開発者：なるほど、これはどうでしょう？

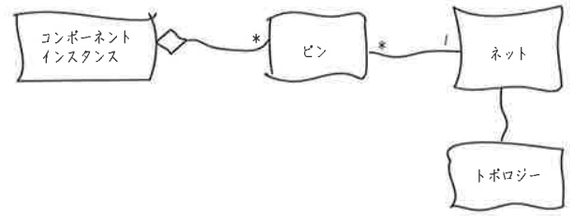

図1.3

調査の焦点を絞るため、当面、ある1つの機能に限って調べることにした。「プローブシミュレーション」とは、信号の伝搬を追跡し、設計において、ある種の問題がありそうな場所を検出する機能である。

開発者：**ネット**によって、そこに接続されているすべての**ピン**へ信号が送られる仕組みはわかりましたが、その先はどうなるのですか？ **トポロジー**が関係するのでしょうか？
エキスパート2：いいえ。コンポーネントが信号をプッシュ（push）して通過させます。
開発者：チップ内部のふるまいをモデル化するのは無理ですね。あまりにも複雑すぎます。
エキスパート2：モデル化する必要はありませんよ。単純化できます。ある**ピン**から他のピンへコンポーネントを通過するプッシュの一覧だけあればよいのです。
開発者：こういう感じですか？

[相当な試行錯誤をしながら、我々は一緒にシナリオの概要を描いた。]

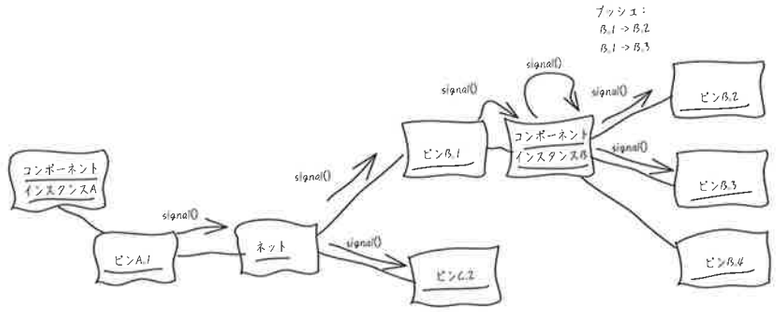

図1.4

開発者：でも、この処理をすることによって、いったい何を知りたいのですか？

エキスパート1：見つけたいのは、信号の長時間の遅延、例えば、2～3回以上ホップしている信号の経路です。大まかな目安ですけどね。経路が長すぎると、信号はクロックサイクル内に届かない可能性があります。

開発者：3回以上のホップですか…。ということは、経路の長さを計算する必要がありますね。何を1回のホップと見なすのですか？

エキスパート2：信号が**ネット**を通過することです。それが1回のホップとなります。

開発者：では、ホップの回数を渡して、**ネット**がホップ回数を加算すればよいですね、こんなふうに。

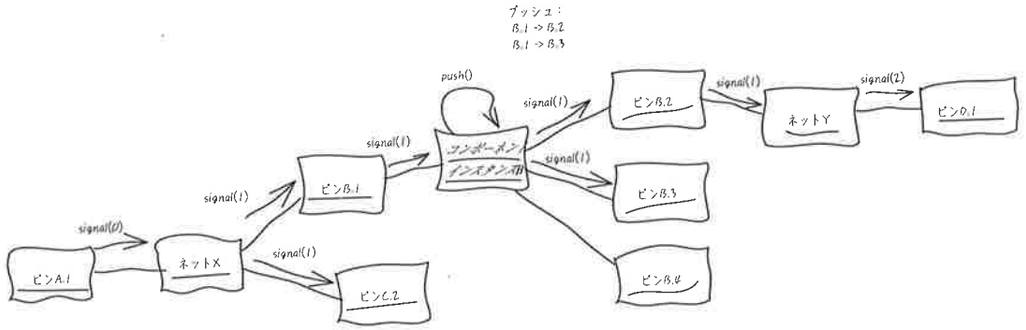

図1.5

開発者：1つわからないのは、「プッシュ」がどこから生じるのかです。すべての**コンポーネントインスタンス**に対して、プッシュのデータを格納するのですか？

エキスパート2：プッシュは、あるコンポーネントのすべてのインスタンスで同じになります。

開発者：それなら、コンポーネントタイプによってプッシュが決定されますね。どのインスタンスでも同じですか？

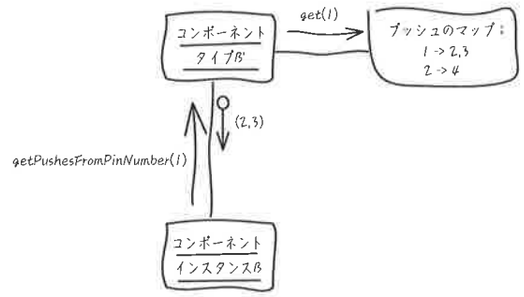

図1.6

エキスパート2：意味がよくわからないところもありますが、各コンポーネントでプッシュを格納する際には、そういう感じになると思います。

開発者：すみません、詳細に立ち入りすぎました。深く考えすぎていたもので…。それでは、**トポロジー**はどこで登場するのですか？

エキスパート1：プローブシミュレーションには使用しません。

開発者：では、ここは省いていいですね？ その機能を取り上げる時に再考しましょう。

こうして議論は進んだ（実際には、ここに示したよりはるかに多くつまずいたが）。ブレインストーミングを行ってはモデルを改良し、質問してはそれに対して説明してもらった。ドメインに関する私の理解が増し、問題解決にあたってモデルがどのような役割を果たすのかを彼らが理解するにつれて、モデルは成長した。この初期モデルを表すクラス図を、次に示す。

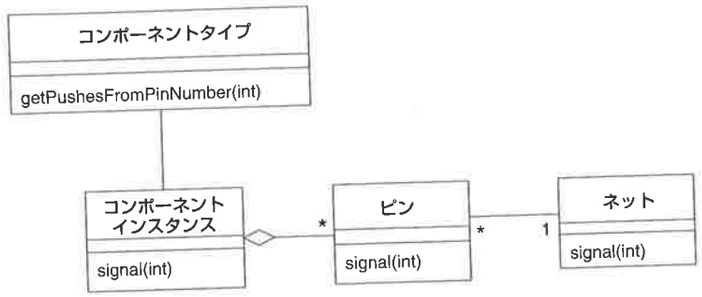

図1.7

こうしたことを、さらに2日ほど行ったところで、試しにコードを書いてみる程度には理解できた気がした。そこで、自動化テストフレームワークを使って、非常にシンプルなプロトタイプを作成した。インフラストラクチャ部分はすべて省いた。したがって、永続化もなければ、ユーザインタフェース（UI）もない。これでふるまいだけに集中することができるようになった。その後、ほんの数日のうちに、簡単なプローブシミュレーションを実演することができた。ダミーデータを使って、加工していないテキストをコンソールに表示しただけだったが、Javaオブジェクトを使用して経路の長さを実際に計算したことには変わりない。それらのJavaオブジェクトには、ドメインエキスパートと私が共有するモデルが反映されていた。

このプロトタイプが具体的であったことによって、モデルが何を意味し、実際に機能するソフトウェアにどう関係しているかを、ドメインエキスパートがより明確に理解できるようになった。それ以降、モデルに関する我々の議論は活発になった。それは、新しく獲得した知識を、私がどのようにモデルに組み込み、さらにはソフトウェアに組み込んでいるかということを、彼らが理解したからである。しかも、彼らは自身の考えを評価するための具体的なフィードバックを、プロトタイプから得ていたのだ。

当然、モデルはここに示したよりもずっと複雑になったが、そのモデルに組み込まれていたのは、解決しようとしていた問題と関連の深い、PCBのドメインに関する知識だった。また、説明に使われた多くの同義語や、ちょっとした揺れが統一された。コンポーネントの実際のディジタル特性など、技術者が理解しているものでも、ドメインと直接関係がないものはいくつも除外した。私のようなソフトウェアの専門家がこの図を見れば、数分でソフトウェアの内容を理解できるようになった。新しい情報を整理して素早く習得したり、重要なものとそうでないものをより正確に推測したり、PCB技術者とより円滑にコミュニケーションしたりするためのフレームワークができていたのである。

技術者が、自分たちの必要とする新しい機能を説明する際には、オブジェクト間の相互

作用に関するシナリオを、私に対して机上で示してもらった。モデルオブジェクトを用いて、重要なシナリオを最後まで実現できなかった時には、新しいモデルオブジェクトでブレインストーミングしたり、以前のオブジェクトモデルを変更したりして知識をかみ砕いたのだ。モデルを改良すると、コードも一緒に進化した。数か月後、PCB技術者は自分たちの期待を上回る、機能の豊富なツールを手にしていた。

効果的なモデリングの要素

我々の行った次のような事柄が、前述の成功につながった。

1. **モデルと実装を結びつける。**あの荒削りなプロトタイプによって、本質的な結びつきが早い時期に作り出され、それが以後のイテレーションを通してずっと維持された。
2. **モデルに基づいて言語を洗練させる。**当初、エンジニアはPCBの基本的な問題を私に説明しなければならず、私もクラス図が意味するものを説明しなければならなかった。しかし、プロジェクトが進むにつれ、我々全員がモデルから用語をそのまま取り出し、モデルの構造と矛盾しない文章にまとめて、通訳しなくても正確に理解し合えるようになった。
3. **知識豊富なモデルを開発する。**オブジェクトには、ふるまいと、守るべきルールがある。モデルは単なるデータスキーマではなく、複雑な問題を解決するのに欠かせないものだった。モデルはさまざまな種類の知識をとらえていたのだ。
4. **モデルを蒸留する。**モデルが完成していくにつれて、重要な概念が追加されていった。だが、それと同様に重要なのは、役に立たない、あるいは核心でないとわかった概念が、取り除かれていったことである。不要な概念が必要な概念と結びついている時には、本質的な概念を切り分け、その他の不要なものを削除できるように新たなモデルを探し出した。
5. **ブレインストーミングと実験を行う。**言語をスケッチやブレインストーミングと組み合わせることで、我々の議論の場はモデルの実験室に変わった。そこではモデルの実験用バリエーションを数多く使用し、試し、使えるかどうかを判断することができた。チームがシナリオを詳細に検討する際には、会話で使われる表現自体が、提案されたモデルを実現できるかどうかについて調べる簡易的なテストになった。表現が明確でわかりやすいか、それともぎこちないかを、聞いてすぐに察知することができたからである。

ブレインストーミングと大量の実験が持つ創造性は、モデルに基づいた言語を通じて活用され、実装を通じたフィードバックループによって鍛えられたものだ。この創造性によって、知識豊富なモデルを見つけ出し、そのモデルを蒸留できるようになる。こうして**知識をかみ砕くこと**で、チームの持っている知識が価値のあるモデルへと変わるのだ。

知識のかみ砕き

金融アナリストは数字をかみ砕く。多くの詳細な数字をふるいにかけ、根底にある意味を求めて何度も組み合わせ直し、本当に重要なものを明らかにしてくれるシンプルな説明を探す。こうした理解こそが、金融上の意思決定を行う際に、基礎となり得るのである。

有能なドメインモデラは知識をかみ砕く。彼らは情報の奔流の中に重要な一滴を探る。次から次へと考えをまとめてみて、大部分の意味を理解できるシンプルな見方を見つけようとする。多くのモデルが試されて、却下されたり変更されたりする。すべての詳細を解明する抽象概念が、一式現れてくれば成功だ。こうした蒸留により、ビジネスと最も関連の深いことが判明した特定の知識が、厳格に表現される。

知識のかみ砕きは独りで行う作業ではない。開発者とドメインエキスパートからなるチームが共同して行うもので、たいていは開発者が率いる。一緒になって、彼らは情報を引き出し、それをかみ砕いて役に立つかたちにする。その材料は、ドメインエキスパートの頭の中や既存システムのユーザ、関連するレガシーシステムや、同一ドメインの別プロジェクトにおける技術チームの過去の経験などから得られる。それは、プロジェクト用に作成されたドキュメントや、ビジネスで使用されるドキュメント、さらにおびただしい量の話し合いというかたちでもたらされる。初期のバージョンやプロトタイプでの経験がチームにフィードバックされて、解釈が変わっていく。

旧来のウォーターフォール手法では、ビジネスエキスパートがアナリストに話をして、アナリストがかみ砕き、抽象化して、その結果をソフトウェア作成者であるプログラマに伝える。このアプローチがうまくいかないのは、フィードバックが完全に欠けているからだ。アナリストはモデルを作成する全責任を負っているが、基になるのはビジネスエキスパートから得た情報だけだ。プログラマから学んだり、ソフトウェアの初期バージョンを体験したりする機会は全くない。知識は一方向に少しずつ流れるだけで、蓄積されないのである。

イテレーティブなプロセスを採用しているのに、抽象化を行っていないせいで、知識の積み重ねに失敗しているプロジェクトもある。開発者は、必要とする機能をエキスパートに説明してもらった上で、それを構築する。その結果をエキスパートに示し、次に何をすべきかを尋ねる。プログラマが習慣的にリファクタリングを実行すれば、ソフトウェアを、継続的に拡張できる程度にはクリーンな状態に保てる。だが、プログラマがドメインに興味を持っていなければ、プログラマはアプリケーションが何をすべきかだけを調べ、その背後にある原理には注意を払わない。そうしたやり方でも、使えるソフトウェアを構築することはできるが、そういうプロジェクトは、既存機能の必然的な帰結として強力な新機能が現れるという段階には、決して到達することがない。

優秀なプログラマは、当然、抽象化することから始め、より多くの処理を行うことのできるモデルを開発する。ただし、その背景にあるのが技術だけで、ドメインエキスパートとの共同作業がない場合、その概念は素朴なものにしかならない。そのように知識が浅薄でも、基本的な処理を実行するソフトウェアを作り出すことはできるが、そのように作られたソフトウェアが、ドメインエキスパートの考え方と深く結びつくことはない。

チームメンバ間のやりとりは、メンバ全員がモデルを一緒にかみ砕くことで変化する。ドメインモデルを絶えず改良すると、開発者は、自分が力を貸しているビジネスにおける重要な原理を習得するように強いられ、機械的に機能を作り出すことがなくなる。一方、ドメインエキスパートは、自分の知っていることを蒸留して本質を抽出するように強いられることによって、しばしば自分の理解を精緻にし、ソフトウェアプロジェクトが必要とする概念の正確さを理解するようになる。

こうしたことすべてによって、チームメンバはよりうまく知識をかみ砕けるようになる。チームメンバは、無関係なものを選り分け、より一層役立つかたちにモデルを作り直す。アナリストとプログラマによって情報が与えられているので、そのモデルはきれいに構成され、また抽象化されており、それによって実装は大いに助けられる。ドメインエキスパートもモデルに対して情報を提供しているから、モデルにはビジネスに関する深い知識が反映されている。その抽象がビジネスの真の原理なのだ。

改善されるにつれて、モデルは、プロジェクトの中を流れ続ける情報を体系化するためのツールとなっていく。モデルは、要件を分析することに焦点を合わせているが、プログラミングや設計とも深く相互作用する。好循環が起きれば、こうしたモデルの性質によってチームメンバのドメインに対する洞察が深まり、状況がより明確に理解できるようになって、モデルがさらに精緻化される。このようなモデルは、決して完成することがない。代わりに、進化していくのだ。ドメインを解明するにあたって、モデルは実用的で役に立つものでなければならない。また、アプリケーションがシンプルに実装され、理解しやすくなるように、モデルは厳密でなければならない。

継続的学習

ソフトウェアを書き始める時、我々は対象を十分に理解しているわけではない。プロジェクトに関する知識は断片的で、多くの人々やドキュメントの間に散在しており、他の情報とも混ざっているから、どの知識が本当に必要なのかということさえもわからない。技術的にそれほど難しくなさそうなドメインには、だまされるかもしれない。つまり我々は、自分たちがどれほど理解していないかを認識していないのだ。こうした無知のせいで、誤った想定をしてしまうのである。

一方、すべてのプロジェクトから知識は流出している。何かを習得した人は他の所へ移動する。再編成によってチームはばらばらとなり、知識が再び断片化してしまう。重大なサブシステムがアウトソーシングされると、コードは納品されても、知識はついてこない。典型的な設計アプローチの場合、コードとドキュメントは、苦労して手に入れた知識を使える形式で表現しないから、口頭での伝承が何らかの理由で途切れたら、知識は失われてしまう。

高度に生産的なチームは、自分たちの知識を意識的に育てて、**継続的学習**（Kerievsky 2003）を実践する。開発者にとって、これが意味するのは、（本書にあるような）一般的なドメインモデリングのスキルと合わせて、技術的な知識を向上させるということだ。ただし、その時に取り組んでいる具体的なドメインについて真剣に学習することも、ここに含まれる。

こうして自ら学んだチームメンバは、最も重要な領域を含む開発作業に集中するにあたって、メンバの中で信頼できるコアとなる（これに関する詳細は、第15章を参照）。このコアチームの頭の中に知識が蓄積されることで、彼らは知識をよりうまくかみ砕けるようになる。

ここで立ち止まって、自問して欲しい。PCB設計プロセスについて、何か学ぶところがあっただろうか？ この例はドメインを表面的にしか扱わなかったが、ドメインモデルについての議論からは、何かしら学んだはずである。私自身は大いに勉強になった。PCB技術者になる方法を学んだのではない。目標はそこにはなかった。私が学習したのは、PCBエキスパートと話をして、アプリケーションに関係する重要な概念を理解し、構築している内容が正しいかどうかのチェックを行うことである。

実のところ、我々のチームは、最終的にプローブシミュレーションの開発の優先順位が低いことを発見し、この機能は結局開発しないことになった。同時に、モデルの中から、信号をプッシュしてコンポーネントを通過させることと、ホップをカウントすることについての理解をとらえていた部分も除去された。アプリケーションのコアは別にあることがわかったので、その側面を舞台の中央へ持ってくるようにモデルが変更されたのである。

ドメインエキスパートがより多くを学んだことで、アプリケーションの目標が明確になったのだ（第15章で、この問題を掘り下げて説明する）。

機能の開発が取り止めになったとはいえ、初期の作業は不可欠だった。主要なモデル要素がそのまま残ったということもあるが、それよりも重要なのは、初期の作業が知識のかみ砕きというプロセスを始動させ、それによって以降の全作業が効果的に行われたということだ。すなわち、チームメンバ、開発者、ドメインエキスパートが同じように知識を得て、共有された言語が芽を出し、実装を通してフィードバックループが完結した。発見の旅は、どこかから始めなければならないのだ。

知識豊富な設計

PCBの例で示したように、モデルによってとらえられる知識は、「名詞を見つける」ことに留まらない。ビジネスの活動やルールも、ドメインに含まれるエンティティと同じように、ドメインにとって中心的なのだ。このように、どんなドメインにもさまざまな種類の概念がある。知識をかみ砕くことによって、この種の洞察を反映したモデルが生み出される。モデルの変化と平行して、そのモデルを表現するべく、開発者は実装をリファクタリングする。それによって、新しい知識をアプリケーションで使用できるようになる。

エンティティや値を超えてその先に行こうとする、このような動きに伴った時こそ、知識のかみ砕きは力を発揮できる。複数のビジネスルールの間に、実は矛盾があるかもしれないからである。通常、ドメインエキスパートは、自分の頭の中で起きているプロセスがいかに複雑かを意識することなく、仕事をする中でこれらのルールをすべて調べて矛盾を調整し、常識で考えて隔たりを埋めている。だが、こういうことはソフトウェアにはできない。ソフトウェアエキスパートと密接に協力する中で、知識をかみ砕くことによって初めて、ルールが明確となり、具体化されて、折り合いがつけられるか、あるいはスコープの対象外とされるのである。

例 1.1 ——隠された概念を引き出す

非常に単純なドメインモデルから始めよう。これは、ある船の一回の航海に対して、貨物を運んでもらうように予約するアプリケーションの土台になり得るものだ。

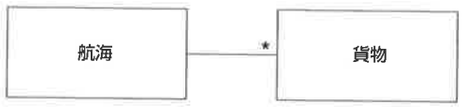

図1.8

予約アプリケーションの責務は、各**貨物**（Cargo）を**航海**（Voyage）と関連づけ、その関係を記録し、追跡することだと言える。ここまでは順調だ。アプリケーションコードのどこかに、次のようなメソッドが入るだろう。

```
public int makeBooking(Cargo cargo, Voyage voyage){
   int confirmation = orderConfirmationSequence.next();
   voyage.addCargo(cargo, confirmation);
   return confirmation;
}
```

土壇場でのキャンセルはよくあることだから、輸送業界の標準的な慣習では、1回の航海でその船舶が運搬できる量より多く貨物を受けつける。これを「オーバーブッキング」という。収容能力の110%を予約するなど、積載量に対して単純なパーセンテージを使用することもあるし、主要顧客や特定の貨物を優先して、より複雑なルールを適用する場合もある。

これは、輸送業界で働く人なら誰もが知っている輸送ドメインの基本的戦略だが、ソフトウェアチームの技術担当者全員が把握しているとは限らない。

要求定義書の中には、次の一文が書かれている。

> 10%のオーバーブッキングを認める。

クラス図とコードは次のようになる。

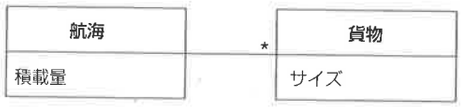

図1.9

```
public int makeBooking(Cargo cargo, Voyage voyage){
   // 最大予約＝航海の積載量×1.1
   double maxBooking = voyage.capacity() * 1.1;
   if ((voyage.bookedCargoSize() + cargo.size()) > maxBooking)
      return -1;
```



```
   int confirmation = orderConfirmationSequence.next();
   voyage.addCargo(cargo,confirmation);
   return confirmation;
}
```

ここでは、重要なビジネスルールが、ガード節としてアプリケーションメソッドの中に隠されている。第4章で、オーバーブッキングのルールをドメインオブジェクトへ移すよう導くレイヤ化アーキテクチャの原理を取り上げるが、ここでは、この知識をより明示的にし、プロジェクトに携わる人全員が使えるようにする方法に専念しよう。そのためには、似たような解決策がある。

1. このままでは、たとえ開発者が手助けしても、ビジネスエキスパートがコードを読んで、ルールが正しいかどうかを検証できそうもない。
2. ビジネスについて知らない技術者が、要求定義書の記述をコードと結びつけることは困難である。

ルールがより複雑になれば、問題はさらに大きくなる。

この知識をより適切にとらえるよう、設計を変更することができる。オーバーブッキングのルールは、ポリシーである。**ポリシー**とは、ストラテジーとして知られている設計パターンの別名だ（Gamma他 1995）。通常、このパターンが使われるのは、さまざまなルールを置き換える必要のある場合だが、現時点でわかっている限り、その必要性はない。しかし、ここでとらえようとしている概念は、ポリシーの意味と適合している。このこともまた、ドメイン駆動設計では重要な動機となる（第12章「デザインパターンをモデルに関係づける」を参照）。

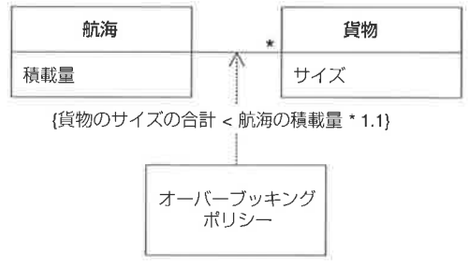

図1.10

コードは次のようになる。

```
public int makeBooking(Cargo cargo,Voyage voyage){
   // オーバーブッキングポリシーに照らして、貨物が航海で許可されるか
   if (!overbookingPolicy.isAllowed(cargo, voyage)) return -1;
   int confirmation = orderConfirmationSequence.next();
   voyage.addCargo(cargo,confirmation);
   return confirmation;
}
```

●例 1.1 ——隠された概念を引き出す

新しい**オーバーブッキングポリシー**クラスには、次のメソッドがある。

```
public boolean isAllowed(Cargo cargo, Voyage voyage){
   return (cargo.size() + voyage.bookedCargoSize()) <=
      (voyage.capacity() * 1.1);
}
```

オーバーブッキングというものがはっきり識別できるポリシーであることは、誰にとっても明白であり、そのルールの実装は明示的で独立している。

ところで、**私は、こうした手の込んだ設計をドメインのあらゆる詳細に適用することを推奨しているわけではない。**第15章「蒸留」では、重要なことに集中して、それ以外のものをすべて最小限にするか分離するための方法を掘り下げる。この輸送ドメインの例は、ドメインモデルとそれに対応する設計が、知識を確固たるものにして共有するのに使えることを示すために用いた。より明示的な設計には、次のような利点がある。

1. この段階まで設計を進められる頃には、プログラマやその他の関係者の誰もが、オーバーブッキングの本質を、単なるわかりにくい計算ではなく、明確で重要なビジネスルールだと理解するようになっているだろう。
2. プログラマは、ビジネスエキスパートに技術的な成果物を見せることができる。コードさえも見せられるのだ。こうした成果物は、ドメインエキスパートに（開発者の手助けがあれば）理解できるはずで、それによってフィードバックループが完結する。

深いモデル

役立つモデルがすぐに見つかる表層にあることはめったにない。ドメインとアプリケーションがやるべきことを我々が理解するようになるにつれて、たいていの場合、表層にあり最初は重要と思われたモデル要素を廃棄するか、それらのモデル要素を違った視点で見ることになる。すると、最初は考えつかなかった、それでいて問題の核心を突くような、うまい抽象が現れてくるのだ。

前述した輸送ドメインの例がおおよそ基にしたのは、コンテナ輸送システムである。これは、本書を通じていくつもの例で利用するプロジェクトの1つである。本書で示す例は、輸送業のエキスパートでなくても理解できるようにしてある。しかし、実際のプロジェクトでは、継続的学習によってチームメンバもドメインの複雑さに対応できるようになるのであり、有用で明確なモデルを作るにあたって、ドメインとモデリングテクニックの両面においてさらなる洗練が求められることも多い。

例に挙げたプロジェクトの場合、貨物の予約行為によって輸送が始まるため、貨物や輸送日程などを記述できるようにするモデルを開発した。これは必要であったし、役に立ちもしたが、ドメインエキスパートは満足していなかった。我々が見落していた、彼らなりのビジネスの見方があったのである。

結局、何か月も知識をかみ砕いた後で我々が気づいたのは、貨物の取り扱い、すなわち物理的な荷積みと荷下ろしや、ある場所から別の場所への移動などは、ほとんど下請け会社や社内の業務担当者によって行われているということだった。輸送業エキスパートから見れば、関係者の間で連続して責任が移動するということだったのだ。そうした荷主から地元の輸送会社へ、さらに輸送手段を複数経由して最終的な荷受人までという法的かつ実務的な責任の移動を、1つのプロセスが管理していた。貨物が倉庫に置かれたまま、重要な手続きが行われることもよくあった。あるいは、貨物の物理的移動を伴うような複雑な手続きが行われても、それが輸送会社のビジネス上の意思決定とは関係ないこともあった。重要視されるようになったのは、輸送日程の手配よりも、船荷証券などの法律書類や支払い免除につながるプロセスだったのだ。

輸送業に対する見方がこうして深まったことにより、輸送日程オブジェクトが削除されることこそなかったが、モデルは抜本的に変化した。輸送に関する我々の考え方は、コンテナをある場所から別の場所へ動かすことから、貨物に対する責任を、ある当事者から別の当事者へ移動することに変わった。この責任の移動を扱う機能は、荷積み業務へとぎこちなく加えられることはなくなった。代わりにこの機能は、そうした業務と責任の間にある重要な関係性を理解することで得られたモデルによって支えられるようになった。

知識のかみ砕きは探究だから、最終的にどこへ行き着くかはわからないのだ。