SEC-FM1-A-1

実務家のための形式手法

厳密な仕様記述を志すための形式手法入門

厳密な仕様記述入門

2013 年 3 月

IPA 独立行政法人情報処理推進機構
Information-technology Promotion Agency, Japan

掲載されている会社名・製品名などは、各社の登録商標または商標です。

Copyright© Information-Technology Promotion Agency, Japan. All Rights Reserved 2013

推薦の言葉

この本は、独立行政法人 情報処理推進機構 (IPA) ソフトウェアエンジニアリングセンター (SEC) において、ここ数年間にわたって高品質高信頼システムの開発技術に関して複数の作業部会(WG)で検討してきた中で、「厳密な仕様記述 WG」がとりまとめた一つの成果物です。ソフトウェア開発においては、仕様が重要であるということは広く認識されているところではありますけれども、仕様をきちんと記述して、それに基づいてソフトウェア開発プロセスを遂行しているかとなると、なかなか実践できていない場合も珍しくありません。この本は、仕様をできるだけ曖昧さなく明確に記述するためのガイドについて述べています。従来あまり触れられていないような観点で仕様を厳密に書くことの意義について述べています。

従って、この本では記述ということにこだわっています。日本人が好む「以心伝心」がコンピュータと人間との間で成り立つとは誰も期待しないでしょう。人間同士の間でも「以心伝心」が幻想にすぎないことは、読者の皆様も日常経験しているのではないでしょうか。人間の知的な共同作業であるソフトウェア開発では、さまざまな立場の開発者間で、また、開発者と発注者あるいはユーザとの間で、複雑で多様な情報共有や意思疎通を伴います。そこでは「以心伝心」などと悠長なことは言っておられません。

厳密に記述された仕様が、ソフトウェア開発プロセスにおいて有用であり大きな役割を果たすことは、本 WG がとりまとめた調査報告書「厳密な仕様記述における形式手法成功事例調査報告書」でも明らかにされています。

この本が読者として第一に想定しているのは、「現場で先輩に文書や仕様書の書き方を聞いてもなんだかよく分からずに悩んでいる、２０代後半から３０代前半のリーダークラスの技術者」だそうです。もちろん、ソフトウェア開発に当事者として責任感と高い意識を持っている方ならどなたでも、本書を読めば、ソフトウェア開発、特に、上流工程における仕様記述に関して、理解を深めることができて、ソフトウェア開発において

有益な知識や認識と、役に立つ技術としての形式手法の存在を知ることができるでしょう。

とはいえ、形式手法を普及させることや習得して頂くことが、この本の目的ではありません。この本を読んで、品質の高いソフトウェアを効率よく開発することを目指せば必然的に形式手法に辿り着くらしいということを認識し、形式手法に基づくソフトウェア開発に関心を持って頂けたなら幸いです。

上述の「厳密な仕様記述における形式手法成功事例調査報告書」は、IPA のホームページ上で公開されています。この調査報告書の姉妹編あるいは副読本として、この本を位置付けることができます。したがって、これらを両方ご覧頂くことを強くお勧め致します。また、IPA/SEC における関連作業部会(WG)である人材育成 WG（旧、形式手法人材育成 WG）が取りまとめた報告書や、全国各地で開催してきた形式手法導入セミナーで使用した教材をもとに作成した実務家のための形式手法教材「厳密な仕様記述を志すための形式手法入門」も IPA のホームページ上で公開されておりますので、これらも併せてご覧頂いて、形式手法に基づく心地良いソフトウェア開発の世界を是非とも垣間みて頂きますよう御案内申し上げます。

荒木啓二郎
九州大学大学院 システム情報科学研究院 教授

まえがき

開発現場ではおそらく日本で初めて、形式手法というものでシステムの仕様を書き、驚いたことがあります。それは、形式手法を使う以前に、ただ仕様記述言語で要求仕様を記述しようとするだけで、次々と曖昧な点や間違いが発見されたからです。このシステムは TradeOne と呼ばれる証券業務用パッケージ・システムでした。

次に開発した、おサイフケータイに使われているモバイル FeliCa チップのファームウェア外部仕様で、同じことが起きました。仕様記述言語で外部仕様を書こうとしただけで多くの間違いが発見できたのです。

もちろん、その後形式手法による正当性検証や妥当性確認でも、多くの間違いを発見することができ、上記 2 つのシステムは、リリース後のバグが 0 で、生産性も従来の開発現場のやり方より高かったことで有名になりました。

実は、我々の使った形式手法はアカデミックの世界では特に目新しい技術ではありません。しかし、開発現場でどうやって使うかは未知の技術でした。

本書では、このように開発現場で役立つことが確実であろう「仕様を書く技術」を紹介します。この技術の中核となるのは形式手法ですが、形式手法と関係なくとも重要な「仕様を書く技術」の要点をも紹介したいと思います。

本書で対象とする仕様は、要求工学などで言う「要求」とか、開発現場で言う「要件定義」ではありません。これらの「要求」や「要件定義」という曖昧な仕様から、厳密な仕様を記述し正当性検証や妥当性確認を行う「要求分析工程」で使用する「仕様」を対象とします。

開発現場では、要求工学などで言う「要求」や開発現場で言う「要件定義」で収集された曖昧な「要求」を使って、すぐに設計を始めることがほとんどですが、これは 30 年前の構造化分析・設計手法や 20 年前のオブジェクト指向分析・設計手法といったソフトウェア工学的手法の時代でも批判されていたやり方です。

ソフトウェア工学の世界では、30 年以上前から「要求分析工程」などの上流工程でバグを発見することが、システムの品質を高め、生産性も向上することが分かっていました。今の開発現場のやり方では、低品質・生産性低下が起こってしまうのです。

本書で紹介する仕様作成技術は「怠け者のための仕様作成技術」です。すなわち、「同じような仕様を何度も書きたくない」「一度書いた仕様はできるだけ再利用したい」「仕様の修正にできるだけ対応した仕様を作成したい」「仕様のチェックはなるべくツールにやらせたい」という人のための技術です。

ただし、このうち「仕様のチェックはなるべくツールにやらせたい」ということだけは、今の技術では完全にはできません。ツールは「仕様の意味」を理解していないので、仕様に記述されたことだけをチェックします。したがって、人間が「システムは何をなすべきか」のチェックを行わなければなりません。

ここで、現在の開発現場のやり方を思い出してください。そこでは、ツールがあればチェック可能な「てにをは」や単純なチェックをレビューとして行なっていないでしょうか？現時点で人間にしか行えない「高度な意味的チェック」をどこまでしているでしょうか？

本書で紹介する技術を使えば、数は多いが単純なミスはツールに任せ、数は多くないがコンピュータには所詮無理な「高度な意味的チェックや分析」を人間が行うことができるようになります。それが、まさにこの本の著者たちが皆さんに先駆けて体験したことです。

佐原 伸

上流品質技術部会 厳密な仕様記述 WG 主査

謝辞

本書の内容を検討するに際し、形式手法の実践と適用に関する長年のご経験に裏打ちされた貴重なご意見をいただき、また巻頭に推薦のお言葉を賜りました九州大学大学院の荒木教授に WG を代表して深く感謝の意を表します。

目次

読書の手引き

この本は**「現場で先輩に文書や仕様書の書き方を聞いてもなんだかよく分からずに悩んでいる、２０代後半から３０代前半のリーダークラスの技術者」**を、最初の読者として想定しています。とはいえ、もちろんそれ以外の「厳密な仕様記述」に関心のある方々にもお読みいただけることを願っています。

頭から読み進めていただいても結構なのですが、ご興味の持ち様により途中の章から拾い読みできるような形にもなっています。ご案内を兼ねて以下に各章が第一にお答えしようとしている疑問を挙げておきましょう。

第１章 仕様とその位置付け

そもそも仕様とはどのような位置付けのものだろう？

開発の世界では一方に解くべき課題が広がり、もう一方にはその課題を解くためのシステムの仕掛け（設計）が広がっています。端的に言えば仕様はその両者を結ぶものです。この章では課題と設計の間に横たわる仕様というものの位置付けを確認していきます。また大切な概念である妥当性の確認 Validation と正当性の検証 Verification に関しても説明します。

第２章 厳密な仕様記述に向けた手法紹介

厳密な仕様記述に使える技法にはどのようなものがあるのだろう？

仕様を記述する際に、その内容としてどのようなものを書くべきかを説明します。その上で厳密な仕様記述を行うための方針を形式仕様記述言語 VDM を用いて解説していきます。実際の現場利用に際しての適用レベル感についてもお話しします。

第３章 仕様と各工程との関係

厳密な仕様記述ができていると、プロジェクトの各工程にはどのように役立つのだろう？

仕様は設計工程だけから参照されるものではありません。要求文書との整合性チェックや、マニュアル作成、品質保証、保守など、あらゆる局面で活用されることが期待されています。この章では厳密な仕様記述がプロジェクトの各工程でどのような役に立つのかを説明します。

第４章 実務者による現場における形式仕様記述の実践に向けて

厳密な仕様記述を実際のプロジェクトの中で活用するにはどのような点に注意すればよいのだろう？

仕様の位置付けや、書かれるべき内容、各工程での活用などについての解説を受けて、ご自身の現場で厳密な仕様記述を始める際に手がかりとなるヒントを示します。どこから手を付けるべきかの検討を行う際にご参考にしていただけることでしょう。

第５章 厳密な仕様記述を支援するツール

厳密な仕様記述を行うために使えるツールにはどのようなものがあるのだろう？

そして、皆さんがいざ厳密な仕様記述を始める際に使える、世の中にある厳密な仕様記述を助けてくれるツールのご紹介を行います。世の中には既に多くのツールが出まわっていて、有力なものでも無償のものが多く存在します。

またそれぞれの章末には、各章に対する「ちょっとした疑問コラム」が掲載されており、それに対する回答も併せて掲載されています。

結果的に盛りだくさんの内容になっています。また似たような話が切り口を変えて繰り返されていたりもします。繰り返しになりますが、どうぞ気になる場所から気軽に拾い読みをしつつ、お手元に置きながらときどき読み返していただければ幸いです。

平成２５年２月

執筆・編集チーム一同

謝辞

本書の執筆編集に当たり、ご議論ご協力いただきました皆さまに深く感謝致します。

第１章 仕様とその位置付け

この章について

本章では「仕様とは何か」を概説します。一言で言ってしまえば、なんらかの解きたい課題があり、それを実際に解いた解があるときにその両者を「つなぐもの」が仕様です。ソフトウェア開発でよく使われる用語に対応させれば、要求と設計の「仲立ちをする」位置にあるものが仕様ということになります。

とはいえ「つなぐもの」とか「仲立ちをする」と言われても意味ははっきりしません。ただなんとなく想像できるのは「仕様とは他の定義との関係が大切なものである」ということです。以下さまざまな側面から「仕様」の位置付けを紹介していくことにしましょう。

1. 仕様とは何でしょう

多くの人や組織はなんらかの解くべき課題を抱えています。
この解くべき（解きたい）「課題」がどこから来るかといえば

（1） 日頃不便だと思っていることを改善したいとか
（2） こんなことができたら凄いなというアイデアを得たとか
（3） ある計画を実行するのに必要な解を探しているとか
（4） 新しいデバイスを用いて新しい利用者体験を創出したいとか
（5） あるいは単純に他人（お客さんや上司）に「この問題を何とかしてくれ」と頼まれたとか

その源泉はさまざまです。

もちろん毎回「全く新しい」課題を解いているわけでもなく、繰り返し類似の課題を解く局面も圧倒的に多いのですが、その場合は既存の「解」を如何にうまく活用し、改善しながら再実装できるかがその「プロジェクトの課題」の一つとして重視されることになります。

多くの場合、最初は課題の形ははっきりしていません。「こんなことができたらいいな、こんなことがしたいな」という形で述べられる希望は、つかみどころがなくもやもやとしています。もちろん、そのまま「どうしよう」と悩んでいるだけでは、課題はちっとも片付きません。もやっとした「課題の種らしきもの」を切り分けて、とりあえず「解くべき課題」が何かをはっきりさせるのが第一歩です。

いわゆる上流工程では課題を生み出すこうした「源泉」からのもやもやした情報を整理し、課題を解決したときに生み出される「価値」に着目しながら要求分析を進めることになります。ここで整理された解くべき課題を具体的に解決するために、システムの開

発が選択されるとすれば、要求分析を進めた結果の一つがシステムを駆動するソフトウェアの外部仕様として結実します。

こうして抽出されたソフトウェア（システム）の外部仕様は、もともと抱えていた課題の解決に寄与しなければなりませんし、もちろんその課題を解決することにより得られる「価値」の生成を助けるものである必要があります。

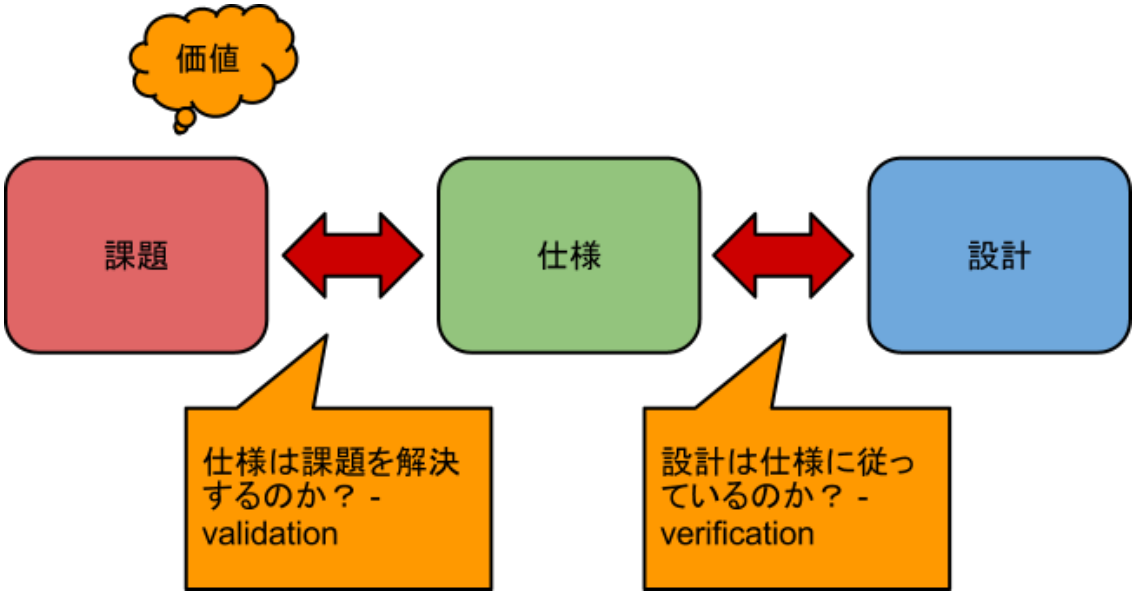


図 1-1 課題・仕様・設計の関係

まずはソフトウェアからは離れて、もっと卑近な例で考えてみましょう。

たとえば夕方空腹を感じている程度なら、解くべき課題は「空腹を満たす」ことで、その課題は自分に由来しています。それをどのように解決するか、たとえば「自炊する」「外食する」「誰かに食事を持ってきてもらう」「部屋の中にあるものを適当にそのまま食べる」などのやり方がそのあとに続くことになります。

まあその日の気分によっては、そもそも空腹を感じる前に「今日は来々軒のラーメンが食べたいなあ」という願望が先行しているかもしれません。この場合にはやがて顕在化

する「空腹を満たす」という課題は、かなりの確率で「来々軒のラーメンを食べる」という解法へ向かうことが予測されます。

さて、いささか強引ですが、ここで解法と呼んでいるものはさらに「仕様」と「設計」に分解することができます。ここで仕様とは解法が満たすべき条件を示すものでたとえば「来々軒のラーメンを食べる」という目標を表します。そして設計はその仕様を満たすために具体的に遂行されるべき手順を記述したものです。

たとえば

- 課題：鈴木くんの空腹を満たす
- 仕様：来々軒のラーメンでお腹が満ちた状態になる
- 設計：部屋を出る、ラーメン屋へ行く、ラーメンを食べる、勘定を払う、部屋へ帰る

といった分類が考えられるでしょう。もしその日が真冬だと外出するのが嫌になって設計が変わるかもしれませんね。

- 課題：鈴木くんの空腹を満たす
- 仕様：来々軒のラーメンでお腹が満ちた状態になる
- 設計：来々軒に電話する、ラーメンを受け取る、代金を払う、ラーメンを食べる、食器を部屋の外に出す

ここでは、「課題、仕様、設計」を適当に分類しましたが、この分類方法に厳密な正解があるわけではありません。課題一つとっても「空腹を満たす」でも「栄養を補給する」でも「血糖値を上げる」でも何でもよいわけで、解法も「食事する」「点滴する」などさまざま、そしてそれぞれの設計方法もさまざまに考えることができるのです。

さてここで

- 課題：鈴木くんの空腹を満たす
- 仕様：来々軒のラーメンでお腹が満ちた状態になる

の部分の記述を見てみましょう。いまここで想定した「来々軒のラーメンでお腹が満ちた状態になる」という仕様は、もともと存在していた「空腹を満たす」という課題の解決に寄与しているでしょうか？おそらく課題の解決には寄与するものと思われますが、実はこの判断は慎重に行う必要があります。たとえばもしこの記述が単に

- 課題：鈴木くんの空腹を満たす
- 仕様：ラーメンを食べる

と書いてあった場合は、先の記述と何が本質的に違うのでしょうか？ 一見似たようなことを言っているようですが、「来々軒の」という文言が仕様に入っているために、この仕様は後者に比べてより「厳密な記述」になっているのです（おおげさですね)。また先の記述では食べるという行為ではなく、その結果「お腹が満ちた状態になる」ということに重きを置いています。

たとえば店を特定すれば、麺の量（カロリー)、具の種類（栄養素)、スープ（味わい）などが特定されますから、その店のラーメンがある人物の空腹を満たすには十分か否かの議論が、少なくともより具体的なデータを用いて行うことができることになります。これは実際に設計（実行動）を起こす前に、この仕様が妥当であるか否かの議論を容易にします。

「来々軒のラーメンを食べに行こうか？」という問いかけになら「あそこの麺は物足りない」「あそこの麺なら満足できる」といった答を返すことができますが、「ラーメンを食べに行こう」と誘われて連れて行かれたラーメン屋では全く満足できない場合、問題

の発見と修正が随分遅れてしまうことになります。要するにラーメン屋の前まで出かけて行った時間と労力（そして、食べてから不満ならお金も）が無駄になってしまうのです。

仕様は当初の課題の解決という観点に照らして「妥当」であるか否かが判断されなければなりません。ところが多くの開発現場で書かれている仕様は、いわば単に

仕様：ラーメンを食べる

と書かれているだけのものがとても多いのです。しかも課題に関してははっきり書かれていないことが多く（議事録や、メモや、メールの中に断片的に分散していることが多い)、仮に書かれていても

- 課題：空腹を満たす

とだけ書いてあるようなものがとても多いのです。これでは「誰の」空腹を満たせばよいのかがはっきりしません。現実世界の問題の解決を検討しているのですから、仕様の妥当性を検討するためにも

- 課題：「鈴木くんの」空腹を満たす
- 仕様：「来々軒の」ラーメンでお腹が満ちた状態になる

くらいは書いて欲しいところです。ところが、繰り返しになりますが現実世界の仕様は

- 仕様：ラーメンを食べる

とだけ書いてあるようなものが多く、多くの開発者はいろいろ想像を巡らせながら

- 設計：部屋を出る、快楽亭に行く、ラーメンを食べる、勘定を払う、部屋へ帰る

というようなシステムを構築して、発注者の課題（＝鈴木くんの空腹を満たす）を解決できないハメに陥るのです。

このあまりにも単純な喩え話の中でも仕様を考える際には、その仕様が「本当に課題を解決しているのか」という問が大切になることを示されています（そしてもう一歩踏み込むならその課題解決は本当に「必要な人への価値」を生み出しているのか？と問いかけることができるでしょう）。

2. 仕様に含まれるものは

なによりも大切なことは、これから作るシステムがどのような性質を備えているものなのかを明快に示すことです。性質の示し方にはいくつかの方法があります。入出力に着目して両者の関係を示す方法、望ましい状態遷移を示す方法、常に守られるべき条件の範囲を示す方法などなど。より詳しい手法の紹介は第 2 章で行います。

さて、しばしば見受けられるのは、どのようにシステムが動作すべきかの細かい指示が書き連ねてあるような文書を「仕様書」と呼んでいるケースです。

繰り返しますが仕様に含まれるのは「性質」であって、具体的な実現手段ではありません。特にソフトウェアシステムの性質は型や条件式や関数、状態遷移などで示されます。とはいえこの「性質」はたとえば何を（ソフトウェア）システムの入出力として取り扱うかの決定に依存して決められたものですから、実現に向けてのなんらかの意思決定が行われた結果であることは事実です。

3. システムとハードウェアとソフトウェア

ここで一歩下がって考えましょう。わたしたちはなんらかの課題を解決するために「システム」を構築します。登場する人間もシステムの一部としてとらえなければ問題解決のためのシステムとしては厳密に言えば不足しています。こうしたシステムは複合要素

の組み合わせでありさまざまな組み合わせを考えると可能性は膨大なものとなります。個々の要素がそれなりの規模のシステムであることもあり得ます。こうしたシステムの仕様を実現するためには通常ハードウェアとソフトウェアの組み合わせが必要なのですが、多くの場合私たちはもっと狭い意味でシステムをとらえ、ハードウェアなども既存の PC などを想定する限りはあまり意識しなくて済んでいます。しかし時に応じて解決しようとしているシステムが複合システムの協調によって成立していることを思い出すことが必要になることでしょう。

しかし本書ではこうしたシステムのシステムや、ハードウェアシステムとソフトウェアシステムの間の責任分担といった話題にはこれ以上踏み込みません。

4. 課題を抽出し仕様にまとめる社会学

要求は「そこにあるもの」か？

要求が役割としての発注者の頭の中に既にあるわけではありません。しばしば「要求を発注者がはっきりさせてくれないから仕様が決まらない」という不満が聞かれますが、そもそも要求とははっきりしないものなのです。ごく一部の要求だけが（とりあえず）はっきりしています。たとえば「列車の予約が行えるシステムを作りたい」など。

しかしその先に一歩踏み込むと、「列車」とは何か「予約」とは何かすら厳密に考えると曖昧であることに気付かされることがあります。

安定しているもの、創りあげるもの

課題（要求）を検討していく際には比較的安定している環境（事実、物理法則、法律、既存システムなど）と不安定な「これから創りあげるもの（サービスや製品）」の関係の整理が大切です。

検討の過程ではありとあらゆる仮説が登場し、仮の仕様と組み合わさって、課題に対する仮想的な確認が行われます。ともすると開発の中身ばかりに目が向いてしまいそれが課題に対して満足できるものなのかの検討がおろそかになったりします。

なお課題の領域の中で要求を引き出すための動機には「単純な課題解決」を超えた「変革への意志」が必要になることがしばしばありますが、この話題にはここではこれ以上踏み込みません。

5. 課題・仕様・設計

本書は仕様を中心に解説をしていきますが、さらに詳細な説明に踏み込む前に、課題・仕様・設計の関係について確認しておきましょう。簡単に言ってしまえば「課題」とは問題領域におけるさまざまな動機と要求と制約条件、「仕様」とは「課題」に現れた要求を満たすべく提案された開発対象システムの性質、「設計」とは「仕様」が求める開発システムが満たすべき性質を実現するための具体的な仕掛けを定義したものです。

「課題」は原則として現実に対応し、「設計」はシステム内部に対応しています。言い換えると課題は発注者の視点であり、設計は開発者の視点です。もし課題の視点を思い付くままに設計として実装していったとすると、多くの場合はバランスを欠き保守も難しいシステムになってしまうでしょう。課題の世界はさまざまな視点がそれぞれの言葉で表現されているからです。

「仕様」は「課題」を分析し開発すべきシステムの性質を定義するものです。開発対象が何かをここでまとめ、「設計」のための出発点となります。その意味で「仕様」は「課題」と「設計」の橋渡しを行う接点と考えることができるのです。

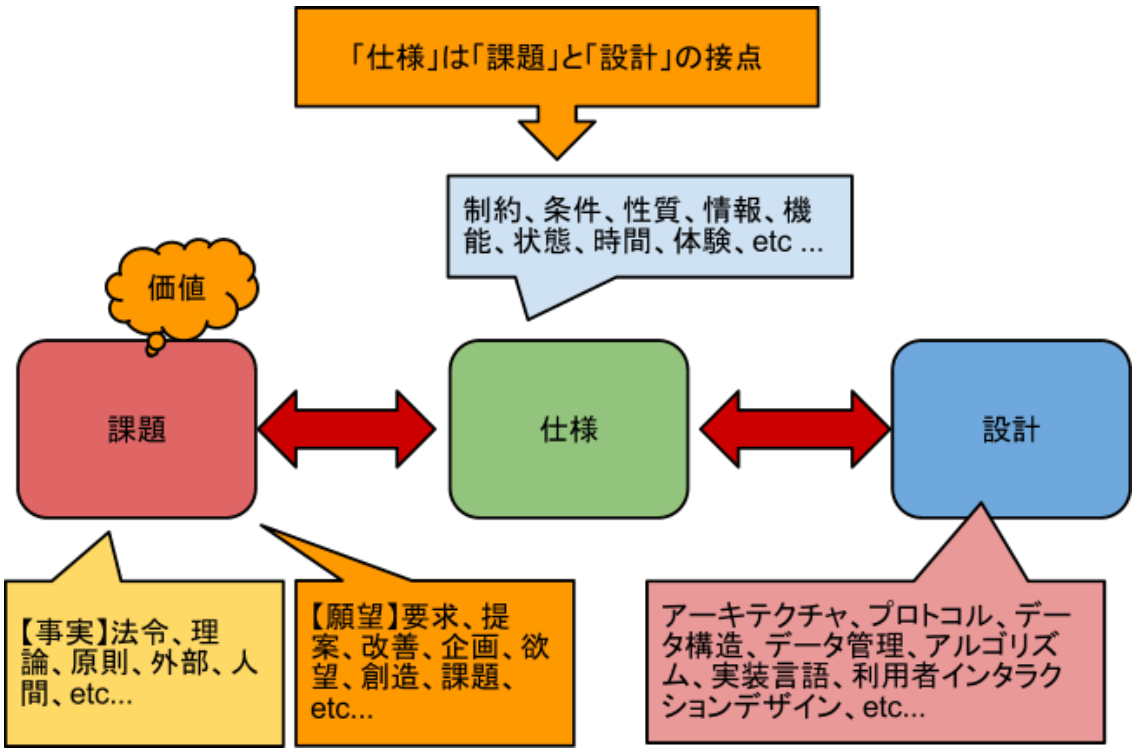


図 1-2 課題・仕様・設計に含まれるもの

こう考えると「課題」に現れる記述は発注者とか顧客の世界のものであり、「設計」に現れる記述は受注者もしくは開発者の世界のものであることが分かります。
そして「仕様」は誰のものなのかといえば、発注者、受注者それぞれにとっての意味をもつ記述になるのです。では仕様は誰が記述するべきなのか、という疑問が湧いてくると思いますが、仕様を記述する役割の人物（グループ）に求められる要件は

課題検討側で集められた材料を、設計者に渡せる程度に厳密なシステムのモデルとしてとりまとめることのできる人物（グループ）

ということになります。
現実的には設計につながる厳密性を持つモデリングの技法をいろいろ知っているという意味で、開発者側の人間が手を動かし、問題領域の知見を持つ人間が課題側の材料をとりまとめるチームを組んだ方が進み方はスムースかもしれません。こうした話題は第３章や第４章で扱います。

なお、課題・仕様・設計をこのように併記して書くと、課題が決まるまで仕様が決まらず、さらに仕様が決まるまで設計が始められないような印象（すなわちウォーターフォール型の開発スタイル）を受けるかもしれません。
しかし実際にはこれら三つの記述は独立に並列して進む場合が大部分ですので、その内容も細かい要素に分けて検討していくことができます。

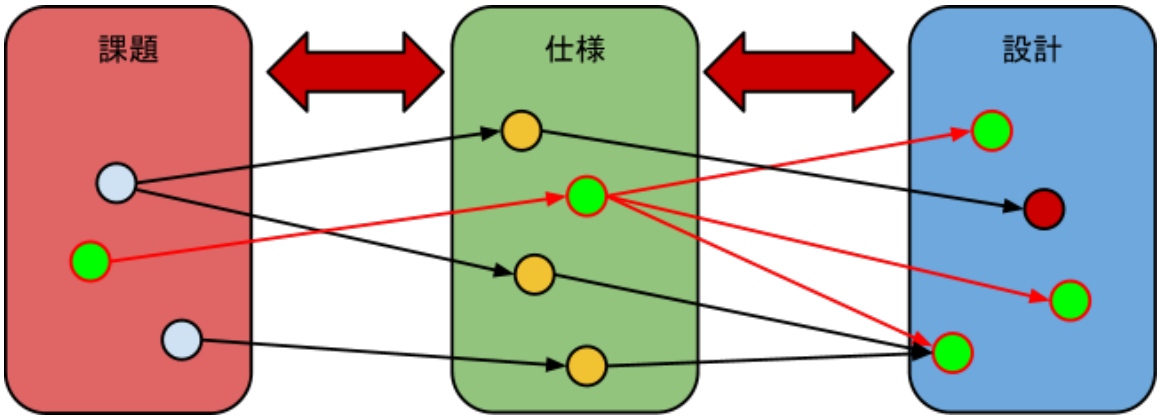


図 1-3 要素ごとの追跡性

前の図は、課題の中に挙げられた各項目が、仕様や設計に反映される様子を概念的に示したものです。このようにそもそもの課題要素が、仕様や設計のどの部分に反映されているかを追跡できることは、プロジェクトが確かに望まれているシステムを、確実に構築しつつあるのかを確認するために必要な特性となります。
たとえばプロジェクトの特性によっては、特定の課題がまず解決可能なものなのか否かを先に確認する必要があります。つまり「ある問題が解けないならこの開発はしない」という場合です。こうした場合には本格的な検討に先立ち「実現可能性検証 - feasibility study」が行われることがあります。たとえば想定した制御方法で対象のデバイスは制御できるのか、必要十分な性能を発揮できるアルゴリズムは開発可能なのか、市場における保守費用はどの程度のものになるのかなどなど。単に課題レベルの検討だけではなく、仕様レベル、設計レベルまで要素を掘り下げていき実現可能（受容可能）であるかどうかを判断する必要があるのです。もちろん簡単なものであれば作り捨てのプロトタイプを用いたものを使うだけで十分な場合も多々ありますが、こうした先行検討要素も

立派なプロジェクトの成果物なのですから、「確認されたこと」はきちんと「課題」「仕様」「設計」の各記述要素の中に追跡可能な形で残されていることが理想となります。なお注意して欲しいのは、こうした場合にプロトタイプを無理に拡張して実際の製品の中に流用せよと言っているわけではありません。検討・検証の過程で明らかになった「課題」「仕様」「設計」の各要素をそれぞれのレベルのモデルとして記録するべきだということです。

ここまでの説明を読んで「あれ、実装は？」と思われた方もいらっしゃると思います。本書では実装は、設計の中に含まれるものとして説明をしています。これは最近の開発環境の目覚ましい発展によって、コーディングの占める割合が極端に少なくなっていることにも起因します。むしろ適切なコンポーネントを選ぶ方が大切だ - ということは設計時に多くの実装結果は決まってしまっていることになります。

実際には実装基盤に関する細かい議論は存在するのですが、ともあれ本書では「仕様」が議論の中心ですので、設計に関しては実装を含んだ大きな概念でとらえることで進みたいと思います。

6. いわゆる「要求」と「仕様記述」の関係について

仕様は課題から抽出された問題を解く「システムの性質」を記述したものです。既に説明したように「課題」にはさまざまな要素が含まれています。

課題を構成する２大要素「事実」と「願望」のうち、「願望」の中に含まれるのが「要求」です。UML を使っている場合には、この部分の記述に UseCase が使われたりすることになりますが、もちろん要求の記述方法は UseCase だけに限られたものではなく、単に要求項目の箇条書きである場合も多く見ることができます。

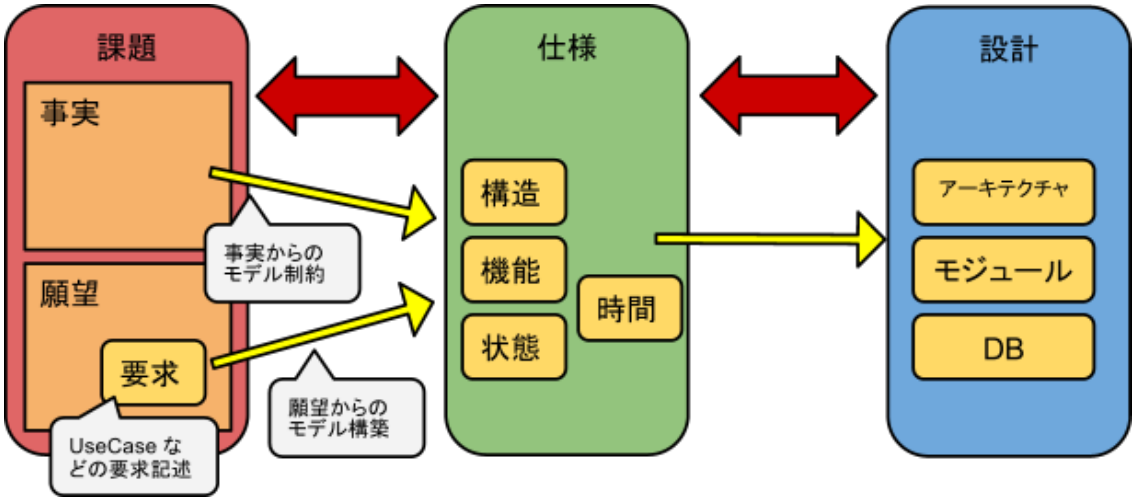


図 1-4 「課題」の構成要素と、他の要素との関係

なお本書では「要求」を「課題」の中の「願望」を構成する一要素として説明していますが、実際の現場に見られるいわゆる「要求文書」は「事実」と「願望」の記述があまりうまく分離していない場合が散見されます。

本書は「実際の文書の名前」ではなく、仕様以前に存在している実世界の要素全体を「課題」という名前で呼び、その中をさらに「事実」と「願望」という形で分類しています。そして「願望」の中の要素の典型的な呼び名が「要求」であるというとらえ方をしています。

これは絶対的な分類ではありませんし「事実」と「願望」の区分が怪しい問題領域も沢山ありますが、本書ではそうした俯瞰的な見方で説明を続けていきます。

単純な例を示しましょう。

- 事実（１）：小田急特急ロマンスカーの停車駅は { 新宿, 参宮橋, 成城学園前, 新百合ヶ丘, 町田, 相模大野, 本厚木, 伊勢原, 秦野, 新松田, 松田, 開成, 小田原, 箱根湯本, 大和, 藤沢, 片瀬江ノ島, 小田急永山, 多摩センター, 唐木田 } である。
- 事実（２）：特急料金は新宿⇔町田、新宿⇔相模大野間は４００円、新宿⇔本厚木間は６００円、。。。etc. 事前に特急券を買わずに乗車すると、車内料金として２５０円が加算される。

- 要求（1）: 小田急の二つの駅を指定して、ある特急ロマンスカーに対して４名までの座席予約を行う
- 要求（2）: 座席予約は４名まで一括で行うが予約取消は１席ずつ行う

なお「願望」として含まれ得るものとしては : 事前予約を可能にしたい、発車１５分前までに購入されない事前予約はキャンセル、座席予約は窓際や通路側を優先して確保できる、優先希望が叶わない場合は予約を行わないというオプションも用意、発車前１５分以内のキャンセルには手数料１００円を課す。

といったものなどが考えられます。これらの願望は仕様として整理していく過程で、他の要求を修飾したり、新たな要求になったりするでしょう。

こうして「事実」と「願望」をうまく切り分けながら、要求の変更に柔軟に対応することが現場の開発では求められています。厳密な仕様記述を支える技術である形式手法はこうした際に、開発者たちを強力に下支えしてくれる手段を提供してくれるのです。

7. 課題と仕様

問題解決の場面で「課題」を検討する際には、取り組む問題領域によって記述される対象は異なります。ビジネス系アプリケーションであれば、ビジネスゴールから導かれるビジネスモデルや事務処理のフロー、セキュリティポリシー、問題領域の中核をなす情報モデル（いわゆるドメインモデル）などが必要ですし、組み込み系アプリケーションの場合は制御対象の物理的特性に関するモデルや、安全性や性能に関する制約条件を検討する必要があります。もちろん最近の状況は単純なビジネスソフト、組み込みソフトという分類を無意味にしていますが。

仕様を書いたとき、その評価は何に基づいて行うべきでしょうか。仕様の評価はそれがもともとのシステム開発の動機となった課題に対して解を提供し、関係者に価値を提供するか否かという視点で行われます。こうした価値の確認作業は「妥当性確認 validation」と呼ばれます。

8. 指示と定義

さて、とにもかくにも、現実世界の問題を解くためには、現実世界に現れる存在や現象を記述の世界に取り込まなければなりません。課題の世界に現れる「事実」としての法律や鉄道路線や駅やカレンダーなどを必要に応じて記述して利用することになります。こうした「課題」のレベルで検討記述された「現実世界の項」は「指示 designation」と呼ばれます。何が指示として採択されるべきかは、問題領域の特性に依存します。[1]

「指示」を組み合わせてさらに「定義 definition」を行うことができます。「定義」や「指示」を組み合わせてさらに複雑な「定義」を得ることも可能です。こうした「指示」や「定義」が現実世界とモデルの世界をつないでくれる役割を果たします。

特に「仕様」の世界では「課題」側で挙げられた材料を基に得られた「指示」や「定義」を用いて性質や条件の定義を行い、完全で（漏れがない）一貫性のある（矛盾がない）モデルを記述するのが目的となります。

なおここで現れた「モデル」という言葉は現在開発対象として考えている対象のさまざまな性質を抽象し表現したものです。詳細は第２章に譲りますが、仕様を記述する「構造」「機能」「状態」の観点、そして仕様を動作可能にしてアニメーションを行い、動的に評価可能にするための計算式、制約式などの定義を合わせたものがここでいう「モデル」です。

9. 仕様と設計

「仕様」で定義された性質を具体的に実現（実装）するために行われるのが設計です。とりあえずハードウェアの開発は行わないと仮定しても、たとえば全体アーキテクチャの決定、利用者経験を実現するデバイス界面の選択、データ構造とデータ管理手段の選択、通信手段の決定その他が必要になります。こうした決定の一部は既に仕様定義の段階でも行われていて、それを設計段階以降で具体的な実装言語で記述していくことになります。

記述された設計の評価は基になった仕様に対して行うものと、その時点での利用者価値に対して行うものとに分かれます。後者は仕様の際にも挙げた妥当性の確認 = Validation ですが、前者は正当性の検証 = Verification と呼ばれます。

10. 仕様の品質

仕様の品質を考える際にはどのような基準が考えられるでしょう。たとえば下に示したのは IEEE830-1998 に挙げられた仕様の品質基準ですが、どのような基準を検討したとしても、似通ったものになるでしょう。

特性	説明
正当性	ソフトウェアが持つべきすべての要求が含まれており、それ以外の要素は含まれていない
無曖昧性	個々の要求の意味が明確
完全性	全ての必要な要求が含まれている 全ての入力と状態に対する応答の定義が含まれている 用語及び図表の説明が含まれている
一貫性	要求間で矛盾がないこと
順位付け	要求が重要性や安定性に関して順位付けられている
検証容易性	全ての要求が有限のコストで検証可能
修正容易性	容易かつ完全に一貫性を保って要求の変更を行うことができる
追跡性	要求の根拠が明確で開発工程全体で参照できること

表 1-1 仕様の品質特性

11. 妥当性確認と正当性検証

妥当性確認 Validation

狭い意味での Validation と広い意味での Validation が存在します。前者は「仕様」が「利用者の価値」に見合うかという視点、後者はシステムそのもの、あるいはプロジェ

クト全体が「利用者の価値」に資するかという視点です。どちらも利用者の価値が判断基準になりますが、「仕様」だけでは開発されるシステムの真価は評価できません。もちろん価値を全く生み出さない「仕様」では意味がありませんが、たとえば実際の使い勝手などに関する Validation は仕様段階では制約だけを記述しておき、最終的には実際のシステムを用いて行われるべきものです。

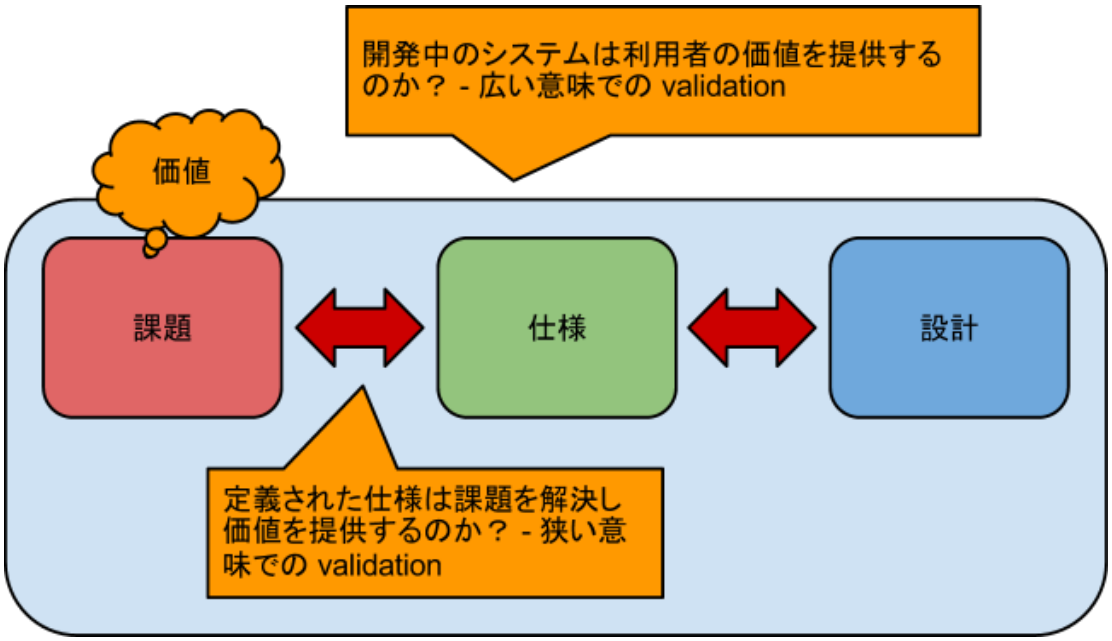


図 1-5 広い意味での Validation と狭い意味での Validation

正当性検証 Verification

どのような小さなステップにも、仕様-設計の対応関係があります。要求-設計、上位設計-詳細設計など、どのような言葉を使っても構いません。要点は「上流」が要求する性質を「下流」がきちんと満たしているかどうかが大切なことなのです。

システムとしての全体の有用性ではなく、この「上流」「下流」の部分的な対応関係に着目して検証を行うのが Verification です。このときには「上流」そのものの正しさは問われません。ただ上流が下流と矛盾していないか否かが問題となります。

テストと Validation・Verification の関係

ここで一般に使われる「テスト」という用語と、Validation、Verification の関係に触れておきましょう。一般にテストは「検証」という意味をなんとなく指す言葉として使わ

れています。しかし Validation、Verification の観点から眺めると、テストはそれぞれを行うための手段の一部ということになります。広い観点からみれば Verification も Validation の一部ということになります。

広い意味での Validation を行うためには機能テストを行うだけではなく、さまざまなリスク分析（耐故障性、セキュリティ、性能など）や、もともとの利用意図/利用者の要望への合致度合い、エンドユーザーによる実利用などのさまざまな確認作業が必要となります。

また Verification の際にはやはり機能テストだけではなく、性能や利用資源の効率性、保守性、安全な設計といった観点でのレビューが必要とされるのです。

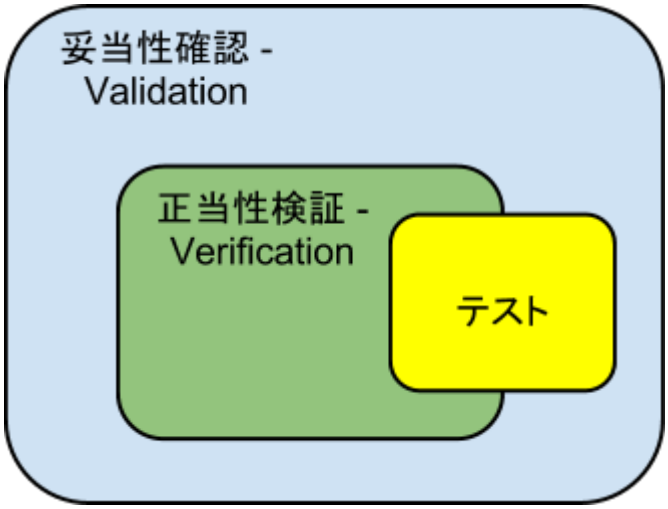


図 1-6 Validation、 Verification とテスト

12. 情報ハブとしての仕様

仕様の置かれた状況

さて、ここまでの説明では「仕様」は大切なものであり、きちんと書かれるべきだという前提で行われてきましたが、実際の現場では「仕様」はどのように扱われているのでしょうか。

多くのプロジェクトにおける仕様書のイメージとは「要求の中でシステム化する部分を必要に応じて書き留めたもの」というレベルに留まっていることが多そうです。もちろ

んこの「必要に応じて」のレベルはさまざまで、単なるメモ書きレベルから、組織としてのフォームが決められたもの、UML などの共通表記を用いるものなど、形式仕様記述を用いるものなどが考えられます。

どの記述レベルを採用するかはプロジェクト毎に異なりますし、多くの場合には同じプロジェクト内であっても記述レベルの濃淡が存在する場合はとても多いのですが、こうした記述レベルのバラつきや厳密さの不足によって、仕様は設計を行う前の使い捨ての定義のように思われていたり、設計を始めるに際して最低限決めておかなければならない部分だけを最低限仕様として決めておいて、あとは設計をしながらその場で（開発者によって暗黙的な）仕様が決められていったりということが頻繁に起こります。

「仕様なんか延々と書いていないでさっさと設計（実装）をしろ」と上司に言われたエンジニアは枚挙にいとまがありません。

不完全で曖昧な「仕様」を、後から Validation や Verification の役に立てようとすることは極めて困難であることを、この前までの節では説明してきました。

こうした現場の状況では仕様というものは「なるべく素早くすり抜けるべきもの」であり、「基本的に使い捨てのもの」あるいは「顧客が納品しろというので、設計から遡って作成するもの」となってしまうのも無理からぬことです。

いずれにしろ「仕様」が一度作られて参照されたあとその保守もままならず、一貫性や完全性も確認されないままに放置されてしまう現実が多く見受けられます。

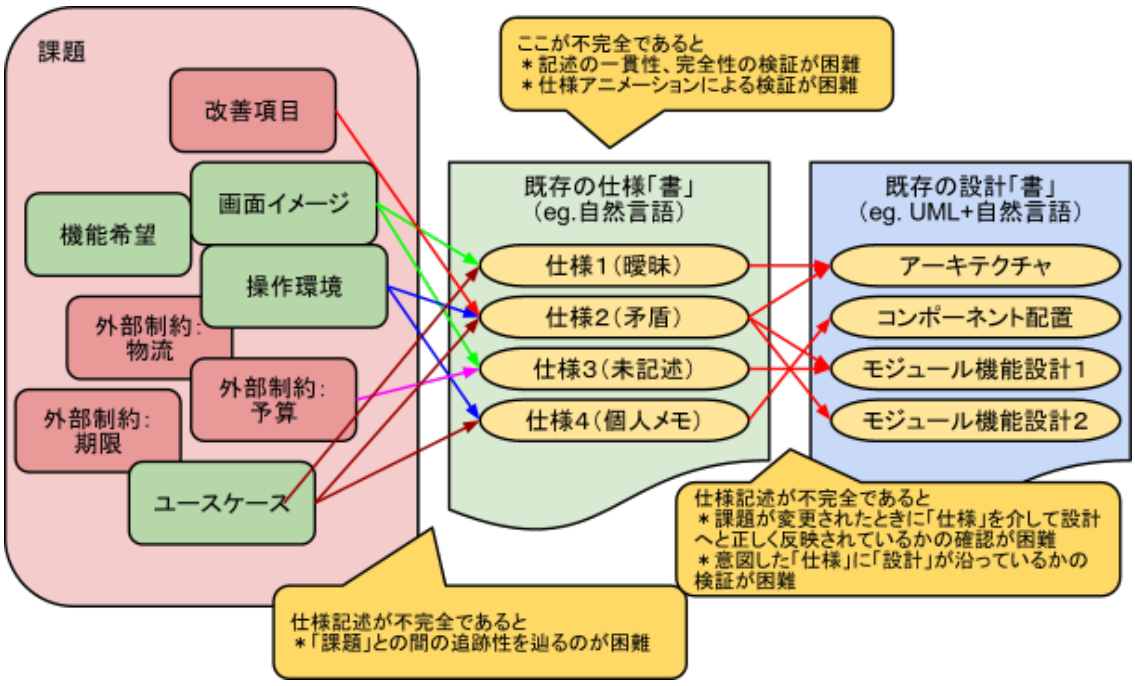


図 1-7 仕様が不完全であることの弊害

こうした「仕様をなるべく軽くして課題を直接設計してしまいたがる慣行」が仕様の地位を低めているわけですが、上記の図にも注記されているように、仕様記述が不完全なことによる弊害にはさまざまなものがあり、出荷後の障害や出荷前の大幅な手戻りなども発生しやすくなります。

ありがちな「ユーザの要求を聞いて（課題）、システム要件をまとめて（仕様）、プログラムを開発する（設計）」という開発プロセスが、ともすれば「手順の遵守」にばかり目を向けがちで、開発を通して作られていく成果物の品質そのものをおざなりにしがちです。正確に言えば「大事なのはプログラムだし、最後にテストするのだから途中にあまり時間をかける必要はない」という主張が多くなりがちだということですが。

こうした事態を自然に避けるようにするための方法の一つは、仕様そのものの位置付けを「一過性」のものから、プロジェクトの全ての工程と結び付き保守段階に至っても活用されるような「使う価値のあるもの」へと引き上げてやるというものです。

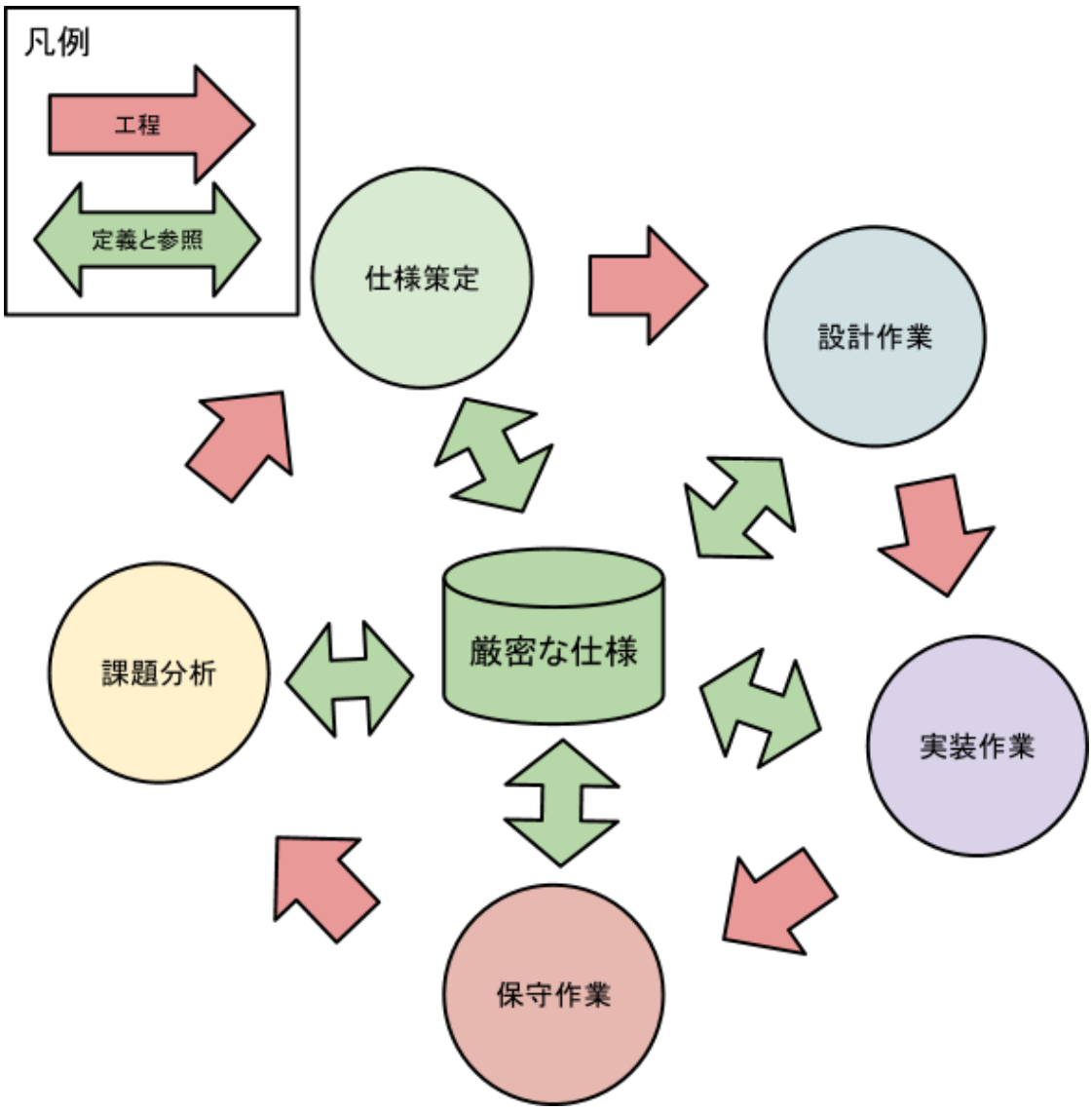


図 1-8 情報ハブとしての厳密な仕様

上の図は「厳密な仕様」を中心にプロジェクトの各工程が連携する様子を概念的に示したものです。一度作られた仕様が保守を経て継続的に利用されていく様子を表わしています。

ここでは仕様は全ての作業から参照（場合によっては定義）されるため、いわゆる仕様変更などもきちんと定義に反映されるようなエコシステムが構築されやすくなるのです。

こうした「情報ハブとしての仕様」を手に入れることにより、仕様の活用は行いやすくなります。この形態は多くの形式手法成功事例で観察されたものです [2]。

次の図は「厳密な仕様記述における形式手法成功事例調査報告書」[2] からの引用ですが、日本国内における代表的な形式手法適用の成功例である mobile FeliCa チップ開発プロジェクトにおける「情報ハブとしての仕様」の位置付けを DFD モデルを用いて表現したものです。
外部仕様を中心にプロジェクトのアクティビティが連携している様子を示しました。

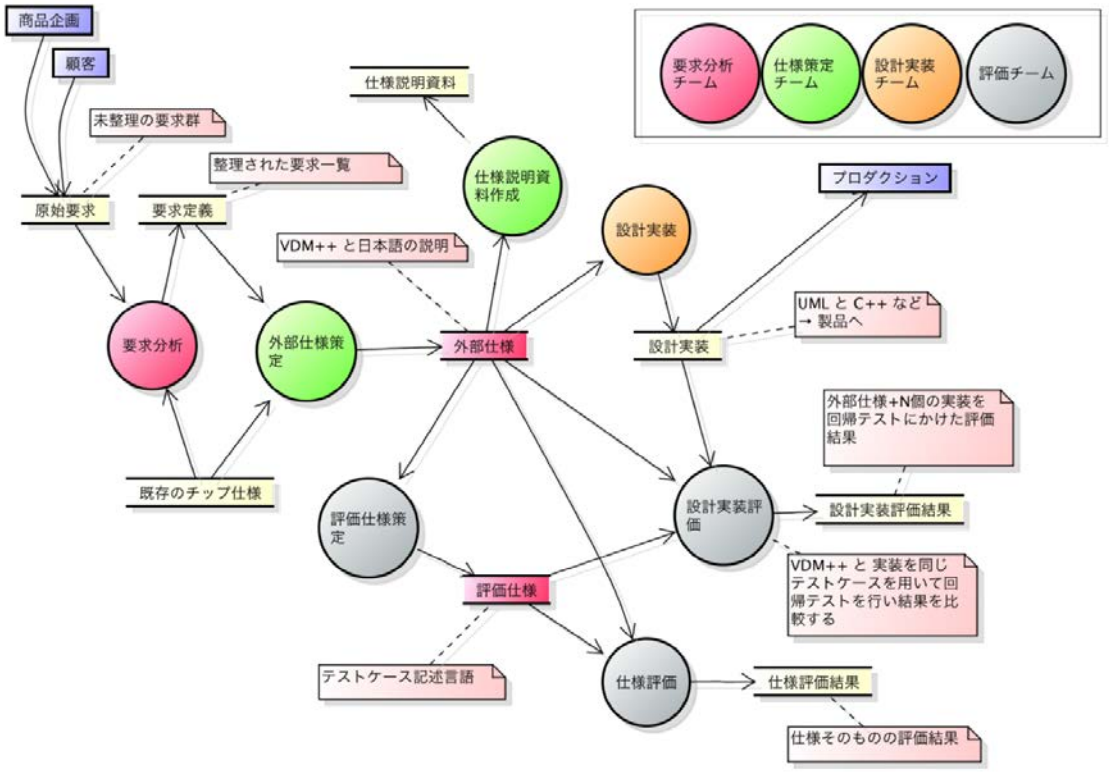


図 1-9 Mobile FeliCa プロジェクトにおける仕様の位置付け

この章のまとめ

この章に登場した主要な概念を以下に挙げておきます。

課題・仕様・設計

仕様の位置付けは、課題と設計をつなぐ接点です。

課題とは何かといえば、問題を解決したい世界における「事実」と「願望」の構成されているものです。「事実」には「法律」「規格」「外部システム」といった既存の構成要素、「願望」には「要求」「提案」「企画」といったこれから実現したい要素などが含まれています。こうした「課題」の中から、解かれることによって生み出される「価値」を見いだして、その価値を生み出すためのシステムの性質を定義したものが「仕様」です。

これに対して設計は、仕様で定義された性質を実現するための仕掛けを定義したものです。仕様が誤っていた場合には、いくら「正しい」設計を行なったとしても出来上がったシステムは価値を生み出しません。

本書ではここに示した意味での「仕様」を、厳密に記述するための手段について解説を行います。第２章では日本語の曖昧さを取り除きながら、仕様としてどのような性質を記述すべきかについての説明を行います。特に厳密な仕様記述の実現手段の一つである「形式仕様記述」を形式仕様記述言語 VDM による例題を通して紹介します。

妥当性確認 (validation) と正当性検証 (verification)

システムはなんらかの「価値」を生み出すために考案され構築されるものです。そのため、そのシステムが本当に「価値」を生み出せるか、生み出しているかを常に確認する必要があります。これを妥当性確認 Validation と呼びます。

一方、ある定義から１段記述を詳細化した場合（たとえば仕様から設計を作成した、仕様を１段詳細化したなど）には、前段で定義されている性質が、次の段でもきちんと保たれているかを検証しなければなりません。これを正当性検証 Verification と呼びます。

Validation と Verification は似て非なる概念です。それぞれ一言でいえば以下のようになります。

- Validation - 正しいものを作っているのか
- Verification - 正しく作っているのか

仕様の品質

仕様自身の品質としては「正当性、無曖昧性、完全性、一貫性、順位付け、検証容易性、修正容易性、追跡性」などのキーワードを挙げました。これらはみな仕様の形式に関する品質です。しかし、仕様の形式の品質が高いからといって、仕様の表す「内容」が高品質とは限らないことに注意しましょう。

どちらが大切かと問われれば、もちろん仕様の表す「内容」が高品質であることの方が、仕様そのものの形式が高品質であることよりもはるかに重要です。しかしながら仕様が大規模になるに従い、形式が低品質な文書は管理の手間が指数的に上昇し、本来は仕様の表す内容に注ぐべき労力を奪ってしまいます。

結局ある程度大規模または複雑な仕様を、継続的に保守していく際には、少々のオーバーヘッドには目をつぶって仕様の形式の品質にも注意を払うべきなのです。本書ではこのオーバーヘッドを低く抑えつつ形式の品質を上げてくれる形式仕様記述を第２章以降で解説しています。

情報ハブとしての仕様

形式的な品質の高い仕様は、定義内容の検索や変換、そして抽出や機械的な検証を行うことが容易になるため、プロジェクトのさまざまな局面での活用が可能になります。「厳密な仕様記述における形式手法成功事例調査報告書」[2] にもあるように、仕様をプロジェクトの情報ハブとして利用した多くのプロジェクトでは、Validation や Verification そして文書化やテスト設計などに仕様を活用できています。専用のツールがなくても、構文と意味の決まったモデル表現なら、それを変換するツールを作成するのは比較的容易です。

このためたとえば形式仕様記述言語を用いた仕様記述はそれ自身の評価機構も加わり、広範囲にわたる仕様の活用を促進してくれるのです。

参考文献

[1] マイケル・ジャクソン「ソフトウェア要求と仕様」2007

[2] IPA/SEC「厳密な仕様記述における形式手法成功事例調査報告書」2013

章末コラム [仕様を記述するちょっとその前に]

第１章を読んで、私は、大きく三つの疑問を抱きました。一つめの疑問は、「発注者自体が認識をしていない課題をどのように過不足なく抽出できるのか」ということです。しかし、この点は、要求工学の専門領域であり、本書の主題ではないと思いますので、

ここでは疑問に思ったということで止めておきます。二つめの疑問は、「仕様を厳密に記述するとは何か」ということです。そして、三つめの疑問は、「曖昧な仕様書が渦巻く状態からどのように脱出できるのか」ということです。

まず、二つめの疑問である「仕様を厳密に記述するとは何か」という点について、第 1 章で記述されている内容を参考にして考察したいと思います。第 1 章に掲載されている「仕様の品質基準」には、「厳密性」という言葉はありません。しかし、該当の表を参照すると、厳密性を噛み砕いて、「仕様の品質基準」となっているような気がします。また、第 1 章では、「「仕様」は「課題」と「設計」の接点」であるとありますので、接点である以上、「課題」「設計」という双方から理解できる必要があるものと考えられます。ここで、“理解できる”とは、「課題」と「設計」で考えていることが合致するということを意味しています。私は、仕様を厳密に記述するとは、「仕様の品質基準」を満たしながら、仕様を利用する人が、統一の理解を得られるように記述することであると理解しました。

次に、三つめの疑問である「曖昧な仕様書が渦巻く状態からどのように脱出できるのか」ということです。第 1 章で述べられているように、既存の仕様書は、記載レベルが揃っていない場合も少なくなく、仕様の存在価値を高めるというところに行き着くこと自体が難しいのではないかと感じました。そもそも、仕様書は、「書」である必要があるのでしょうか。「書」であるからこそ、記述に時間を要する可能性があります。さらには、記述の量が膨大になればなるほど、人が把握できる限界を超えてしまうと思います。記述の内容を検討すると同時に、記述された内容をシステムに関わる人が必要な情報として理解できるようにする方法を見つける時期に至っているということを感じました。そのためにも、第 1 章で述べられている仕様に対する意識の刷新が重要であるのだと思います。

以上、疑問に思ったことに対する私の考察を記載してきました。第 1 章を通読すると、私自身が「分かったようなつもりでいる」言葉が非常に多いということに気が付きました。当然ながら、すべての人が私と同じ言葉を「分かったようなつもりでいる」とは限りません。しかし、一度立ち止まり、自分自身の理解と他人の理解は違うかもしれない、ということを意識することが仕様を記述する際にも重要なのではないでしょうか。

[第 1 章執筆者のコメント]

確かに第 1 の疑問に答えることはこの本の主要な目標ではありません。とはいえ本書で扱う形式仕様記述言語は、ひととひとがとりあえず合意したことを俯瞰的に眺めること

を可能にしてくれます。第２章でも触れられているように構造、機能、状態という仕様の切り口はそれぞれ可視化の手段を（UML などの功績により）与えられています。
こうした正確で網羅的なイメージが提供されることによって、課題に対する “What if ... （もしこうならどうなる？）” の試行を行いやすくなります。これは課題の中で新しいアイデアを発見するためのインスピレーションを与えてくれるかもしれません。
次に「仕様を厳密に記述するとは何か」ですが、（１）課題側から眺めたシステムの性質が Validation という観点で厳密に確認できること、（２）設計側からみたときに開発されたシステムが仕様に従っているということを厳密に Verify しやすくなること、というのが簡単なお答えです。しかしその「厳密さ」にはレベルがあり、証明レベルの厳密さからプログラムのテストレベルでの厳密さまで程度はさまざまです。この話題は第２章の最後の方で触れています。もし日本語で仕様書を書いていたとしたら、テストはおろか構文や相互参照の一貫性などを効率良く確認することもできないのですから、厳密な仕様記述を目標に機械処理可能な形式仕様記述言語を利用する価値はあります。
最後に「曖昧な仕様書が渦巻く状態からどのように脱出できるのか」ですが、ご指摘のように仕様「書」が沢山ある状況はそれだけで重複や転記ミスを誘発しやすく、誰が持っている仕様書が最新なのか、確認済なのかよく分からないという問題もしばしば引き起こします。言葉で書いてあるが故に、人間の認知の癖に引きずられて誤解が生じたり、仕様を批判しているだけなのに、何となく人格を否定されているように感じたりということも現場では見かける風景です。形式仕様記述を用いた記述は、記述対象の問題を人間から切り離し、客観的な議論へと誘う効果があります。しばしば説明に使うのが「問題解決の構図を、あなた vs わたしから、問題 vs 私達へと転換しようというものです」コミュニケーションの質が上がり、課題が客観視できるようになれば、仕様もまた扱いやすいものになるでしょう。

第２章 厳密な仕様記述のための手法紹介

この章について

第１章では仕様の位置付けを、課題・仕様・設計の切り口で説明したり、妥当性確認 - Validation と正当性確認 - Verification の観点から説明したりしました。また仕様の品質についても説明を行いました。第２章ではこの文脈に基づき具体的に厳密な仕様記述を行うための手法を紹介します。なお本書では厳密な記述に関わる箇所では形式仕様記述言語の一つ VDM を具体的な説明の道具に使用します。

一口に形式手法と言ってもその適用の程度はさまざまです。本章の最後では形式手法の適用レベルを４段階に分けて、現場での適用の観点からの説明も行います。なお形式仕様記述を用いた実際の開発工程の姿や具体的に現在のプロジェクトへの導入を行うためのヒントと解説は第３章、第４章で説明します。

1. 仕様に書くべき項目とその表現方法

第１章でも説明したとおり、課題の世界で生み出された「要求」を、開発対象のシステムの性質として定義したものが「仕様」ということになります。さてここで「性質」と一口に表現してしまいましたが、その具体的な表現方法にはどのようなものがあるのでしょうか。

そこで記述される「性質」は課題の要素に対して確認 - Validate できなければなりませんし、作り出された設計から見て検証 - Verification の対象になるものでなければなりません。

かつての構造化分析設計[1]の時代に端を発し、初期のオブジェクト指向方法論[2]や、現在の UML[3] の中にも受け継がれている代表的な仕様記述のスタイルがあります。

それらは構造、機能、状態の記述です。またそれらを修飾する形で時間の制約も記述することができます。

以下にまずそれらの概要を説明します。

構造の記述

「構造」は仕様化しようとしているシステムの中に現れる各種の実体間の関係を記述したものです。ER モデルとか UML のクラス図といった「実体間の関係を表すモデル」を仕様ではしばしば作成します。

たとえばこれから特急券の予約システムを開発したいと思っている場合、その中に現れる「路線」や「駅」という概念を表現する必要があるでしょう。以下の図では UML 表記を用いて「路線」が複数の「駅」から構成されていることを表現しています。この図は UML の「クラス図」と呼ばれるもので、具体的な個別の駅（たとえば「新宿」「町田」など）を抽象した「駅」クラスと、具体的な路線（たとえば「小田急線」）を抽象した「路線」クラスの関係を表しています。

各駅は複数の駅と「隣接駅」というラベルのついた関連で結び付けられています。

「路線」「駅」などは具体的な「路線」や「駅」の実体の抽象を表し、その中に現れる「路線名」「駅名」などが、各実体の属性を表しています。この図から読み取れることは沢山ありますが、データ構造の設計の開始点などとして用いることができるでしょう。

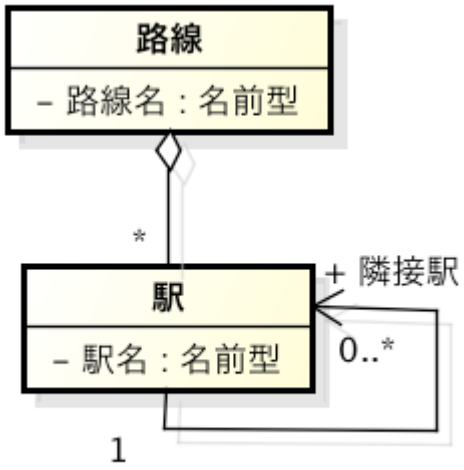


図 2-1 「路線」と「駅」の関係

しかし、図だけでは定義しきれないことがあります。たとえばこのモデルには「ある駅の隣接駅として指定された駅は、もとの駅を隣接駅として指し返さなければならない」といった暗黙的な制約が欠落しています。先のクラス図を展開し、実際のインスタンス同士の組合せのレベル（オブジェクト図という）で考えてみましょう。

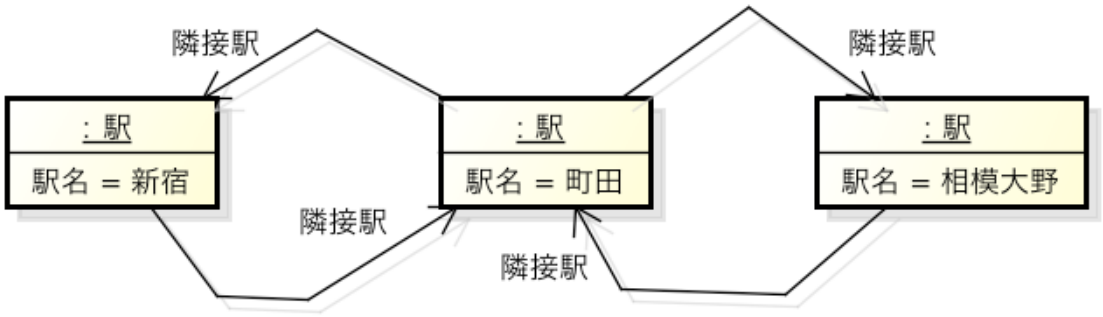


図 2-2 駅同士のリンクを示したオブジェクト図

ここでは小田急線の「新宿」「町田」「相模大野」駅同士を「隣接駅」というリンクで結んでいます（なお実際の小田急線では「新宿」と「町田」の間には沢山の駅がありますが、ここではとりあえず上記のように並んでいるとします)。町田から見て新宿と相模大野が「隣接駅」リンクとして指している駅ですが、同時に新宿と相模大野からもそれぞれ「町田」を隣接駅として指しています。これは「隣接駅」という関連の制約なのですが、それは最初のクラス図にはうまく表現されていないのです。

もちろんこうした制約を、図の「注釈」として書き込んでいくことは可能です。しかしそうした、アドホックな記法で制約を書き込んだ場合、モデルの正当性を検証しようとする際に機械を使った自動化をあまり望むことはできません。人間が上のような図を見てその正当性を検証してやらなければならないのです。モデルの中で「駅」同士が「隣接駅」というリンクを介してつながっていることは、クラス図の方の制約から保証することができますが、より細かい制約条件を注釈だけで正確に定義してそれをきちんと評価することは難しいことです。

まして仕様変更が頻繁に起こるような状況ではそれを間違いなく遂行することは困難です。

厳密な仕様記述はこのような「図では記述しきれない」部分も補いながら、人間の作業を支えてくれる技術になります。

上記のような例でも、形式仕様記述言語を用いることにより厳密な仕様を実現しやすくなるのです。より詳しい説明は後述の例題を通して行います。

構造の　Validation は、ある実体を構成する属性の過不足や型、ある実体から他の実体への誘導可能性、関連の多重度、そしてその構造の上に定義された表明（＝成り立つべき条件）などを見ていくことで行うことができます。

機能の記述

ある意味「機能」が一番「仕様らしい」記述かもしれません。ここで言う「機能」とは簡単に言えば、入力に対してどのような出力が得られるべきかを定義したものです。

ただし、ここで述べる「機能」とはあくまでも情報間の関係であって、何かの「動作」を表すものではありません。「動作」はむしろ次の節で解説する「状態」（とその遷移）によって仕様が表現されます。

機能仕様を表現する際には【事前状態】と【入力】から、どのような【事後状態】と【出力】を得るかが目的になります。なにか変なことが起きたら【例外】が起きるべきですが、これもまた仕様として定義することができます。

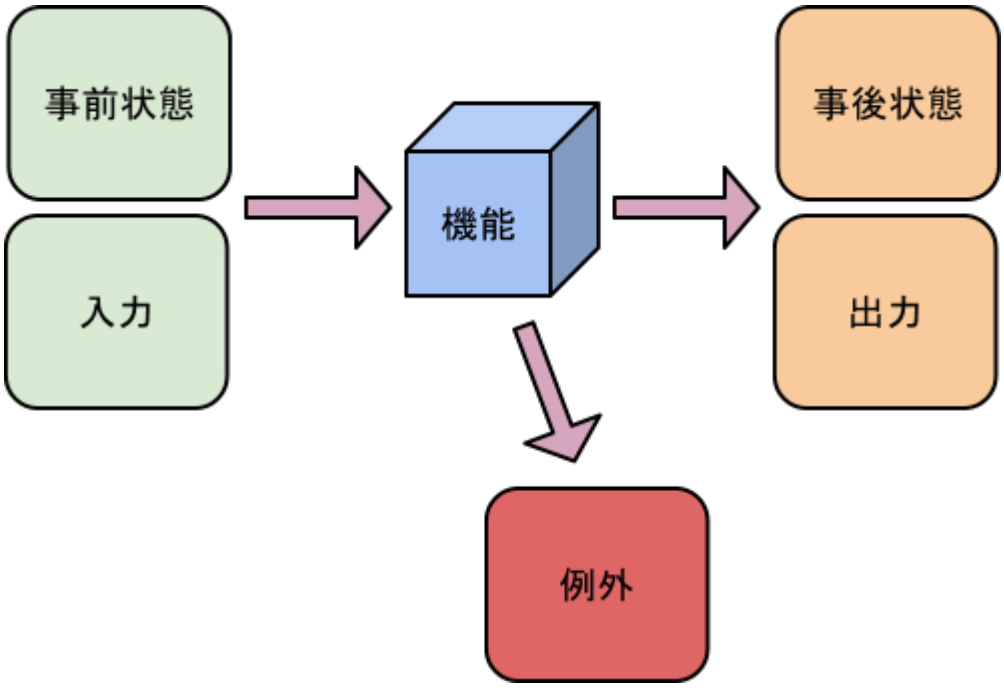


図 2-3 「機能」の定義を取り巻く要素

大切なことは「どのような『事後状態と出力』を得るか」を定義することであって、「『事後状態と出力』をどのように得るか」は設計の仕事であるということです。

たとえば簡単な例題として以下のようなものを考えましょう。

ある映画館の座席予約を考えます。事前状態は「現在の予約状況」です。これに対して入力として席数を指定して「機能」を適用します。機能適用の結果、「事後状態」として「更新された現在の予約状況」が返され、出力としては確保された座席の情報が返されます。

ここで「座席の予約」はある基準に沿って行われるものとします。「ある基準を満たすように座席を選択するアルゴリズム」はここではまだ決まっていません。

こうした場合、多くの「機能」仕様書は「具体的に座席をどのように割り当てるのか、そのアルゴリズムはどのようなものか」の記述に力が注がれています。もちろんそれも

大切なことなのですが、その前に仕様としては「どのような結果が得られれば、検証 - verification ができるのか」をはっきりさせなければなりません。

```
予約席割り当て : 現在座席予約 * 席数 -> 現在座席予約 * set of 座席
```

図 2-4 「予約席割り当て」機能のシグネチャ

この表現は「予約割り当て」という名の機能が、現在座席予約型（事前状態）と席数型（入力）を受け取って、現在座席予約型（事後状態）と座席の集合を返すものであるということを表わしています。この部分はシグネチャと呼ばれます。さらに詳細な部分を見てみましょう。

```
予約席割り当て : 現在座席予約 * 席数 -> 現在座席予約 * set of 座席
予約席割り当て(予約席, 数) == is not yet specified
pre
  card 空席(予約席) >= 数
post
  let mk_(新予約席, 今回予約席) = RESULT in
     予約席 inter 今回予約席 = {} and
     新予約席 = 予約席 union 今回予約席;
```

図 2-5 「予約席割り当て」の事前条件(pre)と事後条件(post)

こうした記述は後ほどより詳しく解説しますが、ここで示された「予約席割り当て」という機能の仕様は大きく分けて三つの部分から成り立っています。

（1）シグネチャ：入出力の状態、入出力などの型を定義する部分

```
予約席割り当て : 現在座席予約 * 席数 -> 現在座席予約 * set of 座席
予約席割り当て(予約席, 数) == is not yet specified
```

図 2-6 シグネチャ

ここでは仮引数として、予約席、数を指定しています。それぞれ現在座席予約型、席数型の引数ということになります。残りの二つは事前条件と事後条件と呼ばれる部分で、機能適用の前後に成り立っている条件を定義した部分です。

事前条件は事前状態と入力、事後条件は事後状態と出力から構成された条件式です。

（２）事前条件：機能適用に際して最初に成り立っていなければならない条件

```
pre
  card 空席(予約席) >= 数
```

図 2-7 事前条件

予約席から分かる空席数が、指定した数よりも多いことがこの機能の適用の条件です。もしこの条件が成り立っていない場合には、この機能を適用してはいけないことになります。

（３）事後条件：機能適用後に成り立っていなければならない条件

```
post
  let mk_(新予約席, 今回予約席) = RESULT in
     予約席 inter 今回予約席 = {} and
     新予約席 = 予約席 union 今回予約席;
```

図 2-8 事後条件

RESULT は機能の戻り値が入っていることを前提とした特殊変数です。ここでは、機能の戻り値が「どのような条件を満たせばよいのか」についての定義を行なっています。RESULT は「新しい予約席全体の情報」と、「今回の予約割り当てによって新たに割り当てられた座席の情報」で構成されています。これをそれぞれ「新予約席」「今回予約席」に割り振ります（主に読みやすさの観点です)。

このあと

（１）機能を呼び出す前の、もともとの「予約席」と、「今回予約席」の間に共通要素がない（２）もともとの「予約席」と「今回予約席」の和集合が「新予約席」と一致することの二つが「満たすべき条件」として指定されています。

この段階ではどのように「今回予約席」を求めるべきかが書いてありませんが、「機能の満たすべき性質」は理解できたことになります。機能を厳密に定義するより詳しい説明も本章の中で後述します。

機能の Validation は、定義した入出力の関係が課題の要求する計算結果に一致しているかを確認することによって行われます。

状態の記述

「状態」とは仕様記述を行おうとする対象のシステムの中で人間が区別したい状況を区分したものです。モデルの内部では、属性の組合せから構成される式が満たしている条件を、Validation を行うために意味のあるグループへと分類したものとして表現されます。とはいえ、意味のある属性が見つからず単に「状態」を表す属性を使って状態の違いを区別することもよく行われます。

しばしばイベントとも呼ばれる事象の発生と関連付けられて、「事象を契機とした状態の遷移」がモデル化の対象となります。

たとえば自動ドアの制御を考えてみると、制御システムは少なくとも【ドア開】と【ドア閉】の状態を区別しておく必要はありそうです。なんらかの事象が起きることによって、制御システムは状態を変化させるべく動作を行うことになります。

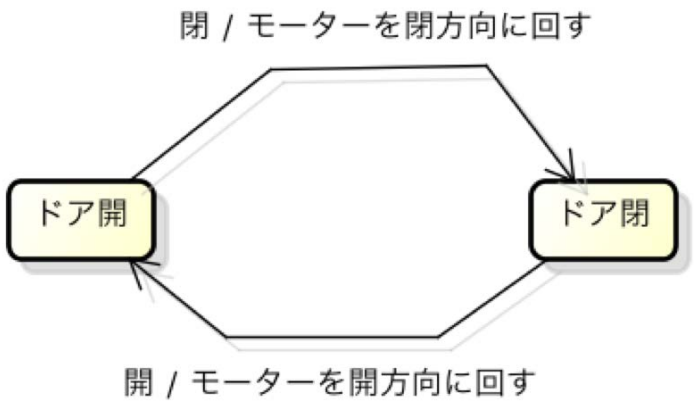


図 2-9 ドアの開閉に関わる単純な状態遷移図

ここでは極端に単純化した振る舞いとして示しました。先の節で状態遷移モデルに関するもう少し詳しい解説を行います。

状態の Validation は望ましい振る舞いを表す各「状態」が、構造と機能の組み合わせによって間違いのない範囲できちんと遷移しているかを確認することによって行われます。

時間の記述について

仕様記述を行う際に扱いたいものの一つに「時間」があります。この「時間」は特定の時刻に起きるべき事象、留まるべき状態、とり得る情報の構造などを表現したり、ある機能の遂行にかかる時間を指定したりといった場所に登場します。

先に考えた自動ドアの制御をまた使いましょう。誰かが自動ドアのスイッチを押すとドアは開き、一定の時間が経過したあとでドアを閉じ始めなければなりません。

つまり【ドア開】の状態に入ってから一定時間がすぎたら【ドア閉め中】の状態に移行し実際にモーターを回してドアを閉めなければならないということです。

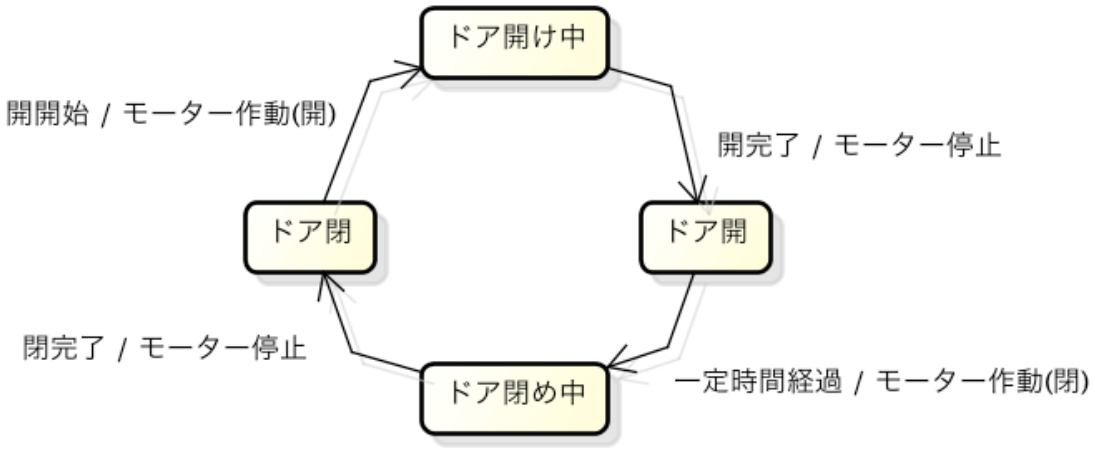


図 2-10 途中の状態も含めたドアの開閉に関わる単純な状態遷移図

時間特性が仕様の中に入り込んでくるようになると、仕様のレベルだけでその性質を確認/検証することは難しくなってきます。時刻は物理的特性と結びついていて結局設計実装された対象システムの振る舞いが仕様で指定された条件範囲内に落ち着くということを、実際のテストを通して確認することしかできないからです。

読者の方へ

さてここまでで、仕様を記述するための切り口である「構造」「機能」「状態」（と時間）の切り口を簡単に説明しました。本章の次節以降では、それぞれの内容をもう少し詳細に説明していきます。細かい話もありますので、技術的詳細に踏み込む前に、プロジェクトに対する影響や導入に関する話題をもっと読みたいという方は、この後を読む前に先に第３章または第４章へお進みください。

2. 厳密な仕様定義の手順例

仕様を構成する個々の要素を説明する前に、本書が想定する厳密な仕様定義の大きな流れをまず説明しておきましょう。第１章で説明したように、仕様に先立ちなんらかの形で課題（事実と願望）が抽出されていることが前提です。この課題の抽出方法そのものは本書では議論しませんが、たとえば（１）「規格書」、「法律」、「業務ルール」「外部システムの仕様」などが「事実」の例で、（２）いわゆる「要求文書」や「企画」「改善案」などが「願望」の例となります。

これらの課題を基に仕様（本書では主に構造、機能、状態のモデル）が定義され、仕様の課題に対する妥当性確認 - Validation と 正当性検証 - Verification を経て、必要ならば繰り返し仕様が改訂されるという流れになります。概念的な流れを次の図に示しました。

この図では、「課題（事実＋願望）の管理」と書かれた部分では外部とのコミュニケーションを取りながら、さまざまな課題が抽出定義されています。この内容を基に、課題の世界における価値を生み出すための仕様が「仕様の策定」という場所で定義され、続けて「仕様の確認と検証」により妥当性確認と正当性検証が行われます。「仕様の策定」の部分では厳密な仕様記述を用いることで、仕様策定担当者同士でのコミュニケーションの品質が向上します。また「仕様の確認と検証」の部分では厳密な仕様記述を用いることで品質保証を行うチームと仕様策定チーム間の協力が推進しやすくなります。

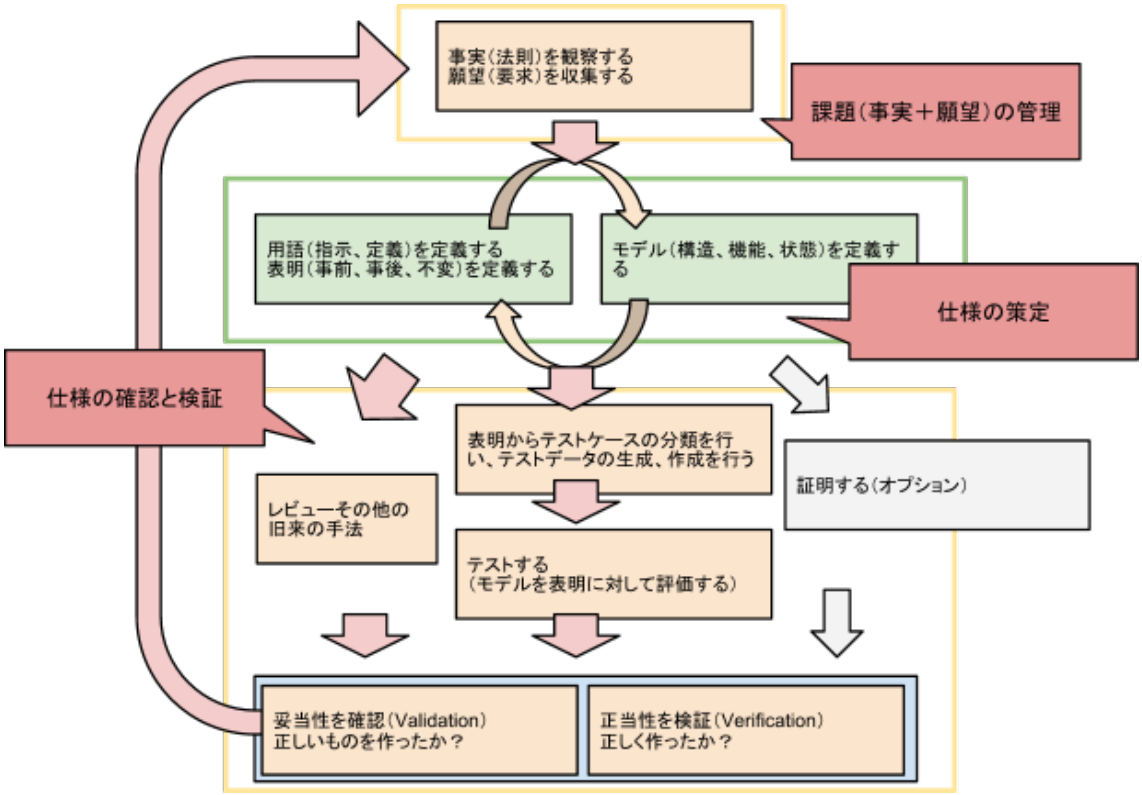


図 2-11 厳密な仕様定義の手順例

なお本書では、前図に示したように「証明」はオプションとして扱っています（8 節「 形式仕様記述と適用のレベル」 で解説）。

3. 自然言語と厳密な記述、そして形式仕様記述言語

より詳しい構造・機能・状態の記述方法を説明する前に、開発現場で普通に馴染んだ「自然言語」 と、 厳密な記述、 そして形式仕様記述言語の関係について説明をしましょう。

自然言語（日本語も、英語も）は人と人がコミュニケーションを深めるための道具です。もちろん芸術的なメッセージを伝える用途もありますが、ここでは「仕様」という文脈で、正確な意味の伝達に的を絞って考えることにします。

以下に示したのは、厳密な記述を支える要素である論理記号と、自然言語である日本語と英語、そして厳密な記述を機械処理することを目的として提供される形式仕様記述言語 VDM の対応表です（なおこの表は [4] に掲載のものを翻案したものです）。

論理記号	日本語	英語	VDM
¬	でない/以外	not	not
∧	かつ/及び/さらに/と/,	and	and
∨	または/あるいは/や/か/,	or	or
⇒	ならば/のとき/とすると	implies then	=>
⇔	ならば/同値/つまり/言い換えると	if and only if equivalent	<=>
∃x	ある/存在する	there exists x such that	exists x
∀x	すべての/必ず/任意の	for all x ...	forall x

表 2-1：論理記号と自然言語と形式仕様記述言語の対応

この表に挙げた論理記号を用いて、仕様に現れる自然言語の記述を論理式に置き換えていくことが可能です。もちろん任意の文章を置き換えられるわけではなく、仕様として表現される、構造、機能、状態などの記述がまず対象となります。

たとえば「なぜそのようなことをしたいのか」といった「理由や動機」は仕様ではなく第 1 章で説明した「課題」の中の適切な部分に自然言語で書かれることが普通です。

本章では仕様としての論理式を、機械処理可能な形式仕様記述言語で表現していく方法を例を使って説明しますが、まずは日本語の文章と論理式の対応を考える部分から始めましょう。

指示と定義と論理式

どのような手法を使うにせよ、自然言語による記述も含めて開発文書の中では「指示 - designation」と「定義 - definition」の区別をはっきりさせなければなりません。

ここで、「指示」とは現実世界との直接的な対応関係を表す「用語」であり、「定義」とは「指示」ならびに他の「定義」を用いて作られる「用語」です。[5]
このように「指示」と「定義」を分ける理由は、現実世界との接点をなるべく狭い範囲に保ち仕様の中に現れる用語の管理を分かりやすいものにしたいからです。

まず「指示」の簡単な例を挙げてみましょう

指示の例：

親子(x,y) ＝ x は y の親である

年長(x,y) ＝ x は y より先に生まれた

男(x) ＝ x は男性である

女(x) ＝ x は女性である

指示は現実世界と対応する用語ですので、右辺は普通の言葉で書いてあります。もちろん既存の数式などによる表記を書いても構いません。こういう指示を得た後で、文書中に**親子(x,y)** という用語を書いたら、それは「**x は y の親である**」という意味だということです。現実の世界における「親」という概念は人間が知っているだけで、システム側ではそれを既知のものとしてただ記号化しているだけです。

次に簡単な「定義」の例を示しましょう。定義は既存の「指示」または「定義」だけを使い、論理記号で式として組み上げたものです。

定義の例：

兄弟姉妹(x,y) ＝ x ≠ y ∧ ∃p･親子(p,x) ∧親子(p,y)

ちなみに、この定義はもう少し普通の日本語で書いたならば以下のようになるでしょう。

異なるxとyに共通の親pがいるとき、xとyの関係は兄弟姉妹である。

ただし、この文章では最初の論理式との対応関係がはっきりしません。最初の「兄弟姉妹」の定義を少しずつ日本語に置き換えてみましょう。

兄弟姉妹(x,y) ＝x ≠ y ∧ ∃p･親子(p,x) ∧親子(p,y)

↓

兄弟姉妹(x,y)とは、xとyが異なるとき、あるpが存在して
･親子(p,x) かつ 親子(p,y)

↓

親子(p,x) かつ 親子(p,y)となるpがいるとき、相異なるx,y間に兄弟姉妹(x,y)が成り立つ

↓

相異なるxとyに共通の親pがいるとき、兄弟姉妹(x,y)が成り立つ

↓

相異なるxとyに共通の親pがいるとき、xとyの関係は兄弟姉妹である。

回りくどいやり方のようですが、先に示した「定義」は、既に定義された指示である「親子」だけを項として使って記述されています（日本語の表現の方では「親子」という用語を使わず同じ概念が「親がいるとき」といった表現の中に埋め込まれています)。兄弟姉妹は「親子」にだけ依存して定義されていますので、「親子関係」さえきちんと現実世界と対応付けられれば、兄弟姉妹関係はその情報から判断することが可能です。

より複雑な定義も、既存の指示や定義を組み合わせて作り上げていくことができます。そうした際には先の表で挙げた ∃ や ∧ といった論理記号を使うことになります。ここで大切なことは「現実世界と接しているところは指示だけ」であり、後は定義を用いて論理的に構築されているだけなので、現実世界の変化に対応しやすいということなのです。

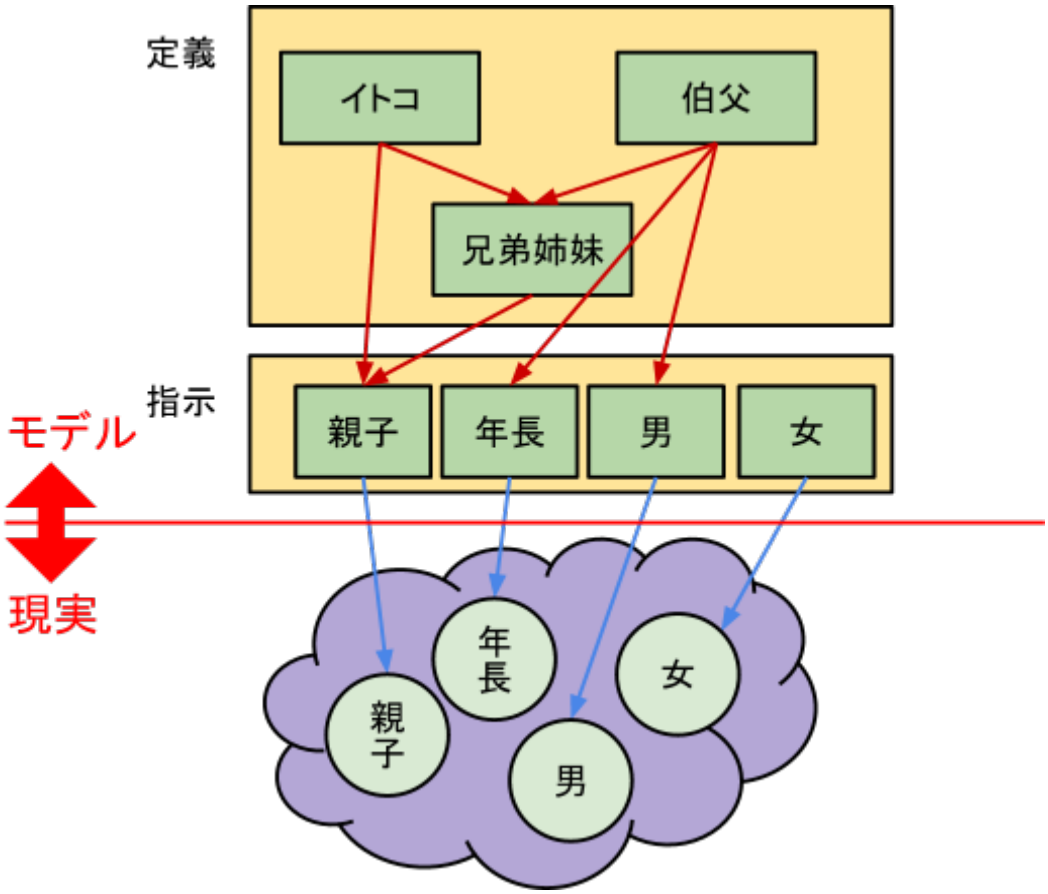


図 2-12 現実世界とつながる指示と定義

それでは次の文はどうでしょうか。

x が y の親よりも年長の男の兄弟姉妹である場合、x は y の伯父である。

これを論理結合子と既存の指示、定義を用いて、新しい定義として書き直すと以下のようになります。

定義の例：

$$伯父(x,y) = \exists p \cdot 年長(x,p) \wedge 男(x) \wedge 兄弟姉妹(x,p) \wedge 親子(p,y)$$

なお、叔父の場合は定義の一部が変わって

定義の例：

$$叔父(x,y) = \exists p \cdot 年長(p,x) \wedge 男(x) \wedge 兄弟姉妹(x,p) \wedge 親子(p,y)$$

となります。

ここで説明のために使っている、伯父、叔父、年長、男、女、親子、兄弟姉妹という用語は、読者にはお馴染みの概念ですので、あまり違和感がなかったと思います。むしろ「分かりきったことをなぜ難しく書くのだろう」とさえ思ったかもしれませんね。
しかし、実際の開発で登場する用語はそれほど「分かりきった」ものばかりではありません。むしろ細かい用語の解釈の違いが無数に存在する場合が普通です。

たとえば以下の定義があったとします。

定義：

$$休日(x) = 土曜日(x) \vee 日曜日(x) \vee 年末年始(x) \vee 祝祭日(x)$$

これを見たときに分かることは「土曜日、日曜日、年末年始、祝祭日は休日だな」ということですが、土曜日、日曜日はともかく、年末年始やそして祝祭日の定義はここから

はすぐには読み取れません。いや祝祭日は簡単なのでは、と思われるかもしれませんが、いま何年の暦のことを考えているのかによって答は変わります（祝祭日は法律が変わると変化するので、どの年の話をしているかが大切です）。
通常の開発作業では、こうした「何時の時点の何の定義をしているのだ？」という疑問が山のように飛び交います。こうした定義が普通の日本語の文章の奥に書かれていたとしても、検索することもできなければ、テストすることもできません。レビューで考えられる範囲にも限りがあります。そして何より人間は「忘れてしまう」いきものです。読者の皆さんはあるプロジェクトで書いた一つの定義を１年後に戻ってきて思い出すことができるでしょうか。

さて、ここまでに示してきた論理式の中に現れる x, y, p という記号は何でしょう。ここまでは読み手の常識として

$$伯父(x,y) = \exists p \cdot 年長(x,p) \wedge 男(x) \wedge 兄弟姉妹(x,p) \wedge 親子(p,y)$$

この x, y, p は「人」を表わしているな、とか

$$休日(x) = 土曜日(x) \vee 日曜日(x) \vee 年末年始(x) \vee 祝祭日(x)$$

この x は「日付」のようなものかな。といった判断を行なっていたと思います。しかし、この判断は問題領域（ここでは血縁関係や休日に関する話題）についてある程度知っているからこそ働く判断に過ぎません。もしもっと複雑な問題領域（たとえば税金の計算とか）を扱わなければならなくなった場合には「この x はどんな性質のものだっただろう。そもそも x にこの式を当てはめてよかったのだろうか」ということを考えなければならなくなります。

そしてそこで、人間がいちいち考えなければならいとすると、結局間違いがさまざまなところに入り込んでいしまいますから、現在の仕様書の状況と大きくは変わらないということになってしまうかもしれません。

x, y, p がどのようなものであるかを正確に表現するためには「型」と呼ばれる概念が必要です。型を添えて論理式を書くことにより、「何について定義しているのか」がより明確になります。

たとえば

$$伯父(x,y) = \exists p \cdot 年長(x,p) \wedge 男(x) \wedge 兄弟姉妹(x,p) \wedge 親子(p,y)$$

ただし x, y, p は「人型」

と書くこともできますし、あるいは

$$伯父(x:人型,y:人型) = \exists p{:}人型 \cdot 年長(x,p) \wedge 男(x) \wedge 兄弟姉妹(x,p) \wedge 親子(p,y)$$

といった書き方をすることもできるでしょう。とはいえこのまま記述がどんどん増えていったとしたら、普通のエディタで管理するのも大変ですし、相互に式が矛盾しているかのチェックも膨大な手間になりそうです。ではどうしたらよいのでしょう。

ここでようやく形式仕様記述言語の出番となります。

形式仕様記述言語の導入

さて「用語」とはなんらかの「意味」を表現するために使われるものです。

たとえば「伯父」という用語の意味は？と訊ねた場合には、定義によりそれは

$$\exists p \cdot 年長(x,p) \wedge 男(x) \wedge 兄弟姉妹(x,p) \wedge 親子(p,y)$$

ということになります。ではその中の年長、男、兄弟姉妹、親子の意味は？と訊ねるとさらにそれらを定義している他の「定義」や「指示」へと辿られていくことになります。ここまでの例題では、用語はみな論理値を返すだけでしたので、式も純粋に論理値を組み合わせる論理式だけでしたが、論理値の代わりに任意の型の値を返すようにすることも可能です。この場合「式」は任意の型を用いて定義できるものとなり、末端まで展開したあとで計算された値を式の中で評価していけば、最終的にもともとの「用語」が表している「値」を計算することも可能になります。

たとえば

売上金額 (x)

という「用語」の「定義」を考えてみることにしましょう。

売上金額 (x) ＝ 小売額(x) ＋ 消費税額(小売額(x))

消費税額 (a) ＝ a ＊ 消費税率

消費税利率 ＝ 0.05

売上金額という用語の「定義」は、物品　x の小売額に消費税額を加えたものであり、消費税額の「定義」は売上額 a に消費税率を掛けたものです（ここまで説明したように、どこかで指示に辿り着きます。小売額や消費税率などはおそらく指示として扱うことができるでしょう）。
結局「指示」も「定義」も形式仕様記述言語の世界では「型」や「関数」を組み合わせて表現されることになります。

多くの形式仕様記述言語は、前節で説明した論理式表現を素直に書けるような道具立てを提供しています。もちろん任意の型を用いた計算式を定義することもできます。本書では形式仕様記述言語を用いた例は全て VDM で記述しますが、他の形式仕様記述言語（Z, B など）でも似たような記述が可能です。

さて前節で説明したさまざまな血縁関係の論理式を VDM を使って表現してみましょう。
まずは必要な型を定義します。

```
public 文字列型 = seq of char;
public 日付型 = int;
public 性別型 = <男>|<女>;

public 人型 :: 名前 : 文字列型
              誕生日 : 日付型
              性 : 性別型
              父親 : [人型]
              母親 : [人型];
```

図 2-13 型の定義

ここでは四つの型が定義されています。文字列型、日付型、性別型、そして人型です。中心となるのは人型です。人型の定義は、普通のプログラミング言語におけるレコード型もしくは構造体のようなもので、複数の属性を持つ型を定義しています。public という指定は（上の兄弟姉妹のところでも出てきましたが）それぞれ対応する要素が外部から見えるということを示しています。
この定義によれば、人型には五つの属性が定義されています。すなわち、名前、誕生日、性、父親、母親です。それぞれの属性の型も同時に定義されていることが分かります。

父親、母親属性の型の定義の場所に [人型] と書かれているのは、この属性に指定されるデータの型が人型のものであるか、あるいは nil （即ち定義されない）であることを意味しています。

これらの型を用いた関数をいくつか定義します。

```
public 男？：人型 -> bool
男？(x) == x.性 = <男>;

public 女？：人型 -> bool
女？(x) == x.性 = <女>;

public 年長？：人型 * 人型 -> bool
年長？(x, y) == x.誕生日 < y.誕生日;

public 親子？：人型 * 人型 -> bool
親子？(p, c) == (p = c.父親) or (p = c.母親);

public 兄弟姉妹？：人型 * 人型 -> bool
兄弟姉妹？ (x, y) ==
   x <> y and
   exists p in set {x.父親, y.父親, x.母親, y.母親}
    & 親子？(p, x) and 親子？(p, y);
```

図 2-14 基礎的な関数の定義

男？、女？、年長？、親子？、兄弟姉妹？という関数が定義されています。ここでは日付を単なる整数として扱っているので、年長？の中身は誕生日の数値の大小だけで決まります（この先の例ではもっと複雑な日付の例も説明します)。

ここで通常の論理式で書かれた兄弟姉妹の定義を再掲します

$$兄弟姉妹(x,y) = \exists p \cdot 親子(p,x) \wedge 親子(p,y)$$

VDM の定義と、この論理式の一番大きな差は、VDM では x, y, p の型や定義されている範囲が明確にされていることと、論理値(bool) を計算して返す関数として定義を行なっているところです。この例では論理値を返す関数名の最後に”？”を付加して、可読性を上げています。こうした定義を積み上げて、より大きな「用語集」を作り上げることが可能です。

これを VDM の言語処理系で処理すると、型の整合性が機械によって検査されます。

論理値 (bool) を返す関数だけではなく、人型を扱う関数も用意します。

```
public 親 : 人型 -> set of 人型
親(x) == {x.父親, x.母親}\{nil};
```

図 2-15 親の集合を求める関数

この関数は、ある人型のデータを与えると、その両親を人型の集合 （set of 人型）として返す関数です。もし単純に {x.父親, y.母親} とだけ書くと、 x の父親や母親が未定義の場合には nil を要素に持つ集合が返されてしまうことになります。しかしそれは型が合わないことになるため、要素に nil があるときにはそれを取り除くようにしています。 “＼{nil}” という部分がそれを指定しています。 “＼” （バックスラッシュ）は集合演算子で、前の集合から後の集合の要素を取り除く役割を果たしています（端末によっては¥で表示されます）。よってもし x の両親が共に nil である場合にはこの関数は “{}” （空集合）を戻り値として返します。

もう一つ関数を定義しておきましょう。

```
public 子 : set of 人型 * 人型 -> set of 人型
子(allp, x) == {c | c in set allp & 親子？(x, c)};
```

図 2-16 子の集合を求める関数

この関数「子」は、人型 x の子供の集合を set of 人型として返します。なおこの関数は入力の引数としても set of 人型 を渡して、それを allp という仮引数で参照しています。それはなぜでしょうか。ここで人型の定義を思い出してみましょう。

```
public 人型 :: 名前 : 文字列型
              誕生日 : 日付型
              性 : 性別型
              父親 : [人型]
              母親 : [人型];
```

図 2-17 人型の定義（再掲）

このように人型は、親に関する知識は持っていますが、自分の子に対する知識は持っていないのです。このため引数で与えられた人型の集合 allp の中から、自分 x と親子関係にある人型の要素を選び出して返そうとしているのです。
集合を作っている部分だけをクローズアップします。

```
{c | c in set allp & 親子？(x, c)}
```

図 2-18 親子関係にある要素だけを抽出

この式は VDM の集合内包表記という記述形式です。

{ c | … } という部分をみると、この式は c に関する集合を作り出すものだということが分かります。次にその … の部分をみてみると、 c in set allp によって c が allp という集合の要素であることが分かります。もしこれだけだと、返される集合は allp と同じものになります（allp から集められた要素をそのまま使って新しい集合を作っただけなので）。
しかし & とそれに続く式を使って、選ばれる c を絞り込むことが可能です。 & の後に書かれる式は論理値 (bool) を返さなければなりません。この例では論理値を返す「親

子？」という関数が指定されています。ここには 「親子？(x, c) 」と書いてありますから、結局入力で指定された x に対して、allp から選ばれた全人型要素のうち「親子？(x, c)」が 真(true) を返す c だけが選ばれて、{ c | … } の集合の値となります。
これは結局「子」という関数の戻り値として返されることになります。

さて、ここまで道具が揃えば、

$$伯父(x,y) = \exists p \cdot 年長(x,p) \wedge 男(x) \wedge 兄弟姉妹(x,p) \wedge 親子(p,y)$$

に対応する VDM 記述を与えるのは簡単です。伯父だけではなく、ついでに叔父の定義も与えておきましょう。

```
public 伯父？：人型 * 人型 -> bool
伯父？(x, y) == let parents = 親(y)
            in exists p in set parents
              & (年長？(x, p) and 男？(x) and 兄弟姉妹？(x, p));

public 叔父？：人型 * 人型 -> bool
叔父？(x, y) == let parents = 親(y)
            in exists p in set parents
              & (年長？(p, x) and 男？(x) and 兄弟姉妹？(x, p));
```

図 2-19 伯父と叔父の問合せ関数の定義

「伯父？」の定義の中をみてみましょう。
最初の let parents = 親(y) in の部分では、まずそれ以降で使う集合 parents を求めています。
let 式はまだ完結していなくて、in 以降の式を in 以前の式を参照しつつ評価します。
ここでは

```
exists p in set parents
        & (年長？(x, p) and 男？(x) and 兄弟姉妹？(x, p))
```

図 2-20 「伯父？」問合せ関数の一部

この式が評価対象です。普通の論理式だと

$$\exists p \cdot 年長(x,p) \wedge 男(x) \wedge 兄弟姉妹(x,p)$$

と書いたものに相当します。おや？前に出てきたときにはこの部分は

$$\exists p \cdot 年長(x,p) \wedge 男(x) \wedge 兄弟姉妹(x,p) \wedge 親子(p,y)$$

と書いてあった筈です。どうして省略されているのでしょう。実は let parents = 親(y) の部分で y の親の集合を求めていますから、仮に後半の式の中で 親子?(p, y) と書いても必ず true になるため省略してあるのです（もちろん全部記述しても構いませんし、仕様の読みやすさから言えば全部書く方が望ましいとも言えます)。

ともあれ、この後半の式

```
exists p in set parents
        & (年長？(x, p) and 男？(x) and 兄弟姉妹？(x, p))
```

図 2-21 「伯父？」問合せ関数の一部（前図の再掲）

は（y の親の集合である） parents の各要素 p の中で、年長？(x, p) and 男？(x) and 兄弟姉妹？(x, p) を満たす p が一つでもあれば真 (true) となります。

伯父？と叔父？の定義の違いは、自分の親の兄が伯父で弟が叔父というだけですので、定義もその部分だけが異なっています。具体的には「年長？」の引数の順序です。もちろん後者の場合「年長？(p, x)」と引数を入れ替えるのではなくて「not 年長？(x, p)」と書くこともできます。
付録に完全なソースを示した「血縁.vdmpp」の中には、例題用のデータも含まれています。

```
源義朝　= mk_人型("源義朝", 11230000, <男>, nil, nil);
由良御前 = mk_人型("由良御前", 11320000, <女>, nil, nil);
常磐御前 = mk_人型("常磐御前", 11380000, <女>, nil, nil);
源頼朝　= mk_人型("源頼朝", 11470509, <男>, 源義朝, 由良御前);
源義経　= mk_人型("源義経", 11590000, <男>, 源義朝, 常磐御前);
北条時政 = mk_人型("北条時政", 11380000, <男>, nil, nil);
伊東祐親の長女 = mk_人型("伊東祐親の長女", 0, <女>, nil, nil);
北条政子 = mk_人型("北条政子", 11570000, <女>, 北条時政, 伊東祐親の長女);
大姫　　= mk_人型("大姫", 11780000, <女>, 源頼朝, 北条政子);
源頼家　= mk_人型("源頼家", 11820911, <男>, 源頼朝, 北条政子);
若狭局　　= mk_人型("若狭局", 0, <女>, nil, nil);
足助重永の長女 = mk_人型("足助重永の長女", 0, <女>, nil, nil);
一幡　　= mk_人型("一幡", 11980000, <男>, 源頼家, 若狭局);
公暁　　= mk_人型("公暁", 12000000, <男>, 源頼家, 足助重永の長女);
源実朝　= mk_人型("源実朝", 11920917, <男>, 源頼朝, 北条政子);
静御前 = mk_人型("静御前", 0, <女>, nil, nil);
源義経の長男 = mk_人型("源義経の長男", 11860729, <男>, 源義経, 静御前);
```

図 2-22 源氏の家系を定義したデータ

ここでは

定数名 = mk_人型(名前, 誕生日, 性, 父親, 母親)

という形式で、データの定義を行なっています。本来は一番左の定数名の部分は不要な場合が多いのですが、説明のしやすさからあえて意味のある名前の定数として定義してあります。こうしたデータを用意した上で

```
叔父？(源義経, 源頼家)
```

図 2-23 式の評価

という式を評価してみると、確かに真 (true) が返されることが観察されます。

形式仕様記述言語で文の曖昧さを取り除く

さて、改めて通常の日本語表記の曖昧さを考えてみましょう。一見普通の表記のように見えても思わぬ曖昧性が隠されている場合があるのです。以下いくつかその曖昧な例と、それらの曖昧性がどのようにして形式仕様記述言語で取り除かれるかを示しましょう。

例１：A ならば B ではない

もし仕様書の中に A ならば B ではないという表現が出てきた場合、これは何を意味しているでしょうか？少し噛み砕いてみましょう

（1）A が成り立つ場合には B は成り立たない。A が成り立たない場合のことは分からない

（2）A が成り立つ場合には B は成り立たない。A が成り立たない場合には B は成り立つ

（3）「A が成り立つ場合には B が成り立つ」というわけではない

少し正確になったような気もしますが、これでもあまり分かりやすくなったとは言えませんね。ということでこの部分を論理式として考え VDM で表現した上で、A と B に関する真理表を書いてみましょう。

（1） A => not B

A	B	A => not B
true	true	false
true	false	true
false	true	true

false	false	true

表 2-2 真理表

（２）(A => (not B)) and ((not A) => B)

A	B	(A => (not B)) and ((not A) => B)
true	true	false
true	false	true
false	true	true
false	false	false

表 2-3 真理表

（３）not (A => B)

A	B	not (A => B)
true	true	false
true	false	true
false	true	false
false	false	false

表 2-4 真理表

ここで見たように、（１）、（２）、（３）の解釈は形式仕様記述言語を用いた論理式として表現すると明確になります。実際には機械的にこの評価を行うことができますので、人間は膨大な組合せの正しさを「目で」検査する必要から解放されます。

例２：A かつ B または C

これは文章の区切り方の曖昧性の例です。書き手の意図は

（１）（A かつ B）または C

（２）A かつ（B または C）

のどちらかであったかが曖昧なのです。もちろん注意深い人が日本語を書いた場合には、それぞれ

（１）A かつ B、または C

（２）A かつ、B または C

といった類の書き分けを行うかもしれません。しかし、書き間違いは常に起きますし、読み手の側も頭のなかで「、」を無視して続けて読んでしまうかもしれません。こうした状況も VDM 表記の場合

（１）(A and B) or C

（２）A and (B or C)

といった形で書き分けることができ、曖昧さを減らすことができます。

例３：すべての A に対して P(A, B) となる B が存在する

これはもう少し具体的な例として説明しましょう。たとえば以下のような文章はどうでしょうか。

「誰でも休日が好きだ」

特に問題は無さそうな気もします。まあこの位の例では誤解は起きないのかもしれないのですが、現場で書かれている仕様書の文中にはさまざまな可能性が残されています。たとえばもう少し詳しく書き込んでみたら、以下のような意味なのかもしれません。

（1）誰でも全ての休日が好きだ : つまり誰でも仕事が休みになれば嬉しいという意味
（2）皆が好きな休日が少なくとも一つある : つまり誰もが好きなある特定の休日 - たとえば「文化の日」- が一つはあるという意味
（3）皆が好きな休日が丁度一つだけある：つまり誰でもある一つの特定の休日 - たとえば「文化の日」 - が好きでそれ以外には全員に好かれている休日はないという意味
（4）誰でも自分が好きな休日が少なくとも一つある：たとえば太郎は「子供の日」が好きで花子は「春分の日」と「秋分の日」が好きといったこと
（5）誰でも丁度一つだけ好きな休日がある：たとえば太郎は「元日」だけが好きで、花子は「敬老の日」だけが好きという意味

これを VDM で書き直してみましょう。なお「好き」というのを表現するのに

```
好き : 人型 * 日付型 -> bool
```

図 2-24 「好き」関数のシグネチャ

という関数が定義されているとします。この関数は人型と日付型を与えると、真理値 (true か false) を返します。
また「全員」は人型の集合（set of 人型）で対象とする全員が要素として含まれているとします、また「全日付」は日付型の集合（set of 日付型）で対象とする全休日が要素として含まれているとします。

（1）forall p in set 全員 & (forall d in 全休日 & 好き(p, d))
（2）exists d in 全休日 & (forall p in set 全員 & 好き(p, d))
（3）exists1 d in 全休日 & (forall p in set 全員 & 好き(p, d))
（4）forall p in set 全員 & (exists d in 全休日 & 好き(p, d))
（5）forall p in set 全員 & (exists1 d in 全休日 & 好き(p, d))

些細な違いに気が付いたでしょうか。この表現も慣れないと簡単とは言えませんが、先の日本語で書かれた（1）～（4）の定義は、機械で処理することができませんので、仕様を実際のデータで検証しようとしても人間がレビューするしかありません。
これに対して VDM を使った記述なら、上の（1）～（4）のような論理式に対応する式を、機械的に検査することができるのです。
こうした特徴は、特に「要求が不安定で仕様が頻繁に変更される」ような状況には有利に働きます。なぜなら変更に対して仕様上矛盾が発生していないか否かを、機械を用いた「仕様の回帰テスト」を実施することができるからです。

4. 構造を記述する

仕様を表現する際には、記述の対象であるさまざまな概念間の構造をきちんと表現する必要があります。たとえば UML などに含まれるクラス図は、そうした目的を表現する一つの手段です。しかし「図であること」は、人間の直感に訴えるという点では優れた点もありますが、それを Validation あるいは Verification の手段として使おうとするときには、いろいろと物足りない点も出てきます。

さてここからは説明のために一つの例題を使いましょう。とある鉄道の特急券予約システムを考えることにします。もちろん説明のためですので大幅に簡略化されています。

例題：特急券予約

開発に際して、このような「初期要求文書」が渡されてきたとしましょう（まあこんな簡単で大雑把な文書であることは実際にはありませんけれど）。

- 特急券の予約を行う
- 特急の停車駅は「新宿」「向ヶ丘遊園」「新百合ヶ丘」「小田急多摩センター」「町田」「相模大野」「藤沢」「片瀬江ノ島」「本厚木」「小田原」である
- 予約に際しては列車を指定し、発駅、着駅、人数を与える
- 列車は複数の車両が連結されたものである

- 各座席は車両番号と座席番号で一意に指定される
- 一つの車両には複数の座席がある
- 一つの予約で確保される座席は複数の車両にまたがらない
- 指定した人数分の空席がない場合には予約は行われない
- 路線は新宿を起点とし相模大野で二つに分岐して、その先片瀬江ノ島と小田原それぞれを終点とする。前者は江ノ島線、後者は小田原線と呼ばれる
- 複数の座席を確保する際にはなるべく近くにする

この文書を整理して、システムの仕様を検討しやすくするためにはどうすればよいでしょうか。まずは整理の第一歩として、第1章でも説明した事実と願望の分離を試みましょう。

事実

- 特急の停車駅は「新宿」「向ヶ丘遊園」「新百合ヶ丘」「小田急多摩センター」「町田」「相模大野」「藤沢」「片瀬江ノ島」「本厚木」「小田原」である
- 列車は複数の車両が連結されたものである
- 各座席は車両番号と座席番号で一意に指定される
- 一つの車両には複数の座席がある
- 路線は新宿を起点とし相模大野で二つに分岐して、その先片瀬江ノ島と小田原それぞれを終点とする。前者は江ノ島線、後者は小田原線と呼ばれる

願望

- 特急券の予約を行う
- 予約に際しては列車を指定し、発駅、着駅、人数を与える
- 一つの予約で確保される座席は複数の車両にまたがらない
- 指定した人数分の空席がない場合には予約は行われない
- 複数の座席を確保する際にはなるべく近くにする

もちろん、実際の文書はもっと複雑ですからきれいに分離できるとは限りません（上の分離も少し怪しいところがあります)。しかし事実と願望を分けることによって、仕様をまとめていく際に、前提とすべき部分と、構築で考えるべき部分の区別がつけやすくなります。

次に用語の候補を検討してみましょう。

特急券、予約、停車駅、列車、発駅、着駅、人数、車両、座席、空席、路線、起点、分岐、終点

これだけだと、まだどれが「指示」でどれが「定義」かもはっきりしません。また全てが仕様を構成する用語としてすべて取り込まれるとは限りません。

候補を少しずつ眺めながら、お互いの関係をまとめていきます。たとえば、列車、車両、座席の関係は、先の「事実」の中の記述を眺めると

列車 = 車両が複数連結されたもの
車両 = 複数の座席を含むもの

といった定義候補が浮かび上がります。もちろんこのままでは、機械処理ができませんから、形式仕様記述言語を用いて厳密化することにします。完全な記述は付録(Reservation.vdmpp) に示しましたが、その中に以下のような定義を置きました。

```
public 編成型 = seq of 車両型;
public 車両型 = set of 座席型;
```

図 2-25 「編成型」と「車両型」

もともとの文章では「列車」とされていたものは「編成型」とされています。これは「列車」には編成情報だけではなく、発着駅の情報や列車を指定するための識別子である列

車番号をさらに指定してやりたかったからです。一方、個々の車両は少なくともこの段階では座席の集合としてみなせば十分だとして扱っています。

なおここに出てくる seq of、set of という定義はそれぞれ VDM における「列」（順序のあるデータの集まり）と「集合」（順序のないデータの集まり。数学の集合と似た概念）を表すキーワードです。

列車は列車型としてまとめることとし、以下のような定義としてみました。

```
class 列車仕様 is subclass of 路線仕様
types
 public 車両型 = set of 座席型;
 public 編成型 = seq of 車両型;

 public 列車型 ::
        列車番号 : 列車番号型
        発駅 : 駅型
        着駅 : 駅型
        編成 : 編成型;

end 列車仕様
```

図 2-26 「列車仕様」クラス

列車そのものに関する定義を「列車仕様」という class にまとめています。class はオブジェクト指向言語のクラスに相当するものですが、さまざまな定義をまとめるための場所を提供します。実際ここでは、クラスの中に types というセクションを用意して「車両型」「編成型」「列車型」の三種類の型を定義しています。他にも「座席型」、「列車番号型」、「駅型」などがあるように見えますが、「class 列車仕様」中には定義されていません。

それらは「列車仕様」のスーパークラスである「class 路線仕様」の中に定義されています。路線仕様の説明は先で行いますが、ここでは共通の型などが定義された場所と考えていてください。ここで、Reservation.vdmpp の中に定義されたクラス構造をみてみましょう。以下の図は VDMTools というツール（第５章を参照）を用いて

Reservation.vdmpp から生成した XMI （UML の共通データ交換形式）を、Astah* という UML ツールで読み込んでクラス図として描いたものです。

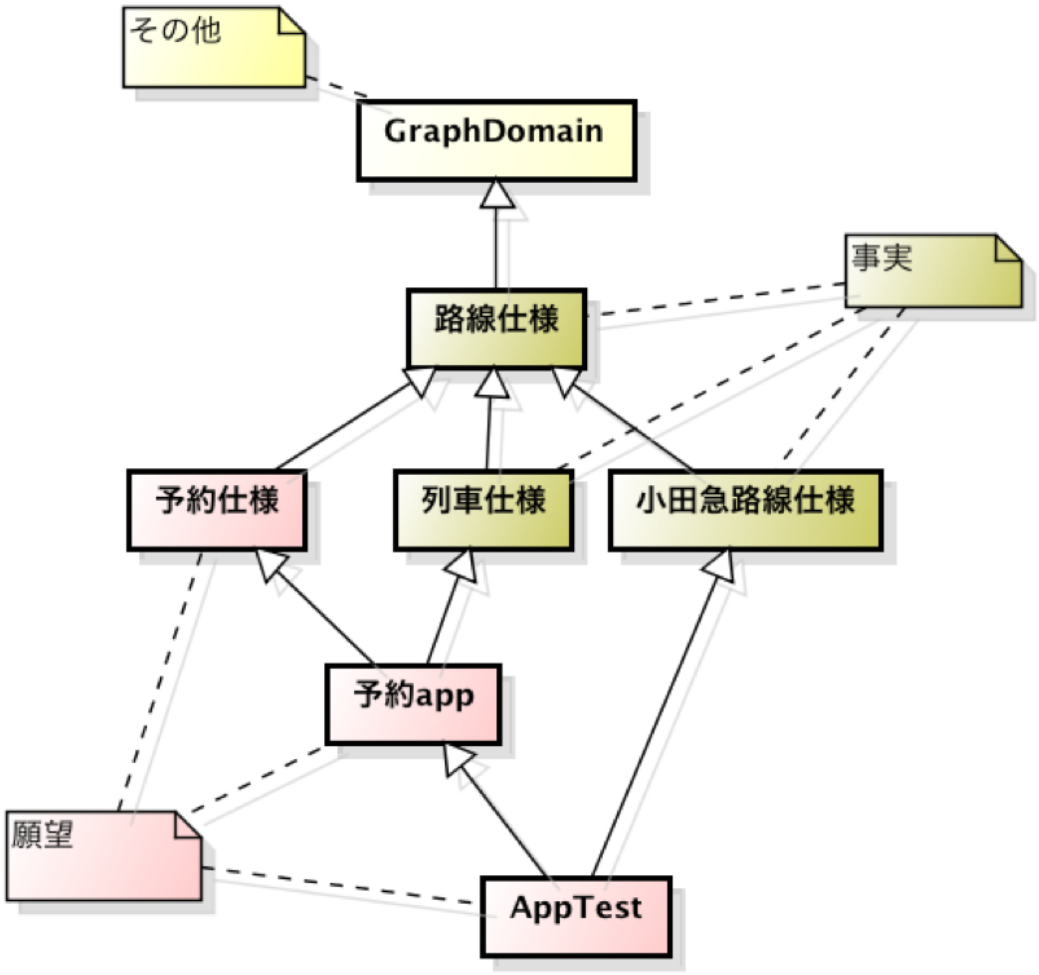


図 2-27 ツールから生成した仕様を構成するクラス図

先に示した「列車仕様」というクラスが、ほぼ真ん中に置かれています。コメントにも書かれていますが、「路線仕様」「列車仕様」「小田急路線仕様」が事実に対応するもの、「予約仕様」「予約 app」「AppTest」が願望に対応するものです。 AppTest は本格的なテストではありませんが、小田急線の路線仕様の下で機能を試すためのいくつかのメソッドが定義されています。またその他の「GraphDomain 」は路線を内部でグラフとして表現するための仕掛けが定義されています。

「予約仕様」という名前は重要そうですが、例題では簡単な型が定義されているだけです。今回はむしろ「予約 app」の方に大事な定義は詰め込まれています。とはいえリファクタリングを繰り返すうちに「予約仕様」の中に定義されるべきものが少しずつ増えていく可能性はあります。
以下に例題の「予約仕様」の中身を示しておきましょう。

```
class 予約仕様 is subclass of 路線仕様
types
 public 予約型 ::
        列車番号 : 列車番号型
        予約発駅 : 駅型
        予約着駅 : 駅型
        車両番号 : 車両番号型
        座席番号 : set of 座席型;

end 予約仕様
```

図 2-28 「予約仕様」クラス

この中では予約が、列車番号、予約発駅、予約着駅、車両番号、座席番号で構成されることを定義しています。

- 各座席は車両番号と座席番号で一意に指定される

という事実に対応して車両番号と座席番号（の集合）が予約型にはもたされています。

予約 app は予約仕様と列車仕様から型を取り込み、さらに列車予約に関連したさまざまな「用語」（関数）を定義しています。たとえば以下のような「用語」があります。

```
-- ある列車に対する現在の予約群が、指定された発駅着駅の区間で確保している
席を答える
  public 予約席集合 : 路線型 * set of 予約型 * 列車型 * 駅型 * 駅型
                                  -> map 車両番号型 to set of 座席型
  予約席集合(gr, rsv, tr, fs, ts) ==
        .... 以下略 ....
```

図 2-29 「予約席集合」関数

これは「予約席集合」という関数です。引数と戻り値の型をみると

路線型 * set of 予約型 * 列車型 * 駅型 * 駅型 -> map 車両番号型 to set of 座席型

となっています。路線情報、予約の集合、列車情報、発駅、着駅の情報を受け取って、指定された発駅と着駅間でその列車が予約されている席の情報を返します。ここでも車両番号型から座席型の集合への対応 (map) という形で座席の情報が表現されています。

不変条件の記述

構造を記述した際に大切なことが「不変条件」の記述です。不変条件は定義しつつあるデータの上に成立するさまざまな制約条件を定義します。「型」を定義しただけでは不足するような制約条件を定義し、仕様を厳密化するのが目的です。

たとえば、例題の「予約型」という型を考えます（再掲）

```
public 予約型 ::
        列車番号 : 列車番号型
        予約発駅 : 駅型
        予約着駅 : 駅型
        車両番号 : 車両番号型
        座席番号 : set of 座席型;
```

図 2-30 「予約型」

たとえば「座席番号」というのは「座席型」の集合であるということは分かりますが、その数に関する制約は特に書かれていません。もし考えている「予約」の仕様ではなんらかの制約が必要となった場合には、これに対して不変条件を指定することができます。

たとえば一つの予約に許される席数は 1 から 4 だとすると、この部分の定義は以下のようになります。

```
public 予約型 ::
      列車番号 : 列車番号型
      予約発駅 : 駅型
      予約着駅 : 駅型
      車両番号 : 車両番号型
      座席番号 : set of 座席型
 inv r == card r.座席番号 >= 1
                  and card r.座席番号 <= 4;
```

図 2-31 不変条件を加えた「予約型」

新しく加えられたのは

```
inv r == card r.座席番号 >= 1
                  and card r.座席番号 <= 4;
```

図 2-32 不変条件の例

の部分ですが、これは「座席番号集合の大きさは 1 以上 かつ 4 以下である」という不変条件を表したものです。

不変条件は「機能と無関係に」定義されます。このため新しい機能仕様を付け加えた際に、それが不変条件を満たしている機能仕様なのか否かを、仕様を実行評価する（仕様アニメーションを行う）ことによってテストすることが可能です。

5. 機能を記述する

契約書としての機能仕様

先に説明したように、機能仕様を定義する方法の一つに事前条件、事後条件を定義するやり方があります。事前条件、事後条件は「何を満たせばよいか」を示しているだけで、「どのようにその条件を満たすべきか」を指定してはいません。このことは、本書の中でも特に大事なことですので強調しておきましょう。

「事前条件」と「事後条件」を指定して、機能仕様を定義することができます

条件ですので、それをどのように満たすべきかは定義されていません。

どのようにその条件を満足すべきかを考えるのは「設計」の仕事です

条件式の中で使われるさまざまなデータ項目（とその型）は前節で説明したような、構造を定義する手段を用いて記述を行います。本書では VDM を利用していますが、開発者以外への説明の手段として、UML の図などを生成して利用するのは便利です。

事前条件、事後条件を用いた機能仕様書は、一種の「契約書」のようなものとして考えることができます。誰かがある機能を満たして欲しい場合に、事前に何を用意すれば結果として何を得られるのかが明確になるからです。

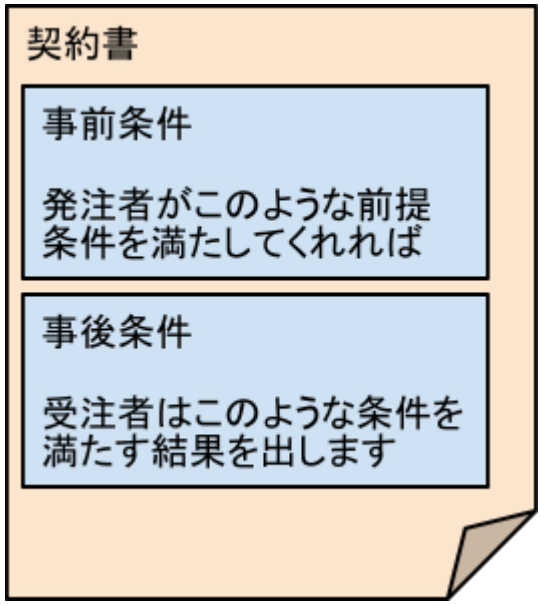


図 2-33 契約書記載項目としての事前条件と事後条件

たとえば、「銀行口座」というデータ型が定義されているとします。さらに「銀行口座」の属性の一つとして「残高」があるものとします。

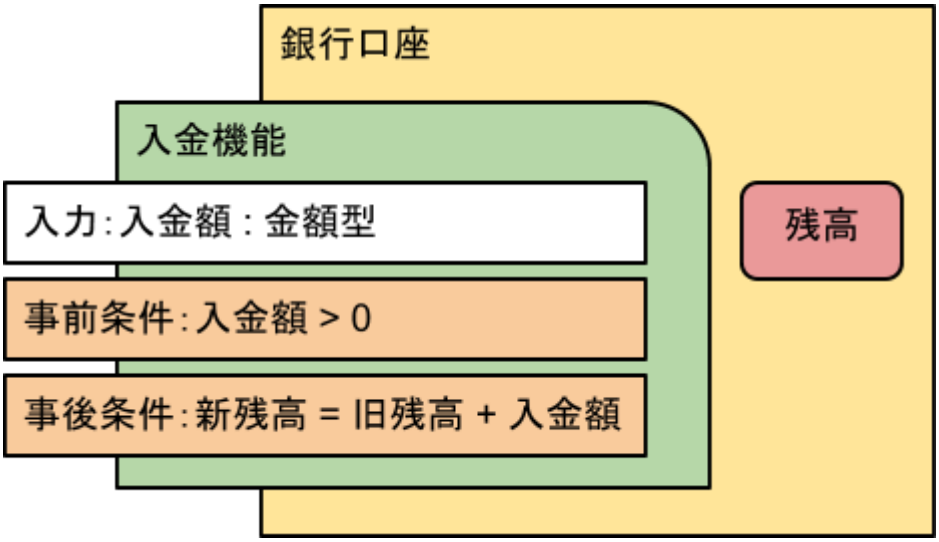


図 2-34 銀行口座の入金機能に付随した事前事後条件

もし日本語で事前事後条件も使わずにこの機能仕様を説明したとしたらたとえば次のようなものになるでしょう。

機能名	入金機能
入力引数	入金額（金額型）注：正の値であること
適用対象	銀行口座
説明	操作対象の銀行口座の残高に入金額を加える

図 2-35 表の形で記述した入金機能の仕様例

この程度の簡単な仕様でも、場合によっては疑問が起こります（たとえば上の記述内の「銀行口座」は正確に言えば「銀行口座」のインスタンス」です。またここでは赤字で

「注：正の値であること」と注記を書いてありますが、こうした注記は言葉だけを用いた文書の中では見失われやすいものです)。

このとき VDM を用いた「入金機能」の「仕様」はたとえば以下のもののようになります。

```
class 銀行口座

types
  public 金額型 = int;

instance variables
  残高 : 金額型 := 0;

operations

  public 入金機能 : 金額型 ==> ()
  入金機能(入金額) == is not yet specified
  pre 入金額 > 0
  post 残高 = 残高~ + 入金額;

end 銀行口座
```

図 2-36 VDM++による「銀行口座」クラスと入金機能の仕様例

銀行口座の属性である「残高」の特性（型や不変条件)、入金機能の事前事後条件がはっきりと定義されています。なお「残高~」という表記は、この機能が適用される「前」の残高の値を表します。すなわち post に書いてある内容は「残高の旧値に入金額を加えたものが、適用後の残高になる」というものです。

この定義では「入金機能」の実現方法そのものは指定されていないことに注意してください。ここでは “is not yet specified”（まだ定義されていない）と書かれています。VDM ではこの表記があるものは、とりあえず構文チェックと型チェックを通過しますが、仕様をアニメーション実行する際に「評価できない」というエラーが発生します。

pre と post すなわち事前条件と事後条件だけが定義された仕様は

陰仕様 （Implicit specification）

と呼ばれます。これは仕様を条件だけで表したもので評価実行はできません。
これに対して “is not yet specified” の部分に、評価用のコードを書き込んだものは

陽仕様（Explicit specification）

と呼ばれます。ここを定義しておくと、仕様を評価実行すること、すなわち仕様アニメーションが可能になります。しばしばこの部分を実際のプログラムのレベルまで詳細に書き込んでしまう事例が見受けられますが、それでは普通のプログラムをしているのと変わりません（コード生成を行うなら別ですが）。
陽仕様を記述する際には、細かい実装上の都合は省略して（たとえば RDBMS の特性とか、ネットワークのビットオーダーといったものは、上位の仕様を考えている際には無視できる可能性が高くなります）、**あくまでも機能そのものの核心をきちんと定義することに注意を払う必要があります**。そのためには仕様記述から利用できるライブラリを充実させておく必要もあるでしょう。
上記で示した例題で “is not yet specified” の部分を埋めたものの例は以下のようになります。

```
class 銀行口座

types
  public 金額型 = int;

instance variables
  残高 : 金額型 := 0;

operations

  public 入金機能 : 金額型 ==> ()
  入金機能(入金額) ==
    残高 := 残高 + 入金額
  pre 入金額 > 0
  post 残高 = 残高~ + 入金額;

end 銀行口座
```

図 2-37 VDM++による「銀行口座」クラスと入金機能の仕様例（陽仕様追加）

まあ最も単純な加算を用いてアルゴリズムを示してあります。このように簡単な例題では陽仕様を与えることも簡単ですし、これを見た設計者が仕様に従った設計を行うことも簡単ですが、複雑な仕様の場合には設計も一直線には行えません。その場合は設計者は事前条件、事後条件を満たしているかに注意を払いながら設計を行うことになります。なおここで定義されている「入金機能」は、VDM の世界では「関数」ではなく「操作」と呼ばれているものです（細かい話になりますが、関数にはインスタンス変数へのアクセス権限がなく、したがって副作用がありませんが、操作はインスタンス変数にアクセスすることが可能で、値を変更するという副作用を起こすこともできます。この例では「残高」というインスタンス変数を操作しています)。

特急券予約例題の機能の定義

では前に説明したような構造が特急券予約問題の中で定義されているときに、その上の機能はどのように定義されるのでしょうか。

ここでは新しい予約を行う「新規予約操作」を考えることにします。

特急の新規予約を行うためには、「要求文書」の中に含まれる「願望」の記述

- 予約に際しては列車を指定し、発駅、着駅、人数を与える

から、列車、発駅、着駅、人数を指定して、新しい予約を得ればよさそうです。

以下に VDM で定義した「新規予約操作」の定義を示します。

```
public 新規予約操作 : 列車型 * 駅型 * 駅型 * 人数型 ==> [予約型]
 新規予約操作(a列車, a発駅, a着駅, a人数) ==
  let mk_(nr, nrsv) = 新規予約(全路線, 全予約, a列車, a発駅, a着駅, a人数) in
     (全予約 := nrsv;
      return nr)
 pre
  let 駅集合 = 停車駅集合(全路線, a列車) in
   a発駅 in set 駅集合 and
   a着駅 in set 駅集合 and
   a人数 > 0
 post
  let 新予約 = RESULT in
   if 新予約 <> nil then
    新予約 not in set 全予約~ and
    全予約 = {新予約} union 全予約~ and
    新予約.列車番号 = a列車.列車番号 and
    新予約.予約発駅 = a発駅 and 新予約.予約着駅 = a着駅 and
    card 新予約.座席番号 = a人数
   else
    全予約 = 全予約~;
```

図 2-38 「新規予約操作」の仕様例

では少しずつ内容を確認していきましょう。

（1）ヘッダ部

```
public 新規予約操作 : 列車型 * 駅型 * 駅型 * 人数型 ==> [予約型]
 新規予約操作(a列車, a発駅, a着駅, a人数)
```

図 2-39 「新規予約操作」のヘッダ部

この操作の名前と、引数の型と戻り値の型の定義です。名前は「新規予約操作」で目的はその名のとおり新しい予約を行うことです。

引数に予約対象の列車、発駅、着駅、そして予約人数を指定して呼び出すと、予約型の情報が返されてきます。[予約型] のように角カッコで囲まれているのは予約型以外に nil も返される可能性があることを示しています。事後条件を見ると分かりますが nil が返されるのは席の予約ができなかった場合です。

（2）ボディ部

```
let mk_(nr, nrsv) = 新規予約(全路線, 全予約, a列車, a発駅, a着駅, a人数) in
        (全予約 := nrsv;
         return nr)
```

図 2-40 ボティ部

仕様アニメーションを行うためには、なんらかの形で仕様を評価する仕掛けが必要となります。ここでは下働きである「新規予約」という関数を呼び出しています。

a 列車、a 発駅、a 着駅、a 人数というのは、機能の外側から与えられるものですが、全路線、全予約とは何でしょう。実はこれらはインスタンス変数として、システムの中に保持されているものです。新規予約は「関数」なのでその内部ではインスタンス変数にアクセスすることはできませんし、したがってシステムの状態（たとえば予約の状態）を変えることもできません。新規予約「関数」のヘッダは以下のように定義されています

```
新規予約 : 路線型 * set of 予約型 * 列車型 * 駅型 * 駅型 * 人数型
          -> [予約型] * set of 予約型
```

図 2-41 「新規予約」関数のヘッダ部

戻り値は二つの型の組み合わせで、一つめが [予約型] 、二つめが set of 予約型 です。このボディの中の記述では、新規予約「関数」にさまざまな引数（インスタンス変数や外部からの引数）を引渡して結果を得ています。一つめの戻り値が nr, 二つめの戻り値が nrsv に束縛されますが、nr は新規予約操作の戻り値となり、nrsv は全予約インスタンス変数に新しい値として代入されています。

（３）事前条件

```
pre
   let 駅集合 = 停車駅集合(全路線, a列車) in
     a発駅 in set 駅集合 and
     a着駅 in set 駅集合 and
     a人数 > 0
```

図 2-42 事前条件

事前条件はこの関数を使う側の義務です。ここでは発駅、着駅は停車駅の集合に含まれることや、指定する人数は１人以上であることなどが指定されています。この条件を満たさない場合には設計時に例外を返すなどの決定を行うことになります。なお事前条件違反は呼び手側のバグとして看做すべきですが、開発者の手の届かない本当の外部から呼び出される場合には、間に事前条件を保証してシステムを防衛するための層を設けるなどの策を考える必要があります。

（４）事後条件

```
post
   let 新予約 = RESULT in
     if 新予約 <> nil then
       新予約 not in set 全予約~ and
       全予約 = {新予約} union 全予約~ and
       新予約.列車番号 = a列車.列車番号 and
       新予約.予約発駅 = a発駅 and 新予約.予約着駅 = a着駅 and
       card 新予約.座席番号 = a人数
     else
       全予約 = 全予約~;
```

図 2-43 事後条件

事後条件はこの関数の果たすべき義務です。よって事後条件が一番狭い意味での「仕様」に近いイメージのものとなります。さてここに書いてあることは何でしょう。

ここには**「新規予約操作」の結果として満たされるべきこと**が書かれています。

返り値は [予約型] ですから、nil の場合と予約型の場合があることが分かります。事後条件でも、まず戻り値を新予約という名前で参照できるようにしたあと、 if 新予約 <> nil then ~ else という形で、戻り値が nil であるか否かの分類をしています。列車番号や、発駅、着駅、人数などが入力として与えられたものと一致するという条件は分かりやすいと思います。少し説明が必要なのは以下の部分です。

```
.
    .
新予約 not in set 全予約~ and
全予約 = {新予約} union 全予約~ and
    .
    .

全予約 = 全予約~;
    .
    .
```

図 2-44 事後条件の一部を抜粋

「全予約~」といった具合に、名前の後ろに「~」がついているものは何でしょうか。前の節でも説明しましたが、実はこの「~」表記は、この機能が呼び出される前の値（旧値）を参照するという意味なのです。
よって 「新予約 not in set 全予約~」という式は「新予約はこの機能が適用される直前の全予約に含まれていてはいけない」という意味を表しています。他の２行も説明するならばそれぞれ以下のようになります。

全予約 = {新予約} union 全予約~ ：呼び出し前の全予約の集合に新しい新予約要素を union で加えると戻り値の全予約となる。

全予約 = 全予約~ ： もし戻り値が nil だったら（即ち空席が確保できなかったら）全予約は機能の呼び出し前後で変化しない。

以上、事前条件、事後条件を用いて仕様を定義した様子、さらに陽仕様部分を定義して仕様アニメーションのための準備を行う様子を説明しました。

データフローと関数、そしてシステム記述

さて機能仕様の記述の説明の際に「インスタンス変数」といったオブジェクト指向プログラミングの用語が出てきたことに違和感を覚えたひともいるかもしれません。この本で例題の記述に用いているのは、 VDM の中でも、特にその派生系である VDM++ というオブジェクト指向 VDM を用いています。「インスタンス変数」という言葉を聞くと細かいプログラミングの議論のような気もしますが、オブジェクト指向本来の概念に立ち返ってみれば、インスタンスがシステム全体を表現し、その中のインスタンス変数はいわばシステムが保持している状態であると、大きくとらえることもできます。

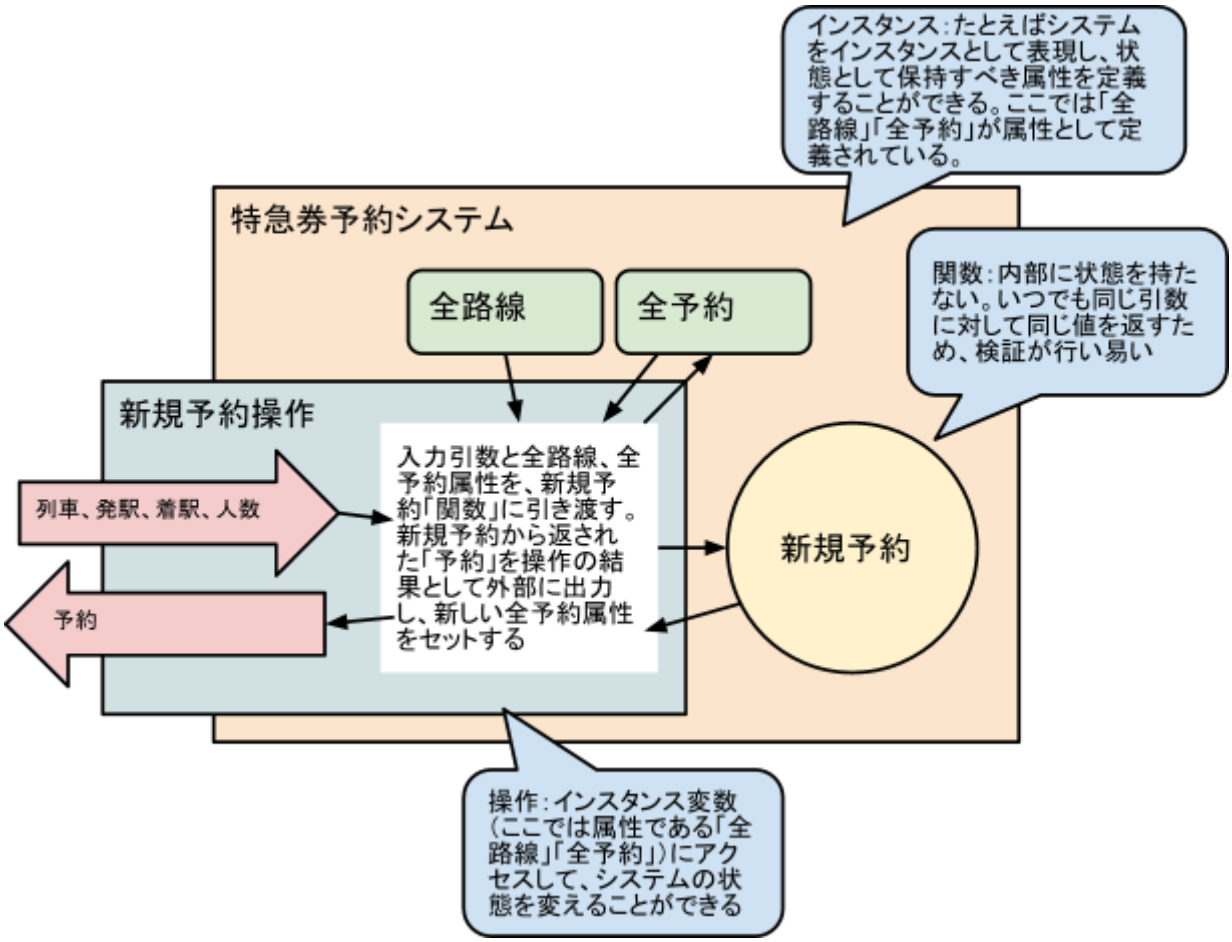


図 2-45 特急券予約システムのコンポーネント図風表現

上の図は、「特急券予約システムオブジェクト」に「新規予約操作」が定義されている様子を UML でいう所のコンポーネント図風に表現したものですが、オブジェクト指向ではなく、昔ながらの DFD （データフロー図）を用いてこの構造を説明することもできます。

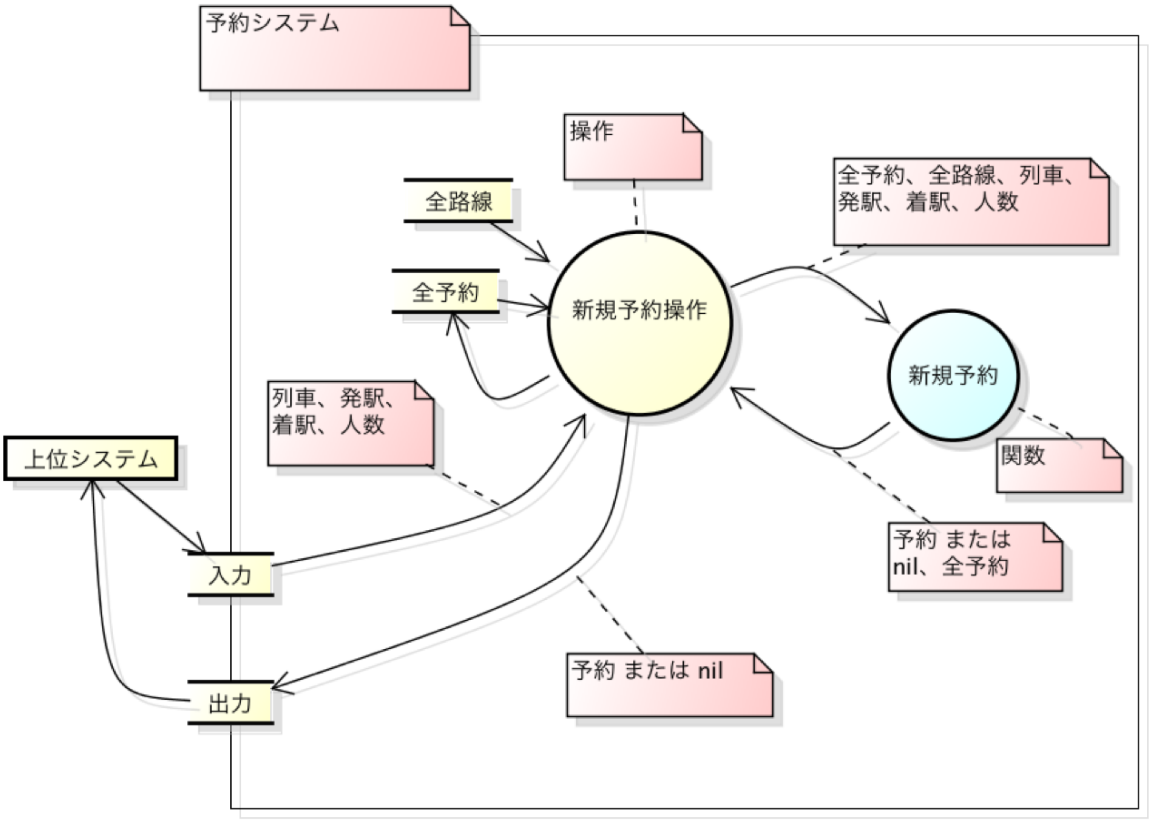


図 2-46 予約システムの DFD 表現

この図は DFD 作成ツールを用いて描いたものですが、全路線や全予約といったインスタンス変数が、ここではシステム内の DB であるように描かれています。予約システムが使う DB を他の操作で共有するような図を描くことも可能ですし、それは VDM で記述した仕様記述から必要に応じて描き出すことも可能です（現時点では VDM と UML のクラス図の相互変換は XMI を介してある程度自動化されていますが、VDM と DFD の相互変換は自動化されているわけではありません。ただ体系的に図として置き換えていくことはできます)。

システムの上位レベルの情報の流れを検討している際には　DFD を使った利用者への説明が有効になることも多く、仕様の Validation に大いに役立つ可能性があります。このように、VDM を用いた機能仕様は、プログラムに近いレベルの詳細なものから、業務フローレベルの粒度の大きなフローを表現する用途にも利用することが可能なのです。

6. 振る舞いを記述する

現実世界に対してなんらかの影響をおよぼすために、状態を管理しなければならないシステムが存在します。状態とその移り変わりを仕様として記述するためによく使われている記法の代表的なものに、状態機械 （State Machine）があります。これは UML の用語ですが、一般的には状態遷移モデルとも呼ばれています。

例題：ポットの温度管理

例題として考えるのは電気ポットです。このポットはハードウェアとしてヒーターと温度計を備えていて、ヒーターへ on/off を指示したり、温度計から現在温度を読み取ったりすることが可能です。こうしたハードウェアとのやり取りを行いながら制御を行う主役はソフトウェアです。ここでは温度管理を行うソフトウェアを「温度制御」と呼ぶことにします。

ハードウェアとソフトウェアの役割分担の概念図を下に示しました。

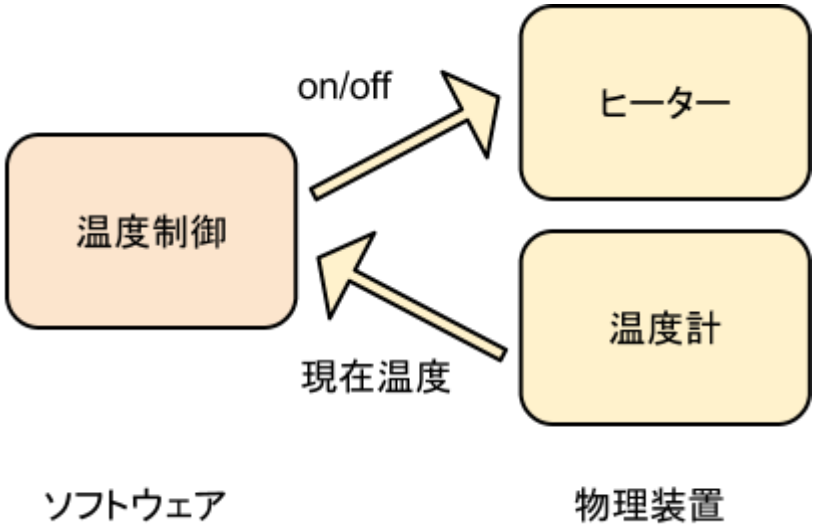


図 2-47 ハードウェアとソフトウェアの役割分担

こうした状況で最終的なシステムの動作テストをするためには、物理装置そのものか物理特性に沿って擬似的に動作するシミュレータが必要になります。とはいえ、温度制御の振る舞いの仕様は記述することが可能です。
たとえば「温度制御」の中を、以下の二つの状態機械で表現することにしてみましょう。

- 「保温モード」: 保温温度設定の切り替えを行う。最初に on になったときには高温モード（９８度）であり、「次モード」を受け取るたびに節約モード（９０度)、ミルクモード（６０度）と保温温度設定が切り替わる。
- 「保温装置」: 保温温度設定より温度が低い場合にはヒーターに対して on を、保温温度設定より温度が高い場合にはヒーターに対して off を行う。

それぞれの状態遷移機械はたとえば以下のようになります。

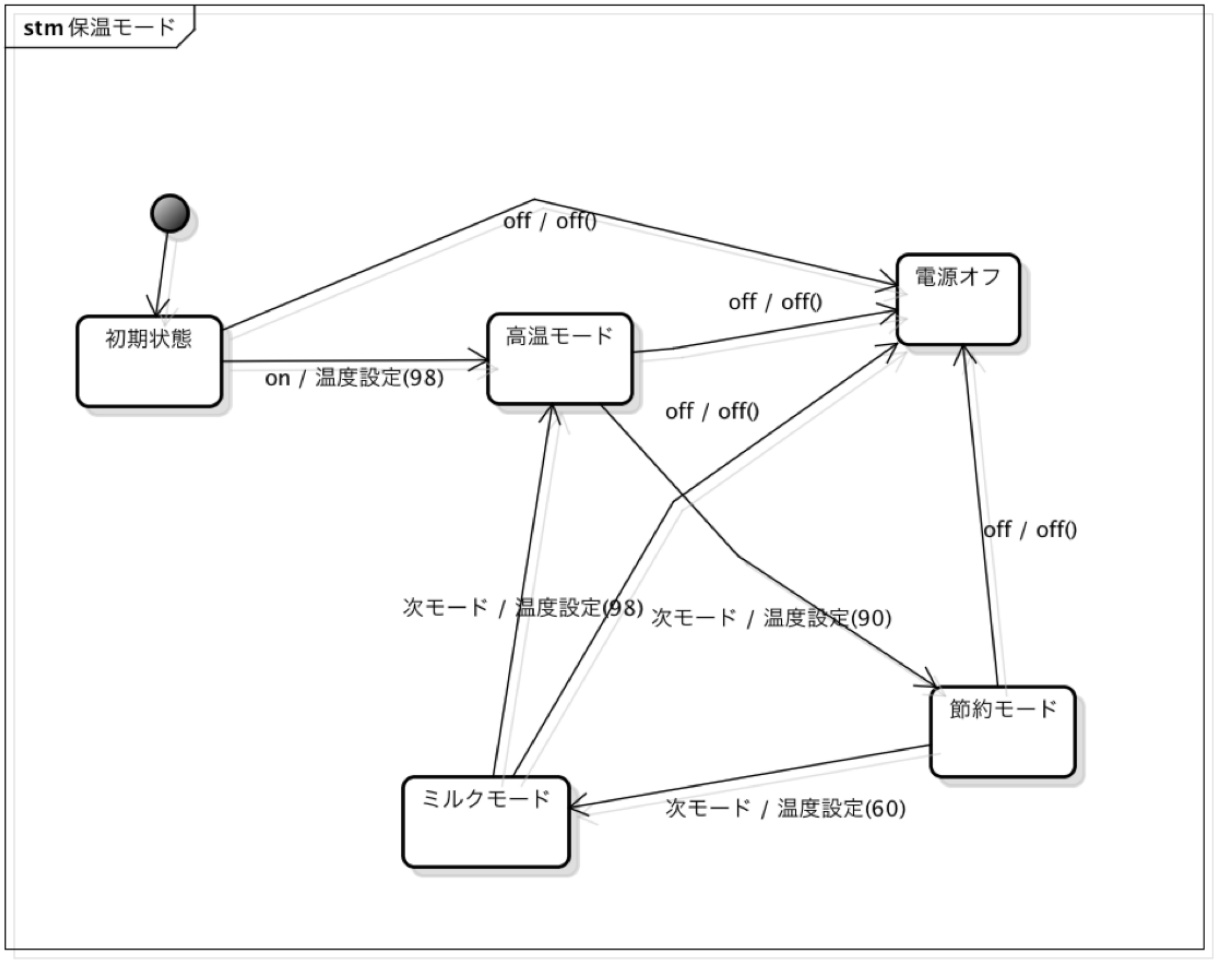


図 2-48 「保温モード」の状態遷移

「保温モード」は初期状態から on が押されると、温度設定（９８）を行いながら「高温モード」へ遷移します。以降「次モード」が発生するたびに新しい温度設定を行いながら「高温モード」→「節約モード」→「ミルクモード」へと次々に遷移します。off を受け取るとどの状態からでも「電源オフ」に遷移します（この状態遷移モデルでは再び on にする遷移は省略されています)。

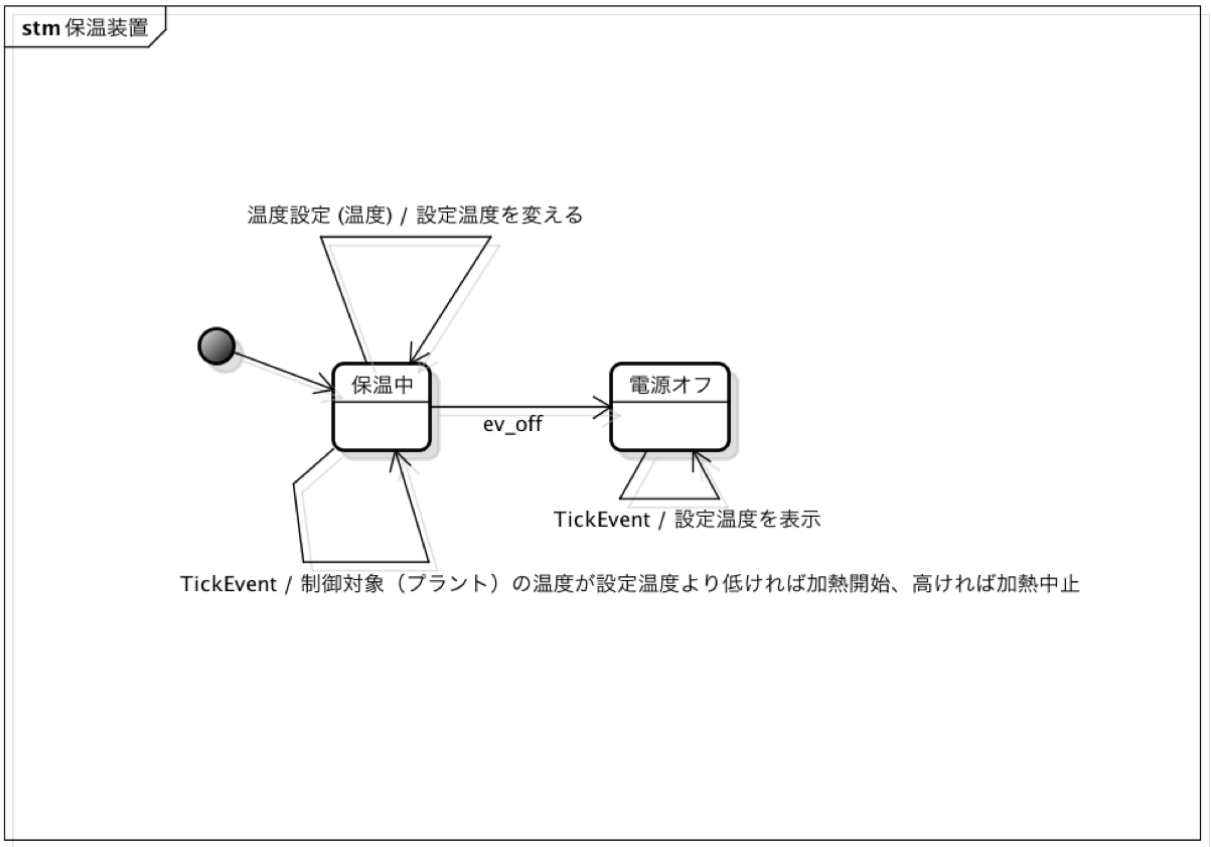


図 2-49 「保温装置」の状態遷移

「保温装置」状態機械は、「保温モード」状態機械に比べるとシンプルです。「保温状態」にあるときには温度設定が行われるか、一定時間ごとの割り込み（TickEvent）が発生しています。そして設定温度よりも現在温度が低ければ加熱し、高ければ加熱中止をするという単純な動作を繰り返しているだけです。

もちろん、この状態機械はまだまだ定義が不足しています。特に「安全性」の観点からの Validation にはまだ合格しないでしょう。

一方、状態機械に基づく設計の Verification は状態機械のモデル上に定義されたさまざまな条件を判定することによって行うことが可能です。また複数の状態機械が動作する際に望ましくない組み合わせの状況が起きないことなどをモデル検査の技法を使って調べることも可能です。

構造と機能を記述する際に用いたように VDM を用いて状態遷移系の仕様記述を行うことも可能です。なお現在の VDM は標準ではモデル検査機能とは組み合わされていませんので、そうした部分に関してはいくつかの工夫が必要になります。

以下に示したものは、VDM で状態機械を記述するために用意されたフレームワークを用いて作成された仕様記述の一部です。実際には VDM-RT (VDM Real Time) という名の、非同期通信と並列動作をサポートした VDM 処理系の上で動作します。

```
private next_state : Event * EventArg * int ==> State
 next_state(aEvent, aEventArg, -) == (
  cases cstate :
    (st_初期状態) -> cases aEvent :
               (ev_on) -> return st_高温モード,
               (ev_off) -> return st_電源オフ
             end,
    (st_高温モード) -> cases aEvent :
               (ev_次モード) -> return st_節約モード,
               (ev_off) -> return st_電源オフ
             end,
    (st_節約モード) -> cases aEvent :
               (ev_次モード) -> return st_ミルクモード,
               (ev_off) -> return st_電源オフ
             end,
    (st_ミルクモード) -> cases aEvent :
               (ev_次モード) -> return st_高温モード,
               (ev_off) -> return st_電源オフ
             end,
    (st_電源オフ) -> return cstate,
    others -> return cstate
  end
 );

 private trigger_event_handler : Event *  EventArg * State * int ==> ()
 trigger_event_handler(aEvent, aEventArg, aNextState, -) == (
  cases mk_(cstate, aEvent, aNextState) :
   mk_((st_初期状態), (ev_on), (st_高温モード)) -> 温度設定(98),
   mk_((st_初期状態), (ev_off), (st_電源オフ)) -> 温度設定(0),
   mk_((st_高温モード), (ev_次モード), (st_節約モード)) -> 温度設定(80),
   mk_((st_高温モード), (ev_off), (st_電源オフ)) -> 温度設定(0),
   mk_((st_節約モード), (ev_次モード), (st_ミルクモード)) -> 温度設定(60),
   mk_((st_節約モード), (ev_off), (st_電源オフ)) -> 温度設定(0),
   mk_((st_ミルクモード), (ev_次モード), (st_高温モード)) -> 温度設定(98),
   mk_((st_ミルクモード), (ev_off), (st_電源オフ)) -> 温度設定(0),
   others -> DebugPrint("**不適切な温度制御遷移です**")
  end
 );
```

図 2-50 状態遷移の VDM による記述例

この定義は「保温モード」状態機械に対応するものです。詳細は説明しませんが、next_state は発生した事象に対して次の状態遷移を決定する部分で、trigger_event_handler では実際の状態遷移が起きる際に行うべき動作（アクション）を定義しています。

この定義を用いて仕様アニメーションを行い仕様レベルでの振る舞いに対する Verification を行うことができます。なお VDM-RT は並列動作時の排他制御の仕様などを定義する言語要素を併せ持っています。

フレームワークを前提に Astah* などの UML モデリングツール状態機械の定義から上記のような定義を生成するプログラムは比較的簡単に作成することができますので、状態機械を示しながら発注者と開発者が打ち合わせを行い、そこから生成されたモデルで仕様アニメーションを行うという手順で開発をすすめることも可能になります。

モデル検査

モデル検査は形式手法の一つです。ある程度複雑な状態機械を動作させた場合に、問題となる組み合わせが発生するか否かを、基本的に指定した範囲で全部検査します。このため通常の人間がテストケースを考えるテストとは異なり、特定の範囲では漏れのない試験が行われたことになって、システムの Verification が行いやすくなります。

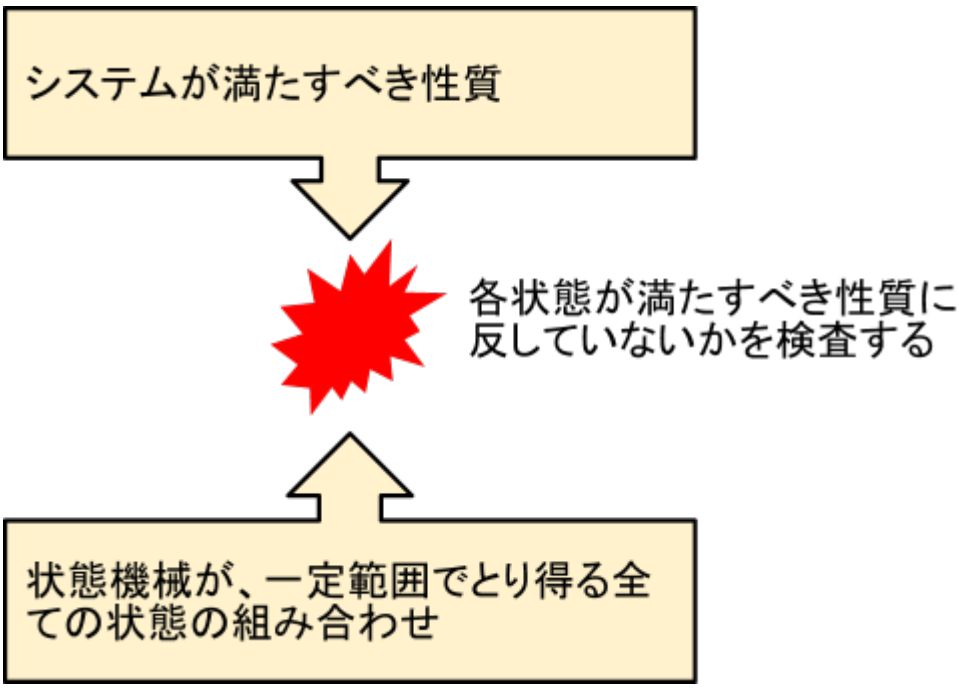


図 2-51 モデル検査の概念

モデル検査は強力ですが、正しいモデルを構成できるかどうかが一つの課題になります。振る舞いに対して Validation が行いにくいような複雑なモデルの場合、そのモデルが

「正しい、正しくない」という議論をしようとしても、モデル自身が正しく書かれているのか、モデルが定義しようとしている内容が正しいのかが混乱しやすく、何を議論しているのか分からなくなってしまいがちだからです。
とはいえ、もし状態機械のフレームワークをうまく用意してやれば、そのフレームワーク上で構築された状態機械からモデル検査用のコードを生成することが可能になりますので、モデルの Verification の部分は行いやすくなります。

7. 非機能要求を記述する

ここまで説明をおこなってきた仕様記述はいずれも「機能仕様」に関する記述でした。ところで仕様記述にはもう一つ、厄介な記述対象があります。非機能要求と呼ばれる一連の要求に対する「仕様」です。
非機能要求は主に使い勝手や保守性、合目的性など数値化しにくい特性に対しての要求を書き並べたもので、機能仕様に直接現れるというよりも、最終的な Validation の対象にしかならないものが多いように見受けられます。

エビデンスベースの非機能要求に対する仕様記述

人間の感性は論理的に表現できないという通説は強いのですが、それでも心理学的な実験結果などからストレスを感じさせない操作時間や色彩などの得ることは可能です。話はソフトウェアから離れますが、米国で政府によって行われている教育改善プログラム（WWC * What Works Clearinghouse）などは、数値化しにくい教育効果に統計学的手法を持ち込んで、効果の高い教授法を科学的に研究しています。[6]
こうした場合、システムの振る舞いや**インターフェース**に関する仕様の一部に、実験結果などから得られた結果（タイミング、色彩、文言、レイアウトなどなど）を不変条件等の形で忍び込ませることも可能でしょう。たとえば（仮の話ですが）画面に一度に２０個以上のチェックボックスを出さないとか、何かのエントリ画面を５ステップ以上の深さにはしないといった類の条件は、構造、機能、状態の機能仕様の中にさまざまな条件として埋め込むことも可能です。
とはいえこうした実際の非機能要求の中身はこれからの研究に期待するしかありません。

リマインダとしての非機能要求に対する仕様記述

エビデンスがきちんと揃わないような非機能要求に対しても、アイデアレベルで仕様記述を行なっておきたい場合があります。たとえばあるシステムが、利用者に対してシステムの診断メッセージを表示しようとする際に、利用者の資格（知識レベル）などに対応して、最初に表示されるメッセージの詳細さを変化させたいという要求があったとします。

「利用者の知識レベルに応じた詳細さ」という定義は曖昧ですので、通常はどこかの備考欄に、「ここに表示するメッセージは「利用者の知識レベルに応じた詳細さ」を考慮すること」といった書かれ方になる可能性があります。

しかしこの部分も厳密な記述を目的に分析することによって、対象メッセージの集合、利用者の分類、詳細さの分類などをとりあえず「指示」として定義しておき、それらの組み合わせで判定する部分を「定義」としておけば、「指示」の一覧として検討事項をリストアップしておくことができます。

リストアップされていれば、スケジュールや優先度に依存して検討をすることができますので、最終的な製品の品質も管理しやすくなります。

8. 形式仕様記述と適用のレベル

形式手法の第一人者の一人である Peter Gorm Larsen 教授によれば、形式手法適用のレベルには以下の四つがあります [7]。

- レベル１：離散数学の概念や記法を用いる
- レベル２：適切なツールの支援と共に形式仕様記述言語を利用する
- レベル３：厳密な仕様を検証する
- レベル４：完全に形式的に開発する（抽象的な仕様を詳細化していく）

それぞれについて説明を次に行いましょう。

レベル１：離散数学の概念や記法を用いる

繰り返しますが、形式手法は数学そのものを扱うわけではありません。あくまでも数学的な記法を用いてソフトウェアの仕様などを扱っているに過ぎません。これに対して数学は数学的な記法を用いて数学そのものを扱っているのです。

すなわち、形式手法は数学からその記法と使用方法を借用して厳密な表現を実現しようとしているだけなのです。

ここで使われている離散数学の概念や記法というものは、先に紹介した「論理結合子と自然言語と形式仕様記述言語」の節でご紹介したものと同じものです。こうした記法を用いて自然言語では表現しにくい厳密な記述を実現するのです。

レベル２：適切なツールの支援と共に形式仕様記述言語を利用する

レベル１の段階では、個々の文章の一部を数学的な記法を用いて厳密化しただけですが、これだけでは仕様をひとかたまりのものとして扱うことは困難です。個別の文章が単独に厳密化していたとしても、集めてみるとさまざまな相互矛盾や、粒度の違いなどが発見されるかもしれません。

形式仕様記述言語を用いて表現した仕様はそれ自身「構文検査」「型検査」などを通して基本的な記述の一貫性を保証することがまず可能です。その上で多くの形式仕様記述言語は仕様をまとめ構造化するためのさまざまな仕掛けを提供しています。

たとえば機能仕様を表現する基本手段として、事前事後条件と不変条件を用いた表明を用いる VDM は、大規模な仕様を記述するために、モジュール (VDM-SL) やクラス (VDM++) といった仕掛けを導入しています。また条件式や関数のシグネチャで用いられる型も大きな仕様記述の一貫性維持を助ける役割を果たします。もちろんこうした記述の一貫性などは機械支援があるからこそ現場における現実的な選択肢の一つとなっているのです。

レベル３：厳密な仕様を検証する

レベル２の段階では、まだ形式的に記述されたとはいえ、検証については構文レベルや型レベルの整合性を静的に検査するものに留まっています。これは人間の目視による検証よりは間違いの少ないものになりますが、定義した仕様の実際の振る舞いの検証についてはそれだけでは不足しています。

後ほど説明しますが、仕様が正しい振る舞いを定義しているか否かを検証するには大きく分けて二通りの方法があります。（１）仕様に対してさまざまなテストケースを適用しその振る舞いを計算して期待値と比べる方法（いわゆるテスト）と、（２）仕様の定義そのものから想定されている性質がいつでも成り立つことを論理的に導き出す証明の採用です。
こうした意味でレベル３は

- レベル３（１）：テストによる検証
- レベル３（２）：証明による検証

の二つのレベルにさらに分解されます。
（２）の証明に関しては現状の支援ツール環境では現場での学習ならびに適用コストがやや高くなってしまうため（数学の証明スキルに近いものが必要とされる）次のレベル４での適用で本格的に行われることが多い技法です。
これに対して（１）はいわゆるプログラムなどに対して行う「テスト」の技法が流用できるため、現場でも導入しやすいというメリットがあります。実際に仕様をプログラムのように扱いテストケースを用いてテストを行う手法は多くの現場で絶大な効果を発揮しています[2]。
本書では全体を通してこのレベル３（１）での現場適用を想定して説明を行なっています。なお仕様を「実行」すると表現する代わりに「仕様アニメーション」という言葉もよく使われます。

レベル４：完全に形式的に開発する（抽象的な仕様を詳細化していく）

レベル３までの適用ではも形式仕様記述言語を用いた厳密な記述を行い、従来のテスト技法を援用した検証を行なっているので、詳細な設計段階に入る以前に仕様そのものを検証し問題点を抽出することが可能になります。一般的には上流で発見された問題ほど低コストで修正が可能なので、仕様段階での検証は大きな意味を持ちます。
一方、テスト技法を用いた検証だけでは心許ないという考え方の場合には、定義された仕様の振る舞いが、その要請に適合しているのか、振る舞いを詳細化して設計へと書き

換えていく個々の段階が正当に行われているかをより厳密に検証できる手段を採用することになります。これは仕様記述を使った「証明」を行うことによって実現されます。証明を行なって正しさを検証した場合、機能に関する「テスト」を行う必要はありません。それは既に証明済だからです。とはいえたとえば実際のプロダクトに組み込まれたソフトウェアは物理的存在になりますから、そうした観点での物理的擾乱を加味したテストを行う必要はあります。ありがちな話としてランタイムライブラリ（OS を含む）やハードウェアにバグが潜んでいる可能性もありますので、結局プロジェクトとして見れば負荷テストや自動生成されたランダムなテストケースを繰り返し適用していく必要性などはなくなりません。
また「証明」を用いた仕様の詳細化にはそれなりの数学的スキルが必要とされるため、現在のツール環境では現場導入はやりにくいというのが実際です。

上にも述べましたが、本書では教育しやすく既存の技法も使えることから原則形式手法レベル３（１）の現場適用を想定してさまざまな解説を行なっています。IPA/SEC の「厳密な仕様記述における形式手法成功事例調査報告書」[8] にはこうしたプロジェクトの報告が掲載されています。

この章のまとめ

以下に本章で説明した重要な概念をまとめておきます。

仕様記述の要素

仕様記述を構成する主要な観点 - 構造、機能、状態を紹介しました。これにさらに時間の特性が添えられて、機能仕様を表現するモデルが構成されます。
それぞれは　UML などで図式表現を与えられていますが、図式表現は解析性や再利用性に劣ります。本書では構造、機能、状態に対する厳密な仕様記述を実現する手段として、形式仕様記述言語、とくに VDM を用いた方法を簡単に解説しました。

日本語と論理式と形式仕様言語

自然言語による記述も、事前に定義された用語をさまざまな論理演算に相当するする言葉（かつ、または、ならば、である、すべての。。。etc）を使って行われています。

本章では、日本語の記述と論理式の関係を示し、相互変換の例に踏み込んで説明を行いました。そのあと数学で使われる論理式を素直に表現して、さまざまな機械的チェックを可能にする形式仕様記述言語について説明し、ここでも簡単な例題を示して解説を行いました。

形式仕様記述言語は確かに外国語のようなものですが、限られた語彙と規則正しい文法を用いた人工言語に過ぎません。C++ や Java に比べても難しいとは言えない言語なのです。

事前条件・事後条件・不変条件

機能仕様を定義する際に、事前条件と事後条件で仕様を表現する方法がよく利用されています。事前条件は機能仕様を使う側の義務、事後条件は機能仕様を提供する側の義務としてとらえることができて、合わせると使い手と提供側の契約関係を表現したものととらえることが可能です。

不変条件は機能そのものではなく、構造上に定義される制約関係を表現するために用いることができます。ある値の範囲や、数値の依存関係、その他さまざまな制約関係を表現するもので、個々の機能とは独立して定義することが可能です。

このため新たな機能を定義した際の二重チェック機構としても用いることが可能になり、仕様に対する検証　Verification の助けを提供してくれます。

形式仕様適用のレベル

形式仕様を適用するいくつかの段階に関して説明を行いました。いわゆる証明を行わずとも、普通の現場の技術者が身に付けているプログラミングとテストの技法でも対処できる、レベル３（１）の段階までなら、導入にあまり大きな困難はなかった事例が多いので、参考になります。

参考文献

[1] トム・デマルコ “Structured Analysis and System Specification” 1979

[2] ジェームス・ランボー他 「オブジェクト指向方法論 OMT—モデル化と設計」1992

[3] マーチン・ファウラー「UML モデリングのエッセンス 第 3 版」2005

[4] 新井紀子「数学は言葉」2013

[5] マイケル・ジャクソン「ソフトウェア要求と仕様」2004

[6] “What Works Clearinghouse”　http://ies.ed.gov/ncee/wwc/

[7] ピーター・ラーセン「形式手法概要」

http://sec.ipa.go.jp/users/seminar/2012/seminar_tokyo_20121023_01.pdf」2012

[8] IPA/SEC「厳密な仕様記述における形式手法成功事例調査報告書」2013

章末コラム [形式仕様言語にとって、数学という言葉は必要か]

形式仕様言語を用いた仕様記述の紹介をする際には、多くの場合「数学」「数理論理学」という言葉が出てきます。これはなぜなのだろう、と私は第 2 章を読んで改めて疑問に思いました。プログラミング言語を学ぶ際に、私の知っている限りでは、「数学の理論に基づいた」などという枕詞はあまり見たことがありません。形式仕様言語と同様に、プログラミング言語も数学理論を基にしているにも関わらず、なぜこのような差が生じるのでしょうか。

私は数学が好きなので、「数学」という言葉に反応し、形式仕様言語に興味を持ちました。ところが、実際には私と逆の反応を示す人たちも多くいると思います。形式手法を推奨している方々は、この事実にも当然気付いているはずです。気付いている方が多いので、難しい数学ではない、などの表現が出てくるのです。もちろん、形式手法の研究（新しい形式仕様言語を開発するなど）を実施する人は、数学、特に、数理論理学を学んでおく必要があると思います。しかし、形式仕様言語を利用したい人々に対しては、プログラミング言語と同様に、形式仕様言語では、どう定義されているのかを説明すれば事足りるのではないか、と思います。つまり、ソフトウェア開発に携わる方々にとって、「数学」という言葉が形式仕様言語に興味を持つことに対して逆効果であるのであれば、予め必要な部分のみを提示すればよいと思うのです。

数学という言葉に戸惑うことなく、形式仕様言語を用いることで、抽象的・論理的に対象をとらえることができるようになるということが、より多くの人の目に留まるようになることを願っています。

[第２章執筆者のコメント]

プログラミングができるということは、技術者自身は自覚していないかもしれませんが、既に「論理的な言葉」を使いこなせているということです。形式仕様記述言語も同様に「論理的な言葉」であるに過ぎません。ものの集合や依存関係を効率的に表現できるよ

うな、集合や写像の演算子や関数型言語の特徴などに少々抵抗があるのかもしれませんが、オブジェクト指向言語（たとえば　C++ や Java など）の複雑さに比べれば特に分かり難いものではないと思います。ということで、おそらく最初のハードルは言語そのものではなく、形式仕様記述言語を使ってモデリング（抽象化）を行う部分にあると思われます。

なぜ抽象化が大切なのか、抽象化することによって仕様記述がどれほど実り多いものになるのか、を理解してもらうことがきっと大切なのでしょう。しかし考えてみれば、プログラミングのレベルでも汎用的な（すなわち抽象化された）ライブラリやフレームワークを書く技術者がほんの一握りしかいないことを考えると、この壁は高いのかもしれません。

しかし必要は発明の母と言います。技術者が特に上流工程の厳密な仕様記述の価値（それは知識の再利用を意味します）に気が付けばこの状況は変わっていくことでしょう。

第３章 仕様と他の工程との関係

この章について

本章では、まだ具体的な利用イメージが湧かないという方に対して、厳密な仕様記述がソフトウェア開発の各工程へ及ぼす影響、考慮すべき点について説明します。第 1 章、第 2 章の説明で、自らの業務にどのように厳密な仕様記述を適用し、どのような効果が得られるかイメージできた方は、ぜひ実際に適用してください。今まで改善できなかった問題を解決できるのは、数あるソフトウェア工学における手法の一つである、厳密な仕様記述である可能性があります。

小さいところからでも、専門家任せではなく、自ら利用することで大きな効果が得られるはずです。

厳密な仕様記述の効果は仕様そのものの改善だけではありません。たとえば、厳密で検証済みの仕様があれば、実装やテストといった開発プロセスも、系統的な方法へ変えるのが効率的かもしれません。また、一定以上の規模をもつソフトウェア開発では、事前に正しくサブシステムへの分割を行い、明確にインターフェースを定義したとしても、複数のサブシステムに渡る関心事の分散や実装制約により、さまざまな設計上の決定にすりあわせが必要になる場合が多く、情報共有が重要になります。厳密な仕様は、このような場合も関係者の合意を集積した「情報ハブ」として、ソフトウェア開発の間、常に参照され、常に更新されるべきなのです。

ソフトウェア開発プロジェクトに、本章の話題すべてを取り入れなくてはならないわけではありません。むしろ、本章ではそれぞれの項目をできるだけ独立して説明するよう心がけました。逆に、厳密な仕様記述と他の手法の組み合わせが有効であることも知られています。厳密な仕様記述だけで全ての問題を解決する必要はないのです。

以下における厳密な仕様記述としては、形式手法を想定しています。いきなり数理モデルはハードルが高いという方は UML や日本語による品質の高い仕様記述と読み替え

てください。しかし、ツールによる機械的な検証だけでも形式手法を利用する価値があるでしょう。

また、形式手法にはさまざまな手法と利用の度合いが考えられます。導入コストとその効果をよく検討し、—できれば定量的に—適切な手法とそれらの使いどころを選んでください。

1. 要求記述から厳密な仕様へ

いまのソフトウェア開発において使用している仕様は、どのように書かれていますか。

操作の順番を書き並べた「操作手順書」や表示される画面を何枚か描いた「画面設計書」だけが、仕様と呼ばれてはいないでしょうか。**仕様には、機能要求、性能目標、設計上の制約、必要な属性などが書かれていなければなりません**[1]。さらに、外部インターフェースや標準規約が加わることもあります。これらに関する要求が明確になっていなければ、ソフトウェア開発において何をどうするかを決定できず、いずれ行き詰ってしまいます。

要求が最初から全て分かっている場合はあまり多くありません。大なり小なり、対象のソフトウェア開発に関わる人たちの間でコミュニケーションをとりながら、何が明らかになっていないかを見つけだし、仕様を確定していく過程があります。

そもそも何を開発したらよいかという対象範囲を決めること、どの事項を表記すれば共通理解が得られるか、どの要求を優先すべきかといった、決定に関する課題はごく一般的なものです。開発コストと完了時期に関する、常にあるプレッシャーとの兼ね合いも考慮しなければなりません。こうした検討過程において、仕様は関係者の共通理解を表すものとして、できるだけ誤解のないように、厳密に書かれているのが望ましいのです。

仕様には、関係者の間における「情報ハブ」として、最新の合意内容が記述されているべきです。つまり、仕様に書かれていることは合意したことであり、書かれていないことは何をしてもよいわけではなく、合意されるとも、されないとも言えないことに当たります。したがって、お互いに後で変更されて困る事項は、仕様として明示するよう努力しなければなりません。

開発期間を通じて仕様を更新し続けることは、悪いことではありません。むしろ、合意事項をより分かりやすく書き直したり、不足していた項目を追加したり、設計情報を追加して設計仕様へ詳細化したりして、積極的に活用すべきです。

ただし、合意した項目を一方的に置き換えるのは、避けるべきです。これまでの開発作業が無駄になり、開発期間も延びてしまいますし、開発者の士気にも影響しかねません。そのような事態が発生しないように、更新した仕様に矛盾がないことはもちろん、妥当性や実現可能性についても考慮しておくのがよいでしょう。

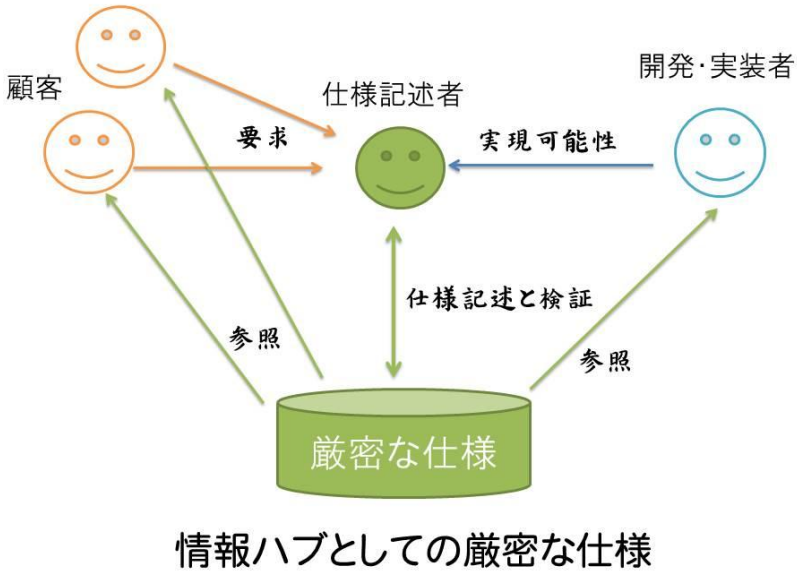


図 3 - 1: 情報ハブを介したコミュニケーション

仕様を厳密に書くにあたって、手戻りになりやすい点として以下のような項目があります[2]。

- 核心となる要求を見落とす
- 利用者の真の要求を見誤る
- 機能要求以外を無視する
- 要求の妥当性を確認しない
- 要求記述を完璧にしようとする

- 設計上の課題を混ぜてしまう

これらは仕様を書く際に、後での変更を避けるために、特に注意すべき点といえるでしょう。

仕様に合意事項を記述するためには、書かれている文書に関する文法と、その要素についての意味が定義され、関係者で共有されていなければなりません。過去に書いたものをすべて憶えておくわけにはいきませんから、これは個人にとっても有用です。しかし、その場そのときに仕様書の文法を定めるのでは、後で意味が分からなくなってしまいがちです。その四角に囲んだ箱はプロセスでしょうか、データでしょうか。その矢印は処理の順序ですか、データの流れですか、時間の経過ですか。 UML や形式仕様記述言語といった、定義された仕様記述言語を用いれば、このような問題を避けて、本質的な課題の分析と記述へ集中することができます。

また、仕様記述言語に定義された構造を利用して、構成管理や版管理といったプロジェクトの進行管理に関わる問題も改善可能です。

たとえば、形式仕様記述言語 VDM++ は、外部クラス単位の仕様をテキストベースのファイルに記述します。したがって、計算機言語によるプログラムと同様に maven や make などの構成管理システムあるいは、 subversion や git といった版管理システムを容易に利用できるのです。

もちろん、厳密に仕様を記述すれば全ての問題が解決するわけではありません。仕様をどのように開発工程で利用するかを決めるためには、なんらかの定量的な見積もりが有効です。

この章では、図 3 - 2 に示す要求を仕様として記述し、さまざまな工程で利用する過程を通じて、厳密な仕様と各工程の関係を示していきます。

この例題は、 1 階に厨房がある 2 階建ての食堂で、 2 階に配膳するためのエレベータを対象としています。エレベータといっても出来上がった料理と空いた食器だけを運ぶもので、主に安全上の理由により人間の移動は行いません。また、ハードウェアも含

めたシステム全体を対象としており、仕様を考える最初の段階では、どの部分を対象範囲とするかも重要な決定になります。 3.2 節で具体的な詳細を示します。

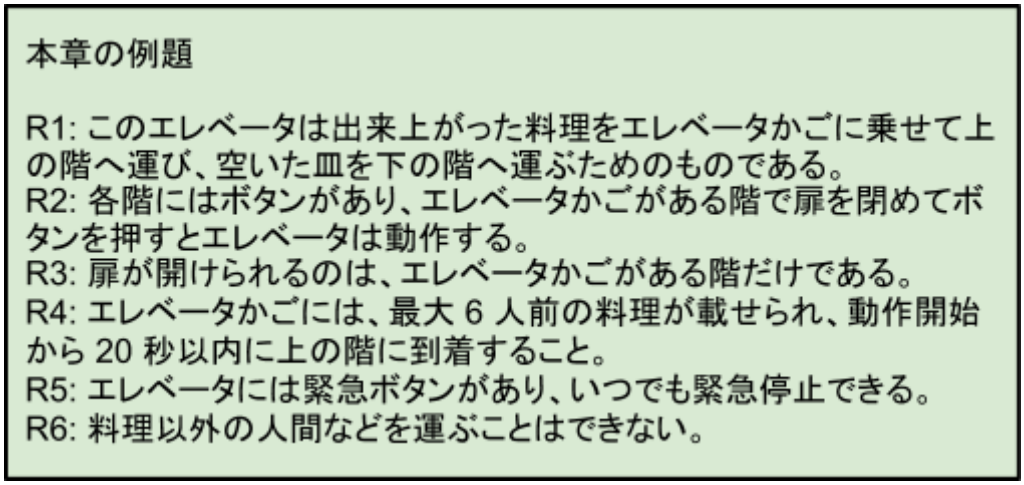


図 3 - 2: 本章の例題

[1] IEEE std. 1012-1986, IEEE Standard for Software Verification and Validation Plans.

[2] Brian Lawrence, Karl Wiegers, and Christof Ebert, The Top Risks of Requirements Engineering, IEEE Software, vol.18, no.6, pp.62-63, 2001.

2. 厳密な仕様とモデル化

仕様をモデルとして記述するとはどういうことでしょうか。

ソフトウェア開発におけるモデルにはさまざまな意味がありますが、共通する性質として、特定の見方に基づいた抽象化が挙げられます。仕様記述の観点からは、**開発対象となるシステムに対するさまざまな立場や性質についての要求を、一貫した視点で統合し、抽象化する作業がモデル化になります。**

具体的には、形式仕様を用いた、数理的な視点に基づく対象の見直しなどがそれにあたります。仕様をモデルと考えると、要求として挙げられた複数の機能の関係がどのようになっているか、どのような条件で共有される変数へのアクセスを許可するかなどの性質を、一貫性をもって記述できます。モデル駆動開発は、こうした一貫性をもったモデ

ルにより設計の正しさを保証する手法であり、形式仕様も数理モデルを用いたモデル駆動開発の一種といえます。

たとえば、 VDM では関数の入力値を集合とみなすことができます。集合は、値の範囲とその条件が明確に定義されるので、すべての場合分けを考えることができます。システムの内部状態に依存しない関数では、このようにして、記法によらず仕様の厳密さを上げることができるのです。これは、テストデータの選択でも用いられている手法でもあり、ここで得られた入力値の場合分けは、そのままテストデータの代表値として用いることができます。

ただし、モデル化にあたって捨象した部分は保証できないということも意識しておかなくてはなりません。すなわち、検証したい性質が予め分かっていれば、その性質に適したモデル化を行い、その上で検証することが重要になります。また、同じ開発対象に対して複数のモデルが存在するかもしれません。たとえば、 UML では表現する性質により用いるべき図表が異なります。このため、同じ対象を記述した複数の UML 図においては、意識して図表間の一貫性を保つ必要があります。

モデル化にあたっては、対象の性質を抽象化します。**抽象化で重要なのは、常に対象の本質を問い直すことと、要素間の関係を明確にすることです。**したがって、対象の抽象化は階層的になる可能性があります。要素間の関係も、モデルの他の部分と同様に、見方によって同じ要素間の関係が複数ある場合も考えられます。

機能の仕様記述には、第 2 章にあるように、事前条件と事後条件のみ規定する陰仕様とアルゴリズムまで記述する陽仕様という二通りの書き方があります。陰仕様は契約による設計を具現化したものといえます。

3.1 節にある本章の例題に対応する『ボタン』クラスとエレベータクラスからなる VDM++ によるモデルの例を次の図 3 - 3 および図 3 - 4 にそれぞれ示します。

```
class 『ボタン』

types
public 『状態』 = <入力可> | <入力不可> | <入力済み>;

instance variables
private 状態 : 『状態』;

operations
public 処理完了 : () ==> ()
処理完了() == 状態 := <入力可>
pre 状態 = <入力済み>;

public 押される : () ==> ()
押される() == 状態 := <入力済み>
pre 状態 = <入力可>;

public 入力を禁止する : () ==> ()
入力を禁止する() == 状態 := <入力不可>
pre 状態 = <入力可> or 状態 = <入力不可>;

public 入力を許可する : () ==> ()
入力を許可する() == 状態 := <入力可>
pre 状態 = <入力可> or 状態 = <入力不可>;

public 入力済み？ : () ==> bool
入力済み？() == return 状態 = <入力済み>;

public 入力不可？ : () ==> bool
入力不可？() == return 状態 = <入力不可>;

public 『ボタン』 : () ==> 『ボタン』
『ボタン』() == 状態 := <入力不可>;

end 『ボタン』
```

図 3 - 3: 『ボタン』クラス

```
class エレベータ
types
public 『扉』 = <開> | <閉>;
public 『階』 = <１階> | <２階>;

instance variables
かご : <１階> | <上昇中> | <２階> | <下降中> | <緊急停止>;
１階扉 : 『扉』;
２階扉 : 『扉』;
１階動作ボタン : 『ボタン』;
２階動作ボタン : 『ボタン』;
緊急ボタン : 『ボタン』;
inv not 緊急ボタン.入力不可？();
public 重量センサ : <OK> | <ERROR>;
動作モード : <通常> | <緊急>;

operations
--初期化
public エレベータ : () ==> エレベータ
エレベータ() == (
 かご := <１階>;
 １階扉 := <閉>;
 ２階扉 := <閉>;
 １階動作ボタン := new 『ボタン』();
 ２階動作ボタン := new 『ボタン』();
 緊急ボタン := new 『ボタン』();
 緊急ボタン.入力を許可する();
 動作モード := <通常>;
 １階動作ボタン.入力を許可する();
 );

--かごの操作
private 上昇開始 : () ==> ()
上昇開始() ==
 if １階動作ボタン.入力済み？()
  then (かご := <上昇中>; １階動作ボタン.処理完了())
pre 動作モード = <通常> and かご = <１階> and １階扉 = <閉> and 重量センサ = <OK>
post かご = <２階>;

private ２階で停止 : () ==> ()
２階で停止() == (
 かご := <２階>;
 動作ボタンを有効にする(<２階>, ２階動作ボタン, ２階扉);
 )
pre 動作モード = <通常> and (かご = <上昇中> or かご = <２階>)
post かご = <２階>;

private 下降開始 : () ==> ()
下降開始() ==
 if ２階動作ボタン.入力済み？()
  then (かご := <下降中>; ２階動作ボタン.処理完了())
pre 動作モード = <通常> and かご = <２階> and ２階扉 = <閉> and 重量センサ = <OK>
post かご = <１階>;
```

```
rivate １階で停止 : () ==> ()
 １階で停止() == (
  かご := <１階>;
  動作ボタンを有効にする(<１階>, １階動作ボタン, １階扉);
  )
pre 動作モード = <通常> and (かご = <下降中> or かご = <１階>)
post かご = <１階>;

--扉の操作
public 扉を開ける : 『階』 ==> ()
扉を開ける(指定階) ==
  cases 指定階:
  <１階> -> ( １階扉 := <開>; 動作ボタンを無効にする(<１階>, １階動作ボタン, １階扉)),
  <２階> -> ( ２階扉 := <開>; 動作ボタンを無効にする(<２階>, ２階動作ボタン, ２階扉)),
  others -> skip
  end
pre (動作モード = <通常> and かご = 指定階) or 動作モード = <緊急>;

public 扉を閉じる : 『階』 ==> ()
扉を閉じる(指定階) ==
  cases 指定階:
  <１階> -> ( １階扉 := <閉>; 動作ボタンを有効にする(<１階>, １階動作ボタン, １階扉)),
  <２階> -> ( ２階扉 := <閉>; 動作ボタンを有効にする(<２階>, ２階動作ボタン, ２階扉)),
  others -> skip
  end;

--ボタンの操作
public 動作ボタンを押す : 『階』 ==> ()
動作ボタンを押す(指定階) ==
  cases 指定階:
  <１階> -> ( １階動作ボタン.押される();上昇開始(); ２階で停止()),
  <２階> -> ( ２階動作ボタン.押される();下降開始(); １階で停止()),
  others -> skip
  end;

private 動作ボタンを有効にする : 『階』 * 『ボタン』 * 『扉』 ==> ()
動作ボタンを有効にする(指定階, 指定ボタン, 指定扉) ==
  指定ボタン.入力を許可する()
pre 動作モード = <通常> and かご = 指定階 and 指定扉 = <閉>;

private 動作ボタンを無効にする : 『階』 * 『ボタン』 * 『扉』 ==> ()
動作ボタンを無効にする(指定階, 指定ボタン, 指定扉) ==
  指定ボタン.入力を禁止する()
pre (not かご = 指定階) or 指定扉 = <開> or 動作モード = <緊急>;

public 緊急ボタンを押す : () ==> ()
緊急ボタンを押す() == (
  atomic(
  かご := <緊急停止>; 動作モード := <緊急>
  );
  緊急ボタン.押される();
  )

end エレベータ
```

図 3 - 4: エレベータクラス

図 3 - 3、図 3 - 4 には、 3.1 節の R2, R3, R5 および、 R1 と R4 の一部が反映されています。これらの関係を一つの状態機械として表現しているのがこのモデルになります。一方、 R1 はシステム全体の目的、 R4 の一部はハードウェア仕様、 R6 は注意書きなどでの対応を想定しており、これらは数理モデルの範囲外になる性質の例になります。

モデル記述には一貫した視点が重要ですので、厳密な仕様記述においても、専用に設計された仕様記述言語を用いるのが適しています。仕様記述言語には、さまざまなバリエーションがありますので、対象とする問題、関係者の経験、言語の安定性などを考慮して、目的にあった自分たちに適した言語を選択してください。

初めて仕様記述言語を利用する際には、その記法を学ばなくてはなりません。しかし、ソフトウェア開発の経験があれば、多くの仕様記述言語の文法は難しくありません。特に、形式仕様はその背景となっている数学理論に関する知識があれば、その数理的概念や性質がそのまま仕様記述言語に表れているのが分かります。とはいえ、現在のコンピュータのプログラムは同じ数学理論に基づいて動作しているのであって、いまソフトウェア開発を行なっている人は既に使っているものですから、難しく考える必要はありません。

実際、仕様記述で最も難しいのは、対象の本質をとらえ、表現する部分です。自然言語のような曖昧さを含む記法では、対象の本質をとらえたとしても、他の人に分かるように表現するには非常な努力が必要となります。

なぜなら、自然言語による相互理解は暗黙の了解を前提としており、それは関係者各人の属するそれぞれのプロジェクト、それぞれの経験、それぞれの時代などによって、語句の意味やとらえ方が微妙に異なるからです。また、仕様を書いた時点で問題がなくとも、読む人によって解釈が変わってしまう原因になります。この違いは小さいことも多く、それが故になかなか認識されず、よく注意しないと、後の工程まで誤解が持ち越されたりしてしまうのです。

繰り返しになりますが、仕様記述言語の習得には手間がかかります。しかし、一度学習してしまえばソフトウェア開発の多くの場面で役立つ、一生使える技術ですので、さまざまに活用してください。論理的な思考は、自然言語による記述にもいい影響があるはずです。

3. 厳密な仕様と検証

設計時の内部仕様が仕様を満たしているか、実装が内部仕様を満たしているかといった確認作業が検証の例です。厳密な仕様記述はソフトウェア開発における初期段階における検証を可能とし、手戻りによる開発コストの上昇を避ける手段として有効です。上流工程での不具合修正は、実装時の修正の 1 割のコストで可能と言われています[3]。これは昔から言われている話ですが、比較的新しいソフトウェアにおいても、あまり変わっていません[4]。

検証は、仕様シミュレーションによる方法と数理的証明による方法があります。シミュレーションは、仕様をプログラムと同様に動かしてみる方法で、いわば仕様のテストに相当します。シミュレーションの問題は、多くの仕様は動かすための情報が不足しているので人間が補完しなければならない点と、試行した環境と入力の組についてしかその性質を保証できない点です。そのため、完全な保証を与えるためには、モデル検査のような総当りの手法と組み合わせる必要があります。

一方、数理的証明は仕様が数理的に書かれている必要があります。形式仕様は数理的なモデルですので、検証のアルゴリズムを同じモデル上で数理的に定義できれば、検証を計算機上のツールにより実行し、かつその結果が正しいかどうかの判定を行えます。ツールの利用により、計算機の性能向上につれて、検証能力を向上させることが可能となります。

モデル検査は、ソフトウェアを有限状態機械とみなし、すべての状態遷移の場合を尽くすことにより異常状態にならないという証明を与える手法です。 Promela/SPIN, NuSMV, FSP/LTSA などが代表的なモデル検査手法です。計算機の能力向上に伴い、このように証明を力ずくで計算機に代行させる方法も実用的になっています。

モデル検査の課題は、対象の性質を表現する適切な抽象度で書き表すのが難しいという点です。状態機械における状態は階層性があるので、適切な抽象度を選ばないと、検証すべき対象の無意味な性質を証明するだけになってしまいます。また、どのような状態が異常であるかも仕様として与えなくてはなりません。多くのモデル検査ツールは、デッドロックやライブロックを検出できますが、それが異常であるかどうかは対象の性質によります。それ以外の異常は、時相論理などにより人間が記述し、モデルと同時にツールへ入力しなくてはならないのです。

つまり、モデル検査はあくまで、計算機により証明を代行するだけであり、何を証明するかという一番難しい課題は検証する人が考える必要があります。

もちろん、数理的検証で検証されるのは記述した数理モデル上の数理的な性質だけであることは、理解しておく必要があります。形式仕様で記述し、検証したからといって、対象となるソフトウェアの不具合が全て除かれるわけではありません。よく用いられる形式仕様記述法は、機能に関する性質を検証する VDM Family や Z 記法/B-Method など、あるいは振る舞いに関する性質を検証する Promela/SPIN や NuSMV, CafeOBJ などです。これらの一部を第５章 に紹介しています。

非機能要求も詳細化することで、機能要求に置き換えられる場合があります。 3.1 節にある本章の課題 R4 における「最大 6 人前の料理」を考えてみましょう。 6 人前の料理がどのようなものかは、注文により異なります。しかし、設計対象が料理を運ぶためのエレベータであることを考えれば、この要求が設計に与える影響は主に「重さ」と「置き場所」になるでしょう。置き場所を決める広さはハードウェアの条件ですが、重さは動作可能な最大重量としてソフトウェアによる制御にも影響します。この例では、これを重量センサによる最大重量を超えていないかの判定という機能要求に置き換えているのです。

ただし、仕様記述は手段に過ぎませんので、本質的に曖昧な要求を対象とすると、「間違いなく曖昧な仕様を記述する」ことになってしまいます。すなわち、仕様を厳密に記述しさえすればよい仕様が出来上がるというわけではないので、何を書くべきかという対象の本質を見いだす努力は常に必要となります。

要求に書かれている機能が無効となる条件が書かれていない、ボタンの機能が変化するモードの情報がないなど、直接機能に関係しない情報が抜け落ちる対象範囲の誤りはよくある仕様の不具合です。

たとえば、図 3 - 5 に示す本章の課題では、動作モードが大域的なモードにあたります。R3 にあるように、通常の動作時は、かごがないときに扉を開けると、かごが動いてきて危険なので禁止されなくてはなりませんが、緊急時にはかごを停止するとともに、かごがどこにあっても扉を開けられるようにするべきでしょう。これは、要求として明示されていませんが、モデル化により書かれていない情報が見つかり、明示的に補完される例にあたります。

```
public 扉を開ける : 『階』 ==> ()
扉を開ける(指定階) ==
  cases 指定階:
  <1階> -> (1階扉 := <開>; 動作ボタンを無効にする(<1階>, 1階動作ボタン, 1階扉)),
  <2階> -> (2階扉 := <開>; 動作ボタンを無効にする(<2階>, 2階動作ボタン, 2階扉)),
  others -> skip
  end
pre (動作モード = <通常> and かご = 指定階) or 動作モード = <緊急>;
```

図 3 - 5: エレベータのモデル（抜粋）

同じように、境界条件を数学的に定義してすべて数え上げれば、見落としやすい場合も見つけられます。このとき、それぞれの変数のとり得る値が集合として十分に定義されているかどうかを注意してください。このように整理すると、複数の変数がある場合も、集合の包含関係や共通部分集合の有無が判定できるようになり、要求の漏れを防ぐことができるのです。

[3] Barry W. Boehm, Software Engineering, IEEE Transactions on Computers, Vol.C-25, No.12, pp.1226-1241, 1976.

[4] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai, Have things changed now?: an empirical study of bug characteristics in modern

open source software, Proceedings of the 1st workshop on Architectural and system support for improving software dependability, pp.25-33, 2006.

4. 厳密な仕様と要求の妥当性確認

ソフトウェア開発は、自然言語(日本語、英語など人間同士で使う言語)による要求の発露から始まるのが一般的です。ソフトウェア開発が人間による活動の一部である限り、それは変わらないでしょう。現在の計算機言語(計算機での実行を目的としたプログラム記述言語)と自然言語には意味的なギャップがあり、ソフトウェア開発は、開発者がそのギャップを埋める翻訳作業とみなすこともできます。

このとき、少なくともソフトウェアを開発して欲しい「利用者」あるいは「発注者」と、ソフトウェアを開発する「開発者」あるいは「受注者」という 2 つの立場があります。これらは役割ですので、実際は同じ人や組織であったり、対象プロジェクトによって変わったり、システムの再分割など開発工程によって変わることもあります。

開発者は、その専門知識を活用して、利用者の要求を正しく反映したシステムが出来上るように仕様を記述するとともに、利用者が理解できるように説明できるべきです。逆に、利用者は仕様からどのようなものが出来上がるのかを読み取り、要求が漏れていないか、正しい優先順位になっているかといった希望が反映されていることを確認すべきです。しかし、どのようなシステムが出来上がるかを仕様からイメージすることは、慣れた人でも難しいことです。

利用者による仕様確認方法の一つは、レビューあるいはインスペクションです。ただし、効果的なレビューにはそれなりの計算機に対する知識が必要であり、計算機システムになじみのない利用者には敷居が高いものです。このような場合、仕様が要求を満たしているかを確認するには、対象システムの一部機能が動作するように、不完全なモデルを作成するプロトタイピングが有効です。プロトタイプにより、仕様からプログラムを実装する前にそのふるまいを知ることができれば、仕様が要求を満たしているか、あるいはその仕様が目的に対して妥当かどうかを確認することができるからです。

厳密な仕様記述の利点の一つが、仕様に入力データを与え、仕様に沿った評価を行い、その結果を出力する仕様アニメーションです。仕様アニメーションは対象となる仕様を、特定の入力に対するプロトタイプとして利用できるのです。もちろん、仕様アニメーションは開発者にとっても有用です。

形式仕様には、対話的な仕様アニメーション機能をサポートするツールをもつものがあります。これらのツールは、計算機上でユーザから与えられた入力に対し、自動的に適用可能な仕様の候補を見つけて、選択された仕様の評価結果を表示することができます。ツールの補助機能により、紙上のプロトタイピングよりずっと効率的な仕様の確認ができます。

本章の例題では、図 3 - 6 のようなテストケースを記述したクラスを用意し、VDMTools 上で実行できます。

```
class エレベータTS

instance variables
public ev : エレベータ := new エレベータ();

values

operations
public runTest : () ==> ()
 runTest() == (
   ev.扉を開ける(<1階>);
   ev.扉を閉じる(<1階>);
   ev.重量センサ := <OK>;
   ev.動作ボタンを押す(<1階>);
   ev.扉を開ける(<2階>);
   );

end エレベータTS
```

図 3 - 6: エレベータのテストケースの例

実行結果は図 3 - 7 のようになります。

```
> load エレベータTS.vpp エレベータ.vpp『ボタン』.vpp
> create test := new エレベータTS()
> print test.ev
objref5(エレベータ):
 <  - エレベータ`かご = <1階>,
    - エレベータ`1階扉 = <閉>,
    - エレベータ`2階扉 = <閉>,
    + エレベータ`重量センサ = <<UNDEFINED>>,
    - エレベータ`動作モード = <通常>,
    - エレベータ`緊急ボタン = <入力可>,
    - エレベータ`1階動作ボタン = mk_token( "objref6" ),
    - エレベータ`2階動作ボタン = mk_token( "objref7" ) >
> debug test.runTest()
> print test.ev
objref5(エレベータ):
 <  - エレベータ`かご = <2階>,
    - エレベータ`1階扉 = <閉>,
    - エレベータ`2階扉 = <開>,
    + エレベータ`重量センサ = <OK>,
    - エレベータ`動作モード = <通常>,
    - エレベータ`緊急ボタン = <入力可>,
    - エレベータ`1階動作ボタン = mk_token( "objref6" ),
    - エレベータ`2階動作ボタン = mk_token( "objref7" ) >
```

図 3 - 7: 仕様に対するテストの実行例

この例では、初期状態から、このエレベータに１階で料理を載せ、重量センサがエラーでなければ、ボタンを押してかごを２階へ送り、２階で取り出すという手順をテストしています。このように、形式仕様を用いると、特定の入力系列に対するモデルの振る舞いを、ツールの補助により実行しながら確認できます。

形式仕様のアニメーションは単なるシミュレーションではなく、背景となっている数理モデルにおける性質の保証が与えられます。たとえば、テストの代表値選択には「２つの入力が近い値であれば、それらの出力も近い値になるだろう」といった漠然とした仮定をおくことが多いのですが、数理モデルを用いればその仮定が適切かどうかを確認できるのです。

厳密な仕様がない場合、実際に動くプログラムが出来てから、その振る舞いにより良し悪しを判断するしかありません。しかも、その判断基準は利用者任せになってしまい、利用者が全ての機能を確認しなくてはならず、またその結果が開発者へフィードバックされるまで時間がかかってしまいます。それに対して、「情報ハブ」としての仕様を参

照できれば、仕様についても、プログラムについても、開発者が自ら良し悪しを判断できるようになるでしょう。

しかしながら、アニメーションを利用するためだけに、仕様として不要なはずの実装アルゴリズムを書いてしまうのは不適切です。本来は、**利用者と開発者の間で必要な合意のみを仕様とし、設計や実装の工程に用いる内部仕様とは区別して扱わなくてはなりません。**さもないと、いつまでたっても仕様記述が終わらなくなってしまい、実装やテストに十分な時間がとれなくなってしまいます。ところが、仕様を決めていく段階で主導的であるべき開発者は、利用者の関わらない実装アルゴリズムやサブシステムへの分割も仕様として記述してしまいがちです。

仕様として合意されたものは、ソフトウェアとして実現されなくてはなりません。つまり、要求の妥当性が確認できないものを仕様に含めると、ソフトウェアとして実現できない部分が含まれることになり、不適切な仕様になってしまいます。最終的に仕様に含めるかどうかの判断は別として、ソフトウェア開発の現実的な制約の下で、要求が実現可能であることの確認は重要なポイントです。

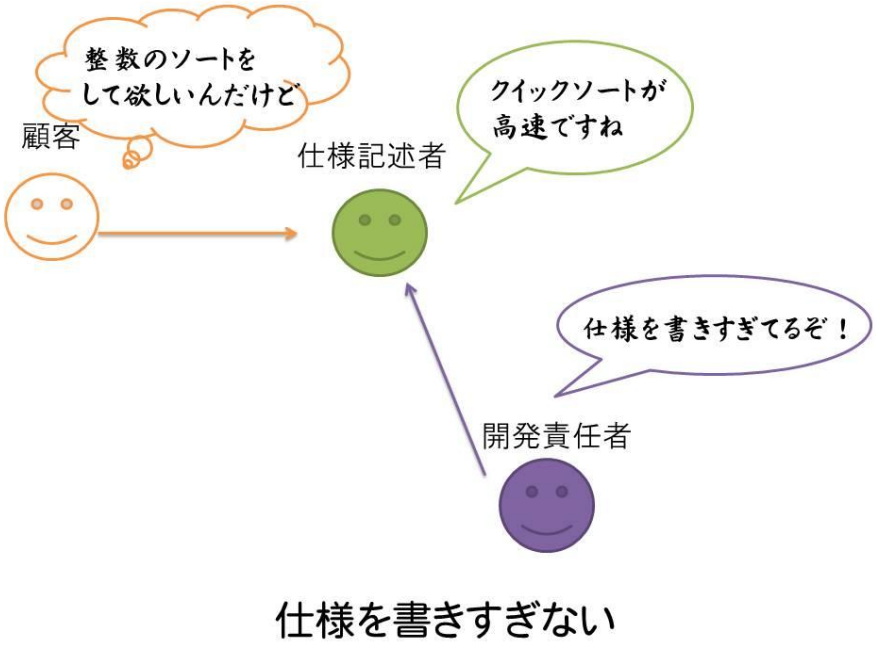


図 3 - 8: 設計者や実装者を信用しよう

形式仕様記述では、より詳細なモデルの自動生成を前提に、モデル記述に全体の 50% 以上の作業量をかけた例もあります[5]。

仕様をどこまで書けばよいのかという問いに一般的な解はありません。対象システムの仕様は、開発者が利用者と何を合意しておくべきかによってのみ決まりますので、開発者と利用者との関係など非技術的な要素も考慮すべきです。

そして、仕様を決めることは、仕様を読む人が必要な情報が何かを決定する作業でもあります。必要な情報を漏らさないのと同時に、不要な情報はできるだけ書かないようにしましょう。

[5] ClearSy and Siemens Transportation Systems, Using B as a High Level Programming Language in an Industrial Project: Roissy VAL, Proceedings of the 4th International Conference of B and Z Users, pp.334-354, 2005.

5. 厳密な仕様と仕様の詳細化

ソフトウェア開発は、仕様を満たすような設計、設計を満たすような実装と複数の工程とその成果物が連続しながら進んでいきます。工程ごとの違いが小さいほど手間はかかりますが、実装が元の仕様を満たしているという保証は与えやすくなります。

ところが、せっかく書いた仕様をその後の工程で活用せず、仕様を斜め読みしてすぐにコーディングを始めたり、仕様に書かれていない機能を作りこんだりというケースがしばしば見られます。再利用される可能性のない使い捨てのプログラムであればともかく、対象システムの全体を理解しておかないと、仕様として合意した機能のバランスを崩したり、誤ったソフトウェア・アーキテクチャを適用してしまったりしかねません。

そもそも、仕様を理解せずにその完成度を評価できるのでしょうか。

仕様策定に参加していたならともかく、**与えられた仕様が後工程で参照するに足るものか**どうかを最初に検討する必要があります。この時点で不具合や曖昧な点があれば解消しておかないと、高い確率で大きな手戻りのコストが発生します。このときに、仕様からすぐに実装を作れるものか、何段階か詳細化して内部仕様を生成するのがよいのかをある程度、見極めなくてはなりません。

ある仕様を満たすシステムは、複数存在します。たとえば、対象となるデータが特別な偏りをもっていたり、メモリ階層など実装上の制約を考慮したりすると、適切なデータ構造やアルゴリズムを選択しなければ、実行効率や互換性の低下を招くことになります。メモリ量やプロセッサ性能などにより、配列のソートに関する、要素間のリンクの付け方、内部ソートや外部ソートといったアルゴリズムの違いが、性能に大きな影響を与えるといった場合が具体的には挙げられます。

また、ソフトウェア・アーキテクチャによっても詳細化の方針に大きな違いが生じます。たとえば、分散処理に関するアーキテクチャは、責務分割や実装上の制約に関する考慮も含め、システム全体の振る舞いを規定する工程を補助するものです。分散処理に関するソフトウェア・アーキテクチャには、クライアント/サーバ、エージェント・システ

ム、Peer to Peer(P2P) などがあり、それぞれは対象システムの性質だけでなく、保守性や耐故障性、あるいはネットワーク帯域などの優先度により最適なアーキテクチャは異なります。

詳細化により大きな影響があるのは、非機能要求です。形式手法は数学的な理論を利用して対象システムを抽象化し、仕様を記述するので、背景となる数理モデルによってシステムの見方が変わり、保証できる数学的性質も異なります。広く使われている形式手法のほとんどは機能記述を対象としていますが、これは仕様として機能要件が対象となる場合が多いこと、またそれらの評価が達成度として定量化しやすいことによります。

これに対して、非機能要件は直接的な評価法が確立していないものが多く、定量的に評価しようとすると対応する機能要件群に置き換えて、それらの達成度で評価する場合がほとんどです。設計や実装においても、特定の指標を設定し、それらの指標の測定および基準との比較という機能により非機能要件の達成度を評価します。

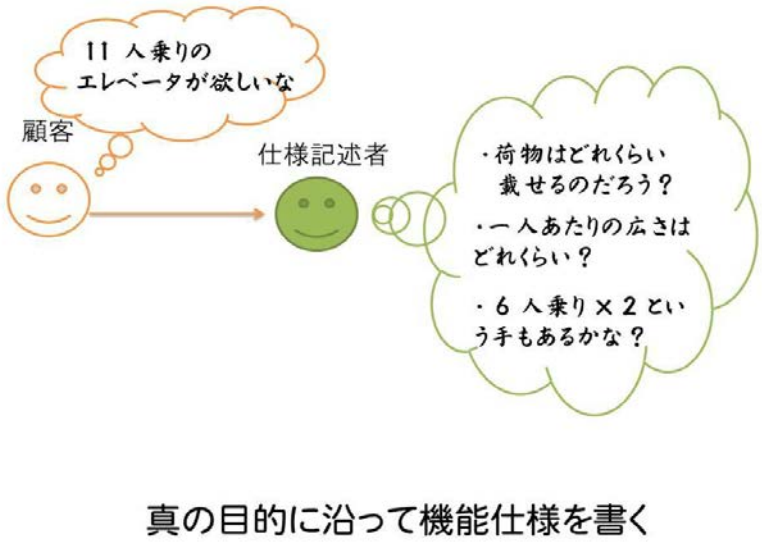


図 3 - 9: 真の要求を見極めよう

それでは、ある内部仕様をもつシステムの仕様は複数存在するのでしょうか。これも同じように複数存在し得るのです。たとえば、入力されるデータサイズの変更といった拡

張性や他の実行環境への移植性をどの程度予期するか、運用時のエラー処理などをどの程度許容するかといった判断により、仕様は異なってきます。

具体的な例をみてみましょう。図 3 - 10 および図 3 - 11 は LTSA を用いて生成した状態遷移図です。LTSA の詳細については、第５章を参照してください。

本章の例題では、かごの状態を 1 階か 2 階かという要求に表れる状態だけではなく、次の図のように上昇中、下降中という遷移状態を区別しています。これは、システムの抽象度を少し実装に近づけて、エレベータ制御の中心となるかごの制御の特徴である、動き出しと停止を識別するためです。

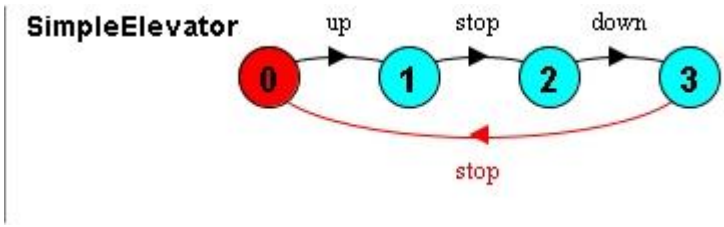


図 3 - 10: 上昇下降のみの状態遷移

人が乗るエレベータでは乗り心地を考慮し、動き出しや停止をさらに細かな段階に分けて制御するものもありますので、そうした場合はさらに細かい区別が必要になります。この例題では、乗り心地を考慮する必要はあまりありませんので、図 3 - 10 のような抽象度が適切と考えられます。

この例題も、実際にプログラムとして動かすためには、扉の開け閉めやボタンの有効化・無効化といったより詳細な状態の定義が必要になります。 3.2 節のモデルに沿ってこれらの情報を追加した状態遷移図はあっという間に次の図 3 - 11 のようになります。しかも、これには大域的な緊急停止は含まれていません。

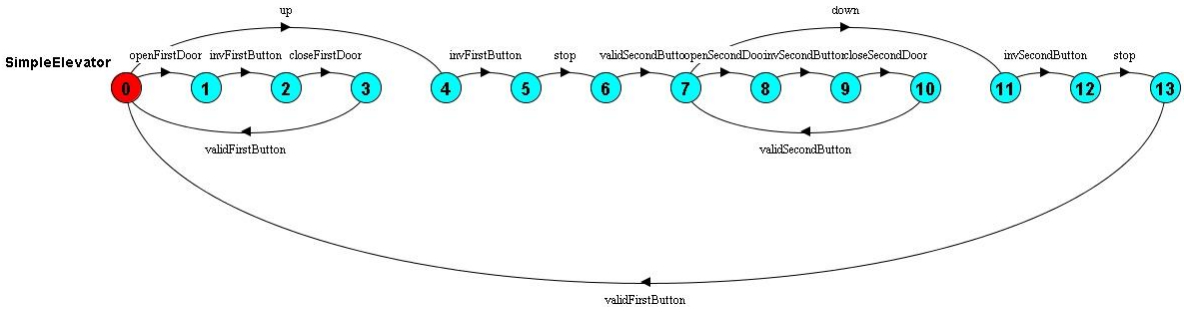


図 3 - 11: 扉やボタンの状態も含めた状態遷移

つまり、**仕様は開発期間を通じて詳細化されたり、解釈し直されたりする**ということです。神棚に上げてしまったかのように、仕様を大事に眺めても活用したことにはなりません。仕様をどのように実現するかという内部仕様へ書き換え、常にその正しさを検証しつづけることで、最も詳細化された実装が仕様を満たしていることを保証できるのです。

6. 厳密な仕様とレビュー

仕様を確認する手法として最もよく用いられている手法はレビューでしょう。レビューはソフトウェア開発における段階的な詳細化の途中でドキュメントやプログラムを人間の目により確認し、それら中間生成物の不具合を早期発見・修正したり、完成までの作業を見積もったりできます。独立した専門家のレビューにせよ、関係者によるピアレビューにせよ、**レビューの根拠となるのは仕様であり、プログラムだけではその正当性を示すことはできません**。

レビューは、ソフトウェアの品質向上に有用な手法であり、関係者と対象となるシステムの知識を共有できるだけでなく、プログラミングのテクニックなどを学ぶこともできるなど、価値の高い工程になります。特に、利用者と開発者が一同に介して行うレビューは、相互に素早いフィードや疑問点の解消が行われるので、高い開発効率が期待できます。しかし、複数の関係者へレビューを依頼すれば、その間チームは別の作業をできなくなります。効率的なレビューは、開発期間の短縮に欠かせません。

形式仕様を利用する場合、プロトタイピングによる仕様の検証も有力です。しかし、VDM でプロトタイピングを行うためには、仕様を動作させるためには陽記述が必要となります。ところが、仕様は必ずしも陽に記述されるとは限りません。

これに対して、レビューには厳密な手順が定められた方式から、ごく簡単な手順による方式までさまざまな提案がなされており、期間や開発者の技量に応じてさまざまな手法を使い分けることが可能です。レビュアの立場もレビューの専門家へ依頼する場合と関係者が相互にレビューしあうピアレビューがありますし、対象も開発者およびレビュアの能力次第で、陰仕様、陽仕様、自然言語によるドキュメント、図や表など扱うことができます。この場合、仕様はツールによる文法検査、型検査、人間によるレビューという静的な検査により検証されます。

ただし、レビューは、比較的短時間で中間生成物をみていくので、問題領域やプログラムについてある程度の専門知識を必要とします。

レビューにおけるもう一つの大きな問題は、仕様が変更された場合の影響を見極めにくい所です。一部でも仕様の変更があれば、仕様中の他の部分に直接あるいは間接的に影響する可能性があり、多くの部分をレビューし直さなくてはなりません。そして、ソフトウェア開発の途中で要求が変更されることはよくあるのです。

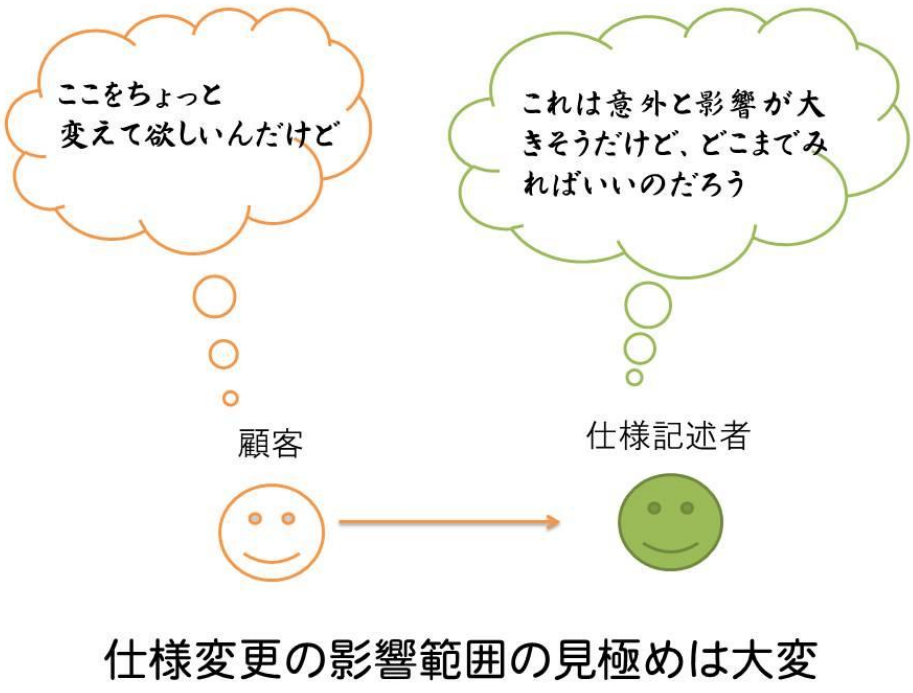


図 3 - 12: 見掛けの小ささに惑わされないようにしよう

厳密な仕様は、短時間でのレビュアによる仕様の理解を可能にします。これは、自然言語による仕様でしばしば要求される「行間を読む」必要がなくなるからです。

用語の曖昧さや関連の曖昧さは、読み手の解釈を必要とし、異なる理解による問題を生じる原因となり得ますが、これらは厳密な仕様で最初に解消すべき点です。

また、修飾語がどこまでかかるかは書き手の意図が読み手に伝わりにくく、誤解の元となります。たとえば、「赤いボタンのついたキーボード」は、赤色でしょうか、それとも黒色でしょうか。

もう少しやっかいなのは、含意による表現です。「問題が発生したら、処理の後にエラーを通知してください。」という要求は、問題が発生しなければ何も通知しないことを意味しているのでしょうか。それとも、エラー以外の何かを通知することを意味しているのでしょうか。

R1: このエレベータは出来上がった料理をエレベータかごに乗せて上の階へ運び、空いた皿を下の階へ運ぶためのものである。
R2: 各階にはボタンがあり、エレベータかごがある階で扉を閉めてボタンを押すとエレベータは動作する。

図 3 - 13: 本章の課題（抜粋）

上に示した図 3 - 13 でも、ボタンを押せばかごが動作することは分かります。しかし、ボタンを押さなくても動作してよいかは不明です。この例題では、 R1 の目的から、同じ階で同じときに扉を複数回開け閉めする可能性があって、ボタンが押されなければ動かないのが望ましいと想像できますが、もっと大きな問題で、部分的な仕様しか分からない場合、解釈の相違が生まれやすいところです。

もちろん、 3.2 節のようにモデルとして記述すれば、どの条件でかごの移動が起こるか、あるいは起こらないかが明確になります。厳密な仕様では、こうした仕様の各部分の関連も明確にしなければならず、抜けている条件も見つけやすくなるのです。

たとえば、話題沸騰ポットの場合[6]、簡易な形式記法からの状態遷移表の自動生成により、 109 件の表記のゆれや条件もれなどの指摘事項が見つかっています[7]。

さらに、厳密な仕様があれば、レビューに必要な資料を改めて作成する必要がなくなります。レビューのたびに資料を書き起こすのでは、作業量が大きくなりますし、資料と実際のソフトウェアが同じであるという保証を与えることもできません。仕様をレビューし、その結果に基づいて仕様を改善すれば、常に最新の仕様に基づいたソフトウェア開発が可能となります。

より進んで、形式仕様により厳密な仕様を記述すれば、文法および型の検査をツールにより行うことができますので、自然言語による記述のように、表現や書き誤りへの注意を大幅に減らせます。これにより、分かりやすさを向上させるための識別子の名前付け規則や入力データに適したアルゴリズム選択といった、より本質的な部分にレビュアの意識を集中させることが可能になります。

仕様のレビューは、計算機言語によるプログラムよりも抽象度が高く、より一般性の高い数学的な概念に基づいていますので、計算機に関する知識がない人でも、コードレビューより対象となる仕様の理解は容易です。また、実装のためのテクニックや最適化に煩わされることもありません。プログラムのレビューも基準となるレビュー済の仕様があれば、スムースに進むことでしょう。

仕様のレビューでコードレビューが不要になるわけではありませんが、**仕様はあるべき姿、コードは現在の姿**とお互いのレビューする目的がしっかり分けられるので、レビュアは見るべき点を整理して、指摘するべき内容に集中できるようになります。厳密な仕様をうまく活用すれば、このような効果も期待できるのです。

[6] SESSAME WG2, **組込みソフトウェア開発のためのオブジェクト指向モデリング**, **翔泳社**, 2006.

[7] **水口 大知, 漆原 憲博, 簡易な形式仕様記述と状態遷移表を併用した要求仕様のレビュー方法, 第 8 回クリティカルソフトウェアワークショップ**, A2, 2011.

7. 厳密な仕様と実装および単体テスト

仕様からコードを自動生成するツールがあります。これらはドライバ生成など特定の分野では有用ですが、汎用のソフトウェア開発向けに利用した場合、コードと同じことを書かなければならなかったり、可読性が著しく悪いために変更が困難であったり、まだ不十分なものであることが分かるでしょう。

設計や実装には、環境の制約や最適化といった仕様とは異なる情報が必要なのですから、これはある意味であたりまえです。自動生成がうまく働くのは、環境が限定され、必要な情報が分かっている場合です。そうでなければ、仕様に必要な情報を付加していかなければなりません。

ある程度以上の規模におけるソフトウェア開発における仕様書は、使用するプロセスや開発規模により異なりますが、一般的には複数の段階があり、代表的なものとして、次の仕様書があります。

- 要求仕様書(ユーザ仕様書)
  発注者が求める開発対象システムが満たすべき仕様を記述したもの。非機能要求や実行環境要件(OS やハードウェア)が含まれる可能性もあります。

- システム仕様書(外部仕様書、機能仕様書)
  システムがどのような機能を満たさなければならないかを記述したもの。要求仕様書との大きな違いは、非機能要件を機能的に定義している点です。

- 詳細仕様書(内部仕様書、技術仕様書)
  システム内のサブモジュールをどのような仕様で実装するかという設計情報を記述したもの。通常は設計やコーディングを行う開発者向けに記述され、ユーザには公開されません。

この他に必要に応じて、画面仕様書、テスト仕様書、手順仕様書などが加わることもあります。

それぞれの仕様書はそれぞれの目的があり、対象となる読み手も書くべき内容も異なります。形式仕様は、汎用的な数理モデルを用いており、いずれへも適用可能なので、書き手が意識して書き分けないと、要求仕様を書いていたはずが詳細仕様書が出来上がるといったことになりかねません。

仕様は、テスト仕様を開発する際にも用いられます。プログラムによる実装は、仕様を満たしていなければなりません。プログラムが仕様を満たしていることを確認する最も簡単な方法は、入力と仕様から予想される出力とプログラムの実行結果を比較するブラックボックステストです。

同様にして、確定した仕様が意図せずに変更されていないかどうか確認することができます。たとえば、 VDM++ には回帰テストを行うための VDMUnit ライブラリが用意されています[8]。 VDMUnit は 他の xUnit 同様に、単体回帰テストのテスト対象モジュール、対応するテストケース、対応するテストドライバを用意します。

対象モジュールは VDM++ の陽仕様で記述されたクラス単位でテストされます。テストケースは TestCase クラスを継承したクラスとして、それぞれ前処理、テスト本体、後処理に対応する setUp, runTest, tearDown の 三つの operation を定義します。テストドライバは、単体テストの初期化、複数のテストケース実行、集計など後処理を行うよう記述します。

VDM++ の単体回帰テストはクラス単位で行われ、ツールの仕様アニメーション機能を利用します。したがって、 VDMUnit を利用するには、対象モジュールが陽仕様で記述されていなければなりません。逆に、このような条件を満たす全てのモジュールに対して、ツールにより回帰テストを自動実行させることができます。VDMUnit を活用することで、仕様の変更が他の部分に影響を及ぼしていないかといった確認が容易に可能です。

陽仕様のテストデータは、コードのテストデータとしても利用できます。前提として、陽仕様のテストに十分なテストケースが用意されていなければなりませんが、両者の結果が一致すれば、ブラックボックステストは満たしているといえるでしょう。

本章の例題は、状態遷移が主ですので返値はほぼありませんが、テストシナリオは共通です。図 3 - 14 に示すように、 3.4 節の正常系のシナリオは、そのままシステムテストのシナリオに使えますし、コードのテストにおけるシナリオとしても利用できます。ここでは省略しますが、異常系や他のシナリオについても同様の再利用が可能です。

```
テストシナリオ:
初期状態から、このエレベータに1階で料理を載せ、重量センサがエラーでなければ、
ボタンを押してかごを2階へ送り、2階で取り出す。

仕様のテストシナリオ:
public runTest : () ==> ()
 runTest() ==
   (
   ev.扉を開ける(<1階>);
   ev.扉を閉じる(<1階>);
   ev.重量センサ := <OK>;
   ev.動作ボタンを押す(<1階>);
   ev.扉を開ける(<2階>);
   );
```

図 3 - 14: 共通のテストシナリオ

陽仕様に対しては、内部の状態遷移を確認するホワイトボックステストも可能であり、変数名を合わせれば、こちらもコードに対するホワイトボックステストとして利用できます。性能の確認など実装についてのテストは別途用意しなくてはなりませんが、仕様に関する問題は、積極的に共通のデータを用いることで、誤りが生じた段階を特定しやすくなるのです。

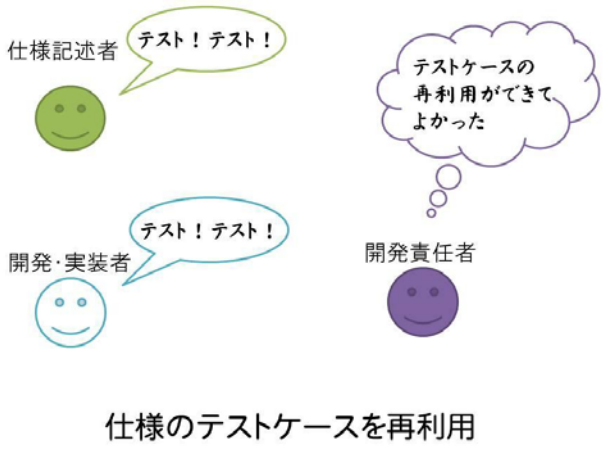


図 3 - 15: 仕様は積極的に活用しよう

逆にしばしばみられる失敗に、厳密な仕様を記述しようとして書き過ぎる、オーバーモデリングがあります。これはプログラミング言語におけるデータ型に影響されたり、実装にとらわれたりして対象の本質を見失う状況です。

仕様を書くときには、常に対象の本質を洞察し、仕様として表現できているかどうかを問い直さなくてはなりません。初心者はもちろん、経験のあるソフトウェア開発者こそ、仕様がコード作成のためだけのものではないことを意識し、対象の本質を明らかにするよう心がけるべきなのです。

[8] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef, 酒匂 寛(訳), VDM++によるオブジェクト指向システムの高品質設計と検証 , 翔泳社 2010.

8. 厳密な仕様と結合テストおよびシステムテスト

複数のモジュールを組み合わせてソフトウェアを構築する作業をビルドとよびます。結合テストはソフトウェアをビルドし、実行可能なプログラムを生成し、それが仕様どおりに動くことを確認する工程です。ビルドは実行環境にも依存しますので、複数の実行環境が想定される場合はそれぞれで確認しなくてはなりません。

システムテストは、ビルドしたプログラムを実際に使用される環境でソフトウェアの動作させる工程であり、さまざまな使われ方に対して問題が生じないことを最終的に確認します。ここでは、ユーザの誤操作やハードウェアとのすり合わせなど、多角的なテストが重要になります。

仕様に基づいて適切なモジュールへ分割し、個別のモジュールをそれぞれの仕様どおりに作成し、単体テストをパスしても、ビルドが失敗することもあります。このような問題が発生する原因は、相互のモジュールにおけるインターフェースのミスマッチです。具体的には、各モジュールで異なる仕様を参照していたり、ライブラリや実行環境のバージョンが異なっていたりといった場合が該当します。

これを設計あるいは実装の段階で防ぐには、適切な版管理ツールを用いたり、常にビルド可能であることを確認したりする継続的インテグレーションといった手法があります。

版管理ツールは、開発中のソフトウェアの複数の版を保持し、簡単に切り替える機能を備えており、不具合が発生した時に以前の版へ戻したり、新しい機能の試作を別の版に分けたりといった作業を容易にします。最近の版管理ツールのほとんどは、共同作業における変更の衝突を検知する機能を備えており、個人の作業成果とチームとしての作業成果を分離し、うまく統合することができるようになっています。

継続的インテグレーションは、自動的なテスト実行環境を構築し、ビルドを細かい間隔で繰り返すことによって、プログラムの変更に伴うビルド失敗を早期に発見します[9]。この段階で発見される不具合はかなり多いので、早期かつ自動的なビルドは手戻りを小さくする意味で大きな効果があります。

しかしながら、設計あるいは実装の段階で対策をしても、モジュール間で仕様レベルの誤解や誤りによる不具合が生じることがあります。

この工程で生成されるソフトウェアが正しいかどうかは、直接、仕様と照らし合わせることになります。正しいソフトウェアを作るためには、正しく仕様を記述するだけでなく、正しく仕様を理解しなくてはならないのです。

開発者によって参照する仕様の版が異なるといった状況は、真っ先に排除すべきです。紙による仕様書をつかっていると、どうしても仕様が行き渡らなかったり、不具合の修正に時間がかかったりします。厳密な仕様をモデルとして記述し、「情報ハブ」として活用すれば、何も追加投資を行わなくても、設計や開発を行う人とテストを行う人の間で、最新かつ検証済みの仕様を共有できます。

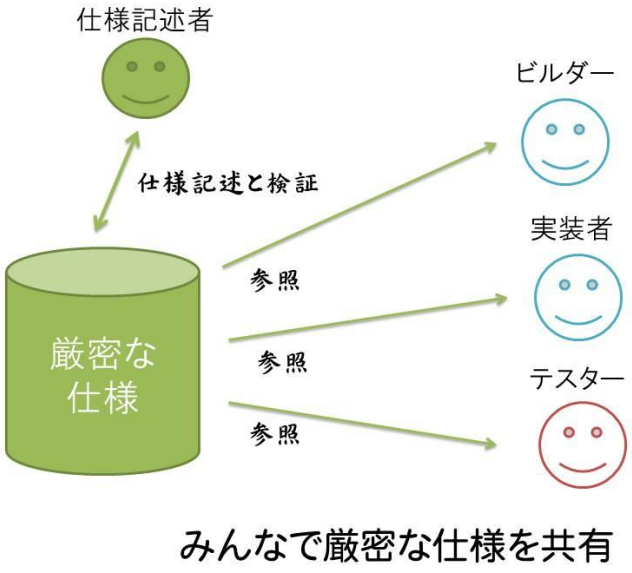


図 3 - 16: 紙ではなくモデルを共有しよう

確認手段としてのテストの規範になるのも仕様です。仕様の観点からは、テスト結果が正しいかどうかを判定するためのテスト仕様を仕様から導出し、テスト仕様に基づいてテストを生成することにより、効率的なテストを実現することが有効な対策です。

テストで利用するためには、どのような仕様を記述すればよいのでしょうか。形式仕様の一種である VDM では、関数や手続きに事前条件、事後条件を定義します。事前条件および事後条件は、仕様を表現した数理モデルの状態に関する特定の条件を与えます。これらにより、実行直前に事前条件が成り立っている場合、対象となる関数や手続きが実行されれば、その直後に事後条件が成り立つという関係が保証されます。

このとき、対象となる仕様の事前条件および事後条件が、実装の事前条件および事後条件として成り立たなくてはなりません。したがって、仕様の事前条件および事後条件から、実行するテストケースおよびそのテストデータを得ることができます。

```
private 上昇開始 : () ==> ()
上昇開始() ==
 if 1階動作ボタン.入力済み？()
   then (かご := <上昇中>; 1階動作ボタン.処理完了())
pre 動作モード = <通常> and かご = <1階> and 1階扉 = <閉> and 重量センサ =
<OK>
post かご = <2階>;

private 2階で停止 : () ==> ()
2階で停止() ==
 (
 かご := <2階>;
 動作ボタンを有効にする(<2階>, 2階動作ボタン, 2階扉);
 )
pre 動作モード = <通常> and (かご = <上昇中> or かご = <2階>)
post かご = <2階>;

private 下降開始 : () ==> ()
下降開始() ==
 if 2階動作ボタン.入力済み？()
   then (かご := <下降中>; 2階動作ボタン.処理完了())
pre 動作モード = <通常> and かご = <2階> and 2階扉 = <閉> and 重量センサ =
<OK>
post かご = <1階>;
```

図 3 - 17: 事前条件・事後条件の例（抜粋）

図 3 - 17 の pre 部が事前条件、 post 部が事後条件になります。同様に、変数に対して不変条件を定義することができます。不変条件が守られているかどうかを常に検査するのは、実行効率を低下させてしまいますが、実際は変更される可能性がある部分で検証すれば十分です。

このような機械的な変換を行わない場合でも、さまざまな場合をカバーしたテストを設計する際には、厳密な仕様が役立ちます。特に有用なのが、網羅的な場合からのテストケース選択とテスト結果の妥当性確認です[10]。

実用的な問題では網羅的なテスト数が膨大になりますので、モデル検査のようにすべてのテストを行うことは、現在の計算機の能力をもってしても困難です。したがって、一部が重複しているテストケースや冗長なテストケースを除くことでテストケースを最適化しなければなりません。適切にテストケースの選択を行うためには、属人的な知識に依存せずに対象を理解できるような厳密な仕様が役立ちます。

また、テスト結果についても、仕様から導かれるテスト結果とプログラムのテスト結果を突き合わせることで、不具合を見つけることができます。このとき、大規模なソフトウェアになればなるほど、計算機による機械的な突合せが有用になります。厳密な仕様記述による仕様アニメーション、さらに実行可能な形式仕様記述は、このような目的に適しています。

現代的なプログラミング言語にもこうした機能を備えたものがあり、表明あるいはアサーションと呼ばれています。表明はコードの実行中に満たすべき状態を記述するもので、実行時にその条件を満たしているかどうかが検査されます。仕様から導かれる不変条件や事前条件、事後条件を表明として記述することで、実装と仕様を結び付けることができるようになるのです。

[9] Matyas, P., Glover, S. and Duvall, A.: Continuous Integration: Improving Software Quality and Reducing Risk, Addison-Wesley, 2007.

[10] 幡山 五郎, **自動改札機ソフトウェアの品質向上の取組み** ―厳密な仕様、もらさないテストを目指して―, ソフトウェア品質シンポジウム 2012, セッション C2, 2012.

9. 厳密な仕様と保守作業

実行可能なソフトウェアが完成したからといって、仕様の役割は終わりではありません。運用や保守といった工程でも仕様は活用できるのです。

利用者の要求を完全に満たすソフトウェアは、そうあるものではありません。実際には、利用者の要求に優先順位を付け、開発期間や費用を勘案して適切な仕様を決めることになります。

市場投入のタイミングを重視し、いったんリリースしてから、優先度の低い要求を追加したり、不具合の修正をしたりするという方法も増えてきています。このような状況は、問題なく動作しているソフトウェアにおいてさえ、常に不足があるとみなすこともできて、保守作業の重要性を示しています。

保守作業はソフトウェアライフサイクルにおいて 60% から 80% のコストを占めると言われています[11]。さらに、保守作業の 60% は 保守対象となるソフトウェアの構成や機能を読み取り、仕様を理解することであるという報告もあります[12]。

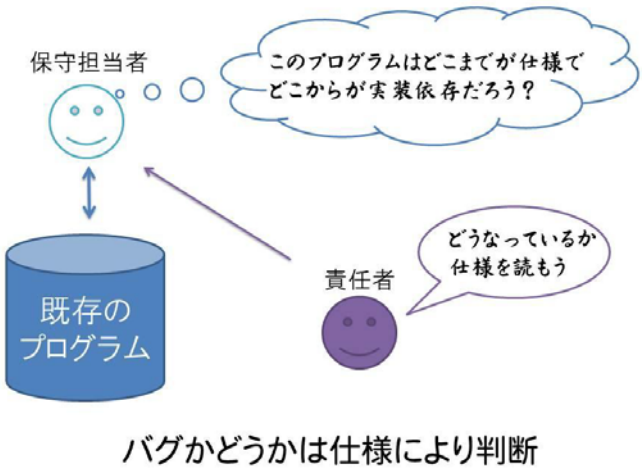


図 3 - 18: 仕様には関係者の合意を記録しよう

ソフトウェア保守の基準は、開発段階で用いた文書群になります。しかし、現実にはこれらの文書の不備や実装との乖離がこのような保守作業の困難をもたらしています。ソフトウェア開発の期間に、要求や実装条件の変更によって仕様が修正されることはしばしばあり、文書とコードの不一致の主な原因となっているのです。つまり、ソフトウェアのライフサイクル全体におけるコストを下げるためには、プログラム作成の出発点としてのためだけに仕様を書くのではなく、仕様が実装と一致していることが保証され、かつ人間にとって理解しやすい仕様であることが必要です。

開発者が仕様と実装が一致していると信じられるためには、仕様が常に一箇所で管理され、最新かつ検証された状態になっていなければなりません。なぜなら、個々の開発者がそれぞれの解釈で内部仕様を作っていたり、そもそも参照している仕様が異なっていたりしていては、保守作業の拠り所が失われるからです。

厳密な仕様を利用し、仕様の変更の度に検証するようにしていれば、一時的に内部矛盾を生じたり、意図しない誤りが入り込んだりしても、すぐにそれを検出し、一貫性を取り戻すことができます。これは、厳密な仕様があれば、仕様変更の影響範囲を見極めやすく、設計や実装との対応付けもしやすいからです。せっかく見つけた不具合も、最新版では仕様が変わっているかもしれないとなると、その確認にかかる時間や手間も見過ごせません。こうした手戻りにどの程度のコストがかかっているか定量的に測ってみるのは、ソフトウェア開発の定量的な効率改善に役立つでしょう。

本章の例題を例に考えてみましょう。 3.2 節のモデルでは、図 3 - 19 のような扉の開け閉めを各階の扉単位で扱うのではなく、図 3 - 20 のようにこのエレベータの扉に共通の操作として定義しています。これにより、エレベータの動作階数が増えても対応しやすくなっています。

```
public 1階扉を開ける : () ==> ()
1階扉を開ける() == (
  1階扉 := <開>;
  動作ボタンを無効にする(<1階>, 1階動作ボタン, 1階扉)
  )
pre (動作モード = <通常> and かご = <1階>) or 動作モード = <緊急>;
```

図 3 - 19: 個別の扉モデル

```
public 扉を開ける :『階』 ==> ()
扉を開ける(指定階) ==
  cases 指定階:
  <1階> -> (1階扉 := <開>; 動作ボタンを無効にする(<1階>, 1階動作ボタン, 1階扉)),
  <2階> -> (2階扉 := <開>; 動作ボタンを無効にする(<2階>, 2階動作ボタン, 2階扉)),
  others -> skip
  end
pre (動作モード = <通常> and かご = 指定階) or 動作モード = <緊急>;
```

図 3 - 20: 共通の扉モデル

しかし、これらの仕様はどちらも正しいものであり、また、どちらかが優れているということもすぐには言えません。図 3 - 20 には同じものを拡張しやすいという利点がありますが、個別の扉についての修正が多いのであれば、図 3 - 19 の方針が対応しやすいかもしれません。製品化されたシステムがどのように使われ、どのように保守されていくのかを事前に見極めるのはなかなか困難ですが、対象の問題領域をよく知り、仕様記述の経験を積むことでより適した仕様へ近づくことができるでしょう。

逆に言えば、**形式仕様であっても、そこには書き手のあるいは顧客の意思が反映されます。**未熟な仕様記述者が厳密に仕様を書こうとしても、よい仕様になることは難しいのです。つまり、厳密な仕様を書く時には、個別の文章や行間ではなく、より高いレベルの意思や考え方が反映されるのです。保守工程でも、仕様に込められた開発時の関係者の合意と、その先の拡張や移植についての考え方は（もしあれば）大いに参考になるでしょう。

もう一つ、保守の観点からもできるだけ紙に印刷した仕様書を参照するのではなく、電子的に参照できるようにすることをお勧めします。形式仕様を利用すれば、ツールにより仕様を検証することができ、少なくとも利用している数理モデル上で矛盾した記述がないことは保証できます。修正や変更が即時に反映され、いつでも頼りになる最新版を参照できるようになっていれば、間違いなく開発の効率は上がるはずです。

それでは、理解しやすい仕様とはどのようなものでしょうか。一つは、実世界における対象となる問題における概念と、それに対応する計算機システム中で表現された概念のギャップを小さくすることです。

たとえば、保守対象のシステムで用いられている専門用語が、そのまま識別子としてソフトウェア中で利用できれば、利用者の理解している概念や手順と、ソフトウェアとして実現された概念やアルゴリズムの対応付けが一目で分かり、仕様の理解が容易になります。

厳密な仕様の価値は、こうしたソフトウェアの満たすべき性質の定義から曖昧さを取り除くことです。この結果、ソフトウェアで実現されようとしている概念やアルゴリズム

が、利用者の専門知識や理解とあっているかという判断を、顧客も開発者も判断可能になります。当然、仕様と実装の対応付けも自然言語による仕様と比べて、はるかに容易になりますので、プログラムの理解にも役立ちます。

このとき、一つの用語が複数の意味で使われていると、そこで用語の解釈に曖昧さが生まれてしまい、せっかく厳密な仕様を書いても分かりにくくなってしまいます。このような場合は、元の要求を整理し、それぞれの用語を分離するべきです。

開発するソフトウェアがある規模以上になる場合、こうした対応付けの確認は、形式仕様を利用して計算機にさせた方が簡単になります。各形式手法ならびにツールにおける日本語の識別子への対応状況については、第５章をご覧ください。

[11] Gerudo Canfora and Aniello Cimitile, Software maintenance, in Handbook of Software Engineering and Knowledge Engineering, Shi Kuo Chang(ed.), pp. 91-120, World Scientific, 2001.

[12] Thomas A. Corbi, Program understanding: challenge for the 1990's, IBM Systems Journal, Vol.28, No.2, pp.294-306, 1989.

10. 厳密な仕様と派生開発

最近のソフトウェア開発現場では、一から作るような新規開発は滅多にみられません。多くの場合は、既存のソースコードを修正して新しい機能を追加したり、別の製品やシステムで稼働しているソフトウェアを部分的に流用したり、異なるミドルウェアへアプリケーションを移植したりといった再利用が行われています。

その理由として、プログラム言語や OS の寿命が長くなったこと、ソフトウェア開発期間の短縮が求められていること、ソフトウェアの高機能化に伴って開発規模が増大し、新規に開発するリスクが高くなったことなどが挙げられます。かつては、プログラムのライブラリとしての再利用が中心でしたが、近年は仕様の再利用についても実用化が進んでいます。

派生開発に固有の事情として、その多くが短納期であることが挙げられます。納期が短いために、必要な工程や中間成果物が省略されがちであり、派生を繰り返すうちに、ドキュメントとコードが一致しなくなり、派生元ソフトウェアを理解するにはプログラムを解析しなくてはならなくなって、結局のところ期待されるような動作品質のソフトウェアの再利用が困難になってしまいます。

「情報ハブ」としての厳密な仕様は、このような状況を防止するために役立ちます。それは、**曖昧さのない厳密な仕様であることで、属人的な知識に依存せず、誰もが同じ理解に達することが可能**だからです。

現在の派生開発で起きている代表的な問題点として、ソフトウェアのデグレードがあります。デグレードとは、ソフトウェアの変更が、変更していないはずの部分に予期しない影響を与え、変更前まで動いていた機能の品質が低下することを指します。派生開発におけるデグレードを防ぐためには、仕様あるいはプログラム変更の影響範囲を特定しなければなりません。これまで述べてきたように、厳密な仕様記述とその検証は、仕様における影響範囲を特定する有力な手段です。

一方、コードに関しては回帰テストがデグレードを防ぐために有効な手法です。形式仕様を陽仕様で与えれば、仕様についても回帰テストを行うことができます[13]。

さらに、実装が仕様を満たしていれば、抽象度の違いを除いてコードのテストも仕様のテスト結果と一致します。したがって、陽仕様について正しいと確信できるテストケースを作成できれば、実装の回帰テストに共通のテストケースを用いることができます。

派生開発が特に多い分野では、より積極的に派生開発を前提とした開発法を採用するという方法もあります。ソフトウェア開発における可変性を分析する手法としてソフトウェアプロダクトライン(Software Product Line Engineering, SPLE) があります[14]。

SPLE は、適用する製品群に対して有効な資産を抽出するために、フィーチャー分析を行います。フィーチャー分析は、製品がどのような特性を持っているのかを体系的に分析し、製品群の共通部分と書く製品に固有の可変部分に分類します。フィーチャーという抽象度の高い要素と、対応する設計や実装を一体として管理し、共通部分の再利用

性を向上させます。実開発への応用事例も多くあり、開発プロセスや社内環境を整え、適切なスコーピングを行えば、短期間で効果が上がる場合もあることが報告されています[15]。

SPLE は、共通の参照アーキテクチャを使用する製品群に対して、有効であることが知られています。厳密な仕様を利用すれば、設計上の決定に関する複数の選択肢およびそれらの適用条件も厳密に記述可能です。選択肢の適用条件が明示されることから、それらの依存関係を明確にすることが可能になり、機械的な検証を行うための設計情報として活用することが可能となるのです。

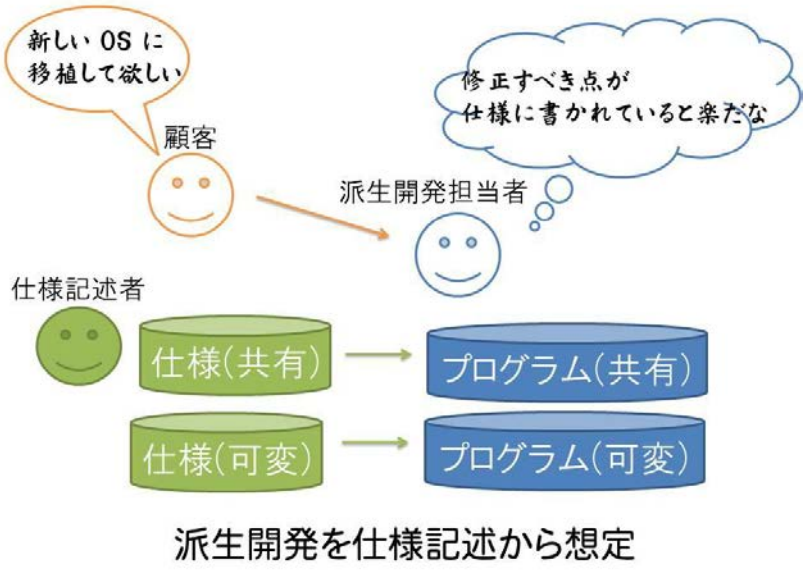


図 3 - 21: 派生開発も予測しよう

最後に、実装と仕様がきちんと対応付けられて更新されていれば、チームへ新しいメンバが入ってきたときにもすんなりと対応できるはずです。適切なドキュメントがない場合、ソフトウェアを理解するためにはプログラムを解析するしかありませんが、それではどこが仕様に基づく部分で、どこが最適化や実装制約による製品固有の部分かは分からず、どんなに優秀な人でも学習に時間がかかってしまいます。特別な教育用ドキュメ

ントを作るよりも、実際に使われているプログラムの仕様書を更新する方が効率がよいのではないでしょうか。組織的な観点では、こうした教育コストも考慮しなくてはなりません。

厳密な仕様を書こうとして対象の本質を考察することは、個々の技術者の視点を広げ、ソフトウェア開発能力を向上させます。とにかく短期間で納品するばかりでなく、長期的な効率改善への投資は無駄ではないはずです。ただし、このようなリスクをとることは、プロジェクト単位では難しい部分ですので、経営的な観点による判断による支援が望まれます。

本章の例題からの派生開発にはどのようなものが考えられるでしょうか。これは、読者の皆さまへの自由課題といたしましょう。

たとえば 2 台の連携はどのようにすればよいでしょうか。すぐに思い浮かぶ 3 階建てへの応用は意外に難しいです。時間のあるときに、チャレンジしてみてください。

[13] 佐原 伸, VDM++関数型回帰テスト支援ライブラリ, ソフトウェア・シンポジウム 2005, 2005.

[14] Paul Clements and Linda Northrop, Software Product Lines: Practices and Patterns (3rd Ed.), Addison-Wesley Longman, 2001.

[15] 大塚 潤, 河原畑 光一, 岩崎 孝司, 内場 誠, 中西 恒夫, 久住 憲嗣, 福田 晃, 大規模移動体ネットワーク機器ファームウェア開発へのソフトウェアプロダクトライン適用事例, 組込みシステムシンポジウム 2011 予行集, pp.26-1-26-10, 2011.

この章のまとめ

本章では、ソフトウェア開発における「情報ハブ」としての厳密な仕様記述に注目し、各工程において厳密な仕様がどのように役立つかを述べてきました。幅広い活用法を挙げてきましたので、ここで簡単にまとめておきましょう。

- 厳密な仕様はコミュニケーションの基盤となる

厳密な仕様は、書き手の意図を正確に表現します。読み手にとっても、書き手の意図を正確に理解しやすくなります。このため、仕様に関する誤解が生じる可能性は小さくなります。明確な仕様は開発の原点となるだけでなく、ソフトウェアの詳細化を進めていく際の指針にもなります。仕様を「情報ハブ」として活用できるようにするためには、後工程で問題になりそうな点について、関係者でコミュニケーションを図り、積極的に合意をとるようにするのがよいでしょう。

- 厳密な仕様は適切なモデル化である

  厳密な仕様は、統一された視点に基づいて対象を抽象化するモデル化です。したがって、モデル化する視点は対象を特徴づけている性質を表現できるものでなければなりません。抽象化によって、一部を捨象し対象の本質を明らかにすることが「情報ハブ」としての合意に近づく早道なのです。

  モデル化するには、形式仕様を例にとっても、多様な手法の中から適切な手法を選ばなくてはなりません。一般的なソフトウェア開発では、一階述語論理に基づく VDM や B-method 、あるいはプロセス代数に基づく Promela/SPIN や FSP/LTSA がよいでしょう。

- 仕様の一般解はない

  仕様は、関係者間のソフトウェア開発における合意を表すものです。仕様は合意が必要なだけ書き、合意が不要な部分は書かないのが原則です。つまり、仕様をどこまで書くかは単独で判断できるわけではありません。環境や外部要因も関係しますし、関係者の知識や経験値など属人性もあります。しかし、関係者の入れ替わりなどを想定すれば、できるだけ仕様を明示するのが望ましいでしょう。どのようなプロジェクトもいつかは終わりますが、製品はその後も使われるのですから。

- 仕様の正しさをどうやって確かめるか

  仕様の妥当性確認と正当性検証には、それぞれ適した手法があります。前者はレビューや仕様アニメーション、後者は数理的証明やモデル検査が挙げられま

す。要求の正しさは計算機には分かりませんので、妥当性確認は人間が行う必要があります。正当性検証は、数理モデルなどを用いればその論理体系上で計算機による補助がある程度可能です。

以下に、本章で用いた術語の一部の定義を示します。これらの定義は IEEE 標準によります[16]。

- 正当性検証は「各開発工程の成果物がその工程の開始時に与えられた制約を満たしているかどうかを判定するために，システムまたはコンポーネントを評価するプロセス」と定義されます。
- 妥当性確認は「各開発工程の間あるいは終了時に、要求を満たしているかどうかを判定するために、システムまたはコンポーネントを評価するプロセス」と定義されます。

正当性検証と妥当性確認の違いは、判定が工程の開始時の制約について行われるか、工程の途中あるいは終了時に要求に対して行われるかという点です。

- 要求は「利用者が課題を解決したり、目的を達成したりするために必要とする条件あるいは能力に関する表明」です。

つまり、第 1 章の課題・仕様・設計の関係で言えば、仕様は要求および前提となる事実を明確に表したものになります。

[16] IEEE std. 610.12-2012, IEEE Standard Glossary of Software Engineering Terminology

章末コラム [手法と手法の接着剤？]

数学と聞いて私が真っ先に思い起こすのは、公理や定義、ある定まった記述方法で共通理解を形成することができるということです。先ほどのトピックで、数学という言葉を

出さない方がよい、と言っておきながら、数学を引き合いに出したのには理由があります。もし、ソフトウェアの仕様記述が数学と同じ効果を発揮するのであれば、どこの会社・国の開発者、どんな年代の開発者であっても、仕様を理解することができるという大きな効果が得られると思うからです。このことは、現状の開発環境（オフショア開発、技術者派遣等）を考えると、大変に大きな効果ではないでしょうか。
そして、数学は、何も数式だけで形成されているわけではありません。自然言語を用いて頭の中で検討している内容を自然言語と数式を合わせながら巧みに表現します。さらに、必要に応じて図やグラフを描くこともあります。面白いのは、図と数式の間にも関係性が明確に成り立っているということです。このようなことを思い浮かべると、現状提唱されている形式仕様記述や UML などの図式化言語は、お互いに力を合わせることができるという思いが強まります。
特に、形式仕様記述が数学を基礎にしているというのであれば、形式仕様記述は、現状までに提唱されてきているソフトウェア開発に関わる各種技術をまとめ上げる力を秘めているのではないでしょうか。実際に、ソフトウェアの周辺の仕様記述に関する技術を思い浮かべてみると、双方の歩み寄りが始まっていると思います。第 3 章で述べられているように、どれか一つのみ手法を適用すれば完璧であることはなく、むしろ各手法をうまく組み合わせて利用できるようになることが、重要であるのです。
本書を通じて、仕様を大切に扱いたい、と思うようになるほど、各手法間でお互いの弱点を指摘し合っている場合ではないと感じます。それこそ、ソフトウェア開発ということを抽象的にとらえ直し、全体を包括する理論を構築できると面白いなと、想像してしまうのです。

[第３章執筆者のコメント]

「コンピュータは電気で動いているのではない、数学で動いているのだ」というのは、誰の言葉だったでしょうか。バベッジの解析機関は機械式でした（動きませんでしたが）。何年か後には、光コンピュータや量子コンピュータが広まっているかもしれません。それでも、コンピュータは数学で動いていることでしょう。数学的な性質は普遍性があり、さまざまな場所でさまざまな状況で表れます。論理はその典型といえます。
厳密な仕様は、ものごとの本質を明らかにし、お互いの関係をはっきりさせます。数学は手段にすぎませんが、コンピュータとの相性は抜群によいのです。それが、情報ハブとして、さまざまな手法の中心に適していると感じられる理由でしょう。
一方で、現実を見失ってはいけません。厳密な仕様で検証できるのはあくまで仕様の

話であり、コンピュータを間違いなく動かすためには、実装のテストや効率に関する工夫が重要です。そうした観点からもさまざまな手法の組み合わせは有用なのではないでしょうか。

第４章 実務者による開発現場における形式仕様記述の実践に向けて

この章について

本章では、実務者が開発現場における形式仕様記述の実践に向けて検討するとよいことがらを紹介します。過去の課題や経験と照らし合わせて、厳密な仕様の記述やシステム開発の全般に関する、具体的な検討に役立ててください。

1. この章を読む方へのヒント

この章では、若い読者の皆さんが携わっている開発、とくに皆さんの開発における仕様はどうあるべきなのか、仕様書に限らず、どのようなプログラムや文書をどのように組織的に書いたり、読んだり、確認したり、残したりしていくべきなのか、そもそも仕様や文書やプログラムやソフトウェアは、一般的にではなく、どのようにあるべきなのか、について、読者の皆さんのそれぞれの事情に合わせて考えていく視点の一端を提供します。ソフトウェアの開発や厳密な仕様の記述、文書の作成に対することがらを自分なりに整理したり、現在取り組んでいることがらとの隔たり（ギャップ）について分析したり、考えをまとめたりするときの参考にしてください。

一方で、書きたい仕様やその書き方、検証方法が明確になっている場合や、仕様の記述に限らず、解決したい課題が明確である場合は、すぐに具体的に仕様を書いてみたり、課題の解決に向けた取り組みを直接的に考えたりすればよいでしょう。そういった方々は、この章を読む必要はないのかもしれません。

ここでは主に、形式仕様記述手法を用いてソフトウェアの何らかしらの機能の仕様を記述することを想定します。形式仕様記述に関する議論において、必ず投げかけられる質問、疑問の一つは、形式仕様記述手法は、ある種のデータ構造や機能仕様を記述するのに適していることは分かるが、それでははたして非機能仕様の記述に対しては有用なのか否か、というものです。この質問に対する回答としてはさまざまなものがあると思いますし、また、質問に対してただ一つの回答があるべきだとも思いませんが、ここで一つの考え方を提供しておくと、形式仕様記述とは、ある見方においては、仕様の決定と記述と維持に向けた、思索やコミュニケーション、レビュー、テスト、そして何よりも文書の読み書きの積み重ねを、機能仕様書として厳密に反映させていくための手法であり、かつ、その他の、機能仕様の検討、記述以外の難しいことがらに意識と時間を向けていくための手法である、という言い方もできるということです。厳密に書くことができることがらは形式仕様の記法に則って書き、さらにたとえば、レビューや、ツールの利用、テストにより仕様の品質を向上させ、確実に変更やその管理や回帰テスト等を行

うことにより仕様の品質を維持しつつ、その他の、定めたり、厳密に書いたり、検証したりすることがより難しい非機能仕様の策定に時間を振り分けるとよい、ということです。

あなたの開発現場では、機能仕様が曖昧であること、機能仕様書を曖昧に記述していることによる課題が山積していますか。もしそうなのであれば、形式仕様記述手法を利用することにより、早期に問題の解決を図ることができるようになるかもしれません。しかし、もちろん、これによりすべての課題を解決することはできないでしょう。もしそうではないのであれば、課題の解決に最適な方法を探りましょう。課題の解決方法の一つは形式手法かもしれませんし、そうではないかもしれません。それから、必ずしも一つの手法に頼ることもないのだ、というよりもそれは不自然なことなのだ、ということも憶えておいてください。

繰り返しになりますが、ここでは主に、形式仕様記述手法を用いて機能仕様を記述することを想定しますが、そうではない場合、とくに自然言語を用いて仕様を記述する際にも役に立つと思います。最終的に、形式仕様記述手法を使わなくとも、従来の課題を解決する可能性がある手法について知ることができれば、現場における具体的なさまざまな課題がどのように解決できる可能性があるのかを知る、考えることができるようになります。このことにより、自然言語を用いて仕様を記述するときに、どのような課題をどのように解決できる可能性があるのか、ないのか、困難であるのか、ということについての見通しを持つことができるようになります。

さらに、そもそも、どのような開発であっても、すべての仕様書、文書を形式仕様記述言語で記述するわけでもないでしょう。形式仕様記述手法を用いるのだとしても、同時に自然言語を用いた文書も作成することになります。ですから、形式仕様記述手法を利用するのか、あるいはしないのかに関わらず、自然言語で書く仕様書やその他の文書の記述とその品質の向上をどのように行うのか、という議論を避けて通ることはできません。

いずれにせよ、形式仕様記述について知り、その直接的な、そして間接的な効果について考察することは、開発工程全体で作成する文書群や、開発工程全体について考え直す、よいきっかけになることでしょう。

2. 組織や成果物の課題の分析

仕様の記述をどのように行うのかを考えていくときに、まず行うべきことは、(1) プロジェクトの目的の見すえ、仕様記述をどうすべきなのかを考えること、同時に、仕様記述に限らず、開発や運用、保守の工程全体について考えていくことと、(2) 現状や過去の良い点（良かった点）と悪い点（悪かった点)、課題の整理をすることです。

このときに、仕様はチームにおける開発、品質確保、その後の運用、保守等の、すべての活動の起点となるものですから、できるだけ多くの関係者の意見を集め、議論をしていくことが大切です。

いま、できるだけ多くの関係者の意見を集めて議論、と簡単に書きましたが、一般的に、多くの人々が読み書きをする仕様に関係する組織や人はとても多く、立場も多岐にわたることから、また、仕様に関しては、それぞれの立場で一家言あるため、意見の一致には多くの困難がともなうかもしれません。これは、仕様の決定やレビューが難しいのと同じことです。

形式仕様記述手法は、あえて言えば手法としてはまったく目新しいものではなく、歴史があるものですし、誤解を恐れずに言えば、ソフトウェア開発技術者にとってまったく難しいものではありません。しかし、たとえばあるテストチームのごく限られた人たち、あるいは、個人が新たに用いるテスト設計の技法とは、関係者の数や関係者のそれぞれの仕事への影響度が異なります（この例であることに、手法や技法の導入の影響範囲以外のことに関する特別な意図はありません)。だからこそ、開発や運用、保守の全体に仕様に関する課題が横たわっている場合、形式仕様記述手法が仕様に関する課題を解決することにより、開発全体と各工程の多くの課題を同時に解決することができるかもしれないという可能性が期待できるのです。

繰り返しになりますが、形式仕様記述にかぎらず、仕様に関する議論、仕様そのものに関する議論は難しいものです。このことと、形式仕様記述手法特有の導入の困難性については、それぞれ分けて考えるようにしましょう。なんでも手法や技法、そしてツールの問題にしてはなりません。

課題の棚卸しに関しては、既に、組織やチームがある場合は、その組織における過去の課題を整理し、対応案を考えていくとよいでしょう。とくに、これから開発するソフトウェアと何らかしらの意味合いで同じようなものを開発してきた場合、過去の具体的な課題を検討することが、仕様の検討に限らず、非常に役に立ちます。

一方で、既存の組織やチームがなく、新規の組織による、新規の開発の場合においても、チームに過去の課題の累積はなくとも、チームのメンバはまったくの初心者ばかりではないでしょうから、それぞれの過去の経験を持ち寄り、新しい組織で新しい開発を行うにあたり、一般的な課題に対してどのように立ち向かっていくべきなのか、そのときにどのように仕様を記述していくべきなのかということについて考えていきましょう。

また、仕様について考えるのと同時に、仕様のことだけについて検討するのではなく、繰り返しになりますが、仕様はチームにおける開発、品質確保、その後の運用、保守等の、すべての活動の起点の一つとなるものですから、関係する活動、すなわち、要求分析、設計、実装、テスト、運用、保守等々のすべての活動について考慮し、活動の連動について思いを巡らせ、いま検討しているこの開発において仕様がどうあるべきか、ということについて考えていくことが大切です。

仕様に関する検討は、開発や運用・保守のすべてのことを検討していくことがらにつながり、そして、すべてのことがらを考えていこうと指向していくことが、仕様について考えていくことにつながります。そしてその結果、多くの時間と費用を掛けてまで多くの情報を仕様として厳密に書き表し、レビューやテストも含めて確認した方が良いのか、あるいは、極端な場合かもしれませんが、仕様書は一切書かなくてよいのか、などといった、自分たちなりの結論に達することになるのでしょう。

3. 具体的な課題の整理

それでは具体的には、どのような観点で良い点（良かった点）と改善すべき悪い点（悪かった点、課題）を整理していったらよいのでしょうか。

ここでは、(1) 課題の整理、(2) 欠陥の整理、(3) プロセスの整理に分けて考えていきたいと思います。

(1) プロジェクトの課題の整理

課題の整理とは、開発や品質確保、運用、保守等の、すべての活動の全体をとらえ、継続していきたいこと、より良くしていきたいことを考えることです。一般的には、過去のプロジェクト全体の課題リストを紐解いていくとよいでしょう。このときに、仕様に関する課題はもちろんのこと、仕様に直接関係しなさそうな課題であっても、もし仕様を厳密に書くことができていたら解決できるかもしれない、という視点で課題を一つひとつ点検し、仕様に関係する課題を洗い出していきましょう。

(2) 欠陥の整理

その上で、過去の開発における欠陥の整理を行いましょう。この欠陥の整理は、一般的には、過去のプロジェクトにおける欠陥や、欠陥の候補となるリストを分析することになるでしょう。欠陥の候補とは、最終的には欠陥とは判断しなかったが、開発の過程で欠陥の候補になったもののことですが、これも整理の対象に加えた方が良い理由は、たとえばテストによって欠陥の候補が挙げられ、その後の切り分けの結果、欠陥ではないことが分かったが、欠陥の候補となってしまった原因が、仕様を曖昧に記述していた、あるいは、欠陥であるかどうかの切り分けに時間と労力が掛かった、というようなことが考えられるからです。

欠陥や、欠陥の候補の分析にあたっては、その真の原因を考えていく必要があります。しかし同時に、追究しすぎては、何でも上流工程や利害関係者間の調整、考慮の不足、技術者の能力や教育等の問題になってしまうかもしれませんから、注意しなければなりません（原因の追及が悪いわけではありません）。また、特定のサブチームや開発者をみだりに批判することにもつながりかねないですから、気を付けましょう。

以上の点に留意しながら、チームの活動として仕様が原因となっている欠陥を洗い出し、どのような記述、たとえば曖昧な仕様の記述、あるいは、仕様が書かれていなかったことが、従来のどのような課題の原因になっていたのかを整理をしていきましょう。

(3) プロセスの整理

課題や欠陥の整理と同時に、開発プロセスに関する検討も行いましょう。とくに、開発の手順全体における情報とその流れの整理、運用、保守や、派生開発に関する課題の整理を行いましょう。ここでの目的は、情報とその流れを見直すことにより、開発工程の全体に関わる重要な情報や文書を浮かび上がらせる、重要な情報が各工程でどのような意味をもっているのか、各工程間でどのようなつながりをもっているのか、つながり（トレーサビリティ）を確認する必要があるのであれば追跡できるのか、新たに整理した方が良い情報がないかどうか検討することです。

以上のような整理を積み重ねて、仕様に起因する問題がどれくらいあるのか、仕様が明確だったら円滑になることがどれくらいあるのかを整理しましょう。これにより、曖昧な仕様に起因する問題解決にどれくらいの期間と予算が割かれているのか、手戻りがどれくらいあるのか、どれくらい無駄なコストが掛かっているのかが分かるかもしれません。仕様の策定には、仕様について複数の案を考えたり、関係者間で調整をしたりする必要があること、この場合、仕様策定者間で同期を取って一貫した仕様を記述していく必要があること、さらに言えば、仕様策定は既に頭の中にある情報を記述するだけではないので、仕様策定に掛かる時間や費用の見積もりは難しいのですが（まったく新規のプログラムの設計や実装を行う場合も同様です。仕様の策定が特別なのではありません）、厳密な仕様の記述にかかるであろう時間や費用が、これまでに曖昧な仕様記述に起因して発生していた無駄なコストよりも大きいのであれば、厳密な仕様の記述を行うことは、品質を向上させたり、開発の期間を短くしたり、予算を削減したりすることができるかもしれないと考えることができるかもしれません。一方で、仕様の記述に大きな問題がないのにも関わらず、厳密な仕様記述を追究していくことは、コストという視点においては無意味なことです。厳密な仕様記述は手段であり、その手段の上位に、無意味なことはやめて、より重大な課題の解決にあたるべきというセオリーがあることを忘れてはなりません。

また、この文章の読者として想定している若い方が考えるのには少し早いかもしれませんが、現在はソフトウェアや開発に問題がなくても、今後のリスクも考慮した方が良い

こともあります。将来、開発したものに対して、どのような責任、説明責任が生じるのか、ということについても考えておくべきです。問い合わせに対して確実に答えようとしたときに、プログラムのソースコードが基準となっていては、そのプログラムをどのようなつもりで作ったのか、将来の派生や拡張のための開発に耐えられるのか、担当者が変わったときに問題は生じないか、また第三者による仕様やソフトウェアの客観的な評価や認証が必要になることはないのか、ということについて適切に答えることができるかどうか頭の片隅で考えておく必要もあるかもしれません。

4. 課題について考えていくときの留意点

仕様について考えていくときに、まず、(1) 曖昧さを残さず仕様を決めて厳密に記述することと、(2) 仕様がなかなか決められないこと、仕様が変わってしまうこと、あるいは意志を持って仕様を変えていくこととは、独立したこととして認識する必要があります。

(1) の厳密な記述により、仕様書に書く仕様を決めることができること、仕様を厳密に記述することにより検討や議論が深まり、より厳密に仕様について考え、決定し、そして読み書きができるようになること、よりよいコミュニケーションができるようになること、これらのことにより、より良い仕様を検討できる可能性が増すでしょう。しかし、(2) の課題の本質を厳密な仕様記述で解決することはできません。記法とは別の解決方法を模索する必要があります。

一方で、(1) の活動において、仕様を決めることができない、従来は曖昧なままにしてきた仕様をいざ厳密に書こうとしたときに、あらためて仕様が決まっていないことに気が付く、決めようとしても決めることができない、ということがあるかもしれません。また、厳密な仕様記述により、仕様や開発領域（ドメイン）に関する組織的な再認識が促進され、あらためて仕様を見直すことができるようになる場合もあるかもしれません。その結果、仕様に関する議論が可能になり、それにより仕様が簡単に決まったり、あるいは、反対に仕様にあれやこれやの条件が複雑に込み入り、簡単には仕様を決めることができなくなったりするかもしれません。また、仕様に関する議論の結果、より良い仕

様について考えることができるようになったり、さらに、仕様に関するコミュニケーションが促進され、よいチームワークにつながったりしていくということも考えられます。以上のように、活動を活発にしていくことは、おおむねよい方向に向かうと思いますが、その是非を一概に言うことはできません。状況に応じて、関係者でよく話し合っていくしかないでしょう。従来、仕様について議論をしようとしても、議論の基盤になる仕様が存在せず、あるいは、存在していても曖昧な記述や日本語の問題に議論が終始し、建設的な話し合いを行うことができなかったということはないでしょうか。仕様書のレビューを行う時間の大半を、日本語の記述に関する指摘や議論に費やしていたということはないでしょうか。開発プロセスをよりよい方向に向かわせていくためには、状況に応じて、関係者でよく話し合っていくしかないでしょう。

厳密な仕様記述は、この章のはじめに述べたとおり、まず書くことができる仕様、とくに機能仕様を厳密に記述し、その後、仕様の記述よりも解決が困難な課題に立ち向かっていくためのものであるととらえることもできます。一方で、課題の先にあるものは、もはや形式仕様記述手法や文書の記述とは関係がない可能性があること、また、仕様を記述する以前のことに関しても、記述の方法論が課題ではない可能性があります。多くのことに関係する仕様の記述であるからこそ、以上のことに留意しておきましょう。

5. 記述に関する具体的な課題

このような課題の整理の結果、曖昧な仕様に起因するどのような問題が浮き彫りになるのでしょうか。あるいは、なりましたか。それとも、仕様の記述や仕様の品質とは関係がない問題が多いでしょうか。あるいは、多かったでしょうか。

仕様に関する問題について考えていくうえでの着眼点の一つは、「論理」とコミュニケーションです。現在、「論理」、「論理的」という言葉は、小中学校等における学校教育から、社会人に向けた書籍やセミナーにいたるまで、ありとあらゆるところでその重要性が叫ばれ、また、具体的なことがらが取り上げられていますが、その内容の多くは、書き言葉やコミュニケーションの道具としての話し言葉に関係することがらと、これらの言葉に関係する数学、論理学に関係することがらからなります。そして、ソフトウェ

アの仕様記述ということに関しても、その記述は言葉と言葉の関係に関することがらからなっていて、また、これらが密接に関係し合っているために問題を難しくしています。課題を分析するときには、用語や語法や文法を正しく使っているかなどの言葉の問題と、言葉から構成される条件文などが文脈に応じて適切に内容を指し示しているかという問題、つまり職場で時折使われる、いわば「国語」と「数学」（算数・論理学）の問題にできるだけ分離して整理するとよいかもしれません。

「国語」の問題とは、作文、とくに仕様書の記述の場合はいわゆる技術作文に関することがらです。論理的な文章とは、たとえば、書式や用語や前提知識が読者と合意されていて、文法や語法や表記が正しく、全体として読みやすく、内容に一貫性がある文章のことです。といって、論理的な文章の定義は一つではありません。どういった文章を論理的な文章として考え、開発現場で記述していくのか、ということについてチームで整理をしておくとよいでしょう。

従来の仕様書に関する課題の中に、「国語」やコミュニケーションに関する課題がどれくらいあり、また、その内訳はどのようなものだったでしょうか。これらの課題のうち、どのような課題を解決すべきなのかを検討し、解決に向けた具体的な施策を考えていく必要があります。

このときに、形式仕様記述手法を用いることにより、どのような課題がどのように解決できるのかを考えてみると、読者の皆さんの開発において、形式手法がどれくらいの効果を発揮するのかを想像できるようになるでしょう。たとえば、用語が未定義であったり、不統一であったりする問題が多い場合、形式仕様記述言語とツールを利用すれば、定義をしない用語は用いることができず、また型の確認まで行えるようになります。このような場合には、用語に関する問題を少なくしていくことができるようになります。たとえば、いわゆる用語の表記揺れに関する問題をなくすことができるようになります。また、たとえば日本語の文法に関する課題が多い場合も同様です。文法や意味が定まった仕様記述言語とツールの文法確認機能を用いることにより、未定義の文法を用いて仕様を記述することを避けることができます。従来、文書レビューにおいて、用語や日本語の文法に関する指摘や議論が多かったのであれば、形式仕様記述手法の利用により、時間や費用を抑えることができるようになるかもしれません。以上は一例に過ぎません

が、従来の課題として国語の問題が多く、また、記述する担当者によっても国語問題の傾向がことなる場合には、根本的な解決策は、各自の作文力を向上させる、チームで記述法を合わせる、あるいは、限界はありますが制約された自然言語を用いたり、型（テンプレート）にあてはめたり、ツールを用いたりするなどの対応が必要になります。しかし、現実にはツールや技術や工夫で課題を根本的に解決することはなかなか難しいですし、作文力の向上は一朝一夕に達成できるものではありません。

しかし、繰り返しになりますが、開発において仕様記述言語を用いるとしても、自然言語を用いた記述やコミュニケケーションがまったくなくなるわけではないですから、即座に解決できるものではないと心して、しかし、同時にあきらめることなく努力を継続していく必要があります。

また、国語の問題と同時に、コミュニケーションの問題はどれくらいあるのでしょうか。コミュニケーションの問題とは、たとえば、勘違い、確認の不足、追究の不足等により生じる、現場での具体的な問題のことです。また、各自が書いた日本語の稚拙さに対する羞恥心がコミュニケーションの阻害要因になっているということはなかったでしょうか。これは一例ですが、このようなコミュニケーションの問題を分析し、その背景が、曖昧で不完全な仕様をコミュニケーションの基盤としているために問題が発生しているのだとしたなら、厳密な仕様の読み書きをすることにより、これまでの課題の解決は可能かもしれません。

仕様策定者にとってのコミュニケーションにはどのようなものがあるのでしょうか。一つめは、仕様策定よりもいわゆる上流を担当する要求定義者、企画者等、さらにはプロジェクト外の利害関係者らとのコミュニケーションです。これらの人々とのやりとりに、形式仕様を直接的に持ち出すことはできないかもしれませんが、厳密に記述したことが、厳密な仕様が、確かなコミュニケーションの基盤となるでしょう。二つめは、仕様策定者自身の内省コミュニケーションです。複数人で仕様を策定する場合は、チーム内での仕様の策定に関するやりとりです。こちらも、仕様をチームで厳密に記述していくことが、仕様開発チーム内でのコミュニケーションズのよりどころになるでしょう。最後は、いわゆる後工程に携わる数多くの人々とのコミュニケーションです。仕様に基づいて行われる、設計、実装、テスト等のあらゆる活動を確かに行うためのコミュニケーション

です。仕様に基づいて行われる活動のすべてが、活動に従事する人々とのコミュニケーションが、確かな仕様によって、強固なものになることは、これまでも繰り返し述べてきたことです。

さらに、従来の仕様書の課題の中に、「数学」に関する課題がどれくらいあり、また、その内訳はどのようなものだったでしょうか。「数学」と言っても、私たちの日常の開発から遠く離れたものではありません。「または」であるとか「かつ」といった、プログラミングでも用いる命題論理を用いた複雑な条件を適切に表すことができていない、三段論法等のような基本的な論理の構造であっても明確に表現することができていない、自然言語で書かれていることを論理的に検討することが難しく、必ず成り立つとは限らない「裏」や「逆」、必ず成り立つ「対偶」等について十分に認識したうえで議論ができていない、たとえばデータベースを利用するときに必ず登場する概念である集合や写像の定義や操作が適切に表現できていない、集合に対する述語論理や、時間に関する時相論理が表せない、等の問題はないでしょうか。あるいは、これらの問題の表面的な表れ方から、その裏側にある、数学的な問題を認識することができているでしょうか。

上記の数学に関する知識は、すべて特別なものではなく、極端なことを言えば、すべてのソフトウェア開発技術者が一通りおさえておかなければならない基本的なことがらです。これらの数学的な問題が多い場合には、形式仕様記述手法を用いるとよいかもしれません。形式仕様記述言語を用いることにより、集合や列や写像としてデータやデータ同士の関係を定義しながら、命題論理や述語論理等の論理を厳密に、直接的に表すことが可能になり、論理的なことがらを簡単に表現したり、検証したり、伝えたり、私たちに必要な数学について見直したりすることができるようになります。

6. 仕様記述の目的の設定

前章までで述べてきた仕様記述や形式仕様記述の目的やその効果、本章における以上の考察を踏まえて、開発や運用、保守における仕様の位置付けや、仕様記述の目的を明らかにして、プロジェクトのチームメンバと合意しましょう。

仕様記述の目的はさまざまですが、ここでは、形式仕様記述の直接的な効果と、間接的な効果の一部を列挙しておきます。検討の参考にしてください。

直接的な効果

・厳密な仕様の策定と記述、記録、伝達
・ツールによる仕様記述の補助、仕様の品質の向上
・動作する厳密な仕様のテストによる品質の向上
・仕様に関するさまざまな側面における情報の抽出

間接的な効果

・属人性を排したチームでの客観的な記述
・コミュニケーションズの活性化
・開発領域（ドメイン）に関する再認識

まずは、いくつかの直接的な効果を得ることを目的として、形式仕様記述の目当てを考えていくとよいでしょう。また、直接的な効果を得ることはとても重要ですが、関係する間接的な効果についても、同時に検討しておくとよいでしょう。とくに解決したい従来の開発における課題が、間接的な効果と関係する問題であることもあるため、間接的な効果と、直接的な効果の連動性について整理をするとよいです。たとえば、従来、開発領域（ドメイン）に対する共通認識が不足していたという課題がある場合、属人性を排した網羅的で厳密な仕様を記述することにより、厳密な仕様がコミュニケーションの基盤になります。このことと、また、属人性を排することによりチームとして仕様を客観視することができるようになることとから、仕様に関する議論や派生した検討、記述、レビュー、検証が活性化し、より網羅的で厳密な仕様が記述できるようになります。そして同時に、開発領域に対する組織的な認識が深まっていく、というようなことが考えられます。以上は一例ですが、さまざまな開発工程や技術、その目的と効果は連動して相乗効果をもたらしますから、目的のそれぞれについて考えていくことも大切です。と

同時に、それぞれの関係性についても思いを巡らせていくことが、仕様記述の意義や、仕様の存在理由、開発の工程全体を考えていくうえで役に立つでしょう。

7. 仕様の記述方針の検討

厳密な仕様を記述するときには、記述する仕様の抽象度と、記述の方法を決定する必要があります。仕様は、前章までで述べたとおり、漠然とした要求と具体的な設計の間にあるものですが、それでは、どこまでを要求として表し、どこからを設計とするのかは、このときに仕様にどのような抽象度を持たせるのか、あるいは、具体性を持たせるのかによって決まります。そしてこの決定は、開発の目的や事情よって異なります。また、同時に、どれくらい厳密に表すのか、あるいは、曖昧さを残すのか、ということについても考える必要があります。

仕様を記述する際には、以上の二つの軸があり、抽象と曖昧、具体と厳密を分けて議論をしていく必要があります。このことを混同してしまうと、たとえば、仕様が曖昧である、という指摘に対して、仕様を具体化する、たとえば仕様を設計に近づける、という反応をしてしまい、結果として仕様が曖昧であるとは言われなくなるかもしれませんが、仕様と設計の境界がなくなり、仕様の策定や記述をしているのか、設計をしているのか、よく分からなくなってしまう、というようなことになりかねません。ですから、要求でもなく設計でもない仕様の抽象度を保って、厳密に記述する、ということが重要になります。まずは、抽象的な仕様を厳密に記述する、ということを心掛けましょう。このときに、厳密にしようとしすぎて、具体化しないように注意しましょう。

よく、仕様は「What」で設計は「How」である、などと言われます。設計やプログラミングにおいては、具体的な実行環境や開発環境、たとえば、コンピュータやオペレーティングシステムの種類であるとか、利用するフレームワーク、たとえばセキュリティを確保するための設計ガイドライン、データベースの種類、ネットワーク環境などについて検討し、制約を考慮して開発を行うのが一般的です。「How」に相当する設計やプログラミングには、要求獲得や仕様の策定の工程にはない、固有の事情が多くあります。このような事情や制約から離れて、システムやプログラムに「何を」させるのか、を

「What」として定めるのが仕様であり、これにより、システムやプログラムの役割や機能などを、設計や実装の都合から分離をして明確にするのです。
このことを言い換えると、実装に自由度を持たせるということになります。ある仕様を実現する際に、その方法については踏み込まずに、それはまた別に議論することにより、さまざまな可能性を残すということです。たとえば、ある計算を行うための仕様があるとして、その計算を行うために、愚直に計算を行なってもよいし、なんらかの数学的な工夫により処理速度を向上させてもよいし、複数のコンピュータで分散して計算を行うようにしてもよい、ということです。
一方で、既に設計や実装の環境が確定していて、そのことを前提として仕様を書きたい、という場合もあるでしょう。たとえば、開発に用いるフレームワークやデータベースが既に定まっていて、それらを用いてシステムを構築することが、要求にも深く関係しているような場合です。このような場合には、あえて仕様を抽象化せず、開発環境に合わせた要求の整理や仕様の記述を行う、という選択肢が必ずしも間違っているということではないのです。派生開発において、従来からある仕様の一部を厳密化したい、というような場合も同様でしょう。結局は、開発の事情や目的に合わせて、仕様の具体性を決めていく必要があります。

8. 仕様記述言語とツール

具体的な言語やツールの選定に関する議論は、別の章や本に譲るとして、ここでは、言語の仕様やツールの支援になにを期待するのか、について考えてみましょう。
たとえば、仕様記述言語 VDM-SL や VDM++ 言語、これらの言語を用いた仕様の記述や検証を支援するツールである VDMTools を用いる場合、(1) 仕様の構文検査、(2) 仕様の型検査、(3) 実行可能仕様の逐次実行とデバッグ支援、(4) 実行可能仕様のコードカバレッジ計測、(5) 実行可能仕様から C++ や Java 言語のプログラムの生成、(6) 証明課題の生成、(7) UML 描画ツールとの連携、(8) 仕様の清書等の機能を利用することができます。(1)、(2)、(6) の機能により、自然言語で記述した仕様では困難な仕様の検査や、制約はありますが、事前条件や事後条件からの証明課題の生成を行うことが

できます。さらに、動作する操作的な仕様を記述することにより、(1)、(2)、(6) の機能の利用に加えて、(3)、(4) の機能により、仕様のテスト、デバッグ、テストの網羅性の計測を行うことができます。また、(5) により、制限はありますが、動作する操作的な仕様からプログラムを自動生成することもできます。

仕様記述言語やツールのすべての仕様や機能を利用する必要はありません。というよりも、無目的にすべてを利用すべきではありません。プログラミングにおいても、プログラミング言語の仕様のすべてを利用したり、開発環境のすべての機能を利用したりすることはありません。そのことと同じです。

仕様記述言語の仕様をどこまでを利用するのか、どのようなスタイルで、どのような規約にのっとって仕様を記述するのかを考えるときには、記述する仕様を読む可能性があるすべての人々のことを考慮しましょう。仕様を読む人は、プログラムを読む人よりも多いかもしれません。人々は何を目的として仕様を読むのでしょうか。そのことを考えて、みだりに難しい文法や、奇抜な記法を使わないようにしましょう。また、仕様としての一貫性を保った仕様書群とする必要があります。このために、プログラミングと同様に、コーディングルールの検討や、静的コード解析ツールによる確認が有効です。

ツールの利用に際しては、(5) 実行可能仕様から C++ や Java 言語のプログラムの生成機能に期待しすぎないように注意する必要があります。たえず仕様の記述の目的に立ちかえり、仕様の第一の記述目的が、プログラムの自動生成ではないのであれば、仕様は仕様の記述目的に合わせた抽象度をもって読み書きしやすく記述することを目指しましょう。ツールから自動生成できるのは、仕様の抽象度と同様のプログラムであること、それは、プログラムが持つべき具体性を有していない可能性があることを再確認するとよいでしょう。

同様に、ツールの他の機能に関しても、仕様の位置付けや目的を再確認しながら適宜利用していくとよいでしょう。さらに、仕様記述言語やツールを用いるのではなく、自然言語やワードプロセッサ、校正支援ツール等を用いて仕様や文書を記述する際にも、仕様記述言語やツールの仕様や機能を利用する場合の効果との対比を考えながら、どのような目的で、どのような仕様の言語と、どのようなツールの機能を用いて、厳密な仕様を記述するのかについて検討するとよいでしょう。

9. 仕様の設計

仕様記述言語を用いた、たとえば、VDM-SL や VDM++ 言語を用いた仕様の設計には、プログラムの設計に用いる技術、具体的には、構造化分析設計技法、オブジェクト指向分析設計（コンポーネントやクラスの設計）技法、を用いて行います。プログラムの設計に関する経験があれば、仕様の設計は難しくないでしょう。階層構造の検討、クラスやモジュールの設計、変数や関数・オペレーションの決定、ライブラリやユーティリティの検討を行います。これらの検討により、用語や述語、データ構造の決定を行うことができます。

繰り返しになりますが、仕様の設計と、プログラムの設計を必ずしも合わせる必要はありません。しかし、とくにデータ構造は、どのように決定するのかは設計の意図によりますが、表したい仕様として適切な抽象化を行うとよいでしょう。先に、仕様（「What」）とプログラムの設計と実装（「How」）の分離について書きましたが、また、仕様とプログラムの設計の方式を合わせることもあるのではないかと書きましたが、仕様をどのように書きたいのか、によって、仕様の設計方法は変わります。仕様の設計と、プログラムの設計をまったく別の方法で行なってもよいですし、二つの設計をまったく同じようにしてもよいでしょう。これは開発の事情や意図に応じて適宜判断していけばよいことです。ただし、基本的な考え方としては、仕様は、プログラムの設計と実装とはことなる抽象度のものであり、プログラムの設計と実装の「仕様」にはなっても、制約となるべきものではありません。したがって、できるだけプログラムの設計と実装には自由度を持たせるようにすべきものであり、仕様はプログラムの設計に自由度を与えず、すべてを拘束するような内容であってはなりません。

10. 導入や教育に関する検討

形式仕様記述言語や、形式仕様の学習のために、特別なことを行う必要はありません。プログラム開発を行うための技術に加えて、仕様記述言語の仕様やツールの利用方法を習得すればよいのです。

形式仕様記述言語を用いた仕様の記述を行うためには、開発領域に関する知識や経験、日本語の読み書きの能力以外に、とくに、プログラムの設計や読み書き、デバッグを行うための能力が必要です。このことを言い換えると、プログラムの設計や、プログラミングの経験があれば、形式仕様記述言語の仕様やツールの使い方を学習するのは難しくありません。これまでに述べてきたように、プログラム開発のためのさまざまな考え方や工夫が、仕様記述においてもそのまま応用することができることでしょう。一方で、プログラム開発を行うための技術を一から習得する必要があるのであれば、相当の努力と期間が必要になるでしょう。何も知らない、できない人が、たとえば、一から、コンピュータの基礎やオブジェクト指向分析設計、オブジェクト指向プログラミング言語、開発環境に関して学習し、業務を行うことができるようになるのと同様の期間や努力が必要になります。

ここで、「形式仕様記述とプログラミングは何が違うのですか?」、というよくある質問がまた登場するでしょう。それは、こちらもこれまでに述べてきましたが、記述の目的と抽象度、具体性が異なるのです。ただし、その目的や具体性は、開発の位置付けによって異なりますから、一概に決めることはできません。一方で、設計や記述を行う技術は似通っています。手段が似ていて、目的が異なることを再確認しておきましょう。

また、プログラミングの経験がある開発者は、具体的に実現することに慣れていますから、仕様を、仕様記述の目的に対して具体的に記述しすぎてしまうかもしれません。そして、その結果、仕様の記述をしているのか、最終的なプログラミングに近い記述をしているのか悩み、形式仕様記述の本来の意味を見失ってしまうかもしれません。仕様の記述においては、絶えず記述の目的を見直しながら、具体と抽象の間を行ったり来たりしながら、仕様としての記述を積み重ねていくしかありません。

このような形式仕様記述に特別な技術要素は必要ありません。必要な技術は、プログラムの開発に必要なものばかりです。

・ソフトウェア科学の基礎
・プログラミング言語の基礎
・ソフトウェア工学の基礎
・プロジェクトマネジメントの基礎
・各種設計、アーキテクチャ検討についての基礎

これらがすべてではありませんが、ソフトウェア開発技術者が、形式仕様記述のためだけに獲得しなければならないことではない、誰もが、あえて言えばそれなりに、おさえておかなければならないことばかりですから、どこでも通用するソフトウェア開発技術者になることを目指すのであれば、ある程度網羅的な学習をしておくべきでしょう。
他方、仕様策定工程は、プログラミングとは異なる目的で形式仕様記述を行うことによって、プログラムの開発よりも制約が少ない環境で、抽象的な記述を行うことになりますから、従来の知識や経験を抽象的に理解しながら、再整理していく、ということにもつながります。また、仕様策定工程は、これまでにも考えてきたとおり、さまざまな他の工程と密接に結びついていますから、開発工程の全体や、他の工程のことについて考えていくよいきっかけにもなります。
繰り返しになりますが、形式仕様記述は、さまざまなことがらに関係しますので、仕様記述の学習と実践により、非常に幅広い視点でものごとを学習し、とらえ直し、検討することができるようになるのです。

11. 開発管理について

厳密な仕様の記述、とくに形式仕様記述言語を用いた仕様の記述を行う場合、従来の曖昧な仕様の記述と比較して、プログラムの記述や検証と同様の開発管理を行うことができる可能性があります。あなたのプログラム開発の現場では、どのようなことがらを、管理のための管理ではなく、実際に効果のある管理としておりますか。あるいは、従来、自然言語を用いた文書の作成において、どのような方法で進捗や品質を管理してきましたか。また、このときにどのような課題がありましたか。

そのことをあらためて確認しながら、厳密な仕様記述において、何を管理すべきか考えてみましょう。管理項目としては、規模（行数)、1 行あたりのレビュー時間、テスト項目と欠陥の数、記述の効率、要求と仕様と設計とテスト項目のトレーサビリティの完全性、等が考えられます

もちろん、やみくもになんでも管理をすればよいというものではありません。また、仕様そのものや、仕様に基づく開発における工程や成果物の品質についても、その目的を明らかにしながら、効果について確認できるかもしれません。たとえば、レビューや確認の記録を紐解き、厳密性や、予測性、管理可能性、成果物そのものの品質について追跡することが可能となります。もちろん、理由がないのにむやみに追跡する必要はありません。いずれにせよ、開発の目標と、管理する目的を明確にしたうえで、効率良く品質の高いソフトウェアを開発していくために必要な管理情報について、開発メンバや利害関係者と合意形成しながら決定し、開発組織を適切に運営していきましょう。

12. 仕様の記述に関する留意点

最後に、これまでに触れなかった仕様の記述に関する注意点の一部を紹介します。参考にしてください。

◆ 従来よりも仕様の策定、記述の期間が長くなるかもしれない

仕様を広範囲に厳密に記述したり、仕様のレビューや検証を繰り返し行なったりする場合、従来よりも仕様の策定、記述、検証に時間が掛かるかもしれません。これは、前述のとおり、その後の工程における手戻りの時間の削減や、最終的な品質とトレードオフになるものです。また、これまで考えることを避けてきたさまざまな仕様に関する問題について考える必要性が生じるかもしれません。一方で、曖昧な範囲を曖昧に記述するよりも、期間の見積もりや進捗は厳密に行うことができる可能性があります。もちろん、みだりに管理を行えばよいというものではありません。

◆良い仕様とは限らない

厳密に記述した仕様が良い仕様とは限りません。ある課題を解決するための仕様の案は他にもあるかもしれませんし、そちらの方が優れているかもしれません。何かについて検討するときに、既に検討したことを厳密に書くことが、思考の現在位置を明確にし、その後の検討の役に立つことはあるかもしれません。反対に言えば、厳密な仕様記述が良い自由な発想の妨げになることはあるのでしょうか。あなたにとってないのであれば、厳密に記述をしない理由にはならないかもしれません。あとは時間の問題だけです。以上を踏まえた上で、利害関係者とは、仕様を厳密に書くからといって、書くことの「内容」が良いとは限らないという、あたりまえのことを再確認しておきましょう。

◆すべてを形式仕様記述言語で書く必要があるのか

「必要」はありませんし、仕様の「すべて」を書くことは現実的ではないでしょう。しかしながら、開発の起点となる、開発における情報の基準となる基本的な仕様に関して、とくに従来曖昧な仕様に関する課題があったのであれば、基本的となる機能仕様のすべてを厳密に記述することに価値があるのかもしれません。これはこの章で繰り返し述べてきたことです。文書には、記述の目的と読者がいます。このことをよく考えて文書体系を検討し、さまざまな言語や図表の記法を用いながら、文書の設計をしていく必要があります。

◆形式仕様記述手法に限らないことがら

上記のことも含めて、解決したい問題が、達成したいことが、厳密な記述の問題なのか、形式仕様記述や形式手法で解決できる、達成できることなのか考えてみてください。ひょっとしら、さまざまなことがらが、形式手法やソフトウェア工学とは関係ないこと、形式手法やソフトウェア工学に限らないことかもしれません。そう言ったことがらについて、狭い範囲で思考を続けていくことが適切なのかどうか、考えてみてください。

この章のまとめ

この章で一貫して述べたかったことは、誤解を恐れずに言えば、形式手法や形式仕様記述手法において、こうしなければならないというような絶対的な決まりはなく、すべて

が自由だということです。自由だとかえってどうしたらよいか分からないという声が聞こえてきそうですが、だからこそ、従来の課題や、この開発で本質的に何を達成したいのかについてチームメンバや利害関係者と検討、合意し、その中で、自分達の身の丈に合ったやり方で形式手法や形式仕様記述手法を、道具の一つとして利用していけばよいのです。本章で述べたことは、仕様や形式仕様記述手法、開発全体について考えるほんの一端に過ぎませんが、皆さんの現場における具体的な課題の解決の役に立てばと思います。

章末コラム [現状を分析する]

いざ、現状の問題点を分析し、仕様に形式仕様記述を適用しよう、と考えたとします。この際に、大きな障害があるとすると、「抽象化」、「厳密」「論理」など、分かるようで、実は分かっていない言葉ではないでしょうか。これらの言葉は、日常生活でも耳にする言葉であるためかもしれませんが、ふと聞いたときに理解したような気になってしまうのです。そのため、実際に自分の現場で適用しようとしても、手が出ないという状況に陥ってしまうのではないかと感じています。

第 4 章でもこれらの言葉について説明されていますが、その説明を理解しようとした際に、第 4 章で示唆されているように「たとえば自分の現場では...」と具体的な事例に転換してみると、理解が深まるように思います。そして、現場の具体的な事例に転換した際には、どのような効用が得られそうか、考えてみるといいと思います。効果が実感できなければ、第 4 章でいう問題分析が失敗しているのかもしれません。

また一方で、形式仕様記述を用いながら仕様を記載しする場合に、プログラミングの経験のない関係者（顧客、プロジェクトマネージャの一部など）には、形式仕様記述を土台にして議論することが難しいのではないかという疑問を持ちました。それこそ、現場でどのような仕様策定方法が適しているかという問題に帰するのかもしれません。しかしながら、ソフトウェア開発の現場、特に、上流工程の現場では多種多様なステークホルダが関わっており、共通に仕様というものを理解するにはどうしたらよいか、といったことも併せて検討すると、さらに厳密に仕様を記述することの効果が得られると感じました。

最後に、今までの（おそらくは、ほとんどが図や自然言語からなる）仕様と形式仕様記述との関連をどのように考えていけばよいのかを疑問に思いました。今までの仕様のす

べてを形式仕様記述に書き換えるだけの余力は多くの現場にはないと思います。したがって、現状の問題分析し、さらに、形式仕様記述を用いることを決めた場合に、今までの仕様の取り扱いをどうすべきかの検討を含めた方がいいのだろうと思いました。
仕様の問題点を検討するだけでも、ソフトウェア開発における今まで目をそらしてきた問題点やさまざまな疑問点が浮かび上がる予感がします。まずは、第 4 章の内容を吟味し、自分たちの現状を見直そうとする行為こそが、この第 4 章を読んだ成果になるのだと思います。

[第 4 章執筆者のコメント]

ありがとうございます。目をそらす。私も日々、目をそらし続けています。いろいろなことがらから。
開発する、開発したいソフトウェアが、動くプログラムなのだとすると、そして、マニュアルの作成や、ユーザのサポート、運用・保守、派生開発等の開発以外のことがらもすべて考慮したうえで、動くプログラム以外のものを作らなくてよいのだとすると、その他のものはすべて無駄なものになるわけですから、時間やお金が余っていない限り、よほど仕事が好きではない限り、仕様も含めて、何も書いたり、確認したりする必要もないでしょう。別に極端なことを書いているのではありません。これはよくあることなのです。問題は、組織的に、未来も見据えて、困っているか、困っていないかだけです。
その上で、プログラムとは別の何か、を書く必要はあるのでしょうか。たとえば、ユーザから「このプログラムはもともと、何を作りたくて作ったものなのですか」と問いかけられたときに、「何を作りたかったのかはあらかじめ考えていませんでした」と答えても構わないのであれば、それはとても幸せなことだと思いますし、わざわざ仕様書を書く必要もないのでしょう。「何を作りたいのか」,「何を作りたかったのか」を考えて書いておく必要があるのであれば、動くプログラムを作ったあとに、「こんなものを作ったのです」とあらためて書く必要があるのであれば、それはすなわち、仕様書が必要だということです (必ずプログラムの開発よりも前に仕様書が必要になるとは限りません。プログラムを作ったあとに、何を作ったのかな、と仕様書を書くこともあるでしょう。そういう仕事のし方を頭ごなしに否定する必要はないと思います)。動くプログラムしかなかった場合に、プログラムを見ても、そう書かれた結果が残されているだけであって、何を書きたかったのかは、仕様は、分かりません。テストも同様です。作ったものが、作ろうとしている動くプログラムが、プログラマが書いたプログラムの意図ど

おりに動くことを確認すればよいのであれば、プログラムを見ながらテストプログラムを作り、テストを行えばよいのかもしれません。プログラムが仕様どおりに動くのかどうかを確認したいのであれば、プログラムを見ながらテストプログラムを作らない方が良いでしょう。プログラムをまったく見ないで、仕様だけを参照してテストプログラムを作ることができるようになることが理想です（と私は思います)。動くプログラムとは異なる抽象度の仕様が必要になる理由はさまざまです。このときに、どのような「論理」を、「抽象化」し、仕様として、「厳密」に書くのかの背景や理由はさまざまです。考えてみてください。なぜここで考えることをあなたに預けてしまうのかというと、このことはケースバイケースだからです。プログラムと同じ具体性をもった仕様書を書くことに意味はあるのでしょうか。意味がないのであれば、書く必要はありません。あるいは抽象化する必要があります。意味があるのだとすれば、プログラミング言語とは異なる言語で仕様書を書くことに価値があるということなのでしょう。そのことは，書くことは、プログラミングとは目的が異なるのでしょう。きっと。それでは、抽象化とは何でしょうか。それは、たとえば、具体化をしない、ということです。具体化とはなんでしょうか。それは、たとえば、動作するプログラムに固有の工夫や事情や制約について考えたり、踏まえたりする、ということです。工夫や事情や制約を加味しない仕様が必要なのであれば、それは，工夫や事情や制約を除くことにより、仕様として抽象化することができるようになります。これは例です。抽象化の道は一つではありません。さまざまなアプローチが考えられます。

どのような「論理」を「抽象化」し「厳密」に書くのかはケースバイケースです。このときに、プログラムの設計やプログラミングの経験が必要になる、「論理」を「抽象化」するアプローチもあるでしょう。経験がないのであれば、経験をするか、そのアプローチをあきらめるしかないです。これは、選択肢が一つ減るということです。たいしたことではないのかもしれません。しかし、それは、本質的に、「プログラミングの経験」といってよいことですか、というのが、私の問いかけです。「プログラミングの経験」であるという以前に、「ソフトウェア開発の本質の一つ」なのだとしたら、その本質の一つを捨ててもよいのですか、という問題です。

4 章で繰り返し述べてきたことですが、課題、事情、制約はさまざまであり、ソフトウェア開発に決まったアプローチはありません。取り組みや工夫は一つではなく、いくつかのものごとを組み合わせるのでしょう。言葉も同様で、プログラミング言語に加えて、書き言葉、話し言葉としての自然言語や、さまざまな図や表の記法を用いるのでしょう。このときに形式仕様記述言語も用いるかもしれません。さまざまな言葉を使って、いろいろな、プログラムも含めて沢山の文書をチームで組織的に、何らかしらの論理を、さ

まざまな抽象度で、目的に応じて、厳密に書いて、確認していく。このときに失ってはならないものは何なのか、ということについて、それぞれが考えていく必要があるのだと思います。

第５章 厳密な仕様記述を支援するツール

この章について

本章では、厳密な仕様記述を支援するために有用なツールを紹介します。決して網羅的ではありませんが、さまざまな目的のためのさまざまな言語/記法をさまざまな環境で利用可能なものを集めました。まずは各ツールの言語、方法論、記述対象、文書化、OS、開発環境、静的検査、動的検査、日本語による記述との連携、日本語対応について表で示し、続いて言語ごとにツールを紹介します。

紹介するツール一覧

表 5-1 : VDM family および Z 記法のツール一覧

ツール名	VDMTools	Overture Tool	Z word tool	CZT
言語	VDM family (VDM-SL VDM++ VDM-RT)	VDM family (VDM-SL VDM++ VDM-RT)	Z 記法	Z 記法
方法論	VDM (Vienna Development Method)	VDM (Vienna Development Method)	-	-
記述対象	機能仕様	機能仕様	機能仕様	機能仕様
文書化	LaTeX RTF XMI	LaTeX	MS-Word	LaTeX テキスト
OS	Windows MacOSX Linux	Windows MacOSX Linux	Windows	Windows MacOSX Linux
開発環境	独自 IDE CUI コマンド	Eclipse	MS-Word	独自 IDE Eclipse
静的検査	型検査 証明課題生成	型検査 証明課題生成	型検査	型検査
動的検査	実行 単体テスト 表明の検査	実行 単体テスト 表明の検査		実行
日本語に	日本語識別子	日本語識別子	識別子の索引化	Unicode 形式

よる記述との連携	文芸スタイル	文芸スタイル		文芸スタイル
日本語対応	IDE 識別子	識別子	識別子(不完全)	識別子(不完全)

表 5-2 : B-method とモデル検査のツール一覧

ツール名	Atelier B	ProB	Rodin	SPIN	LTSA
言語	AMN	AMN CSP TLA+ Z 記法		Promela	
方法論	B-Method	B-Method 他	Event-B		
記述対象	機能仕様 設計	機能仕様 設計 並行プロセス		並行プロセス	並列状態機械
文書化	PDF RTF LaTeX	GraphViz によるグラフ			グラフ
OS	Windows MacOSX Linux	Windows MacOSX Linux		Windows MacOSX Linux	
開発環境	独自 IDE	独自 IDE	Eclipse	独自 IDE Eclipse	独自 IDE
静的検査	型検査 証明課題生成 証明支援	型検査	型検査 証明課題生成 証明支援	型検査	

動的検査		アニメーション		モデル検査	モデル検査アニメーション
日本語による記述との連携					
日本語対応	IDE				

以下にツールを言語毎に分類し、個別に説明します。

VDM family

VDM（ウィーン開発手法）はモデル規範型を代表する形式手法の一つです。海外にも国内にも適用事例があり、製品の品質や開発の経済性の点での成功事例が報告されています。

記法として、ISO/EIC 13817-1 として国際標準となっている VDM-SL、オブジェクト指向拡張である VDM++（または VDM-PP とも呼ばれています)、リアルタイムシステム向け拡張である VDM-RT の三つがあります。VDM の特徴の一つとして、陽仕様と呼ばれる定義をすることで、モデルをコンピュータ上で実行可能であることが挙げられます。ファイルフォーマットとして文芸スタイルと呼ばれる LaTeX 形式による記述が可能です。文芸スタイルは VDM family による形式仕様と日本語による記述を併記することによって、形式仕様で記述された厳密な仕様を理解するために必要な補足説明をすることができます。

VDM は日本語の書籍が多数あること、IPA/SEC が教材を提供していること、日本における成功事例が報告されていること、日本企業が提供しているツールがあることから、日本人技術者および企業が初めて形式的仕様記述に取り組む際には第一選択肢として挙げられます。

日本語での主な文献

プログラム仕様記述論(IT Text)

荒木啓二郎、張漢明 著

オーム社、ISBN-13: 978-4274132636

ソフトウェア開発のモデル化技法

ジョン・フィッツジェラルド、ピーター・ゴルム・ラーセン 著、荒木啓二郎、荻野隆彦、染谷誠、張漢明、佐原伸 訳

岩波書店、ISBN-13 978-4000056090

VDM++によるオブジェクト指向システムの高品質設計と検証

ジョン・フィッツジェラルド、ピーター・ゴルム・ラーセン、ポール・マッカージー、ニコ・プラット、マーセル・バーホフ 著、酒匂寛 訳

翔泳社、ISBN-13: 978-4798119618

VDM++による形式仕様記述（トップエスイーシリーズ実践講座）

石川冬樹 著、荒木啓二郎 監修

近代科学社、ISBN-13: 978-4764904095

厳密な仕様記述を志すための形式手法入門

IPA/SEC

http://sec.ipa.go.jp/reports/20121113.html

VDMTools

VDMTools は株式会社 SCSK が提供している開発支援ツールです。ツール、マニュアル共に日本語化されています。VDMTools Lite 版は無償で利用することができ、全ての機能を実装した正規版は商用ライセンスとアカデミックライセンスにより提供されています。

URL: http://www.vdmtools.jp/

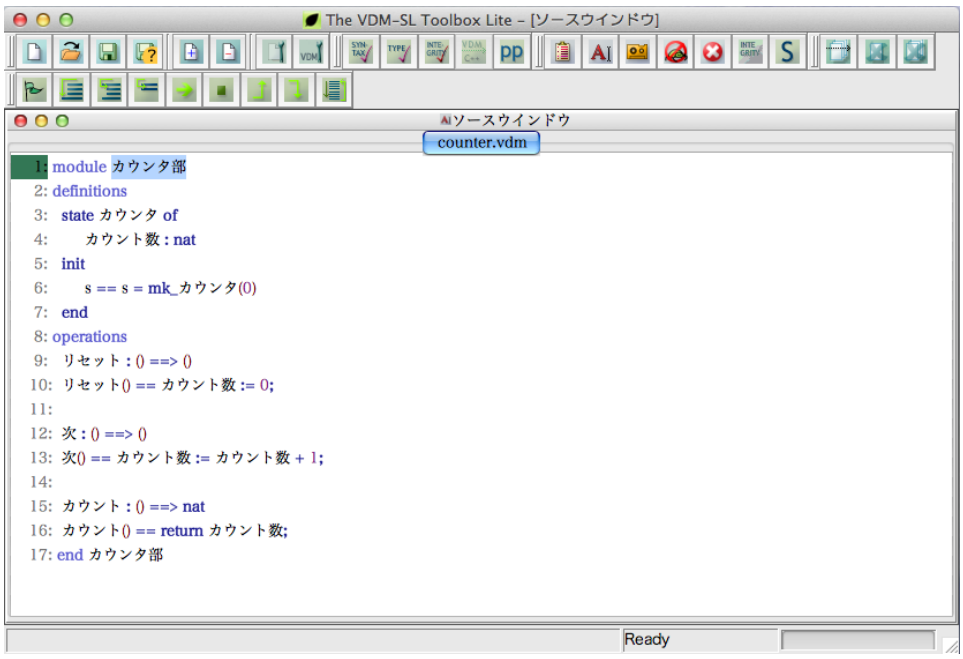


図 5-1 : VDMTools のスクリーンショット

Overture tool

Overture tool は、オープンソースコミュニティにより開発されている VDM 統合開発環境(IDE)です。Overture tool は Java で実装されており、Eclipse プラットフォーム上で動作します。メニュー等は英語。識別子等で日本語も使えるが、プロジェクト名など日本語が使えない部分もあること、また、日本語入力でのキー入力と干渉する場合があるため、日本語を多用する場合には注意が必要です。

URL: http://www.overturetool.org/

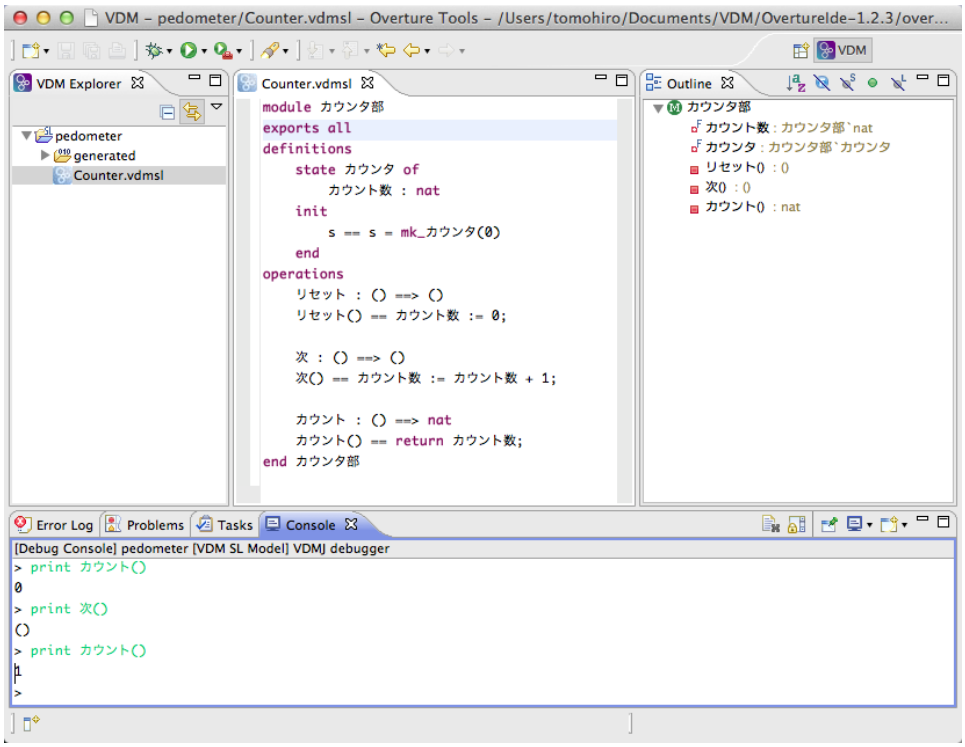


図 5-2 : Overture tool のスクリーンショット

Z 記法

Z 記法は VDM 同様、モデル規範型を代表する形式的仕様記述言語の一つで、ISO/IEC 13568:2002 として国際標準化されています。航空管制などの高信頼性システムの開発において事例が報告されています。Z 記法は非常に強力な抽象的表現力を持っており、厳密かつ簡明な仕様記述が可能です。Z 記法のサブセットを実行するためのツールもありますが、Z 記法は特に構成的正当性(Correctness by Construction)と呼ばれる、証明を中心とした開発の成功事例が報告されています。Z 記法自体は特定の方法論を前提としていません。

Z 記法は比較的早い時期から LaTeX 形式での記述が標準的に行われてきました。VDM family の文芸スタイルと同様に、日本語を含む自然言語と形式仕様の併記が可能です。最近ではマイクロソフト社の Microsoft Word への拡張として、Microsoft Word 文書内での Z 記法の記述を可能にするツールが登場しました。

日本語での主な文献

プログラム仕様記述論(IT Text)

荒木啓二郎、張漢明 著

オーム社

ISBN-13: 978-4274132636

ソフトウェア仕様記述の先進技法 Z 言語

ベン・ポッター、デイビッド・ティル、ジェイン・シンクレア 著、田中武二 訳

トッパン

ISBN-13: 978-4810185638

Community Z Tools

Z 記法の WYSIWYG 編集、型検査、実行を行うことを目標にしたオープンソースによる開発ツールです。Java で実装され、GPL で提供されています。Z 記法とその拡張である Object-Z, Circus, TCOZ をサポートしています。IDE へのプラグインとして、バージョン 1.0 では jedit プラグインと Eclipse プラグインが、バージョン 1.5 では Eclipse プラグインが公開されています。メニュー等は英語で、日本語識別子は構文チェック、型チェックは可能ですが、不要な警告が出るなどの不具合が見られます。

URL: http://czt.sourceforge.net/

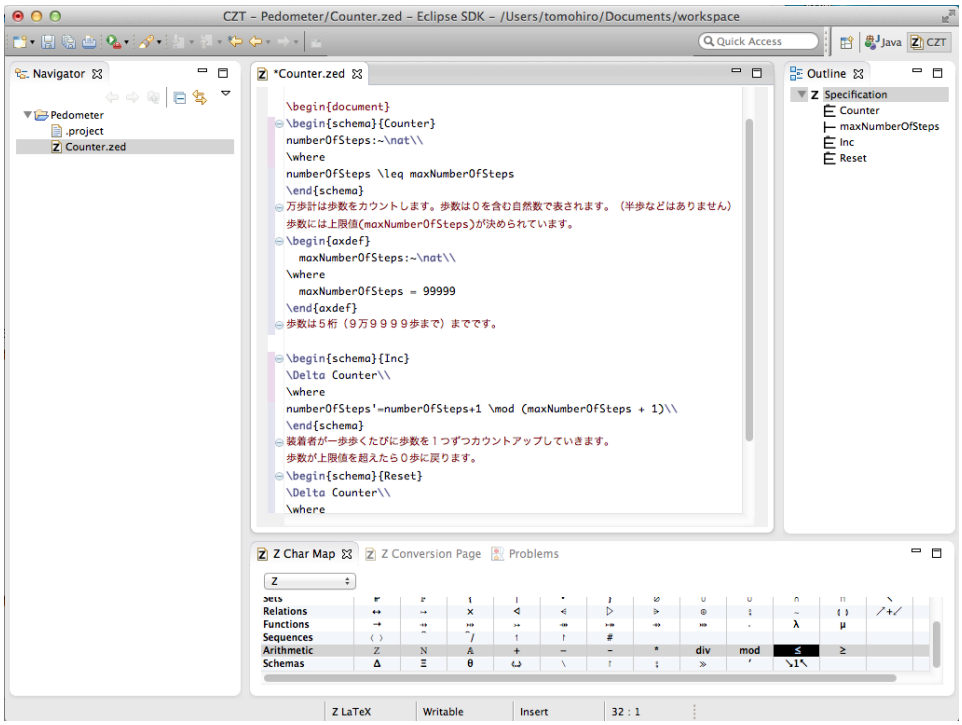


図 5-3 : Community Z Tools のスクリーンショット

Z Word Tools

Microsoft Word 上で Z 記法で仕様を記述するためのツール集です。Z 記法と自然言語の自然な統合をすることで、数学に通じていない関係者への障壁を軽減するとともに、より頻繁な Z 記法の査読と自然言語での説明を促します。型検査、識別子の索引化、ダイアグラム生成等の機能を持っています。Z 記法のバリエーションとして、ISO 標準と通称 Spivey Z をサポートします。日本語は識別子等については ISO 標準(バックエンドとして CZT を使用)では構文検査および型検査が可能です。ダイアグラム生成では日本語は使えません。

URL: http://zwordtools.sourceforge.net/

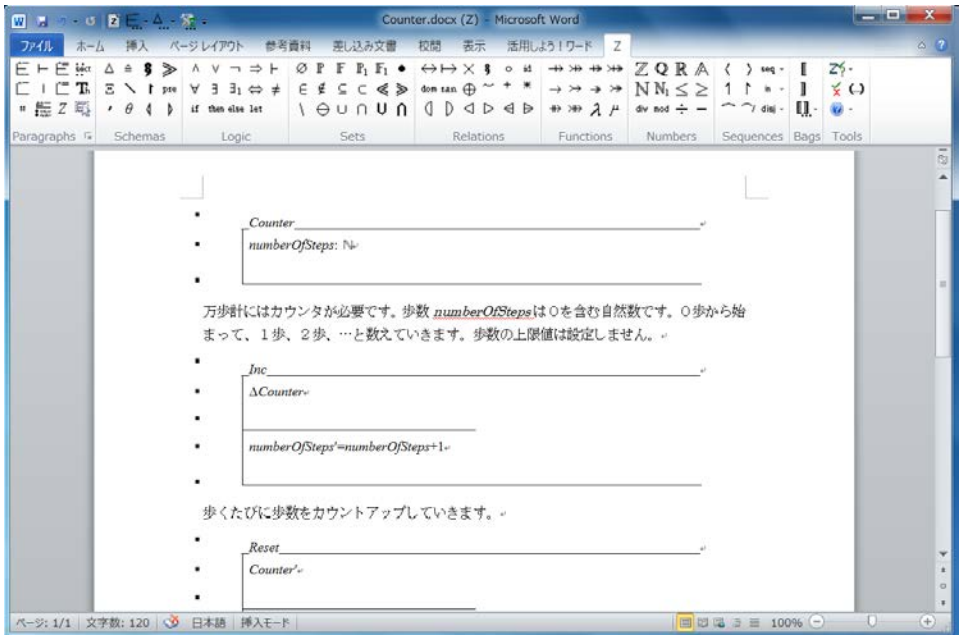


図 5-4 : Z Word Tools のスクリーンショット

ProB

ProB は B-Method のための開発環境ですが、Z 記法で記述された仕様のアニメーションやモデル検査を行うことができます。詳しくは B-Method のツールとして後述します。

B-Method

B-Method は仕様から実装まで AMN（Abstract Machine Notation、抽象機械表記法）によってシステムを表現し、段階的詳細化と呼ばれる技術で徐々に抽象度を下げて最終的な実装を得ます。抽象度を下げる度に、詳細化された記述が元の記述と整合していることを証明することによって仕様に対する実装の正当性を確保します。Z 記法と同様、構成的正当性(Correctness by Construction)と呼ばれる証明を中心とした高信頼性システムの開発を主眼に置いており、パリ地下鉄などで成功事例があります。

日本語での主な文献

B メソッドによる形式仕様記述―ソフトウェアシステムのモデル化とその検証（トップエスイー実践講座）

来間啓伸 著、中島震 監修

近代科学社、ISBN-13: 978-4764903470

Atelier B

Atelier B は AMB で記述されたモデルの型検査および証明責務の生成、証明支援を行います。Atelier B はもともと商用ツールとして利用されていましたが無償版も提供されるようになりました。メンテナンスサポート契約版はより多くの機能を提供しています。また、GUI 内のメニューやメッセージの多くが日本語に翻訳されています。

URL: http://www.atelierb.eu/en/

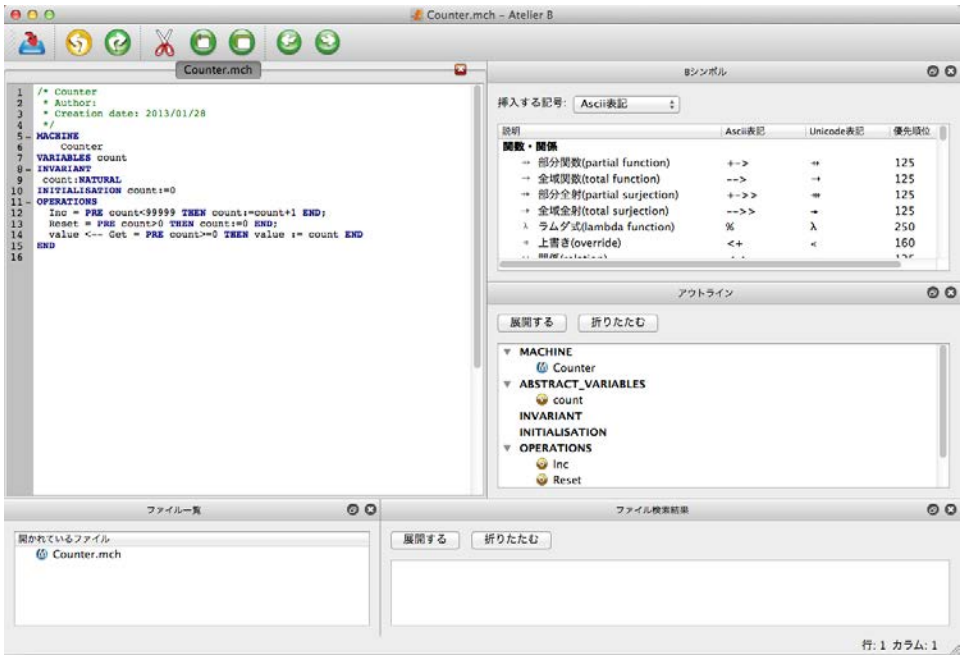


図 5-5 : Atelier B のスクリーンショット

ProB

ProB は主に B-Method でのモデルのアニメーションやモデル検査を行うツールです。B-Method の AMN 以外に、CSP, TLA, Z 記法も扱えます。また、実験的ですが Promela も扱えます。オープンソースのグラフツール「GraphViz」と連携して、状態空間の可視化を行うことができます。識別子等で日本語を扱うことはできません。

URL: http://www.stups.uni-duesseldorf.de/ProB/index.php5/Main_Page

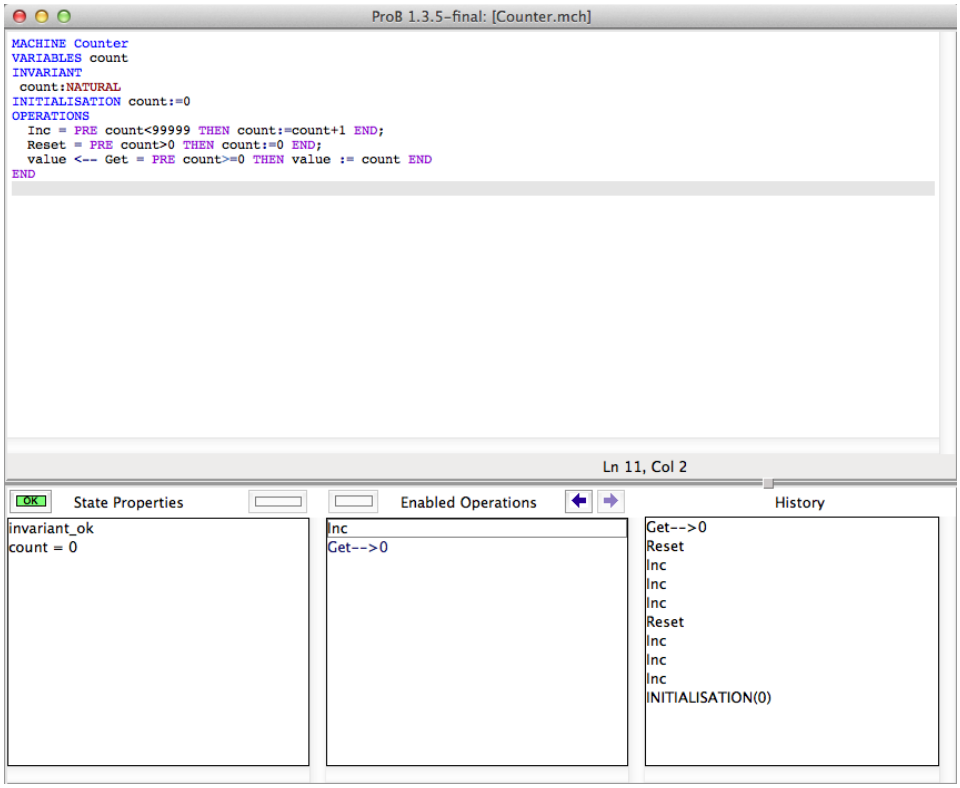


図 5-6 : ProB のスクリーンショット

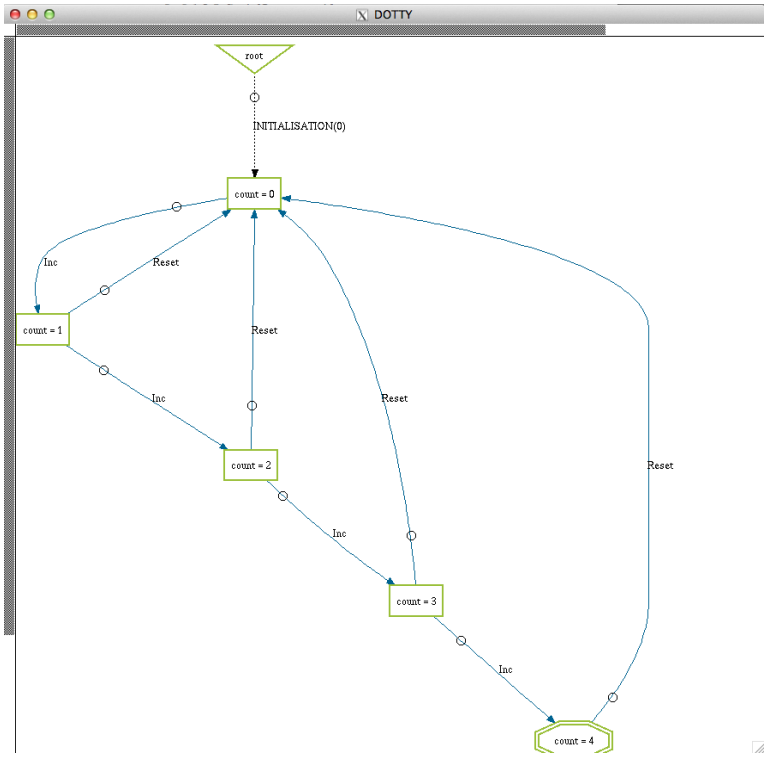


図 5-7 : ProB による *GraphViz を使った状態空間の可視化*

Rodin

B-Method から派生した Event-B の Eclipse ベースの開発環境です。段階的詳細化と証明を支援します。Event-B は B-Method と同様に、集合論を使った記述と段階的詳細化、証明責務により高信頼性システムを開発します。

URL: http://www.event-b.org/

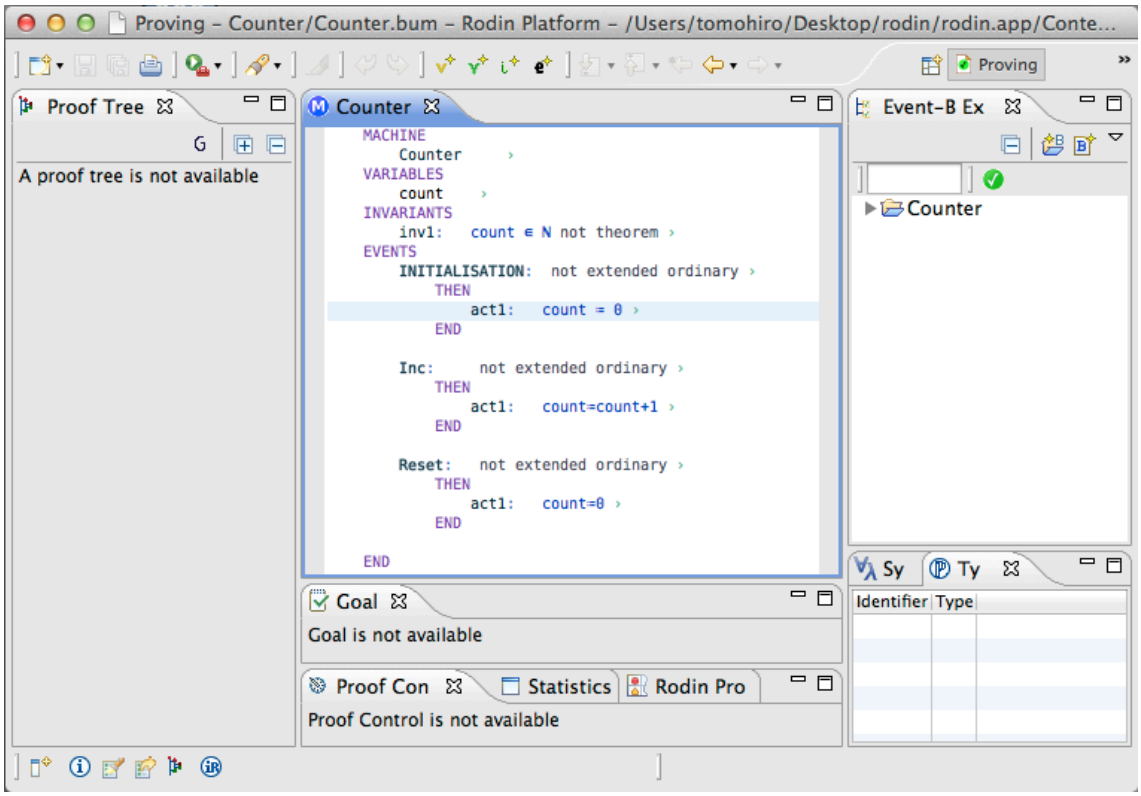


図 5-8 : Rodin のスクリーンショット

PROMELA

Promela（PROcess/PROtocol MEta LAnguage)は並行プロセスルゴリズム記述言語で、モデル検査器として普及している SPIN で検証対象となる並行アルゴリズムの記述に使われています。SPIN は仕様記述よりも検証に焦点を当てたツールですが、マルチコアや分散環境で動作するシステムの振る舞いを記述し、そのプロセスの定義に表明を記述することができるなど、並行プロセスの振る舞いの厳密な記述とも言えます。

日本語での主な文献

SPIN モデル検査 - 検証モデリング技法

中島震 著

近代科学社、ISBN-13: 978-4764903531

SPIN モデル検査入門

モルデチャイ・ベン-アリ 著、中島震、谷津弘一、野中哲、足立太郎 訳

オーム社、ISBN-13: 978-4274208447

Spin

Spin は並行プロセスの振る舞いを有限状態モデルを生成して検査するツールです。

1980 年からと、モデル検査器として比較的長い歴史があり、プロセス間の協調の仕組みとしてランデブー、メッセージ通信、共有メモリという異なる方式をサポートする等、多くの改良が重ねられてきました。書籍も多く発行されており、適用事例も多く、モデル検査器として標準的なツールと考えられます。

URL: http://spinroot.com/spin/whatispin.html

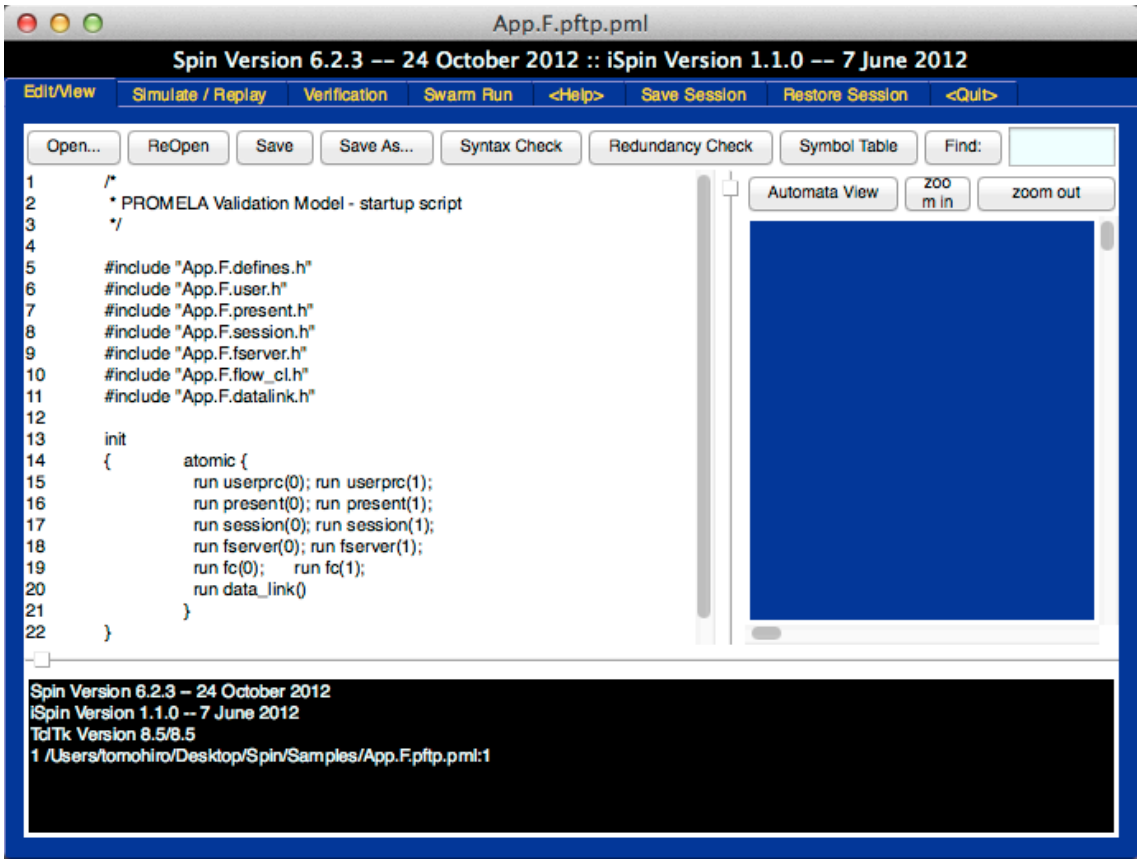


図 5-9 : iSpin のスクリーンショット

FSP

FSP(Finite State Processes)は有限状態機械を代数的に記述する言語です。プロセスをアクション列としてモデリングします。

日本語での主な文献

組み込みソフトウェアの設計&検証

CQ 出版、ISBN-13: 978-4789833448

LTSA

LTSA は並列に動作する状態機械を合成してモデル検査をおこなうツールです。Java で実装されており、合成されたモデルをグラフィカルに表示する可視化機能や、モデルをユーザが操作するアニメーション機能もあります。シンプルでモデル検査器の入門としても、また、検査、可視化、アニメーションによりモデルへの理解を支援するツールとしても使うことができます。

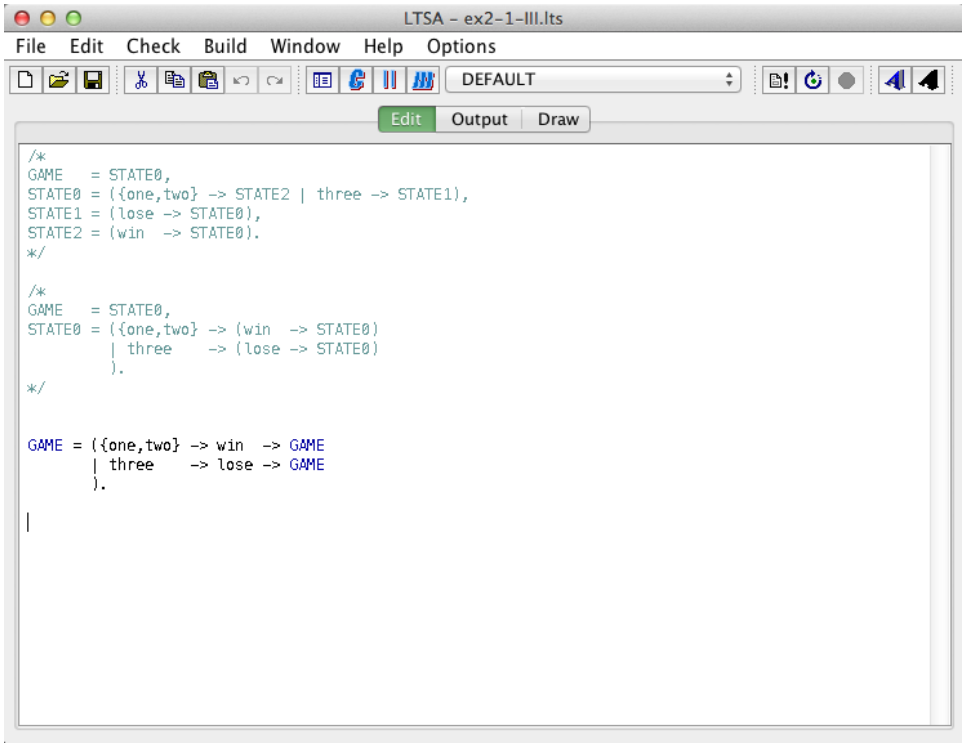


図 5-10 : LTSA のスクリーンショット

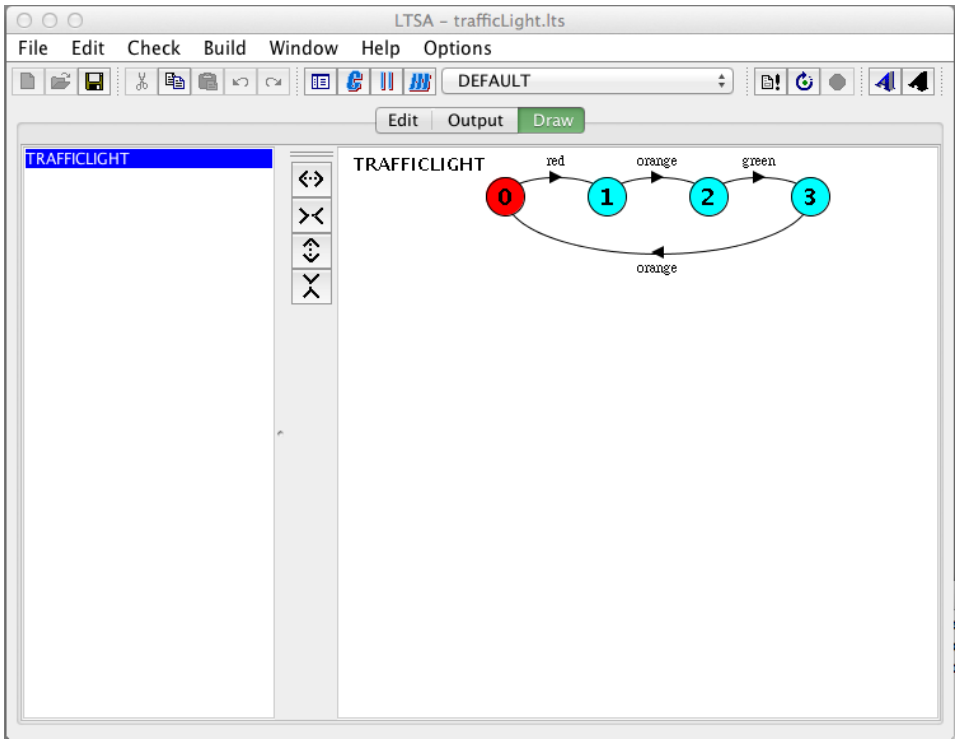


図 5-11 : LTSA による状態機械の可視化

この章のまとめ

本章では、形式的仕様記述として VDM family および Z 記法、B メソッドのツールと、モデル検査ツールとして SPIN と LTSA を紹介しました。各ツールについて言葉による説明に動作中のスクリーンショットを添えて紹介しました。しかしツールの本当のところは、実際に動かしてみなければ分かりません。残念ながら、本章のスクリーンショットは動きません。単なる絵です。ぜひ、「動く」ツールで体験してください。形式的仕様記述言語も、モデル検査用の言語も、紙の上ではなく、コンピュータの上でこそ生きる言語です。ツールを立ち上げた時があなたのスタート地点です。では、厳密な仕様記述の世界を楽しんでください。

章末コラム [まずは、一歩踏み出そう]

第 5 章を読むことにより、多様な形式仕様記述言語それぞれに対し、多様なツールが用意されていることが分かります。共通の機能や画面の構成が似ているツールも多いようなので、まずは、本書に従って、自分が用いたい形式仕様言語を決定した上で、ツール

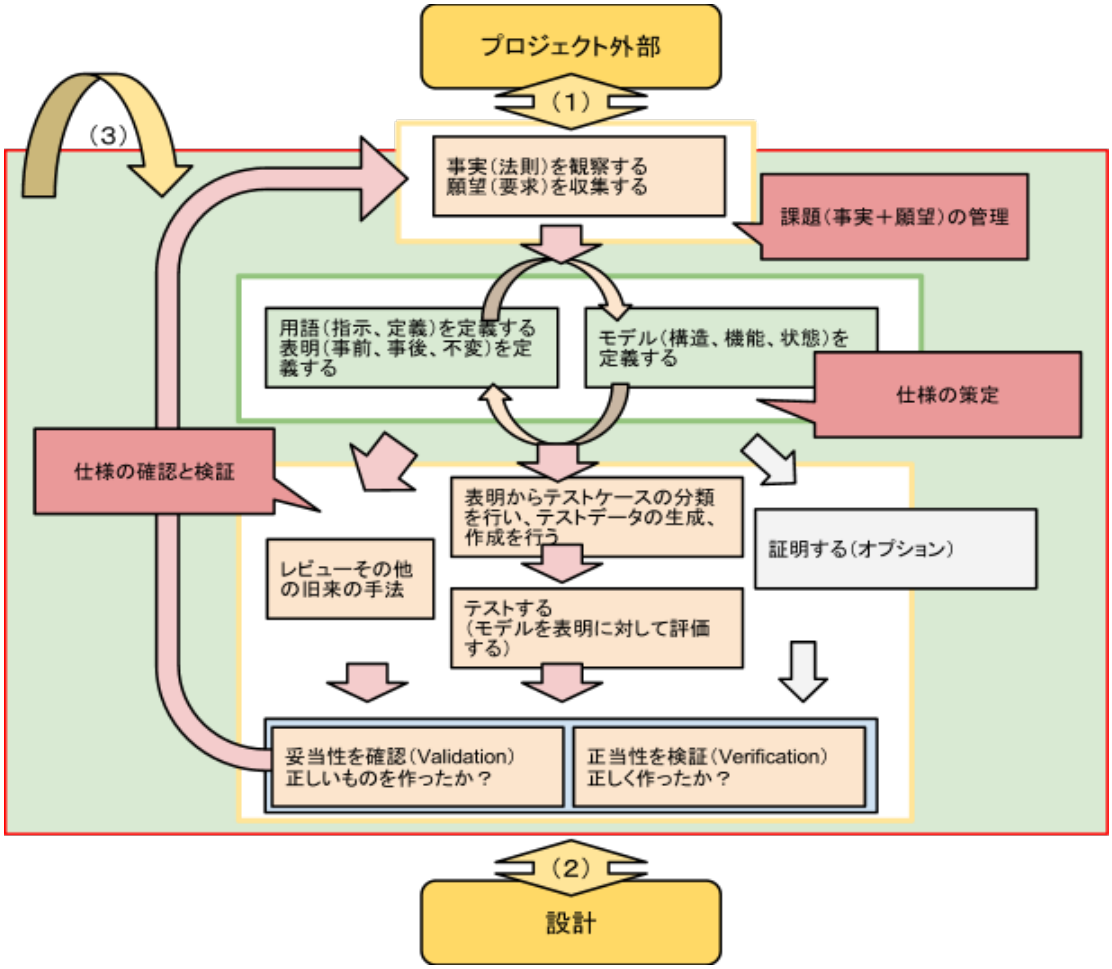
を決定し、使ってみることが大切なのだと思います。
そうは言っても、現実には、その一歩が踏み出せない、ということが往々にしてあるのだと思います。私もそうなのですが、なぜかツールを利用するということが大変に障壁の高いもののように感じてしまうことがあります。それはなぜなのか少し考えてみました。
真っ先に思い浮かぶのは、ツールの多くが英語で提供されているということです。日本語対応がされていても、インストール時点で英語を読まなければならないことは多いかと思います。しかし、同じ専門領域では、同じような言葉が使いまわされていることが多く、特に IT 分野では、カタカナ語としてそのまま翻訳されている言葉も多いのですから、それこそ、少し慣れればこの問題は解消されるような気がします。さらには、ブラウザによっては、単語にカーソルを合わせれば翻訳をしてくれるツールもあるなど、もはや言語は言い訳にしてはいけない環境に到達しているはずです。
次に思い浮かぶのは、会社で利用している環境にインストールするには、各種の申請が必要になる場合があるということです。私物の環境にインストールしたとしても、実際に適用したい対象は、会社から持ち出し不可能で、あまり効率的でない場合もあります。また、ツールをインストールしたいという強い動機がない場合もあります。ただ何となく興味はもったものの、どうしても利用したいと言うほどのものではないという場合です。この場合は、もう一度、自分が何をしたいのかを問い直す必要があるのでしょう。
ここまで、ツールを使い出せない理由を考えてみましたが、いずれも、ちょっと心境を変化させることができれば、難しい問題ではないはずです。この文章を読んでくださっているということは、本書を最後まで読んでしまったのですから、お互い一歩を踏み出してみましょう。

[第５章執筆者のコメント]

ツール選びには仕様記述に取り組む本人の事情、開発組織の事情、ツールの特性、いろいろな要素が絡み合います。全てを完全に満足させるツールというものがあれば素晴らしいことですが、完全に条件が合うツールというものはなかなかありません。まずは第一歩を踏み出す、その第一歩が完全でなくても構わないと思います。いろいろなツールの第一歩を踏み出してみましょう。あなたが手応えのある第一歩に出会えることを願っています。

付録１ 仕様作成の流れ

このページに示した図は第２章に掲載したものに少し手を加えたものです。後からクイックレファレンス的に利用して貰えることを想定しています。



「課題（事実＋願望）の管理」と書かれた部分では外部とのコミュニケーションを取りながら、さまざまな課題が抽出定義されています。この内容を基に、課題の世界における価値を生み出すための仕様が「仕様の策定」という場所で定義され、続けて「仕様の確認と検証」により妥当性確認と正当性検証が行われます。

上図で示した（１），（２），（３）は厳密な仕様記述により改善されるコミュニケーションを表しています。それぞれ（１）プロジェクト外部とのやりとり、（２）プロジェクト次工程への連携、（３）仕様策定者間のコミュニケーションです。

付録２ 例題ファイル

以下に第２章で使った例題の VDM++ による記述ファイルを掲載します。VDMTools に読み込んで実際に関数の評価などを行うことが可能です。

血縁.vdmpp

```
--
-- 例題 : 血縁.vdmpp
-- 血縁関係の「指示」と「定義」を記述するための型と関数を含む
--
class 血縁

types
  public 文字列型 = seq of char;
  public 日付型 = int;
  public 性別型 = <男>|<女>;

  public 人型 :: 名前 : 文字列型
                 誕生日 : 日付型
                 性   : 性別型
                 父親   : [人型]
                 母親   : [人型];

functions

  public 親 : 人型 -> set of 人型
  親(x) == {x.父親, x.母親}¥{nil};

  public 子 : set of 人型 * 人型 -> set of 人型
  子(allp, x) == {c | c in set allp & 親子？(x, c)};

  public 孫 : set of 人型 * 人型 -> set of 人型
  孫(allp, x) == dunion {子(allp, c) | c in set 子(allp, x)};

  public 曾孫 : set of 人型 * 人型 -> set of 人型
  曾孫(allp, x) == dunion {子(allp, c) | c in set 孫(allp, x)};

  public 全子孫 : set of 人型 * 人型 -> set of 人型
  全子孫(allp, x) ==
    let children = 子(allp, x) in
      children union dunion {全子孫(allp, c)| c in set children};
```

```
  public 男？ : 人型 -> bool
  男？(x) == x.性 = 〈男〉;

  public 女？ : 人型 -> bool
  女？(x) == x.性 = 〈女〉;

  public 年長？ : 人型 * 人型 -> bool
  年長？(x, y) == x.誕生日 < y.誕生日;

  public 親子？ : 人型 * 人型 -> bool
  親子？(p, c) == (p = c.父親) or (p = c.母親);

  public 兄弟姉妹？ : 人型 * 人型 -> bool
  兄弟姉妹？ (x, y) ==
    x <> y and
    exists p in set {x.父親, y.父親, x.母親, y.母親}
      & 親子？(p, x) and 親子？(p, y);

  public 伯父？ : 人型 * 人型 -> bool
  伯父？(x, y) == let parents = 親(y)
                  in exists p in set parents
                    & (年長？(x, p) and 男？(x) and 兄弟姉妹？(x, p));

  public 叔父？ : 人型 * 人型 -> bool
  叔父？(x, y) == let parents = 親(y)
                  in exists p in set parents
                    & (年長？(p, x) and 男？(x) and 兄弟姉妹？(x, p));

end 血縁

--
-- 「源氏」クラス。血縁を継承し個別の個人データを定義している。
--
class 源氏 is subclass of 血縁

values

  源義朝   = mk_人型("源義朝", 11230000, 〈男〉, nil, nil);
  由良御前 = mk_人型("由良御前", 11320000, 〈女〉, nil, nil);
  常磐御前 = mk_人型("常磐御前", 11380000, 〈女〉, nil, nil);
  源頼朝   = mk_人型("源頼朝", 11470509, 〈男〉, 源義朝, 由良御前);
  源義経   = mk_人型("源義経", 11590000, 〈男〉, 源義朝, 常磐御前);
  北条時政 = mk_人型("北条時政", 11380000, 〈男〉, nil, nil);
  伊東祐親の長女 = mk_人型("伊東祐親の長女", 0, 〈女〉, nil, nil);
  北条政子 = mk_人型("北条政子", 11570000, 〈女〉, 北条時政, 伊東祐親の長女);
```

```
  大姫      = mk_人型("大姫", 11780000, <女>, 源頼朝, 北条政子);
  源頼家    = mk_人型("源頼家", 11820911, <男>, 源頼朝, 北条政子);
  若狭局    = mk_人型("若狭局", 0, <女>, nil, nil);
  足助重永の長女 = mk_人型("足助重永の長女", 0, <女>, nil, nil);
  一幡      = mk_人型("一幡",  11980000, <男>, 源頼家, 若狭局);
  公暁      = mk_人型("公暁",  12000000, <男>, 源頼家, 足助重永の長女);
  源実朝    = mk_人型("源実朝", 11920917, <男>, 源頼朝, 北条政子);
  静御前 = mk_人型("静御前", 0, <女>, nil, nil);
  源義経の長男 = mk_人型("源義経の長男", 11860729, <男>, 源義経, 静御前);

  登場人物 : set of 人型 = {
     源義朝,
     由良御前,
     常磐御前,
     源頼朝,
     源義経,
     北条時政,
     伊東祐親の長女,
     北条政子,
     大姫,
     源頼家,
     若狭局,
     足助重永の長女,
     一幡,
     公暁,
     源実朝,
     静御前,
     源義経の長男
  };

 -- 以下は評価用関数ならびに操作

functions
  -- 義経は頼家の叔父？
  public ft0 : () -> bool
  ft0() == 叔父？(源義経, 源頼家);

  -- 義経は頼家の伯父？
  public ft1 : () -> bool
  ft1() == 伯父？(源義経, 源頼家);

  -- 頼朝と義経は兄弟？
  public ft2 : () -> bool
  ft2() == 兄弟姉妹？(源頼朝, 源義経);

  -- 源義朝の孫は？
```

```
  public ft3 : () -> set of 文字列型
  ft3() == {p.名前 | p in set 孫(登場人物, 源義朝)};

  -- 源義朝の曾孫は？
  public ft4 : () -> set of 文字列型
  ft4() == {p.名前 | p in set 曾孫(登場人物, 源義朝)};

  -- 源義朝の全子孫は？
  public ft5 : () -> set of 文字列型
  ft5() == {p.名前 | p in set 全子孫(登場人物, 源義朝)};

  public ft6 : () -> set of 文字列型
  ft6() == {p.名前 | p in set 親(源義朝)};

  -- 頼朝と頼朝は兄弟？
  public ft7 : () -> bool
  ft7() == 兄弟姉妹？(源頼朝, 源頼朝);

operations
  -- 義経は頼家の叔父？
  public t0 : () ==> bool
  t0() == return 叔父？(源義経, 源頼家);

  -- 義経は頼家の伯父？
  public t1 : () ==> bool
  t1() == return 伯父？(源義経, 源頼家);

  -- 頼朝と義経は兄弟？
  public t2 : () ==> bool
  t2() == return 兄弟姉妹？(源頼朝, 源義経);

  -- 源義朝の孫は？
  public t3 : () ==> set of 文字列型
  t3() == return {p.名前 | p in set 孫(登場人物, 源義朝)};

  -- 源義朝の曾孫は？
  public t4 : () ==> set of 文字列型
  t4() == return {p.名前 | p in set 曾孫(登場人物, 源義朝)};

  -- 源義朝の全子孫は？
  public t5 : () ==> set of 文字列型
  t5() == return {p.名前 | p in set 全子孫(登場人物, 源義朝)};

end 源氏
```

Reservation.vdmpp

```
--
-- 予約アプリケーションの仕様。
-- 列車仕様と予約仕様を継承してとりこんでいる。
-- このままでは、具体的な路線情報が定義されていないので評価できない。
-- 評価可能な仕様にするためには、小田急線路線仕様などの具体的な情報を
-- とりこむ必要がある。
-- 先の方に定義した AppTest クラスがそうした具体化の例である。
--
class 予約app is subclass of 列車仕様, 予約仕様

types
  public 人数型 = int;

functions

  public 空席有 : set of 予約型 * 列車型 * 駅型 * 駅型 * 人数型 -> bool
  空席有(rsv, tr, fs, ts, np) == is not yet specified;

  -- 指定された区間でずっと空いている席を計算する。
  -- もとい。指定された区間でずっと空いている席の「集合」を計算する。
  -- 各予約に対して、指定した区間で確保している席を答えさせて、
  -- それを union して全座席から取り除く。
  -- 答は「ある区間でずっと空いている席」になるはず
  public 空席集合 : 路線型 * set of 予約型 * 列車型 * 駅型 * 駅型
                          -> map 車両番号型 to set of 座席型
  空席集合(gr, rsv, tr, fs, ts) ==
    let aas = 全席集合(tr), ars = 予約席集合(gr, rsv, tr, fs, ts) in
        aas ++ { cn |-> aas(cn)¥ars(cn) | cn in set dom ars };

  public 全席集合 : 列車型 -> map 車両番号型 to set of 座席型
  全席集合(tr) == { i |-> tr.編成(i) | i in set inds tr.編成 };

  -- ある列車に対する現在の予約群が指定された区間で確保している席を答える
  public 予約席集合 : 路線型 * set of 予約型 * 列車型 * 駅型 * 駅型
                          -> map 車両番号型 to set of 座席型
  予約席集合(gr, rsv, tr, fs, ts) ==
     let 関連列車予約 =
       {r | r in set rsv &
         r.列車番号 = tr.列車番号 and
         len 重複区間(最短経路(gr,fs,ts), 最短経路(gr,r.予約発駅,r.予約着駅)) > 1} in
           {cno |-> dunion {r.座席番号 | r in set 関連列車予約 & r.車両番号 = cno}
                           | cno in set {r2.車両番号|r2 in set 関連列車予約}};

  -- 区間１に対して、区間２が重なっている範囲の列を答える
```

```
  public 重複区間 : seq of 駅型 * seq of 駅型 -> seq of 駅型
  重複区間(区間1, 区間2) ==
      [区間1(i) | i in set inds 区間1 & 区間1(i) in set elems 区間2];

  -- ある列車の停車駅列を求める
  public 停車駅列 : 路線型 * 列車型 -> seq of 駅型
  停車駅列(gr, tr) == 最短経路(gr,tr.発駅,tr.着駅);

  -- ある列車の停車駅集合を求める
  public 停車駅集合 : 路線型 * 列車型 -> set of 駅型
  停車駅集合(gr, tr) == elems 停車駅列(gr, tr);

  -- 空席の中から、人数分の座席を確保し、新しい予約を作る
  -- 全予約に新しい予約を追加したものも返す
  -- 空席がない場合には 予約には nil を返す
  public 新規予約 : 路線型 * set of 予約型 * 列車型 * 駅型 * 駅型 * 人数型
           -> [予約型] * set of 予約型
  新規予約(gr, rsv, tr, fs, ts, np) ==
    let 対象空席 = 空席集合(gr, rsv, tr, fs, ts) in
      if exists cno in set dom 対象空席 & card 対象空席(cno) >= np then
         let cno in set dom 対象空席 be st card 対象空席(cno) >= np in
           let ns in set power 対象空席(cno) be st card ns = np in
             let 新 = mk_予約型(tr.列車番号, fs, ts, cno, ns) in
                 mk_(新, rsv union {新})
      else
         mk_(nil, rsv)
  pre
    np > 0
  post
    let mk_(新予約, 新全予約) = RESULT in
      if 新予約 <> nil then
        新予約 not in set rsv and 新全予約 = {新予約} union rsv and
        新予約.列車番号 = tr.列車番号 and
        新予約.予約発駅 = fs and 新予約.予約着駅 = ts and
        card 新予約.座席番号 = np
      else
        新全予約 = rsv;

functions -- テスト

instance variables

  protected
  全予約 : set of 予約型 := {};

  protected
```

```
  全路線 : 路線型;

operations

  public 新規予約操作 : 列車型 * 駅型 * 駅型 * 人数型 ==> [予約型]
  新規予約操作(a列車, a発駅, a着駅, a人数) ==
    let mk_(nr, nrsv) = 新規予約(全路線, 全予約, a列車, a発駅, a着駅, a人数) in
        (全予約 := nrsv;
         return nr)
  pre
    let 駅集合 = 停車駅集合(全路線, a列車) in
      a発駅 in set 駅集合 and
      a着駅 in set 駅集合 and
      a人数 > 0
  post
    let 新予約 = RESULT in
      if 新予約 <> nil then
        新予約 not in set 全予約~ and
        全予約 = {新予約} union 全予約~ and
        新予約.列車番号 = a列車.列車番号 and
        新予約.予約発駅 = a発駅 and 新予約.予約着駅 = a着駅 and
        card 新予約.座席番号 = a人数
      else
        全予約 = 全予約~;

end 予約app

--
-- テスト用アプリケーションの仕様
-- 予約アプリケーションの仕様と小田急路線仕様を取り込んで定義されている。
--
class AppTest is subclass of 予約app, 小田急路線仕様

instance variables

  protected
  列車1 : 列車型 := mk_列車型(1, "新宿", "小田原", [{1,2,3,4,5,6},{1,2,3,4,5,6}]);

  protected
  列車2 : 列車型 := mk_列車型(2, "向ヶ丘遊園", "藤沢", [{1,2,3,4,5,6},{1,2,3,4,5,6}]);
operations

  public AppTest : () ==> AppTest
  AppTest() == (
    全予約 := {mk_予約型(1, "新宿", "町田", 1, {3,4}),
               mk_予約型(1, "新百合ヶ丘", "相模大野", 1, {5,6}),
```

```
            mk_予約型(1, "新百合ヶ丘", "相模大野", 2, {1,2})
            };
    全路線 := {
               {"新宿", "向ヶ丘遊園"} |-> 5,
               {"向ヶ丘遊園", "新百合ヶ丘"} |-> 3,
               {"新百合ヶ丘", "小田急多摩センター"} |-> 3,
               {"新百合ヶ丘", "町田"} |-> 3,
               {"町田", "相模大野"} |-> 1,
               {"相模大野", "藤沢"} |-> 4,
               {"藤沢", "片瀬江ノ島"} |-> 2,
               {"相模大野", "本厚木"} |-> 4,
               {"本厚木", "小田原"} |-> 6
            }
  );

  -- 状態を変えない
  public t00 : 駅型 * 駅型 ==> map 車両番号型 to set of 座席型
  t00(fs, ts) == return 予約席集合(全路線, 全予約, 列車1, fs, ts);

  public t01 : 駅型 * 駅型 ==> map 車両番号型 to set of 座席型
  t01(fs, ts) == return 空席集合(全路線, 全予約, 列車1, fs, ts);

  -- 状態を変える
  public t02 : 駅型 * 駅型 * 人数型 ==> [予約型]
  t02(fs, ts, np) == return 新規予約操作(列車1, fs, ts, np);

end AppTest

--
-- 路線に関する仕様
--
class 路線仕様 is subclass of GraphDomain

types
  public 駅型 = Name;
  public 路線型 = Graph;
  public 座席型 = int;
  public 車両番号型 = int;
  public 列車番号型 = int;

functions

 public ターミナル駅？ : 路線型 * 駅型 -> bool
 ターミナル駅？(gr, s) == isTerminalNode(gr, s);

 public 最短経路 : 路線型 * 駅型 * 駅型 -> seq of 駅型
```

```
 最短経路(gr, fs, ts) == ShortestPath(gr, fs, ts);

 public 全駅 : 路線型 -> set of 駅型
 全駅(gr) == Nodes(gr);

end 路線仕様

--
-- 列車に関する仕様
--
class 列車仕様 is subclass of 路線仕様
types
  public 車両型 = set of 座席型;
  public 編成型 = seq of 車両型;

  public 列車型 ::
           列車番号 : 列車番号型
           発駅 : 駅型
           着駅 : 駅型
           編成 : 編成型;

end 列車仕様

--
-- 予約の型に関する仕様
--
class 予約仕様 is subclass of 路線仕様
types
  public 予約型 ::
           列車番号 : 列車番号型
           予約発駅 : 駅型
           予約着駅 : 駅型
           車両番号 : 車両番号型
           座席番号 : set of 座席型
         inv r == card r.座席番号 >= 1 and card r.座席番号 <= 4;

end 予約仕様

--
-- 小田急線の路線仕様を「路線仕様」を継承する形で定義
--
class 小田急路線仕様 is subclass of 路線仕様

values

  protected 小田急路線 : 路線型 = {
```

```
                {"新宿", "向ヶ丘遊園"} |-> 5,
                {"向ヶ丘遊園", "新百合ヶ丘"} |-> 3,
                {"新百合ヶ丘", "小田急多摩センター"} |-> 3,
                {"新百合ヶ丘", "町田"} |-> 3,
                {"町田", "相模大野"} |-> 1,
                {"相模大野", "藤沢"} |-> 4,
                {"藤沢", "片瀬江ノ島"} |-> 2,
                {"相模大野", "本厚木"} |-> 4,
                {"本厚木", "小田原"} |-> 6
              };

functions -- テスト

  public ft0 : () -> SPs
  ft0() == SPF(小田急路線, "新宿");

  public ft1 : () -> seq of 駅型
  ft1() == 最短経路(小田急路線, "新百合ヶ丘", "藤沢");

  public ft2 : () -> seq of bool
  ft2() == [ターミナル駅？(小田急路線, "新宿"),
            ターミナル駅？(小田急路線, "本厚木"),
            ターミナル駅？(小田急路線, "小田原")];

  public ft3 : () -> set of SP
  ft3() == APF(小田急路線, "新宿", "藤沢");

end 小田急路線仕様


--
-- グラフを扱うためのクラス
-- SPF (shortest path finder) はダイクストラ法の素直な実装
--
class GraphDomain

 types
  public Name = seq of char;
  public Graph = map set of Name to real;

functions -- graph basics
  public Nodes : Graph -> set of Name
  Nodes(gr) == dunion dom gr;

  public areNeighbors : Graph * Name * Name -> bool
  areNeighbors(gr, a, b) == {a, b} in set dom gr;
```

```
  public Neighbors : Graph * Name -> set of Name
  Neighbors(gr, a) == dunion {d¥{a}|d in set dom gr & a in set d};

  public NearestNeighbor : Graph * Name -> Name
  NearestNeighbor(gr, a) ==
         let neighbors = Neighbors(gr, a) in
           let b in set neighbors
             be st gr({a,b}) = Min({gr({a,t})|t in set neighbors})
             in b;

  public isTerminalNode : Graph * Name -> bool
  isTerminalNode(gr, a) == card {d | d in set dom gr & a in set d} = 1;

functions -- utils
  public Min : set of real -> real
  Min(rs) == let r in set rs in
               if card rs = 1 then r
               else
                  let r2 = Min(rs¥{r}) in
                      if r < r2 then r else r2;

types -- algorithm
  -- shortest path
  public SP   :: path : seq of Name
                 cost : [real];
  -- shortest path map
  public SPs = map Name to SP;

functions -- algorithm
  -- all paths finder
  public APF : Graph * Name * Name -> set of SP -- set of SP
  APF(gr, s, e) == APF2(gr, e, {mk_SP([s], 0)}, {});

  public APF2 : Graph * Name * set of SP * set of SP -> set of SP -- set of SP
  APF2(gr, e, cps, aps) ==
    if cps = {} then
      aps
    else
      let rps = dunion
                {{mu(sp, path |-> sp.path ^ [n],
                         cost |-> sp.cost + gr({sp.path(len sp.path),n}))
                 | n in set Neighbors(gr,sp.path(len sp.path))¥(elems sp.path)}
                | sp in set cps}
      in let naps = {sp|sp in set rps & sp.path(len sp.path) = e} in
          APF2(gr, e, rps¥naps, aps union naps);
```

```
-- pick shortest path between two nodes
public ShortestPath : Graph * Name * Name -> seq of Name
ShortestPath(gr, fn, tn) == let sps = SPF(gr, fn) in
                                    if tn in set dom sps then
                                       sps(tn).path
                                    else
                                       [];

-- shortest paths finder
public SPF : Graph * Name -> SPs -- seq of Name
SPF(gr, s) == let sps : SPs =
                 {a |-> if a = s then
                             mk_SP([s],0)
                        else
                             mk_SP([s],nil) | a in set Nodes(gr)}
                 in SPF2(gr, sps, {});

public SPF2 : Graph * SPs * set of Name -> SPs -- seq of Name
SPF2(gr, sps, ns) ==
              let nodes = Nodes(gr)¥ns in
                if nodes = {} then
                  sps
                else
                  let node in set nodes
                    be st sps(node).cost
                            = Min({sps(n).cost | n in set nodes
                                               & sps(n).cost <> nil})
                  in
                    let nbs = Neighbors(gr, node)
                    in
                      let nsps =
                        sps ++ {b |->
                          if sps(b).cost = nil
                             or sps(b).cost > sps(node).cost + gr({node,b})
                          then
                                 mu(sps(b),
                                 cost |-> sps(node).cost + gr({node,b}),
                                 path |->  sps(node).path ^ [b])
                          else
                                 mu(sps(b),
                                 cost|->sps(b).cost,
                                 path|->sps(b).path)
                                | b in set nbs}
              in SPF2(gr, nsps, ns union {node});
```

```
end GraphDomain
```

索引

実務家のための形式手法
厳密な仕様記述を志すための形式手法入門

厳密な仕様記述入門

2013 年 3 月　初版発行

独立行政法人情報処理推進機構
技術本部　ソフトウェア・エンジニアリング・センター
統合系システム・ソフトウェア信頼性基盤整備推進委員会
上流品質技術部会　厳密な仕様記述ＷＧ委員

主査	佐原　伸	（SCSK 株式会社、法政大学）まえがき執筆
副主査	栗田　太郎	（フェリカネットワークス株式会社）第４章執筆
委員	荒木　啓二郎	（国立大学法人九州大学）推薦の言葉執筆
	岩崎　孝司	（富士通九州ネットワークテクノロジーズ株式会社）
	漆原　憲博	（株式会社ジェーエフピー）
	大森　洋一	（国立大学法人九州大学）第３章執筆
	小田　朋宏	（株式会社 SRA）第５章執筆
	酒匂　寛	（有限会社デザイナーズ・デン） 読書の手引き、第１～２章、付録執筆
	幡山　五郎	（オムロンソーシアルソリューションズ株式会社）
	平田　貞代	（富士通株式会社、芝浦工業大学）
	水口　大知	（独立行政法人産業技術総合研究所）
オブザーバ	木下　聡子	（慶應義塾大学 SDM 研究科学生）章末コラム執筆
	日下部　茂	（国立大学法人九州大学）
	佐々木　千春	（株式会社ジェーエフピー）
	藤本　宏	（株式会社東芝）
	村田　由香里	（株式会社東芝）
事務局	新谷　勝利	
	神谷　慎吾	
	藤原　由起子	

SEC-FM1-A-1