ソフトウェア工学

東京大学大学院総合文化研究科
玉井哲雄

平成 15 年 9 月 11 日

目 次

第1章　ソフトウェアとソフトウェア工学

1.1　ソフトウェアとは

ソフトウェアという語は一般に定着したが，コンピュータ・ソフトウェアだけでなく放送番組や記録された音楽・映像などを指すものとして使われることも多い．実際，Web の検索でソフトウェアを探してみると，上位に「NTT ソフトウェア (株)」と並んで「(株)NHK ソフトウェア」という会社がランクされ，その事業内容は「NHK 放送番組のビデオ・音声商品，キャラクターグッズ，CD-ROM，DVD などの通信販売を行っています」と説明されている．

本書で扱うのはもちろん，コンピュータのソフトウェアである．しかし，インターネットの発達と，音楽や映像記録のデジタル化により,「NHK ソフトウェア」が扱うようなソフトウェアとコンピュータ・ソフトウェアとがきわめて近づいていることも確かである．そもそも現代のいわゆるフォン・ノイマン型コンピュータの一大特徴は，操作の対象となるデータと操作を記述したプログラムとを同一の形で扱うことにあるから，この両者の境界が不分明となってくるのは自然なことである．

このソフトウェアという言葉を誰がいつ言い始め，それがどのように普及したのだろうか．これが実はよく判っていない．ハードウェアという言葉が先にあって，そこからソフトウェアという言葉が作られたことは，明らかである．辞書によれば，ware とは細工物，製品といった意味で，kitchenware や ironware のように他の語幹にくっついて使われることが多いらしい．

比較的最近まで，オックスフォード英語辞典では software という語の最初の用例として 1960 年のものを挙げていた．それより前の 1958 年に，プリンストン大学教授の John W. Tukey が書いて米国数学月報（American Mathematical Monthly)1 月号に掲載された論文の中に software という語が使われているのを，エール大学法学大学院司書の Fred R. Shapiro が発見した [109]．Tukey は高速フーリエ変換 FFT の開発者としても知られるが，bit という語の創始者であるという説があり，それにはかなり確かな根拠があるため，software という語を作ったのが Tukey であってもおかしくはない．しかし，現在のところは彼が最も早くコンピュータ用語としての「ソフトウェア」を使ったことが判明している著者である，というのが正確だろう．

コンピュータ・ソフトウェアは，プログラミング言語という人工言語で書かれる．そのような抽象的な記述でありながら，コンピュータを通して実世界に直接働きかけることができる．つまりソフトウェアは，言語表現による創作結果という意味で小説や論文などの作品と類似する面を持ちながら，実世界への直接的な作用という機能から実利的な工業製品として生産されるという不思議な性質を持つ．そこで，これを製作するプロセスは，言語による表現行為という面と，人工物の設計開発という面とを併せ持つ．

プログラミング言語による記述結果でコンピュータ上で動作するもの自体は，通常プログラムと呼ばれる．プログラムという語とソフトウェアとの使い分けは，前者が具体的にコンピュータで実行されるべき記号列のまとまりを指すのに対し，後者はハードウェアに対比させて議論する場合のように，総称として使われるところにある．だから英語の software は抽象名詞で，softwares という複数形はない．それに対しプログラムは数えることができて，programs という複数形が存在する．ソフトウェアに関連した分野の人でも，英語を母国語としていないと往々にしてこれを間違えるので注意しておく[1]．

1.2　ソフトウェア工学とは

このようなソフトウェアをどう作ればよいかは，現代社会に与えられた大きなチャレンジといえよう．本書の目的は，このようなソフトウェアという対象の持つ独自の性質に注目しながら，それを創るプロセスや手法を探求

[1]同様に information という語にも informations という複数形はない．この誤りも時々見かけるので，ついでに注意しておく．

することにある．とくにソフトウェアが人工的な構造物である点を重視し，これを構築するためのエンジニアリングとして捉える立場をソフトウェア工学という．

ソフトウェア工学の定義として標準的なものは，IEEE Std 610-1990 の次の記述であろう．

> (1) ソフトウェアの開発，運用，保守に対する，系統的で統制され定量化可能な方法．すなわちソフトウェアへの工学の適用．
> (2)(1) のような方法の研究．

工学の中で，ソフトウェア工学はもちろん後発である．伝統的な工学分野の代表である機械工学と電気工学に比較してみると，工学としての開発対象であるソフトウェアを直接明示するソフトウェア工学という命名の仕方は，電気工学よりも機械工学に近いかもしれない．機械工学が機械を作り運用保守する技術であると同様に，ソフトウェア工学はソフトウェアを作り運用保守する技術である．それに対し，電気工学を電気を作り運用保守する技術であると規定するのは，適切とはいえまい．

一方で，ソフトウェアという対象が，機械はもちろん，電気のような物理性を持たず，抽象的なものである点は，一般の工学と異なる大きな特徴である．その抽象性のために，ソフトウェア工学は理学としての計算機科学や数学との距離が近い．さらにソフトウェアの開発は，プログラミング言語による記述や要求仕様・設計の記述という点で，言語表現行為に近いという特性もあり，そのために人文科学とも近親性を持つ．実際，よいプログラムを書く作法を説く本は,「文章読本」の類によく似ている [65].

ソフトウェア工学が他の工学分野と共通する性質をもつことは，以下のような作業を共有することからも見ることができる．

1. モデル化
   なにが問題で，どんなシステムを作るべきかは，最初から明確に与えられるわけではない．対象領域を分析し，問題を発見し，利用者の要求を正確にとらえて，モデル化する技術が重要である．
2. 仕様
   工学はきちんとした仕様を記述することが開発作業の大前提である．
3. 設計
   工学の核は設計技術である．
4. 検証
   仕様を正しく満たすシステムが作られたことを検証しなければならない．
5. 保守
   システムは作りっぱなしでは役に立たない．システムを保守し，環境変化に適応させていく努力が必要である．
6. 組織
   システムは通常一人で作れるような規模ではなく，組織として開発するためのマネジメント技術が必要である．

実際，ソフトウェア工学は上に挙げた「モデル」「仕様」「設計」「保守」を初めとする多くの用語と概念を，既存の工学から借りてきている．

工学として何より重要な特性は，製品としての人工物を作るという点であろう．ソフトウェアを一人で作り自らが使用している分には,「工学」は必要ない．しかし，製品として利用者に，また広く市場に提供するソフトウェアは，一人で作ることはきわめて難しく組織で作らざるをえない．ところがソフトウェアはその抽象度の高さから，数学理論や音楽・文学・美術などの作品に近い面がある．そのようなものを大規模な組織によって開発するという作業は，従来，人類が経験しなかったものといってもよい．実際そこでは，次の 3 つの側面を考慮しなければならない．

- 抽象性の高いソフトウェアを開発し保守するための純粋に技術的な側面,

- 組織的に開発し管理するための管理的側面,
- 利用者の満足度を上げ，また開発者のチーム内の協力体制を築き士気を上げるためのコミュニケーションや認知といった人間的側面.

1.3 ソフトウェア工学の歴史

ソフトウェアの開発という作業は，現代の電子式計算機の誕生とほぼ同時に始まった．それでも最初の電子計算機 ENIAC(1946 年) の場合，プログラミングは電線を繋ぎかえるという原始的な方法でなされた．プログラム内蔵式コンピュータという設計を発明した功績がすべて John von Neumann に帰せられるかどうかについては異論もあるが，ノイマン式コンピュータという名称が今日でも使われていることからも判るように，von Neumann が 1945 年に書いた「EDVAC 報告書の一次稿」はその後のコンピュータの開発に与えた影響は大きい．その一次稿の付録に現代コンピュータ向けに書かれた最初のプログラムが載せられている．

最初の商用計算機 UNIVAC が米国国勢調査局に納入されたのは 1951 年であるが，それ以来コンピュータは多様な分野で使われるようになり，多くのプログラムが開発された．1964 年に発表された IBM の System/360 は，その汎用性と上位から下位機種までの互換性によって，ソフトウェアの有用性を高めた．またそのオペレーティングシステムである OS/360 自身が，アセンブラの原始コードで 5 百万行という規模を持つ現在の基準で見てもすでに巨大なソフトウェアであった．

こうしてソフトウェアに対する社会的な需要は増大したが，それを供給する生産技術と体制が需要に追いつかず，開発スケジュールは常に遅れ，作られたソフトウェアの品質が低いという状況が 1960 年代の後半に顕著になった．この状況は「ソフトウェア危機」と呼ばれるようになる．

ソフトウェア危機に対処すべく，ソフトウェア工学という言葉が生まれ，また分野として認知された時を特定することができる．それはソフトウェア工学というテーマを初めて掲げて，1968 年にドイツのガーミッシュ(Garmisch) で開かれた NATO 主催の国際会議である．それを受け，1970 年代のソフトウェア工学は，構造化プログラミングという主導概念のもとに理論面と実践面の相補的関係がかなりよく機能して，活発な研究，実践活動を展開した．この時代にプログラミング，設計方法論，要求分析手法の基盤ができあがったと言える．

1980 年代に入るとソフトウェアの開発需要は飛躍的に増大し続けるにもかかわらず，ソフトウェア工学が提供する方法論や道具 (tool) が生産性や信頼性の向上にもたらす効果は，未だ部分的でしかないことが意識され始めた．構造化プログラミングに代わる強力な指導原理は現れず，理論と実践との乖離が生じてくる．

開発されるソフトウェアの規模は，巨大化した．原始プログラムの行数で数百万行というのはもはや珍しくなく，数千万行というものが出てくる．このようなソフトウェアの開発保守では，技術面より管理面や人間的な側面がより重大な問題となる．このことから，70 年代のソフトウェア工学の枠内で進められた，段階的詳細化などのプログラミング方法論，モジュール化，抽象データ型，などの分野は，次第にソフトウェア工学から外れていった．代わりに品質管理，構成管理などの管理技術や，開発環境の構築に重点がおかれる傾向がみられた．

80 年代における実践的成果には，査閲 (inspection) やプロトタイピング (prototyping) があり，これらはそれなりの効果をあげた．しかし，技術的な革新性という点での魅力には欠ける．80 年代の後半にはソフトウェアプロセスが多くの関心を呼んだ．プロセスは，CMM や ISO9000 を通して，実務者の興味をもひくことにある程度成功した．

1990 年代はオブジェクト指向技術が注目を集めた．70 年代の構造化技術がプログラミングから設計および分析技術へと範囲を広げ大きなインパクトを与えたように，オブジェクト指向技術も 80 年代初めに Smalltalk の言語仕様が公開され，オブジェクト指向プログラミングが脚光を浴びるようになってから，90 年前後にオブジェクト指向設計やオブジェクト指向分析手法が次々と提案されるという経緯をたどった．70 年代の構造化「パラダイム」に匹敵するオブジェクト指向「パラダイム」が，ソフトウェア工学分野を席巻したようにも見える．

一方で，90 年代のソフトウェアは 70 年代のそれと比べて遥かに多様であった．PC とインターネットの普及に伴い，ソフトウェアに開放化，分散化，小型化，多媒体化の波が押し寄せたが，従来から「大規模・複雑」なシステムにばかり目を向けてきたソフトウェア工学は，この流れに乗り遅れた感もある．90 年代はやはり 70 年代ほど単純な時代ではなく，オブジェクト指向という単一の枠組みですべてが解決するはずもない．しかし，これによりソフトウェア工学がある程度の求心力を取り戻したということは認められる．その後，オブジェクト指向技術は,

パターンやフレームワークといったオブジェクトより大きい単位の再利用技術，コンポーネントベースのソフトウェア開発，Web サービス技術などといった新たなテーマを展開させている．

21 世紀に確実にいえることは，ソフトウェアの社会的需要と依存度が益々高まることであろう．その意味で，誕生から 30 年以上経ったソフトウェア工学の必要性がさらに増大することは当然の帰結である．拡大し多様化するソフトウェアのニーズに適応して，ソフトウェア工学を進化発展させまたよく普及させていくことが社会的に要請されている．

1.4　ソフトウェア工学の範囲

ソフトウェア工学の対象範囲を標準的に定める努力は，様々な形で進められてきた．とくに米国ではソフトウェア工学の標準的な教科書として，I. Sommerville[100], R. S. Pressman[89], S. L. Pfleeger[86] などのテキストが比較的よく読まれて版を重ねている．そこで取り上げられている項目と配列も多かれ少なかれ一致し，ソフトウェア工学が扱う対象範囲の見取り図を与えている．一方，ソフトウェア工学の知識体系を整理して記述することを目的とした IEEE の Computer Society と ACM の共同作業があり，途中で ACM が手を引くというような紆余曲折をたどりながらも，SWEBOK と呼ぶ文書として草案を公開しては改訂を行うという形で進捗している [3]．さらに，計算機科学の標準カリキュラムを作成する活動の中で，ソフトウェア工学も大きな分野として取り上げられている．日本では情報処理学会が発表している標準カリキュラム J97 の中に，ソフトウェア工学が含まれている．米国では，やはり IEEE Compupter Society と ACM がそれぞれ標準カリキュラムを作ってきたが，1991 年に出された両者共同作成の CC’91 の後継として，CC2001 の作成作業が進められ，その中でソフトウェア工学が 1 つの独立した巻として編集されている [30].

1.5　ソフトウェア工学の工学としての成熟度

カーネギーメロン大学の Mary Shaw が，ソフトウェア工学の位置づけを他の工学との関連で論じている [94]. どんな分野にせよ，およそ技術はまず生産と工芸とがそれぞれ生まれ，その両者が結び付いて商用化の段階に入り，さらに科学と結び付いて始めて専門化した工学として成立するという．この理論に従えば，土木工学は材料属性や橋梁の構造力学が明らかにされた 18 世紀から 19 世紀前半に成立し，化学工学は単位操作 (unit operation) の概念が提唱され体系化された 20 世紀初頭に成立したと見られる．土木工学の例を図 1.1 に示す.

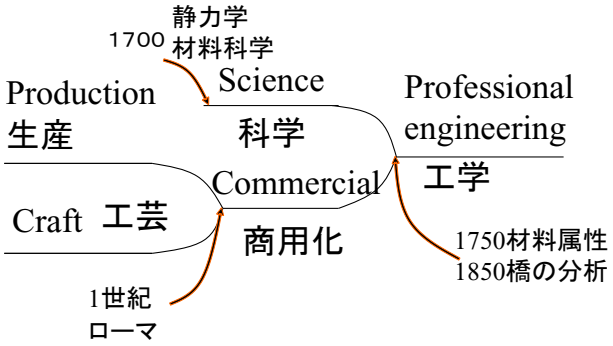


図 1.1: 土木工学の成熟プロセス

それと比べると，ソフトウェアの場合コンパイラなどきわめて限定された個別分野でようやく工学の段階に入ったところだというのが Shaw の主張である．

これ以前にも，ソフトウェア工学の成果に対してきわめて悲観的な評価を示した論文が，ソフトウェア工学の指導的な立場にあると目される人達によって書かれ，大きな衝撃を与えたことがある．

その一つは，D. Parnas による SDI (Strategic Defense Initiative, いわゆるスターウォーズ計画) 用のソフトウェア開発が技術上不可能であることを論じたもの [84] である．Parnas はレーガン政権時代のスターウォーズ計画の推

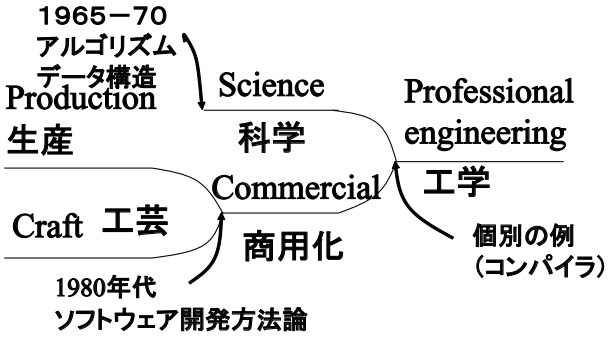


図 1.2: ソフトウェア工学の成熟度

進組織である戦略的防衛発案組織 (Strategic Defense Initiative Organization) の下のコンピュータ支援戦闘管理専門委員であったが，その SDI の核となるソフトウェアの開発が技術的に不可能であるという意見を公表し，委員を辞任した．これは，その理由を述べた文書である．彼はまずソフトウェア工学と他の工学とを比較し，ソフトウェアの信頼性がなぜ低いかを論じている．その論拠は，およそシステムには離散的な状態を持つデジタルシステムと，連続的な状態を持つアナログシステムとがあり，ソフトウェアシステムは前者に属す．既存の数学は連続的なシステムにはきわめて有効だが，離散システムに向いた数学の進展はまだ限られている．しかも，コンピュータの登場によって，膨大な数の状態を持つ離散システムを作成することになったが，その設計や分析の理論は十分に発達していない．とくにソフトウェアシステムは，状態数が多いだけでなく，ハードウェアの電子回路と異なり繰り返し構造を一般に持たないという複雑さがある．その上で，とくに SDI ソフトウェアに要求される種々の機能や性質をあげ，それを満足するソフトウェアの開発は現状の技術では不可能であるとしている．

また，*The Mythical Man-Month*([25] 初版は 1975 年発行) 以来，ソフトウェア工学の分野でその著書や論文が大きな影響力を持ち続けている F. Brooks は，狼男，つまりソフトウェア開発という怪物，を倒す “銀の弾丸” は見つからないという論文を書き，それがきわめて多くの人に読まれて，いまだに話題の種となっている [24]．Brooks は，ソフトウェア開発は本来的に困難な性質を持っているという．そしてこれまで成功してきたソフトウェア技術，たとえば高水準言語，時分割システム，統合プログラミング環境などは，確かにソフトウェア開発の問題の部分的な克服に寄与したが，それが対象としたのはソフトウェアの本来的な問題ではなく，派生的，偶発的なものだという．

また，一般に銀の弾丸となり得るのではないかと期待されている技術の候補として，Ada などの高級言語，オブジェクト指向プログラミング，人工知能，エキスパートシステム，自動プログラミング，視覚的プログラミング，プログラム検証，環境とツール，などを挙げ，それらがいずれも期待に充分応えられそうにないという理由を述べている．

このあたりの論調は Parnas に近いが，しかし Brooks の主張は，論文の巧みなタイトルによって多くの人が思い込まされているほどには，悲観的ではない．確かに怪物を一発で倒す銀の弾丸はないだろうが，いくつかの革新技術の芽は存在しており，それらを地道に発展させ，普及し，利用する努力を続けることが，解決への道だという．そしてそのような見込みのある手段として，次の 4 つを挙げている．

1. 作らずに買うこと
2. 要求の洗練とプロトタイピング
3. 増進的 (incremental) 開発
4. 超技術者

ソフトウェア開発の難しさは，早くから言われてきた．それを悲観的に見るか，挑戦的な目標として前向きにとらえるかは，多分に時代の風潮が影響する．たとえば 1970 年代の初めに，A. P. Ershov は，プログラミングに必要とされる才能として，

- 第一級の数学者の論理性

- エジソンのような工学の才能
- 銀行員の正確さ
- 推理作家の発想力
- ビジネスマンの実務性
- 協同作業を厭わず経営的な関心も理解する性向

を挙げている [38]．これは「そんなことは無理だ」という反応を期待したものでなく，このような知的挑戦に有能な人材が取り組むよう鼓舞するという意図で言われたものである．

第2章　ソフトウェアプロセス

ソフトウェア工学が扱う対象の基本的な構成要素は，プロダクト (product) とプロセス (process) である．プロダクトとはエンジニアリングの過程で生成される中間製品や文書を含めたすべての生産物を指す．プロセスはプロダクトを生み出す工程である．したがって，ソフトウェア工学にとって，ソフトウェア開発のプロセス自身を方法論的な考察の対象にすることは重要である．工学の対象としてのソフトウェア開発プロセスは，通常，単にソフトウェアプロセス (software process) と呼ばれる．

2.1　ライフサイクルモデル

ソフトウェア工学の初期に，ライフサイクルという概念が提出された．これはソフトウェアの開発計画から設計開発，運用，保守そして最後に廃棄されるに至る過程を標準的なモデルとして表すものである．すなわちソフトウェアのプロセス全体を，包括的に示したもので，企業などのソフトウェア開発組織における標準工程は，なんらかのライフサイクルモデルに準拠して作られ，またプロジェクト管理がそれに基づいて実施されている．これは既存の工学におけるプロセスモデルを援用したものに他ならない．

ライフサイクルという用語はもともと生物学で使われ，世代ごとに繰り返される発生・成長の過程を指す (生活環という訳語が当てられることもある)．しかし，一般には，人の一生や商品が市場に出てから消えるまでの過程を指す言葉として使われる．ソフトウェア工学にこの語が入ってきたのは，恐らく生物学というルートを通してではなく，既存の工学のルートを経て工業製品が市場に出てから消えるまでという意味でが取り入れられたものであろう．

ライフサイクルモデルはソフトウェア開発のあるべき姿を示したものと見ることもでき，その意味で規範的なソフトウェアプロセスのモデルを提示しているともいえる．そのようなモデルの目的には，次のようなものがある．

1. 標準的なソフトウェア開発手順を定め，実際の開発担当者の作業をガイドする．
2. 開発プロジェクトを管理するのに準拠する管理モデルとして用いる．
3. 開発の方法論，使用するツールと開発環境，標準文書体系，などを標準的に定めるための基盤としての役割を果たす．

このようなソフトウェアプロセスのモデルは，標準的なものをベースとして，個々の開発組織ごとに，さらに個々のプロジェクトごとに設計される必要がある．実際，大きな組織では，ソフトウェアプロセスを設計することを専門とする部署が作られ，そこに組織内で標準的に使われるプロセスやプロジェクトごとのプロセスを設計し，また運用を観察して適切なフィードバックを行うという役割が与えられている．

2.2　落水型モデル

ソフトウェアのライフサイクルモデルのうち，もっとも古くからあり，現在でもほとんどの企業の標準工程のベースとなっているものは，落水 (waterfall) 型と呼ばれる．図 2.1 に示したのは，典型的な落水型ライフサイクルモデルである．ここで分析，設計，などの工程単位をフェーズと呼ぶ．

落水型モデルはまたテスト・フェーズを分解して，プログラムに対するテストは単体テスト，設計に対するテストは統合テスト，要求に対するテストは受入れテストと分け，それぞれを対応させて図 2.2 のような V 字型のライフサイクルモデルとして描かれることもある．

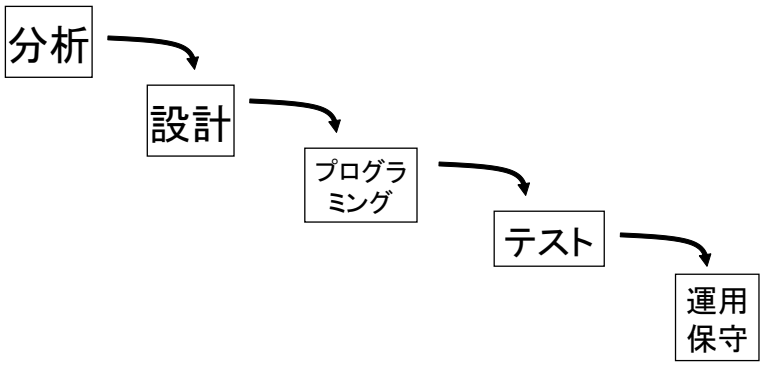


図 2.1: 落水 (waterfall) 型ライフサイクルモデル

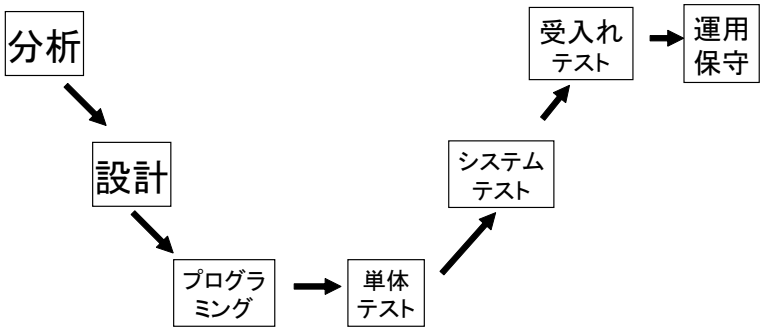


図 2.2: V 字型ライフサイクルモデル

落水型のライフサイクルモデルを最初に提唱した論文は，W. W. Royce によるもの [91] といわれる．この論文では落水 (waterfall) という言葉はおろか，ライフサイクルという語すら使われていないが，図 2.1 に相当する絵が描かれているだけでなく，そのいくつかのバリエーションを示して精緻な議論を展開しており，現在読んでも示唆に富む．

Royce のモデルをベースに，その後一般に落水型といわれるようになったモデルで強調されたのは，次の 2 点である．

1. フェーズ間に明確な区切りを置くと共に，その間はきちんと形式化された文書で受け渡す．
2. フェーズの手戻りを極力なくす．

このような要請には無理があり，落水型のライフサイクルモデルそのものに問題があるという批判が，1980 年代の初めに起こった．それでも落水型モデルが現場では主流であるのは，プロジェクト管理上，都合がよいからである．

2.3 落水型に代わる開発モデル

落水型のモデルには次のような限界があることが，早くから指摘されていた．まず手戻りをなくすという要求であるが，現実のソフトウェア開発プロセスではフェーズの手戻りは常態である．それを最小限に抑えるという目標は誰しも理解できるが，実際に手戻りが起こることを無視したプロセスモデルは現実的ではない．とくに初期フェーズの要求分析が問題で，要求がなかなか定まらなかったり，曖昧さが残ったり，プロセスの進行の過程で変化することは，しばしば起こる．それなのに，要求が完全に確定し仕様としてきちんと文書化されないと設計フェーズに進めないのでは，プロジェクトの進捗に阻害をきたす．また，ある程度の規模の開発では，その部分によって進行の度合いが異なってくるのは自然である．プロジェクト全体として同時にフェーズやその下のス

テップを切り替えていくのは，ほとんど不可能である．実際，製造プロセスの効率的な手法として実践され効果を上げている並行開発 (concurrent engineering) の概念からしても，硬直的な落水型モデルは現実にそぐわない．

このような批判に応じ，落水型に代わるモデルとして，次のようなものが提案されてきている [31].

プロトタイピングモデル　プロトタイピングとは，最終的に運用するソフトウェアシステムを作る前に，実験的な，しかし実際に動くシステム（プロトタイプ）を作り，それを評価するプロセスである．通常，プロトタイピングの目的を，とくにユーザの要求を明確にすることに限定して考えることが多い．その意味でのプロトタイピングプロセスを要求分析フェーズに組み入れた開発モデルを，プロトタイピングモデルという．要求仕様を確定するために，プロトタイプは繰り返し手直しされるが，適当な時点で仕様を固定し，以降は従来からの落水型開発モデルに従う．

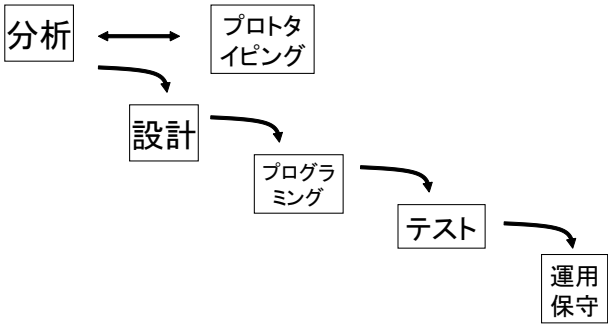


図 2.3: プロトタイピング型ライフサイクルモデル

逐次進化型モデル　要求分析から実装までの開発プロセスをただ一度だけ実行して運用システムを構築するのではなく，初めは小さな機能範囲のシステムを実現し，それを改良していくプロセスを繰り返すことにより，利用者の要求範囲の拡大や変動に対処しながら，柔軟にシステムの完成度を上げていくという開発モデルである．

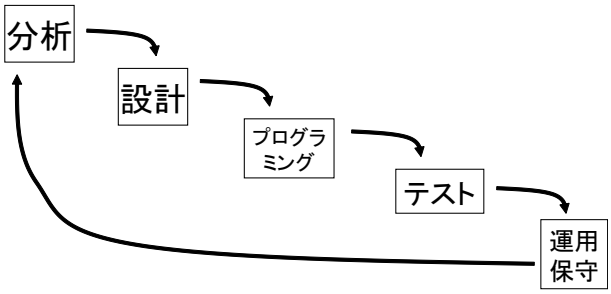


図 2.4: 逐次進化型ライフサイクルモデル

実際に動くシステムを利用者に早く提供するという点では，プロトタイピングモデルと類似する点もある．しかし，プロトタイピング型とは，どのような機能あるいは部分システムを優先的に開発するかという判断に違いが現れる．プロトタイピングでは利用者の要求が曖昧な部分を優先的に取り上げてプロトタイプを作り，その要求仕様を詰める．逐次進化型モデルでは逆に，機能が明確で開発効果も明らかであり，また設計技術上も不確定なところが少ないものを優先して開発し，徐々に新しく挑戦的な機能を付加していくというアプローチを取る．

逐次進化型を極端に推し進めた開発プロセスが，極端プログラミング (extreme programming, XP と略称される) というオブジェクト指向に基づくソフトウェア開発の方法論で，基本的に採用されている [13]．XP ではたとえば次のような方法が提唱されている．

- ソフトウェアの公開の計画をまず決めて，それに基づいて開発計画を定める．
- ソフトウェアの公開は小さな単位で頻繁に行う．

- 開発プロジェクトを小さい単位の作業の繰り返しとして分割する．
- 開発に際し，まずテストケースを作成し，それを満たすコードを書いていく．テストケースが仕様を兼ねる．
- システムの再構成 (refactor) を可能な限り追求する．

これらの手法により，非常に短いサイクルでの逐次進化と利用者への提供が実現される．その結果としてシステム全体がつぎはぎだらけになることは，頻繁な再構成で防ぐ．このようなソフトウェアプロセスは軽快プロセス (agile process) などとも呼ばれる．

2.4 ソフトウェアプロセスの評価

ソフトウェアプロセスの評価手法としては，Carnegie-Mellon 大学ソフトウェア工学研究所 (CMU-SEI) で開発されたプロセスの成熟度評価モデル，CMM(Capability Maturity Model) が著名である．元来は米国国防省がソフトウェア製品を外部から調達する際に，その開発組織を評価するために開発されたモデルである．それが一般にソフトウェアの開発組織で実施されているプロセスを評価し，その改善につなげる道具として使われるようになった．とくにその評価部分が比較的単純な 5 段階評価になっていることで，大きなインパクトを与えたといえる．

5 段階は次のように規定されている．

レベル 1 初期 (Initial)
: プロセスはほとんど定義されておらず，ソフトウェアの開発がうまくいっているとしても，偶然か個人の能力に依存している状態.

レベル 2 反復可能 (Repeatble)
: 費用，日程，機能を管理するための基本的な管理プロセスは確立している．過去に成功した類似システム開発の経験を活かし，そのプロセスを繰り返すだけの仕組みは持っている状態.

レベル 3 既定義 (Defined)
: 管理と開発の両面からソフトウェアプロセスが文書化され，標準化されて組織に組み込まれている．すべてのプロジェクトで定義済みの標準プロセスに手を加えて利用している状態.

レベル 4 管理下 (Managed)
: ソフトウェアのプロセスとプロダクトについて，詳しい定量的データが収集され，それに基づいて分析，管理が行われている状態.

レベル 5 最適化 (Optimizing)
: 定量的な分析のフィードバックによりプロセス改善が不断に行われている状態.

このような段階評価のために，いくつかのプロセス分野に分類された多数の評価項目が定義されており，またレベルを一段上がるためにとくにどのプロセス分野のどのような項目に力を入れて改善すればよいかも示される．

実際，このようなプロセス評価の目的はランク付けにあるのではなく，むしろプロセスの改善につなげるところにある．そこで SPIN と略称されるソフトウェアプロセス改善活動 (Software Process Improvement Network) が地域単位，組織単位で進められている．CMM 自身も発展し，とくにソフトウェアを対象とする SE-CMM(Capability Maturity Model for Software) の他に，システムエンジニアリング用のモデルや，製品とプロセスの統合モデルなども作られ，それらを統合した CMMI が策定されるに至っている．

プロセスを評価することは，規範的プロセスモデルを想定しているということでもある．もっとも古くからある規範的なプロセスモデルが，落水型モデルであった．また，落水型より後から登場したライフサイクルモデルの逐次進化型モデルや ISO/IEC JTC1/SC7/WG7 による SLCP[74](図 2.2 の V 字型モデルを精緻化したもの) なども規範的なプロセスモデルといえる．

これらは開発全体の大きな流れを工程化して示したものであるが，プロセスのもう少し内部の構造まで考え，さらに評価という視点を前面に出したものが，上記の CMM や ISO9000[52]，SPICE[36] などである．

ISO9000

ISO 9000 シリーズは品質管理および品質保証のシステムに関する規格を定めている．ヨーロッパでは，第三者機関によって，申請企業の品質システムが ISO9000 シリーズに規定したシステムに合致しているかどうかを審査し認証する制度が定着し始め，それが米国，日本にまで波及してきた．この制度の基準文書は ISO 9001 である．9001 は一般の製品に対し，とくに 2 者間契約で製造供給される場合につき，その供給者が有すべき品質システムの要求事項を定めている．とくにこれをソフトウェアの開発/設計，供給，保守に適用するための手引として作られた規格に，ISO 9000-3 がある．

SPICE

SPICE は CMM とそれに類似するいくつかのプロセス評価モデルを統合し，世界的な標準としようという目標で進められている．米国生まれの CMM をヨーロッパが取り込んだという感じもあり，その意味で ISO9000 との融合という面もある．

2.5 ソフトウェアプロセスの観察と改善

ソフトウェアプロセスを評価する目的はプロセスの改善にあると述べたが，具体的な改善をきめ細かく立案するには，現在行われているプロセスを定量的に観察し，その結果に基づいて改善案を構築する必要がある．観察と改善案作成のアプローチには，事例研究の分析を積み重ねていくもの，認知科学の手法を適用するもの，品質管理，とくに TQM からのアプローチなどがある．また，プロセスの実態を分析することにより，プロセスの再利用や人材教育に役立てるという利用方法もある．

具体的なソフトウェアプロセスの観察例として，大規模プロジェクトにおけるプロセスの手戻りを分析事例があるので (伊東・玉井 [105, 55])，以下でそれを紹介する．

大規模なソフトウェア開発プロジェクトが管理上の破綻を来す大きな要因の一つとして，要求仕様の不安定さ，すなわち仕様変更が頻発しまた予期不能であるという状況が，しばしば挙げられる．この調査は要求仕様変更が起こる実態を，実際の大規模プロジェクトを分析することによって解明しようとしたものである．その際，要求仕様の変更を捉える方法として，プロセスの手戻りという現象に注目するという方針がとられている．

事例の概要

データが入手可能な 2 つの大規模ソフトウェア開発プロジェクトが対象として取り上げられている．プロジェクトの特徴は，表 2.1 のようである．

表 2.1: プロジェクトの概要

	ケースＡ	ケースＢ
プロセスモデル	落水	プロトタイピング
対象機種	大型計算機	大型計算機
開発環境	RDBMS(MODEL204), 4GL(U/L204), COBOL	RDBMS(MODEL204), 4GL(U/L204), COBOL
開発期間	1985 年 8 月 ~88 年 9 月	1985 年 4 月 ~90 年 12 月
開発規模	550 KLOC	2,000 KLOC

いずれも事務処理分野でデータベースを基盤とするシステムであり，開発期間は長く規模も大きい．両者の大きな違いは準拠する開発モデルで，ケースＡは従来型の落水型モデルを用いているが，ケースＢではプロトタイピング方式が採用されている．

ケースＡ：Ａ社の経理財務システムの再開発　既存のバッチ型のシステムに代わる，全社的なオンライン経理財務処理システムを再開発するというプロジェクトである．全体は２つのフェーズに分けて開発され，この分析の対象となったのはそのうちの第１フェーズに当たる部分であるが，このフェーズに限ってみても，要した工数は 320 人月，投資額にして約３億円という規模である．プロジェクトに参画した要員の数は，開発委託を受けたソフトウェア企業側で延べ 25 名，ユーザの企業側で延べ 40 名に達した．

ケースＢ：Ｂ社の統計情報システムの開発　対象とされているのは，金融機関のための大規模な経済統計情報システムを開発したプロジェクトである．このシステムの規模は，ケースＡよりもさらに一桁大きい．開発工数で 2,500 人月，プログラム規模は２百万行である．やはり，ソフトウェア企業に開発委託がなされた．要員数は，開発側だけで 60〜100 人（時期によって変動）であった．

ソフトウェアプロセスの特徴

両プロジェクトをプロセスの観点でみた場合，いくつかの共通する特徴と，逆に対照的な特徴とがみられた．

計画期間の長さ　両プロジェクトとも，システム計画フェーズに長い期間を当てている．ケースＡでは 19 カ月，全期間の半分であり，ケースＢでは 21 カ月で全期間の 30 %を占める．この計画フェーズの作業には，現行システムの調査，業務とシステムの改善案の列挙，ニーズ抽出のためのユーザ部門の面接調査，最新のハードウェア/ソフトウェアの製品および技術の調査，システム開発プロジェクトの基本計画策定，などが含まれる．この計画フェーズにさらに基本設計フェーズを加えた期間は，プロジェクト全体に対し，ケースＡで７割，ケースＢで６割に達し，平均的なプロジェクトからみるとやや異常に大きな比率を占めているといえるだろう．

計画期間を長くとることは，問題の理解を深め，関係者の共通認識を高め，重要事項の考慮に抜けを生じるという事態を防ぐ効果があるが，全体の方針や計画の斉合性が時間の経過につれて悪化したり，設計，実装期間が不足したり，プロジェクトの初期にあったメンバーの意欲や資源が消失するといった危険を伴う．実際，両プロジェクトに多かれ少なかれこのような弊害が認められた．

役割グループ間の関係の複雑さ　プロセスにはそれを実施する役割が対応し，また個々の要員あるいはチームがその役割を割り当てられる．これらの構成員やチームは，より大きなグループに所属し，それらのグループ間の利害関係，力関係が，割り当てられた機能間に生じる関係に加えて複雑な構図を作る．さらに，プロジェクトが長期に渡ると，同じ役割を受け持つグループ内でも人の異動が起こり，前任チームと後任チームの存在がさらに他の機能グループとの関係を複雑にする．

一般に次のような関係が，プロジェクトの進行に影響を及ぼす．

1. 発注者対受注者
2. 前任者対後任者
3. 発注企業内のシステム部門対ユーザ部門
4. ユーザ部門内で開発プロジェクトの予算を管理する部門と予算権限をもたない単なる利用/関連部門

実際，両ケースでこれらの間に生じるコミュニケーションの問題や意見の相違から，要求や設計仕様の決定に手戻りが発生している．

プロトタイピング　ケースＢは，大規模なソフトウェア開発に対し，単なる画面や帳票の形式を見せるというプロトタイプでなく，機能を部分的に実現したプロトタイプを作成し，顕著な効果を挙げた数少ない例の一つであろう．ケースＡとケースＢの，プロセスとしてのもっとも大きな違いは，ケースＢでプロトタイピングを全面的に採用しているのに対し，ケースＡではプロトタイプを作らない従来の落水型ライフサイクルモデルによっている点である．

ケースＢのプロトタイプは，工数にして 90 人月，期間にして半年をかけている．作られたプロトタイプの大きさは，163kloc である．プロトタイプ開発が全体に占める比率は，表 2.2 の通り．

表 2.2: プロトタイプの相対的な規模

	プロトタイプ	全　体	比率 (%)
人月	90	2,500	3.6
年数	0.5	5.5	9.1
kloc	163	2,000	8.7

プロセスの手戻り

両ケースのプロセスの手戻りに実態が，仕様確認書，システム修正依頼書，設計書，テスト結果報告書，打ち合せメモ，進捗管理会議の議事録，などをもとに調査された．また，プロジェクトのメンバ数名への面接により，文書で不十分な情報が補われている．

ここで手戻りとは，要求分析，概要設計，詳細設計，プログラミング／テストと分けたプロセスの，後ろのフェーズを実行中に起こった前のフェーズの成果物の変更要求に伴う再作業を意味する．ケースＡについてはさらに，運用のフェーズが加えられ，ケースＢではプロトタイピングのフェーズが加えられている．ケースＢの運用時のデータは，この調査時点ではまだ得られていなかった．

ケースＡで 111 件，ケースＢで 196 件の手戻りが認められた．その内訳は，表 2.3，2.4 の通りである．

表 2.3: ケースＡの手戻り件数

	概要設計	詳細設計	実装	運用	合計
要求分析	14	19	3	4	40
概要設計		27	11	4	43
詳細設計			15	2	17

(数値は列フェーズから行フェーズへの手戻り件数の全体に対する割合を%で示している．)

表 2.4: ケースＢの手戻り件数

	プロトタイピング	概要設計	詳細設計	実装	合計
要求分析	38	2	1	1	42
概要設計			8	20	28
詳細設計				30	30

(数値は列フェーズから行フェーズへの手戻り件数の全体に対する割合を%で示している．)

次の点が注目される．

- ２フェーズ以上戻る手戻りが，ケースＡで 44 %，ケースＢで 24 %と大きな割合を占める．
- ケースＢのプロトタイピングは，要求分析フェーズへの手戻りの減少に著しい効果を上げている．プロトタイピングから要求分析への手戻りは，むしろプロトタイピングの本来の目的であるから除外して考えると，概要設計以降から要求分析への手戻りは，ケースＡと比較してきわめて少ない．
- しかし，プロトタイピングは概要設計以降のフェーズ間の手戻りを減少させるのには，成功しているとはいえない．

要求分析と概要設計への手戻りを合わせると，ケースＡで 83 %，ケースＢで 70 %（ただしプロトタイピングからの手戻りを含める）を占める．これらの変更は，ほとんどがユーザの意図の変更に直接または間接に起因する

ものである．すなわち，よく言われる要求の不安定さ (volatility) がソフトウェア開発のプロジェクト管理上大きな問題であるということは，この事例からも裏付けられる．

対策

このようなプロセスの手戻りに対し，どのような対策を取りうるだろうか．次のような点が重要と思われる．

柔軟なプロセスモデル　プロセスに手戻りが生じることは不可避であると，認めざるを得ないであろう．したがってプロセスを不可逆なものとして捉えることは，非現実的である．

ケースＢの結果からは，プロトタイピングの導入が強く示唆される．落水型のモデルを現在も使用している組織では，まずプロトタイピングを標準的に取り込み，その後，プロセスの柔軟性を徐々に上げていくというアプローチを取るのが現実的であろう．

ソフトウェアの規模　すでに両事例の規模の大きさを，強調してきた．両ケースとも規模が大きくなる理由の一つは，新しいシステムが，既存のいくつかのシステムの機能を取り込んだ統合システムとして構想されたことである．統合化にはもちろんよい面が多くあるが，規模が増大することによって生じる問題には，これまで十分な注意が払われてこなかった嫌いがある．ソフトウェアの作り直しの際に生じる規模の増大については別に研究例があるが [112, 108]，プロセスの手戻りという観点からも，無制限な規模の拡大には歯止めが必要であることが判る．

計画期間　日本のプロジェクト開発では，計画に長い期間をかけることが一般的な傾向として認められる [14].

これにはよい面もあるが，たとえばケースＡではその結果要員の交代が起こり，関係グループ間のコミュニケーションに問題を生じ，全体のプロジェクトに重大な影響があった．計画期間が長くなるのが，単に意思決定プロセスの効率の悪さからくる場合が往々にしてみられることから，これを改善して後の作業工程を円滑化するという対策が示唆される．

ユーザプロセス　ケースＢで，ユーザがプロトタイプを使用しシステム使用経験を積むにつれて，機能要求のレベルや範囲に変化が起きるという現象が認められた．

当初，開発側は汎用性があり高度な使用が可能なプロトタイプを提供したが，ユーザから分かりにくいのでもっと簡便に使えるものが欲しいという要望が出た．そこで機能を簡略化したプロトタイプを改めて作ったところ，ユーザがそれに習熟するにつれて，元の汎用的な設計の良さを理解し，両方のプロトタイプを保持するようにと要求が変化した．結果的に，開発側の負荷は増大した．

このことから，ユーザの学習プロセスをもっと重視すべきであるとの知見が得られた．たとえば，次のような点が考慮されるべきであろう．

- ユーザのシステム使用能力や理解のレベルは進化するものであることを，システム開発の前提とすべきこと．
- 初心者向きの機能は，利用者の習熟に伴って不要になりうることを予測して，計画すること．
- 逆に複雑で高度な機能は，必ずしも開発当初に組み入れる必要はなく，ユーザの習熟度に合わせてリリースするといった計画が考えられること．

2.6　プロセスプログラミング

1985 年の第 8 回 ICSE (International Conference on Software Engineering) で，委員長の M. Lehman がプロセスの重要性を強調し，全体テーマとして掲げた頃から，ソフトウェア工学の研究界でもソフトウェアプロセスを研究対象としようという機運が出てきた．

1987 年に L. Osterweil が,「ソフトウェアプロセスもソフトウェアである」という謳い文句でプロセスプログラミングを提唱した [82]．これはソフトウェアを開発するプロセスを手続き的なプログラムとして記述し,「実行」しようというアイディアである．それ以来，プロセスを記述する形式言語やモデル，またそれを動かすプロセス支

援環境の提案が数多く発表された．それらの目的も，プロセスの実証的な分析から，評価，改善，標準プロセスの規定など多岐にわたるが，元の提唱者である Osterweil 等は，記述したプロセスの実行という面を強調し，最終的には開発プロセスの自動化を目指した．

しかし，プロセスをプログラムとして記述できるのは，開発プロセス全体からいうとごく限られた部分でしかないという議論もある．また，プロセスを実行する主体の一つに人間があるが，人間の不確定な行動をなるべく確定的なものとして捉えていこうとする一部の機械主義的な立場には批判もなされた．

ソフトウェアのプロセスとは，実は製造プロセスでなく設計プロセスである．したがって，化学工学などのプロセスとは本質的に異なるところがある点は重要である．他のエンジニアリング分野でも，設計プロセスについてはこれまでそれほど意識的でなかった．それだけに形式化が難しい問題であることは確かである．

プロセスプログラミングの目標　研究としてのプロセスプログラミングは次のような目標で行われてきた．

1. 実際のプロセス，とくに人間の行動が主要な要素となる開発過程を観察し，分析し，改善方法を探る．
2. プロセスの形式的記述に適した形式システム，言語の研究および実際の記述実験を行なう．
3. プロセスという概念を核とするソフトウェア開発環境を開発する．

以下で，代表的な研究例を概観する．

プロセスの形式記述／モデル化　研究例には，階層的関数型言語を用いた HFSP（片山等 [63]），代数的仕様記述を用いた PDL（井上等 [53]），状態遷移モデルを拡張した市販のツール Statemate を用いた例（M. Kellner, SEI[64]）など，多くのものがある．

プロセス中心ソフトウェア開発環境　Osterweil 等は Arcadia というプロジェクトを推進し [110]，プロセス記述用の言語とその記述および実行支援システムの開発を行なった．また，要求分析，設計，テストなど様々なフェーズにおけるプロセスの記述も実際に試している．

コロンビア大学の G. Kaiser を中心に研究開発が進められた Marvel というシステム [62] では，ルールベースによるプロセス記述手段とオブジェクトベースによるソフトウェアの生成物管理とを組み合わせた開発環境を提供している．

他に，落水等による Vela [80]，Ambriora 等による Oikos [8]，Huff 等による Grapple [51]，Schäfer 等による Merlin [85]，などがある．

プロセスの設計と実行　このような技術を実際のソフトウェア開発に適用するには，プロセスの設計と実行という 2 つのステップがとられる．

プロセス設計の考え方には，次の 2 つがある．

- **prescriptive(規定的)**　プロセスを手順として示す．具体的で実行と結びつきやすいが，柔軟性に欠ける．
- **proscriptive(制約的)**　プロセスの満たすべき条件を記述する．プロセスの動的な変化に対応しやすいが，実行との間にギャップがある．

いずれにせよ，プロセスをソフトウェア開発プロジェクトの 1 件ごとに，まったく新たに設計し直すということは考えにくい．基本となる一般的プロセスモデルを前提に，組織ごとに，またプロジェクトごとにカスタマイズするというのが，実際的である．

設計したプロセスを実行するとは，どういうことであろうか．人間を中心に考えれば,「実施」するという表現が適合しよう．コンピュータによる “実行” (enaction) という表現を用いる場合は，開発者を誘導 (navigate) し指針を与えるという動作がまず想定される．また，プロセスが規定された制約に従っているか，あるいは目標の方向に着実に進んでいるか検証するという機能も考えられる．そのためにはプロセスのさまざま特性を自動的に計測し，数値管理を行うことが必要であろう．また，“強い” プロセスプログラミングの立場を取る人は，このような人間としての開発者を支援するだけでなく，可能な限りプロセスを自動実行することを目指している．

スケジューリングの記述方法　プロセスモデルはいくつかの側面を持っているが，ほとんどのモデルが取り扱っているのがプロセスのスケジューリングという領域である．

旧来の PERT と同様に，作業単位の間の前後関係を明示的に指定することによりスケジュールを構成するという方法が，1 つの自然なやり方である．また，作業単位ごとにそれを起動する事象 (event) を記述し，また作業から発生する事象を記述することで，間接的にスケジュールが定められるという方法もよくとられる．さらに間接的な方法としては，作業単位を起動する事象ではなく，作業の入力条件と出力条件を指定し，その条件を満足するようなスケジューラを構築するという方法もある．スケジューラの取る手法には，現状態で入力条件を満たす作業を抽出し，その実行後の出力条件から状態を変更して次に実行可能な作業を探索するという前進型のものと，ゴールから逆向きに条件を生成して実行順序を定めていく後進型のものとがある．

オブジェクト管理　スケジューリングとならんで重要なのが，プロセスで生成され，また利用されるオブジェクト (生産物，artifact) の管理である．プロセスモデルを狭く解釈すれば，オブジェクト管理はその外にあるとも考えられるが，多くのモデルではこの側面を取り込んでおり，またプロセス中心の開発環境では重要なサブシステムとなる．

技術的には，従来からの構成管理やいわゆるリポジトリ・システムで用いられているものがベースとなる．その上で，プロセスとの関連から，より多様な一貫性保持技術が必要となろう．そこではデータベースで蓄積されてきた技術が適用される．

オブジェクト管理システムの標準化を提案している例には，PCTE (Portable Common Tool Environment), IRDS (Information Resource Dictionary System) などがある．

プロセス研究のその後　上で挙げたようなこれらの研究例は，主として 1990 年代前半に集中している．その後は，ソフトウェアのプロセスよりソフトウェアのアーキテクチャ（基本設計思想）の方に研究界の関心は移っていった．これを研究のはやりすたりの現象と見ることもできるが，プロセスに関心が集まる時期とプロダクトに関心が集まる時期とが交互に来る大きなサイクルがあると見ることもできよう．

第3章　開発計画と要求分析

3.1　何を作るか

ソフトウェア開発はどのように始められるのだろうか．あるいは，前章で導入した言葉を使えば，ソフトウェアプロセスの最初におかれる作業は，何だろうか．後者の問いに対しては，落水型などのライフサイクルモデルを見れば，その先頭におかれたフェーズが「分析」となっているから，答えは明らかのように見える．しかし，分析作業に何が含まれるのかという問題に置き換えられただけともいえるし，仮にそれが明らかになったとしても，その「分析」の前はないのかという疑問がさらに生じる．

ソフトウェア開発を始めるには，何を作るかがまず決められなければならない.「分析」とは通常,「要求分析」を意味するものと考えられる．ここで「要求」とは，構築すべきシステムに対してその利用者や関係者が明示的にあるいは潜在的に持つものを言う．要求分析は，そのような要求を引き出し，最終的には要求仕様として定義する作業である．だからまさしく「何を作るか」を決めるものであるが，ことはそう単純ではない．何を作るかを決める前に，そもそも開発に着手すべきか，あるいは少なくとも開発の検討を開始すべきかについて，意思決定が行われなければならない．その意思決定を行うにはまた，どんなシステムかが想定されていなければならない．つまり,「何を作るか」と「作るべきか否か」との検討が並行して進められなければならないわけである．

このような意味で，どのようなソフトウェアを開発すべきか（あるいはすべきでないか）を決定するには，そのソフトウェアがどのような価値をもたらし，その開発にどれだけの費用がかかるかを予測しなければならない．ソフトウェアの価値は，端的にはどれだけ使われるか，とくに商用ソフトウェアの場合はどれだけ売れるか，で測ることができよう．しかし，機能的に優れ商品価値が高いと思われるソフトウェアが，よく売れるとは限らない．個別に開発したソフトウェアでも，よくできているからといって必ずしもよく使われるものではない．一方，開発の費用には，人件費や設備費など単純に金額に換算されるものと，開発期間，その他の必要資源など間接的なものとがある．

要するに，この辺の意思決定の問題は，人間的要素による影響を非常に強く受け，経営学，心理学，市場分析，一般問題発見・解決手法，などがさまざまに交錯する一筋縄ではいかない分野である．そのどこまでに，ソフトウェア工学という基本的に「いかに作るか」を対象とする方法論がかかわるべきは，微妙なところである．機械工学や電気工学の教科書では,「何を作るべきか」を直接取り上げることはないであろう．ソフトウェア工学は何を作るかに大きな自由度があり，また利用者と開発者とのコミュニケーションが持続的に必要であるという特性があるので，要求分析というフェーズがとくに重要であり，またそこで価値判断を含めた問題を扱うことは，ある程度避けられない．

3.2　システム化計画

3.2.1　実際的な開発標準工程における初期フェーズ

企業で実際に使われるている工程では，その最初のフェーズを要求分析とか要求定義とか呼ぶ例はあまりない．そのかわり，システム化計画というような言葉がよく使われる．実際すでに触れたように，ソフトウェア開発計画を進める際に考慮すべき要因としては，どんな機能が要求されるかという要求分析の結果以外に，次のようなものが重要となる．

- 効果，利益
- コスト

- 期間，スケジュール
- 資源
- 経営，組織，市場，制度，社会などによる制約

したがって，システム開発の当初にすべきことは，これらの要因を総合的に判断した上で，プロジェクトを実施するか否かの意思決定を行うことである．この意思決定プロセスには，経営者だけでなくシステム開発者と関係する多様なユーザが，通常参加する．とくに日本では，このフェーズがそのような関係者による合意形成プロセスとしてとらえられることが多い [107, 103].

システム化計画の出力，たとえば「システム計画書」には，標準的に次のような項目が記述される．

- 背景
- 開発目的
- 対象領域
- 業務の現状
- すでにシステム化されている部分の現状
- 新たに開発すべきシステムの概要
- 効果
- 制約条件
- 開発・導入計画
- 開発費用の見積り
- 必要とする資源

したがって，ここには要求仕様（の一部）が書かれるが，その他の事項も大きな役割を占める．また多くの場合，要求仕様の一部は，後続のフェーズの出力にも含まれる．

このような作業のための方法やガイドラインをまとめたものが，多くの企業でマニュアル化され，使われている．表 3.1 に各社が公表している代表例を挙げる．

表 3.1: 代表的なシステム分析手法

技法名	開発元	技法名	開発元
EPG	富士通	F-SPAN	富士フィルム
C-NAP	富士通	BSP	IBM
PPDS	日立	CPS	IBM
STEPS/E	日電	TUPPS	東芝
NUPS	ユニシス		

マニュアルに付随して，ツールが使われることもあるが，その役割は補助的である．その作業は，関係者を集めた合宿形式で行なわれることも多く，また通常インストラクタが同行して，指導・誘導を行なう．

合意形成に時間をかけることは後のステップを円滑に進めるのに有効であることが多いが，計画に時間をかけ過ぎることは，とくに日本におけるソフトウェア開発プロジェクトに往々にしてみられる傾向で，生産性を阻害している可能性がある [14, 105]．その他にも，次のような問題点がありうる．

1. 合宿のような集団による作業は，インストラクタの手腕による部分が大きい．同じ技法を用いても，インストラクタにより成果の質が大いに異なる．一般に，よいインストラクタの数は不足しており，その育成には手間がかかる．

2. 集団作業は，参加者の参加意識を生み，合意形成に役立つ．一方で，システムの利用者，関係者がすべて参加するわけにもいかないので，不参加者に作業結果を伝達する必要があるが，決定のプロセスまで含めた微妙な情報を伝えることには，困難が伴う．

3. 規模の大小によらず，あらゆるプロジェクトに対し，同様な要求分析作業を行なうわけにはいかない．提案されている種々の方法は，かなり手間のかかるものなので，小規模システムの場合どうするかは，別の問題として残る．

4. 作業の結果，問題点の見事な整理ができたり理想的なシステムの姿が構成されても，以降の開発や実現に結びつきにくいケースが往々にしてある．

3.2.2 コスト見積り

開発費用の見積りどの開発組織も昔から行っているが，決定的な方法がない．通常取られている方法は，

1. 開発するソフトウェアの規模を，何らかの方法で見積もる．

2. ソフトウェアの規模から，何らかの見積り式で必要工数（人月）を算出する．

3. 工数から，開発期間と投入要員数を決める．

というものである．

まず，ソフトウェアの規模を見積もるのが難しい．規模を何で測るかについてもいろいろ問題はあるが，原始プログラムの行数を尺度とするのがもっとも一般的である．しかし，ソフトウェアはまだ作られていないのだから，何らかのデータを基にそれを予測しなければならない．

機能点 (function point) 数を数えるという方法が，ある程度実践されている．機能点の数え方には流儀があるが，入力データの種類の数，出力データの種類の数，ユーザ要求にシステムが応答する処理の種類数，アクセスするファイルの種類数，考慮すべき他システムのインターフェース数，に基づいて，重みを付けて算出するというのが一般的である．この機能点数とプログラム行数が比例するとして，比例定数を過去のプロジェクトから推定し，原始プログラム行数を見積もる．

次に，ソフトウェアの規模（原始プログラム行数）から必要工数を算出する式であるが，この関係は線形ではなく，必要工数 E とプログラム規模 S は，

$$E = aS^k, \tag{3.1}$$

という指数関係にあることが経験的に知られている．指数 k は 1 以上で，1.5 というデータもあれば，B. Boehm の調査では 1.05-1.2 という数値が挙げられている [19]．とにかく，この定数の a や k を，過去のプロジェクトから推定することになる．

この必要工数の単位は，人月とか人日という要員の数に時間を乗じたものであるが，これがまた問題である．この工数は，同じ仕事をするのに人を増やせば比例して開発期間が短くなることを暗に仮定しているが，それが実態に合わないことは古くから指摘されている．確かに，10 人で 10ヶ月かかる仕事を 20 人でやれば 5ヶ月でできるかというと，それは無理だというのが現場の感覚であろう．工数を E，要員数を P，開発期間を T として，

$$E = PT, \tag{3.2}$$

の関係を E を固定して考えると，図 3.1 のようになるはずだが，実際は投入要員数を増やすと，たがいのコミュニケーションのオーバーヘッドや教育のために，かえって開発期間が延びてしまうというのが F. P. Brooks, Jr. の主張である（図 3.2 参照）．彼はそれを「遅れているソフトウェア・プロジェクトに要員を追加投入すると，さらに遅れる」と言い表し，自ら「Brooks の法則」と呼んでいる．

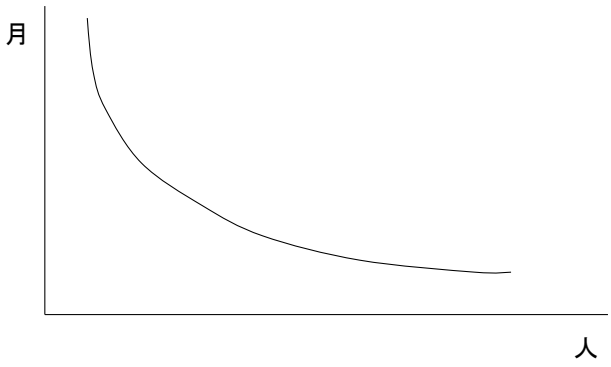


図 3.1: 人月の関係–反比例の場合

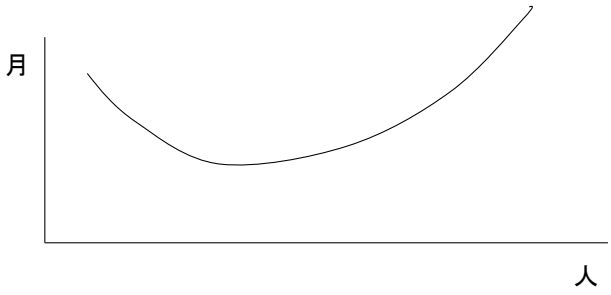


図 3.2: 人月の関係–Brooks の法則

3.3 要求分析

3.3.1 要求工学

何にでも「工学」という語を付けるともっともらしい技術分野が存在するように聞こえるものではあるが,「要求工学 (requirements engineering)」はソフトウェア工学に部分として含まれるものでありながら，それが提唱されてからの歴史はかなり長く，実践的な面からの重要性も広く認識されて，確固とした地位を占めている．遡れば，構造的プログラミングが華やかであった 1970 年代半ばに，要求工学という名のもとで，いくつかの技法が提唱された．代表的なものは次の通り．

1. PSL/PSA
   D. Teichrow(ミシガン大学) 等による ISDOS プロジェクトで開発された言語 PSL と分析ツール PSA を組み合わせた手法である．

2. SREM
   TRW で開発された手法で，要求定義言語 RSL，処理フロー表現 R-Net，分析ツール REVS を用いる．とくにリアルタイムシステム向き．

3. SADT
   Softech 社の D. T. Ross 等によって開発された手法で，作業あるいは動作を箱で表し，その入力，出力，制御データ，使用する資源の 4 つを左右上下からの矢印で表現する．SADT はその後，IDEF0(Integration Definition for Functional Modeling) という規格として，米国の規格協会 NIST(National Institute of Standards and Technology) から 1993 年に公開されている．

いずれも，文献 [2] の要求分析／定義特集号に論文がある．

これらはその後，構造化分析／設計技法に集約されていったが，1980 年代の後半に脚光を浴びた CASE ツールによって，再注目されるようになった．さらに，1990 年代に入ってから要求工学に関する国際会議 (International Conference on Requirements Engineering) が定期的に開かれるようになり，研究分野としても改めて隆盛を迎えている．

3.3.2 要求分析の考え方

要求分析/定義の作業は，大きく 2 つの部分に分けて考えることができる．前半は，自分あるいはソフトウェアの発注者が，そもそも何を意図し何を要求するのかを，曖昧で混沌とした状態から次第に明確なものにしていくプロセスである．この作業は通常，要求分析，問題分析あるいは単に分析と呼ばれる．後半はその分析を基に，決まった仕様記述モデルあるいは仕様記述言語によって，仕様をきちんと記述する作業である．これは要求定義とか仕様記述と呼ばれる．

この要求分析の考え方には様々なものがあるが，以下でそれを 3 種類の型に分類して考察してみよう．

1. 要求抽出型
   初めにユーザのニーズや意図があるものとし，それを要求として抽出する作業を要求分析の中心におく．すなわち，潜在的にでも要求を持っている人の存在が仮定される．ユーザへのインタビューが基本的な方法の一つになるが，その際，シナリオを用いる手法がよく用いられる．構築すべきシステムを利用するシナリオを，ユーザとシステム開発者が共同で作るものである [88]．それをより具体的に記述したものは使用事例 (usecase) とも呼ばれ，システムの仕様を定める基として使われる．

   この方法は，ユーザへのきめの細かい対応が可能なことが特徴で，またシステムの利用イメージも早い段階で明確になる．弱点は，要求を引き出す相手に依存することが大きい点で，人が変わると要求が大きく変わる可能性がある．そこでなるべく多くの関係者から要求を抽出することが勧められるが，そうすると要求が散漫になり，また場合によっては要求同士が衝突することが起こり，その調整が難しい．全体に整合性の高い要求をまとめるのに苦労する傾向がある．

図 3.3: 要求抽出型の要求分析

2. 目標指向型
   システムを開発するのは何らかの目標 (goal) を達成するためであるはずだから，その目標を構造的に明らかにすればシステムへの要求が明確になるという考え方である．目標を意識的に持つのはユーザだから，やはりユーザへのインタビューは自然な方法である．しかし目標の分解・統合をかなり系統的，客観的に行うこともできる．すなわち，1 つの目標を，それを満たすためのより小さな目標に分解する．分解されたすべての目標が満たされる場合に元の目標も満たされる and 結合と，そのうちの任意の 1 つが満たされれば元の目標も満たされる or 結合が考えられる．逆にその目標は何を満たすために作られたかと問うて，より大きな目標にまとめるという形で，目標間の階層構造を階層的有向グラフとして作成する．こうして作られた目標の階層グラフの最下端の目標が，システムへの要求となる [113]．

   実際，このような手法はオペレーションズリサーチや経営科学などの分野で，一般的な問題発見・問題解決の方法として古くから工夫され，また要求工学の分野ではさらに精緻化されてそれを支援するツールも作られてきた．

   この方法の特徴は，要求抽出型より全体的に整合性のとれた要求の集合が得られる点にある．しかし弱点は，この方法に基づいて作られたシステムが，定めた目標に特化した構造となりがちであり，その目標の解決にはきわめて有効でも，少し問題が変わった場合の柔軟性に欠けることが起こりがちな点にある．

3. 領域モデル型
   初めに対象とするドメイン, すなわち問題領域，適用分野，業務，対象世界，などのことばで呼ばれるものがあり，それをなんらかのモデルに写像するという立場をとる．要求や目標は後からモデル上にのってくる，という見方である．モデル化の手法としては，オブジェクト指向モデルなどが用いられる．

   この方法は，一連の製品系列 (product line) 上の多くの少しずつ異なるシステムの要求分析を一度に行う際に有効である．また，対象領域は機能要求と比べて比較的安定しているので，その対象領域のモデルが反映さ

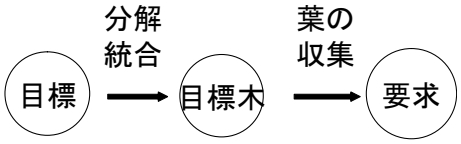


図 3.4: 目標指向型の要求分析

れたシステムは要求の変化に柔軟に対応しやすいという特徴ももつ．一方，この方法の弱点は，システムのイメージがなかなか明確にならず，問題領域内に作られるべきシステムの内部と外部の境界がはっきりしない傾向がある点である．

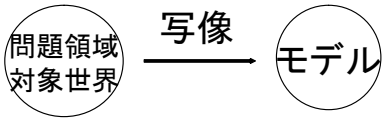


図 3.5: 領域モデル型の要求分析

これらの方法は，必ずしも独立に実施されるわけではなく，むしろこれらを組み合わせて用いる方が有効である場合が多い．しかし，対象となるシステムや開発組織の特性に応じて，適切なものを選びまた組み合わせる必要があろう．

シナリオと目標展開について，例により説明を加える．

飲物自動販売機のシナリオ　街で見かける飲物の自動販売機を，新たに開発するものと想定してみよう．自動販売機を利用者が利用する 1 つのシナリオとして，以下が考えられる．

1. 利用者は紙幣または硬貨，あるいはその両方で金を入れる．
2. 投入金額が表示され，それが価格と等しいか価格を上回る飲物に対応したランプが点く．
3. 利用者が点灯したランプのボタンを押すと，その飲物が 1 本取り出し口に排出され，表示金額からその飲物の価格が減額される．
4. さらにまだランプが点灯している飲物があれば，そのボタンを押して飲物の取り出しを続けることができる．
5. 利用者は 1 に戻って金を追加投入し，2 以下の動作を続けることができる．
6. 利用者は残金があれば，返却ボタンを押してつり銭を受け取る．

このシナリオは通常動作の場合であり，飲物を 1 本も出さないで終了したり，つり銭を取り忘れたり，飲物が在庫切れになる，など特殊な状況に応じたシナリオを別に用意する必要がある．また，シナリオの記述の仕方から判るように，シナリオは動作事象の系列からなり，その動作はいくつかの主体間のやりとりと見ることができる．この例では動作主体は利用者と自動販売機である．

酒屋倉庫管理の目標展開　酒屋の倉庫における在庫管理という問題分野を想定する．図 3.6 の四角い箱がそれぞれ目標を表す．上下方向の線は，上位の目標とそれを分解した目標とを結ぶ．図で，負符号がついている矢印は，効果が負であり目標間に衝突があることを示す．この例では or 展開はなくすべて and 展開である．

3.3.3　要求の種類

要求には機能要求と非機能要求があり，非機能要求には性能，使いやすさ，安全性 (security)，保守性，可搬性 (portability) などがあるといわれる．さらに組織文化や法律などから要請されるものも，非機能要求に含まれうる．

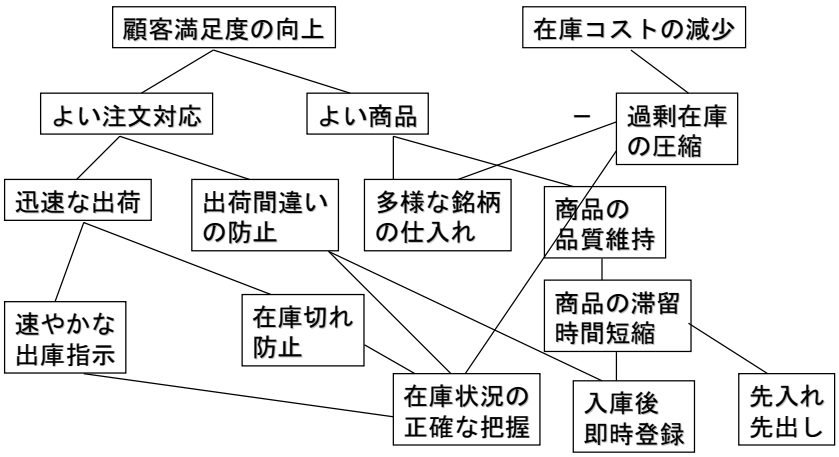


図 3.6: 酒屋倉庫管理の目標展開

機能要求の中でも，要求同士が衝突し相容れないことはしばしば起こり，それをいかに調整するかは要求分析の重要課題の一つである．非機能要求はさらに，そのような衝突を起こしやすい．たとえば性能を追求すれば安全性は犠牲になる，といった関係である．非機能要求といっても，設計段階では機能として実現されるので区別する必要はそれほどないという考え方もあるが，要求分析段階ではやはり区別して考慮し，とくにトレードオフの関係を明確にしておくことは重要である．

3.4 仕様

要求を厳密に定義し記述したものが，要求仕様である．要求分析の段階では図式表現によるモデルを用いることが多いが，仕様の記述には自然言語あるいは形式仕様言語のようなテキスト表現が用いられることが通常である．

3.4.1 仕様と要求の関係

仕様が要求を厳密に記述したものとすると，仕様と要求は実質的に同じものと見なされるが，これをより明確に区別する考え方もある．たとえば M. Jackson は，図 3.7 に模式的に描くように，要求はシステムの外にある環境の現象について述べたものであり，一方プログラムはシステム内の現象について述べたものであるとする．そして仕様はそのシステムの内と外との境界上にあって，両者に共有される現象（事象や状態）について述べたものであるという [56].

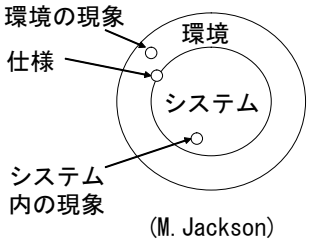


図 3.7: 要求と仕様の一つの見方

そして，仕様と外部世界に関する性質を前提とした時に要求が満足され，またソフトウェアとそれが動く機械の性質を前提とした時に仕様が満たされる，という関係にあるとする．それを論理学的に記せば

$$\text{仕様}, \text{環境の性質} \vdash \text{要求}$$
$$\text{プログラム}, \text{機械の性質} \vdash \text{仕様}$$

すなわち，システムの周囲の環境に関する性質と**仕様**を前提とすると，**要求**が充足されることを示すことができ，またプログラムが動く機械（コンピュータ）の性質と**プログラム**を前提とすると，**仕様**が満たされることを示すことができなければならない．この意味での要求は「目標 (goal)」に近い．

3.4.2　何を仕様として表すか

要求に機能要求と非機能要求があるのに対応し，仕様にも

- 機能仕様
- 性能仕様
- 信頼性仕様
- インターフェース仕様

などが考えられる．機能仕様に限ってみても，それを厳密に記述することは簡単なことではない．

たとえば数学的なソフトウェアに対しては，機能仕様を明確に書くことは比較的容易かもしれない．与えられた 2 つの正の整数の最大公約数を求めるという要求を考えてみよう．この要求を受け取る人（とりあえずソフトウェアの設計・開発者としておく）が，最大公約数という概念を知っていれば，すでに要求定義は済んでいる．そうでないとしたら，最大公約数の定義（2つの数の両方の約数となっているもののうちの最大値）を与えればよい．この表現でもまだ厳密性が乏しいと思う潔癖主義の人は，整数論の公理系にのっとった表現を用いればよい．

多くの場合，残念ながら仕様記述はこのように簡単ではない．その理由には次のようなものがある．

1. 対象とする問題領域が，数学の世界のように明確な論理性をもたない．
2. 「ユーザ」自身が，何をやってほしいか明確に意識していないか，意識していてもきちんと表現できない．
3. ユーザの意図が変化する．あるいは，外部条件の変化により意図が変わらざるをえない．
4. 対象となる問題が大きく複雑すぎて，どう記述したらよいか分からない．

また，実は最大公約数の問題でも，上に述べたような要求だけでは不十分だという点もある．それはたとえば，

1. どういう環境条件でソフトウェアを使うのか（コンピュータは何か，どこに置かれているか，誰が何のために使うか，など）が，述べられていない．
2. 入力や出力のインターフェースの形態が述べられていない．
3. 正常でない状況に対して（例えば，負の数が入力された場合，等）どう動作するか述べられていない．

3.4.3　仕様を誰が書き誰が使うか

要求を仕様として書く必要を感じる人間には，2 種類がある．1 つはその要求を他人に伝えて，ソフトウェアの製作を依頼する発注者である．もう 1 つは，ソフトウェアを製作する人間であり，発注者の意図を確認する目的や，市場型の製品の場合はその製品の機能と動作を明確に示すために記述する．記述された仕様は，後の設計開発過程ではその出力 (設計書やプログラム) が満たすべき要求条件として働く．

仕様は発注者と受注者の間の契約としての性格もあり，これが往々にして仕様記述をさらに難しくしている．契約である以上，一字一句でもゆるがせにすると後で手痛い目にあうかもしれない，と考えるからである．発注者と受注者が同一の人間である場合もある．その場合でも，意図を仕様としてきちんと記述することは，質の良いプログラムを作る上で欠かせない．

仕様を読む人間としては，ユーザ，設計開発者，検査者が考えられる．対象によって，記述の内容（ユーザニーズの記述，設計すべき内容，検査/検証の規範）や形式を変えることも考えられるが，実践されている例は少ない．ここで検査者とは，できあがった製品が仕様を満たしているか否かを検査する人を指す．ユーザ自身が検査者でもあることもあるが，多くの場合は独立した検査専門のスタッフが置かれる．

3.4.4　よい仕様の条件

よい仕様は，以下のような条件を満たさなければならない．

- 正当性
  内容に自己矛盾がないという斉合性（無矛盾性）と，もれなくすべてが記述されているという完全性とを含む．
- 厳密性
  曖昧なところがないこと．内容が一意に定まること．
- 読み易さ
  読んで理解しやすいこと．
- 検証可能性
  斉合性が検証できること，あるいは完全性を何らかの方法でチェックできること．自動的，機械的にできるとなおよい．また，仕様に基づいて開発されたシステムが，その仕様を満たすかどうかの検証が可能であることも，関連して重要である．
- 実現性
  具体的な実装が可能であること．いくら正当で厳密な仕様でも，およそ実現性のないものは仕様として意味がない．一方で，特定の実装方法に依存し設計を縛るような仕様記述は通常望ましくない．

しかし，問題なのは，これらの条件の間に，往々にしてトレードオフ関係が存在することである．たとえば，厳密性を追求すると一般には読みにくくなる．

第4章　モデル化技法とUML

ソフトウェアプロセスの章では，プロセスモデルという言葉を使った．要求分析では対象領域をモデル化するという手法を導入したが，それに対照する形で挙げた要求抽出型の要求分析でも目標指向型の要求分析でも，それぞれに応じた要求モデルを作るものと考えられる．

本章では，とくに要求モデルに限定せず，ソフトウェア開発の広い範囲で用いられるモデル化の技術について，その一般的な特徴を述べる．

4.1　モデルとは何か

M. Jackson がその著書 [56] で取り上げている「モデル」についての議論は，示唆に富む．まず Jackson は数学や論理学でいうモデルと，ソフトウェア工学でいうモデルとでは，現実世界と抽象世界との関係がまったく逆になっていることを指摘する．確かに，数学や論理システムでは理論に対しその実現となっているものをモデルという．たとえば群という抽象代数の理論に対して，加算が定義された整数はそのモデルの 1 つと呼ばれる．逆にソフトウェアの世界では，現実世界の問題領域を抽象化しなんらかの記述体系で表したものをモデルというのが普通である．この指摘は重要である．しかもソフトウェア工学の主要な一角を占めつつある形式手法は形式論理学と重なるから，混乱を引き起こす可能性は高い．

ソフトウェア技術者は論理学で使うような意味でモデルという語を用いてはいけないと Jackson は主張するが，さらに単なる抽象的記述をモデルと呼ぶのも疑問としている．抽象的な記述は，単に記述と呼べばいいのではないかというのである．

Jackson の立場は，それ自身が意味のある物理的存在で，対象となる現実世界の一部に対し類似の動作をするものをモデルと呼びたいというものである．たとえば，パイプ網を流れる流体の動作を電気回路で模したとすれば，その電気回路は物理的な実体であり，かつ流体という領域と類似するモデルになっている．計算機の中に作られるモデルは，物理的に実在するものであり，同時に実世界を模した動作をするので，まさに Jackson の意味するモデルになっているという．

Jackson の言うのももっともであるが，ソフトウェア工学では抽象的な記述そのものをずっとモデルと呼んできた．この後，本書でもとりあげるデータの流れモデルも，ER モデルも，状態遷移モデルも，みな抽象的な記述である．ここでは従来のソフトウェア工学の習慣に従い，これらの抽象的な記述をモデルと呼ぶことにする．すなわち，モデルとは，構造を持った対象を，その性質や動作を理解するために抽象化したものである．もちろん，そのモデルはコンピュータという物理的実体の上で動くソフトウェアとして，いずれ実現されるものであることは，常に意識するものとして．

4.2　ソフトウェアにおけるモデル

4.2.1　モデルという語の使われ方

ソフトウェア工学の世界では，モデルという言葉が偏愛されている．たとえば，1975 年に第 1 回が開かれたソフトウェア工学国際会議 (ICSE) の，過去 20 年間の論文タイトルに現れた単語頻度を調べた鳥居等による研究がある [111]．それによると，当然のことながら Software という語の出現頻度が他を引き離して多いが，それを除けば，次のような順序であった．

1. System,　2. Design,　3. Specification,　4. Model,　5. Analysis

Model という語は第 4 位にランクされただけでなく，1975〜94 年の 20 年間，はやりすたりなく使われているという特徴を持つ．たとえば第 2 位にランクされた Design は，この 20 年のうちのとくに前半に出現頻度が高く後半は減少するが，Model にはそのような変動がほとんど見られない．

それは「モデル」が他の用語と結合して使われてきたからであろう．結合する相手は，たとえば，

> life cycle, design, system, analysis, process, product, quality, domain, object, computation, data, data flow, entity-relationship, function, application, state transition, ...

とおびただしくある．これらの結合相手には，はやりすたりがあったが，モデルのほうは一貫して人気を保ってきたのである．

M.Jackson の「ソフトウェア博物誌」[56] の「モデル」の項でも，ソフトウェアの世界に限らず，モデルということばがいかに素敵な響きを持つものとして使われているかが列挙されている．

> 模範としてのモデル（将軍の鑑）
> 新型車のモデル
> 経済モデル，船のモデル，建築モデル，数学・論理モデル，プラモデル

一方，広辞苑の「モデル」の項を見ると，

> 型，模型，雛型，模範，手本
> 美術家が制作の対象とする人
> 小説・戯曲などの題材とされた実在の人物
> ファッション・モデルの略

となっていて，外来語としては珍しく，もとの使われ方との開きがほとんどないらしい．

4.2.2 モデルと抽象化

工学においてモデル化を行なうのは，対象となる問題領域 (domain) を理解するためにその領域自身をモデルとして構成する場合と，その上で実現するシステムや構築物/生産物を実際に開発する前に，モデルとして構築する場合とがある．建築家が建物を設計する前に発泡スチロールなどを材料としてモデルを作ったり，造船家が船の模型を作ったりするのは，典型的な後者の例である．しかし，建築かもその建物の周囲の町並みをモデル化して，そこに家の模型を置いてみたりする．その場合は，問題把握のための領域モデルと見ることができよう．領域モデルとシステムモデルは密接な関係にあり，同じ枠組の中で両者を同時に表現することも可能な場合があるが，多くのケースでは区別して考えた方がよい．

すでに述べたように，モデル化の鍵は対象の抽象化である．抽象化とは，裏返して言えば本質的でないものを捨てるということである．日本語ではそれを表すのに便利な「捨象」という言葉がある．モデルは，一般に構成要素とそれらの間の関係やそれらの動作の機構として表現されるが，対象の何に注目し，構成要素と関係や動作として何を選ぶかで，適切なモデル表現が定まる．システム分析や設計でよく用いられるモデル化技法を，対象のどのような構造や性質に着目して抽象化しているかという視点から分類してみたのが，以下である．

1. 事象（動作）の抽象化
2. ものと関係の抽象化
3. 情報の流れの抽象化
4. 機能の抽象化

これらはそれぞれかなり異なった対象に着目しているが，それを図式として表現する際には，グラフ構造という共通の形式が好んで用いられる．そこで次節では，この共通点と相違点に注意しながら，モデルのグラフ表現について一般的な視点から考えてみる．

4.3 グラフによるモデル化

4.3.1 さまざまなモデルのグラフ表現

多くのモデル化技法では，なんらかの図式表現が用いられる．図式の例として次のようなものを挙げることができる．ここでは，分析時のモデルだけでなく，概要設計やプログラム設計で使われるモデルの図式も含めている．

1. 制御の流れを表すもの (フローチャート，PAD, HCP, SPD, など)
2. データの流れを表すもの (データの流れ図，SADT，HIPO，など)
3. データ構造を表すもの (ジャクソン構造図，など)
4. データ関連構造を表すもの (ER 図，意味ネットワーク，クラス図，など)
5. 状態遷移を表すもの (状態遷移図，Statecharts, ペトリネット，など)

これらは皆，丸だか四角だか箱といった形が閉じた図形群と，それらを結ぶ折れ線だか点線だか矢印といった線群から構成されるという，共通の特徴を持つ．数学的に言えば，頂点集合と辺集合からなるグラフ構造をとる点が，共通である．

グラフ構造がこれだけよく用いられる理由として，次のような要因が考えられる．

1. 図として視覚化することで「モデル」らしく見える上，描きやすく直観的に分かりやすいこと
2. 対象となる世界の「もの」を頂点で表し,「もの」と「もの」との関係を辺で表すことにより，世界がグラフによって自然に表現されること
3. 「もの」と「もの」との関係には推移律や何らかの推移的な関係が成り立つことが多いが，それはグラフ上の経路の追跡することに対応させて理解でき，また経路探索のアルゴリズムもよく研究されていること

しかし，直観的に分かりやすいことが，恣意的であいまいな使用を助長している面もある．実際，状態遷移図やデータの流れ図の概念を学生に説明すると比較的容易に理解するように見えるが，試しに図を描かせてみると，とても状態遷移やデータの流れを表していないしろものができてしまうことがよくある．実は，学生ばかりでなく，かなりのベテランのソフトウェア技術者でもそういうケースを見かける．

図としてグラフ構造が愛用されるのは，実はソフトウェアの世界に限らない．新聞，雑誌，教科書，報告書，企画書，論文，発表スライドなどに登場する図の多くは，グラフ構造を持ったものになっている．そして，それらの図の意味は往々にして曖昧である．たとえば，矢印が何を表しているのか，因果関係か，時間関係か，物の流れか，制御の流れか，といった点が，同じ 1 つの図のなかで揺れているものをよく目にする．箱の方も同様で，プロセスのような処理主体と，データのような処理の対象が同じ箱だか丸だか表され，混在していることは日常茶飯事である．

グラフはなまじ直観的に理解しやすいがために，かえってその意味（semantics）をあいまいに捉えがちなのであろう．したがって，おなじグラフ構造であるという共通性と，その意味の違いを意識的に考えることが重要だと思われる．

4.3.2 グラフ表現によるモデル化の特徴

すでに述べたように，グラフ表現では通常，頂点に「もの」（概念，オブジェクト，プロセス，データなど）を結びつけ，辺にものともものとの関係を結びつける．その辺の役割によって，モデルを，構造を表す静的なモデルと振舞いを表す動的なモデルとに分類することもできる．

1. **静的なモデル**

   頂点 A と B を結ぶ辺が A と B とのを表す．辺が無向の場合は「A と B がある関係にある」ことを示し，有向の場合は「A は B とある関係をもつ」ことを示す．ER(実体関連）モデル，クラス図，意味ネットワークなどが，その代表例である．

2. **動的なモデル**

   頂点 A から B への辺が A から B への何らかの移動を表す．辺には向きがついている．A から B へ制御などの視点が移動する場合（制御の流れ, 状態遷移など）と，A から B へデータやものが流れる (データの流れ, ワークフロー，物流，交通流など) 場合とがある．

静的なモデルと動的なモデルを混同することは少ないが，動的なモデル同士，たとえばデータの流れと制御の流れ，状態遷移とフローチャートの間の混同はよく目にする．

表 4.1 に，代表的なモデル図式の頂点と辺の組み合わせを示す．

表 4.1: 代表的なモデルのグラフ構造

モデル	頂点	辺
データの流れ	プロセス	データの流れ
ER	実体	関連
状態遷移	状態	遷移
JSD（ジャクソン法)	プロセス	データストリーム結合 データベクトル結合
フローチャート	実行単位	制御の流れ
ペトリネット	プレース，トランジション	発火とトークンの流れ

4.3.3 グラフとモデル上の物理量

グラフは頂点と辺の関係というトポロジーを表している．しかし，グラフを用いてモデル化を行う場合は，頂点や辺に何らかの物理量を結びつけて扱うことが通常である．物理量といったが，ここでは「情報」を相手にしているから，多くの場合は物理量とは呼びにくい．しかし，たとえば電気回路のような典型的な物理を対象としたグラフ・モデルは，多くの示唆を与える．

電気回路の場合，頂点に電位，辺に電流が結びつけられる．辺には電位の差としての電圧も対応する．この物理量とトポロジーとを関係づける法則として，キルヒホフの法則 (Kirchhoff’s Law) がある．

1. キルヒホフの電流法則

   頂点の周りに流入する電流の総量 (流入を正，流出を負として計る）は，0 である．これは，電流が各辺の上を，停滞したりとぎれたりすることなく連続に流れるという，保存則を意味づけるものである．

2. キルヒホフの電圧法則

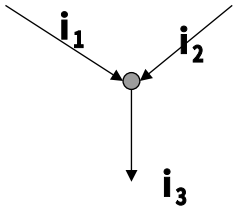
   閉路 (loop) に沿っての電圧の総和は 0 である．

これに関連して，辺の接続に関し直列と並列の 2 種類があるが，それぞれの接続における物理的な性質は，次のように記述することができる（図 4.2 参照)．これはキルヒホフの法則から導びかれる．

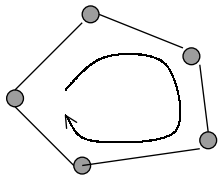
1. 直列：　電流は一定．電圧は加算
2. 並列：　電圧は一定．電流は加算

このような「流れるもの」とそれを駆動する力がそれぞれ辺と頂点に結びつけられるモデルは多いが，すべてのモデルがそれにあてはまるわけではない．しかし，キルヒホフの法則のように，

1. 各頂点の周りに出入りする辺全体についての性質を考える．
2. 辺が接続する経路に沿って成り立つ性質を考える．

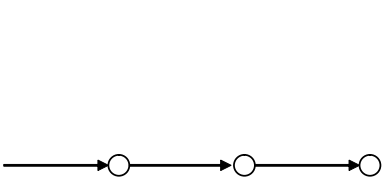


キルヒホフの電流の法則

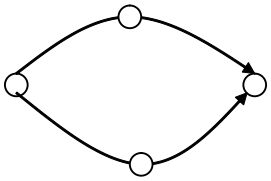


キルヒホフの電圧の法則

図 4.1: キルヒホフの法則



(a) 直列



(b) 並列

図 4.2: 直列と並列

ことは，基本的である．たとえばデータの流れモデル（第 5 章）の場合，頂点であるプロセスに入るデータとそこから出るデータをつき合わせて，プロセスのデータ変換機能を見ることができる．また経路に沿ったデータの流れを追うことで，一連のデータ処理の流れを把握できる．

4.3.4 グラフ理論の初歩概念の応用

グラフ理論というほど大げさなものは必要ないが，教科書の初めに出てくるような初歩概念は役に立つ．
まず，グラフの種類に，いくつかのものが考えられる．それによって，モデルの性質にも違いを生じる．

- 有向グラフ/無向グラフ
- 木，有向無閉路 (Directed Acyclic Graph, 通称 DAG)，一般グラフ

  木は階層構造と対応し，無閉路グラフは共通的な下部構造を持つ階層構造に対応する．一般グラフは階層構造を持たない，いわゆるネットワーク構造と対応する．
- 頂点集合の分割 (e.g. ペトリネット)，辺集合の分割 (e.g. PAD)

グラフの一部をなし，そこに含まれる辺の両端点を頂点集合として含むものを部分グラフという．部分グラフを 1 つの頂点に縮約すれば，その部分グラフをカプセル化，抽象化したグラフとなる (図 4.3)．これは頂点をさらに詳細化してもまたグラフ構造になっているという再帰的な構造を表すものである．これにより抽象の階層を取り扱う自然な枠組みが，提供される．実際，データの流れ図や Statecharts では，この方法が活用されている．

4.3.5 グラフの弱点と対策

原理的にはどんな複雑なモデルでもグラフとして図示できるとしても，実際に 1 枚の図に描ける頂点や辺の数は限られる．1 つの理由は物理的な制約で，紙やディスプレイの画面は限定された面積しかないから，いくら詰め込んでも高が知れた量しか表せない．さらに認知的な制約がある．よく引用される G. A. Miller の「魔法の数字 7 ± 2」[71] が主張するように，人間が自然に認知できるものの数は，案外少ない．また，頂点の数をある程度押さえても，辺の数が多くてクモの巣状になった図は，とても理解できない．
これはグラフによるモデル表現の弱点であるが，これに対して以下のようにいくつかの対策はある．

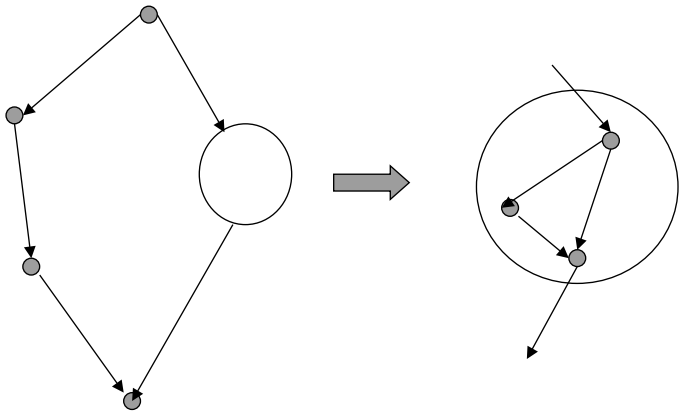


図 4.3: 部分グラフ

- 配置の工夫： 辺の交差をなるべく小さくするような配置や，その他の基準でグラフを見やすくするアルゴリズムがいろいろ提案されている．
- 画面操作の工夫： ズーミングやスクローリングが活用される．
- ハイパーグラフ： グラフの一般化として辺を 2 点間ではなく多点間の関係として扱うハイパーグラフを使うと，場合によっては図が簡素化される．
- 階層化：すでに述べたように，上位グラフの頂点を展開すると下位グラフとなるという階層構造を導入することで，1 レベルの図は簡素化される．この階層構造をツールで支援することも簡単である．

  関連して，頂点の箱に入れ子構造を描けるようにすることも，よく見られる工夫である．

  Statecharts では，この両者の工夫が採用されている．

4.4 UML 記法

UML(Unified Modeling Language) は，オブジェクト指向分析/設計のさまざまな方法論で用いられる図を中心とした多くの記法を統合化し，標準の構文と意味を与えることを目的として設計された「言語」である．言語といっても，文字よりグラフのような図形が主要な構成要素となる．最新の UML の言語仕様については，WWW [1] から入手できる．

4.4.1 経緯

1980 年代の終わりから 1990 年代の半ばにかけて，多くのオブジェクト指向方法論が発表されたが，中でも広く受け入れられたのが G. Booch の Booch 法と，J. Rumbaugh 等の OMT 法である．Rumbaugh が Booch のいた Rational Software 社に移ることにより，この両者を合併して統合化された方法論を作る努力が始められ，1995 年 10 月に，Unified Method として暫定版が公表された．同じ頃，やはり独自の方法論 OOSE を普及させていた I. Jacobson が，自らの会社 Objectory 社とともに Rational 社に加わり，この統合化に OOSE も組み込まれることになった．その結果，オブジェクト指向方法論で用いられる記法の集大成としての UML が，1996 年 10 月に UML0.9 として発表された．

UML の影響力を評価した OMG(Object Modeling Group) は，標準モデル化言語の提案募集を行った．それに対し Rational 社に加えて IBM, HP, TI, Microsoft など多くの企業がパートナーとして参加し UML1.1 という仕様をまとめ提案したが，それが OMG により 1997 年 11 月に標準として採用された．以降，UML の改訂は OMG を舞台

として進められている．2003 年 3 月に第 1.5 版の仕様が発表されたが，その時点で第 2.0 版もほぼ完成に近い状態である．

4.4.2 UML の概要

UML はオブジェクトに基づくモデルの種々の側面を，それぞれ図を中心とした記法で表す．図には大きく分けると，オブジェクトの静的な構造を表すものと，動的な振舞いを表すものとがある．表 4.2 に UML1.5 で定義されている 8 種の図を示す．クラス図と実装図に含まれるものが静的モデルであり，振舞い図に分類されたものが動的モデルである．ユースケース図はユースケースとその主体となるアクタの関係をモデル化したという意味では静的だが，ユースケース自身は振舞い図につながる動的な性格を持つものである．

表 4.2: UML 図

ユースケース図	
クラス図	
振舞い図	Statechart 図
	動作図
	系列図
	協調図
実装図	コンポーネント図
	配備図

UML に対しては，次のような批判もある．

1. 意味 (semantics) がきちんと定義されていない．
2. 単なる記法で方法論がない．
3. さまざまな図式の寄せ集めでごたごたしている．

批判 1 に対しては，現在の UML 仕様書では図を構成する基本要素のメタモデルを与えており，それで完全に意味が定まるわけではないが厳密さを増してはいる．批判 2 に対しては Jacobson による統合プロセス（Unified Process)[59] やそれをベースにした Rational Rose 社の Rational Unified Process(RUP) といったものも発表されている．また以前からあるさまざまな方法論が，記法として UML を採用する傾向も顕著に見られ，UML が方法論から独立していることが逆に強みとなっているともいえる．

批判 3 はかなりもっともなところがある．しかし，本書では，UML をこれまでの分析や設計ノウハウを図を中心に集大成したものと捉える．その立場からすれば，UML の各種の図は取捨選択して適宜用いればよく，また 1 つの図式に関しても，UML で定義されているすべての記法を用いる必要もない．本書ではまた，モデル表現のグラフ構造に注目することから，UML をグラフ構造を持つモデル表現法のカタログとも見なし，それらのグラフ表現の共通性や差異にとくに注意を払うことにする．UML の代表的な図のグラフ構造は表 4.3 のようである．

5 章以降で，さまざまなモデル化技法を，対象の何に着目しているかという観点から整理して説明していく．その中で，それぞれの技法に向いた UML の図式を取り上げる．

4.5 共通例題

これ以降取り上げるモデル化技法の説明と比較のために，共通の例題を用いることにする．1 つは代表的な業務用システムの倉庫管理の問題であり，もう 1 つは代表的な応答型 (reactive) システムの自動販売機の問題である．

表 4.3: UML 図のグラフ構造

	頂点	辺
クラス図	クラス	汎化，集約，関連
状態遷移図	状態	遷移
動作図	動作状態	制御の流れ
協調図	オブジェクト	メッセージの流れ
系列図	メッセージ受発信の時点	メッセージの流れ

4.5.1 酒屋問題

1984 年に，情報処理学会のソフトウェア工学研究会で，種々の設計技法の比較のために「酒屋の在庫問題」という例題が作られた．作成されたのは山崎利治氏 (当時，日本ユニシス) である．その後，多くの論文や研究発表でこの例題が取り上げられたが，種々の設計技法を横並びで比べるという意味でもっとも有益な文献は，学会誌「情報処理」の 3 つの号で組まれた特集 (「情報処理」25 巻 (1984)9 号，同 11 号，26 巻 (1985)5 号) である．

とにかく 20 種類近くの手法を，同じ例題によって横並びに見られるというのは貴重である．しかも，それぞれの著者が各手法の熟練者である上に，実に真剣に課題に取り組んでいる．問題がまたよくできている．事務管理の分野として典型的だが自明でないというものであり，短い文章で充分な問題提示をしており，しかも適度な曖昧性も含む．

まず，問題をもとの形で引用する．

> ある酒類販売会社の倉庫では，毎日数個のコンテナが搬入されてくる．その内容はビン詰めの酒で，1 つのコンテナには 10 銘柄まで混載できる．扱い銘柄は約 200 種類ある．倉庫係は，コンテナを受け取りそのまま倉庫に保管し，積荷票を受付係へ手渡す．また受付係からの出庫指示によって内蔵品を出庫することになっている．内蔵品は別のコンテナに詰め替えたり，別の場所に保管することはない．
>
> 空になったコンテナはすぐに搬出される．
>
> 積荷票: コンテナ番号 (5 桁)
> 搬入年月，日時
> 内蔵品名，数量 (の繰り返し)
>
> さて受付係は毎日数十件の出庫依頼を受け，その都度倉庫係へ出庫指示書を出すことになっている．出庫依頼は出庫依頼票または電話によるものとし，1 件の依頼では，1 銘柄のみに限られている．在庫がないか数量が不足の場合には，その旨依頼者に電話連絡し，同時に在庫不足リストに記入する．そして当該品の積荷が必要量あった時点で，不足品の出庫指示をする．また空になるコンテナを倉庫係に知らせることになっている．
>
> 出庫依頼: 品名，数量
> 送り先名
>
> 受付係の仕事 (在庫なし連絡，出庫指示書作成および在庫不足リスト作成) のための計算機プログラムを作成せよ．
>
> 出庫指示書： 注文番号
> 送り先名
> ⎡ コンテナ番号 ⎤
> ⎢ 品名，数量 ⎥ (の繰り返し)
> ⎣ 空コンテナ搬出マーク ⎦
>
> 在庫不足リスト: 送り先名
> 品名，数量

- なお移送や倉庫保管中に酒類の損失は生じない．

- この課題は現実的でない部分もあるので，入力データのエラー処理などは簡略に扱ってよい．
- 以上あいまいな点は，適当に解釈して下さい．

問題は簡明に記述されているが，見かけほど単純ではない．記述の中で，少し分かりにくいところがある．それは「そして当該品の積荷が必要量あった時点で，不足品の出庫指示をする．」という部分である．よく読まないと「不足品の出庫指示」という意味が分からない．不足なのにどうして出庫できるのか? もう一度読み返してみると，この文は，その直前の文に続いて，出庫依頼があったが在庫が不足のケースを扱っているのだということが分かる．ある時点で不足であったが，その後，入荷があって不足分が充足された時点の話をしているわけである．これがそう素直に読めないのは，在庫がある場合は直ちに出庫指示を出し，在庫不足の場合は出庫指示を出さずに依頼者に連絡するなどの処理をするという場合分けが，明示的に書かれていないからである．

実は，最初に出された問題の記述はこうではなかった．「そして当該品の…」という一文はもともとの問題にはなく，その改定版が出された時に，付け加えられたものである．これが加えられた理由は，不足の場合の処理が，そのまま不足連絡するだけなのか，在庫のある分は注文量に足りなくても出荷するのか，後に在庫が満たされた時点で出荷するのか，といった点があいまいだという指摘が，最初にこの問題に取り組んだ人たちから出たので，それに応じたものだという．その経緯を知っている人にはこの文章は素直に読めるだろうが，最初にこの改定版を見た人には分かりにくいだろう．これは要求仕様のような文書を部分的に変更すると全体の整合に問題を生じるという，一般に文書やプログラムの変更に伴って生ずる問題を，図らずも示したものといえるかも知れない．

4.5.2 自動販売機問題

オブジェクト指向シンポジウム'97 のモデリングワークショップで共通問題として使われた「自動販売機制御ソフトウェア問題」(鰺坂・野呂作成) [5] を，大幅に簡略化し，一部手直しした．

1. 自動販売機は，紙幣および硬貨の投入を受け付ける．紙幣は千円札のみ，硬貨は 10 円以上のものが使用可能．
2. 投入された金額から販売済み代金を差し引いた現在残高を，表示器に表示する．
3. 現在残高で購買可能な商品のうち，売り切れでないものは，その商品に対応する販売ランプを点灯する．
4. 売り切れの商品は，その商品に対応する売り切れランプを点灯する．すべての商品が売り切れの場合は，金銭の投入ができない．
5. 販売ランプが点灯している商品の購入ボタンが押されると，当該商品の販売ランプは点滅し，商品が取り出し口に 1 つ出される．同時に，残高は商品価格分が減額される．この間は，金銭の投入はできない．
6. 返金ボタンを押すと，その時点での残額がつり銭取り出し口に返金される．返金動作中は，金銭の投入はできない．

この問題は，3.3.2 節に挙げた「飲物自動販売機のシナリオ」に対応している．シナリオ記述との差異は,「すべての商品が売り切れの場合は，金銭の投入ができない.」といった制約条件の記述が入っている点と，例外的なケースもカバーしている点である．

第5章　データの流れモデル

本章以降で，代表的なモデル化技法を順に取り上げる．少し大げさな物言いをすれば，モデル化とは世界をどう捉えるかということである．といっても形而上的な話ではなく,「世界」もきわめて限定され切り取られた領域である．しかし，これからのモデル化技法の説明では，あえて「世界をどう見るか」という言い方で，各モデルの特徴を示すことにする．

データの流れモデルでは，世界にはデータが流れていると見る．流れは分岐したり合流したりする．流れの過程でデータは形を変える．その形を変換させるものをプロセスと呼ぶ．

5.1　データの流れ図

データの流れ図 (Data Flow Diagram, DFD と略称)[1]は，データの流れとその変換を記述する図である．グラフによるモデル化の分類でいえば，

データの流れ図	
モード:	動的
対象:	ものの流れ
頂点:	プロセス
辺:	データの流れ
部分グラフ:	下位プロセス

である．

データの流れは，図 5.1 のような有向辺で表される．流れるデータは，その名前を辺の脇に書くことで示され

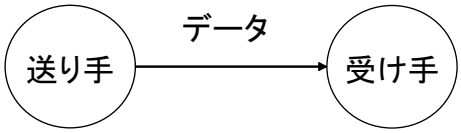


図 5.1: データの流れ

る．データは「送り手」から「受け手」に流れるが，この送り手と受け手はプロセスと呼ばれる．プロセスは，丸の中にプロセス名を書くことで表される．

データの流れ図をプロセスに着目してみたのが，図 5.2 である．プロセスはデータ A をデータ B に変換してい

図 5.2: プロセス

る．すなわち，データ A はプロセスへの入力であり，データ B はプロセスからの出力である．

プロセスへの入力や出力は，図 5.3 に示すように，一般に複数あってもよい．ここで，複数の入力と出力の間の関係は，このデータの流れ図では規定されないことに注意が要る．たとえば，出力 1 が入力 1 から作られるのか，

[1]日本語でもデータフロー図と呼ぶほうが一般的である．

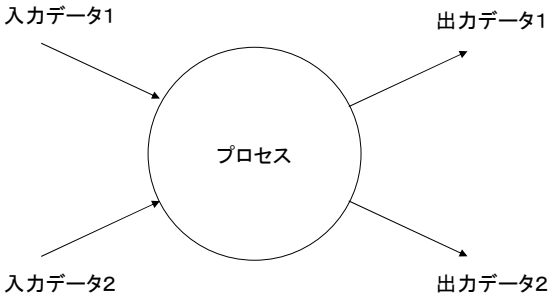


図 5.3: プロセスへの複数の入出力

入力 2 から作られるのか，あるいはその両者から作られるのかは分からない．出力 2 についても同様である．さらに，出力同士の関係も，出力 1 と出力 2 が一緒に出されるのか，出力 1 が出るときは出力 2 は出ず，出力 2 が出るときは出力 1 が出ないというような排他的な関係にあるのか，ということも定めていない．

データの流れ図は，入出力同士の論理的な関係を規定していないだけでなく，データが入ったり出たりするタイミングについても，まったく定めていない．入力や出力は常に連続的に流れているのかもしれないし，ある時点でまとまってデータが入ったり出たりするのかもしれない．2 つ以上の入力データも，同期して同時に入ってくるのかもしれないし，ばらばらに勝手に入ってくるのかもしれない．あるいは，入力 1 が常に入力 2 に先行するというような決まりがあるかもしれない．複数の出力についても，それらの出力順序やタイミングについてはなんら規定がない．

データの流れの中身は，プロセスによる操作の対象となるような内容を持ったデータで，プロセスの起動制御，割り込み，同期信号のようなタイミングを制御するための事象は含めない．それでは応答型のシステムや実時間システムのモデル化には不便だというので，DFD を拡張して制御データや制御用のプロセスを区別して記述する記法を導入した Hatley 法や Ward 法も提案されたが，現在それらのモデル化手法はほとんど用いられず，状態遷移モデル系の手法（たとえば Statecharts）に取って代わられている．

DFD が持つこのような性質は不便のように見えるが，これこそがモデルによる抽象というものである．入出力の関係やタイミングが重要なら，このモデルを使うべきではない．しかし，データの流れモデルは，それらを捨象することによって，どのようなデータがプロセス間を流れ，そこでどのような変換を受けるかという構造に着目する場合は，簡便で強力な記述力を持ったものとなる．

5.2 階層化

データの流れモデルには，プロセスを分解するとまたデータの流れ図になるという再帰的な関係があり，それによって自然な階層化が可能となる．この関係は，部分グラフによるグラフの階層構造の典型例となっている．

たとえば，図 5.4 のような，請求書発行のプロセスを考える．このプロセスを分解して，図 5.5 のようなデータ

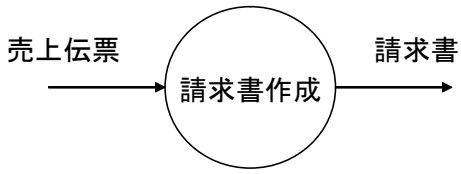


図 5.4: 請求書発行プロセス

の流れ図を想定することができる．ここでは，外部からの入力は売上伝票，外部への出力は請求書という関係 (インタフェース) は図 5.4 と変わらない．しかし，中の構造が 3 つのプロセスと 1 つのファイルに分解されている．ファイルとは，データを溜めておくところである．データの流れモデルでも，データは流れているばかりでなく，溜まるところがどうしても必要になる場合がある．ただ，データの流れモデルの本質からいえば，ファイルは必

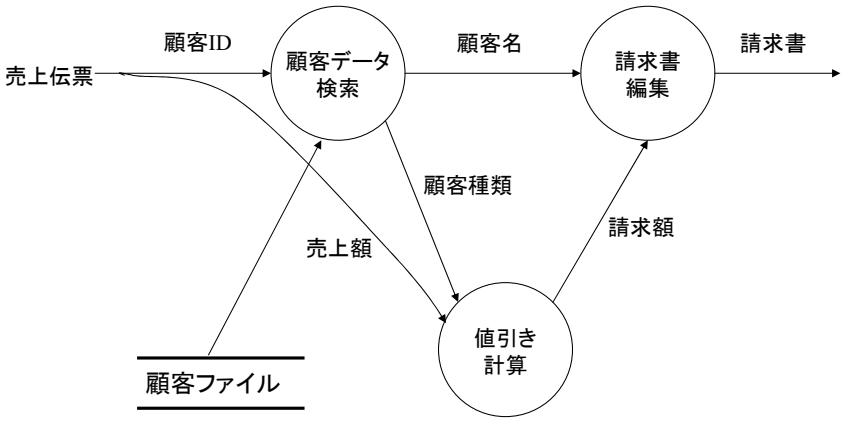


図 5.5: 請求書発行プロセスの詳細化

要最小限に留めるべきである．

さらにここでは，上のレベルでは「売上伝票」というデータであったものを，その内部の構成要素である「顧客ID」と「売上額」に分解して示している．このようなデータの構造については，後述するデータ辞書で管理する．

5.3 Demarco による構造化分析

デマルコによる，データの流れ図 (DFD) を用いた技法は，構造化分析（Structured Analysis, 略して SA）と呼ばれる．これは，データの流れの構造に着目した分析技法の代表例である．Demarco の著書 [33] が基本的な教科書であり，良書である．

SA はオブジェクト指向以前の手法であり，DFD が UML に採用されていないこともあって以前ほど重視されない傾向も見られる．しかし，次のような特徴をもち，現在でも適切に用いれば有効である．

1. 面倒な理論も必要とせず，直観的に分かりやすい．
2. 機能の分解を階層構造として自然に書ける．
3. 図がデータの流れを追うのを助けるとともに，機能は入力と出力の組み合わせによって理解できる．
4. 市販されている CASE(Computer Aided Software Engineering) では, ほとんどがデマルコ流の DFD をサポートしている．

デマルコの方法以外に，データの流れに基づく構造的な仕様化技法としては，D. T. Ross による SADT(現在では IDEF0 として規格化）がある．また，DFD の記法としては Gane-Sarson のものもあるが，本質は変わらない．

SA では，次のような道具を使う．

データの流れ図 (DFD) 図は，次の 4 種類の基本構成要素の組合せで描かれる．

- プロセス: 丸い枠．バブルとも呼ばれる
- データの流れ: 矢印．プロセスとプロセスまたは外部実体またはファイルの間を結ぶ．
- 外部実体: 対象とするシステムの外部にある実体．
- ファイル: データを溜めておくところ．データストアなどともいう．

このうちでも，とくに前節で述べたプロセスとデータの流れが基本的である．

データ辞書 データの内容と構成を定義し，記録したものである．その構造は，列，選択，反復の組合せで定義される．これは記述力として正規表現と等価であり，Jackson によるジャクソン構造図 (7.3.4 節参照) で描くこともできる．

[例] ここでは，列，選択，反復を構成する演算子をそれぞれ，$+, |, *$ で表している．

$$\text{売上伝票} \quad = \quad \text{書籍番号} + \text{顧客番号} + \text{冊数} + \text{金額}$$

$$\text{顧客番号} \quad = \quad [\text{A} \mid \text{B} \mid \text{C}] + \{\ \text{数字}\ \}^*$$

プロセス仕様記述 階層化された最下位レベルのプロセスの定義は，DFD 以外の方法で記述しなければならない．そのためには，構造化英語，決定表，決定木，状態遷移図，形式的仕様記述言語などを用いる．

5.4 データの流れ図の記述方法

構造化分析の方法に基づいて，DFD の記述法を共通例題の酒屋問題を使って説明する．

まず全体文脈図 (context diagram) を描く．全体文脈図は，外部実体とシステム全体の関係を示すものである．そこには原則的に 1 つ以上の外部実体と 1 つのプロセス (すなわち対象とするシステム全体をプロセスと見たもの) が現れる．

問題文に主語として現れるものは，データの流れの送り手となるプロセスまたは外部実体の候補である．また述語として現れるものは，プロセス間のデータの受け渡し動作を指している可能性が高く，その動作の与格（～ に）に当るものはデータの流れの受け手となるプロセスや外部実体の候補となる．さらにその動作の対格 (～ を）に当るものは，プロセス間を流れるデータの候補である．

- 「倉庫係は，コンテナを受け取り」
  ⇒ 倉庫係が受け手のプロセスの候補．コンテナがデータの候補．
- 「(倉庫係は）積荷票を受付係へ手渡す」
  ⇒ 倉庫係が送り手のプロセスの候補．受付係が受け手のプロセス候補．積荷票がデータの候補．
- 「(倉庫係は）受付係からの出庫指示によって」
  ⇒ 受付係が送り手のプロセスの候補．倉庫係が受け手のプロセス候補．出庫指示（書）がデータの候補．
- 「(倉庫係は）内蔵品を出庫する」
  ⇒ 倉庫係が送り手のプロセスの候補．内蔵品がデータの候補．
- 「受付係は（依頼者から）毎日数十件の出庫依頼を受け」
  ⇒ 依頼者が送り手のプロセスの候補．受付係が受け手のプロセスの候補．出庫依頼がデータの候補．
- 「(受付係は）倉庫係へ出庫指示書を出す」
  ⇒ 既出
- 「(受付係は）在庫がないか数量が不足の場合には，その旨依頼者に電話連絡し」
  ⇒ 受付係が送り手のプロセスの候補．依頼者が受け手のプロセスの候補．電話連絡がデータの候補．
- 「(受付係は）在庫不足リストに記入する」
  ⇒ 受付係が送り手のプロセスの候補．在庫不足リストがデータの候補．
- 「(受付係は）空になるコンテナを倉庫係に知らせる」
  ⇒ 受付係が送り手のプロセスの候補．倉庫係が受け手のプロセスの候補．空になるコンテナがデータの候補．

これを素直に図示すると，図 5.6 のようになろう．

図の中で，四角で囲んだものが外部の実体で，丸で囲んだものが対象とする範囲 (システム) を表すプロセスである．全体文脈図では外部実体はシステムの環境を構成するものという位置づけなので，通常は外部実体同士の情報のやりとりは表す必要がない．しかし，環境がどのように働くかを理解するためには，外部実体同士の相互作用も必要範囲で表現してあると便利である．そこで図では，倉庫係へのコンテナの出入りや倉庫係と依頼者とのやりとりも示している．あるいは，倉庫係は外部実体というよりはシステムの内部にいるものと考えて，これを

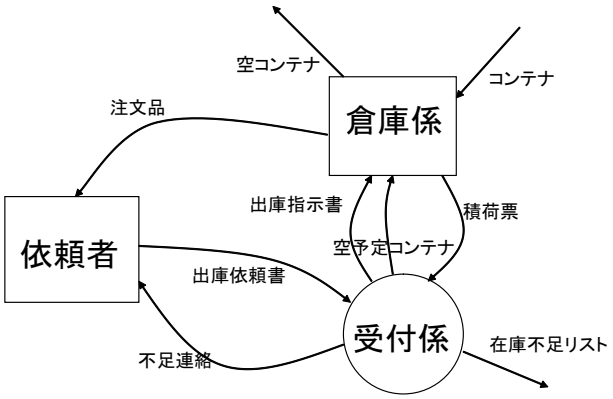


図 5.6: 酒屋の在庫問題のデータの流れ図

丸で囲ってもよい．デマルコの方法では，全体文脈図にはプロセスは 1 つだけ書くように規定されているが，システムの境界内のプロセスが最初から複数見えている場合には，その規則にこだわる必要はない．また，データの流れという意味では，ものの流れは除外されるべきであるが，問題の構造を捉えるには必要なものの流れも記述されていたほうが判りやすい場合がある．コンテナとか注文品というものの流れを図に描き加えたのは，そのような意図による．

このように，問題文がある程度整理されていると，そこから全体文脈図を作ることはかなり自然にできる．その結果得られる図は，対象とするシステムとその周囲の環境との関係を明示するものとして便利である．そのため，全体文脈図は別に構造化分析手法に限らず．多くの開発手法の最初の段階で活用されている．

次に，そのシステムの一番粗い分解を示すものが，トップレベル図である．そこに現れるプロセスには，1 から順に番号がふられる．酒屋問題では受付係の仕事を分解した図 5.7 が，トップレベル図となる．

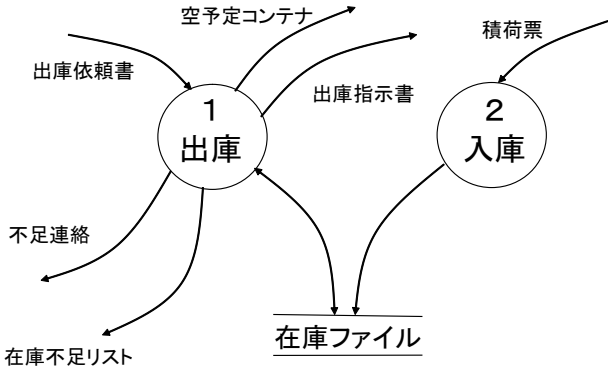


図 5.7: 酒屋の在庫問題のトップレベル図 —その 1—

しかし，この図は問題の古い版に対応している．在庫不足の場合，あとからの入荷によってその注文に応じられるようになった時点で出庫するようにするとすれば，トップレベル図はたとえば図 5.8 のように変わるだろう．すなわち，在庫不足リストを外部への出力とするのではなく，内部ファイルとし (それに応じて全体文脈図は自明な変更が必要になる)，それを入荷時に参照してしかるべき処理をするプロセスを加えることになる．

次にプロセス i の分解図を作る．その外部とのデータの流れのやりとりは，上のレベルで示された対応するプロセスが周囲のプロセスとやりとりしているデータフローと完全に整合がとれていなければならない．プロセス i の分解図に現れるプロセスには，$\mathrm{i.i}_k$ のような章節番号システムと同じ形式の番号がふられる．

酒屋問題のトップレベルのプロセス 1(出庫) を分解したのが，図 5.9 である．これは図 5.7 の問題その 1 に対応したものである．

ここで「在庫あり出庫依頼」と「在庫なし出庫依頼」という表現に注目してほしい．データの内容としては同じ「出庫依頼」であるが，その性質を修飾詞をつけて明示するように工夫したものである．これにより「出庫受

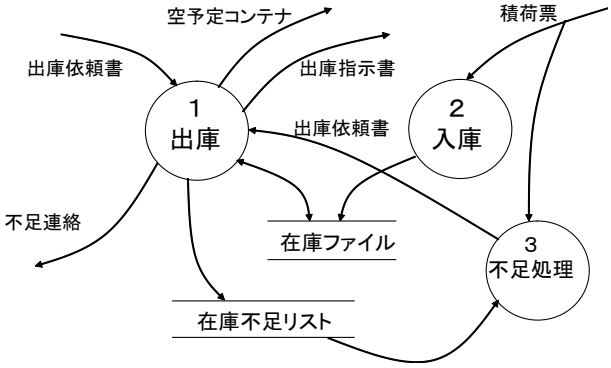


図 5.8: 酒屋の在庫問題のトップレベル図 —その 2—

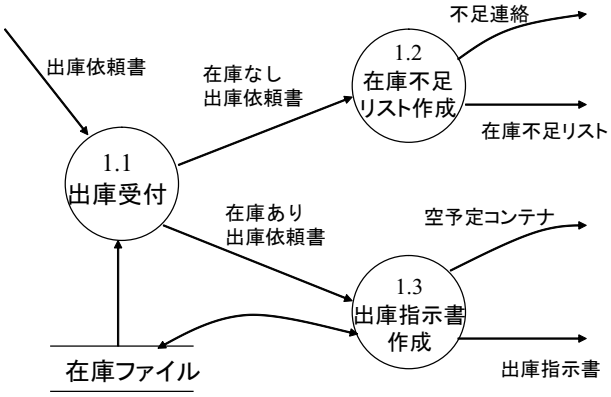


図 5.9: 酒屋の在庫問題の第 2 レベル図

付」の後のデータの流れが，二者択一の分岐であることが暗示される．もちろんこのような修飾詞の工夫だけでは曖昧性が残る．それを厳密に記述するのに，データ辞書を用いる．データ辞書の「在庫あり出庫依頼」の項目に，出庫依頼データの内「在庫あり」という条件（この条件は論理式で厳密に記述することが可能）を満たすもの，という定義を書き入れればよい．

さらに 1.1 や 1.3 のプロセスを分解しようとすると,「在庫ファイル」というファイルの構造を分解しなければならないことがわかる．そこで 1.1 や 1.3 のプロセス分解に応じてファイル構造も分解し，それはまたデータ辞書に登録していく．

このような構造的分解は，プロセスが単一の仕事を表すまで繰り返す．これ以上分解しないと定めたプロセスに対しては，プロセス仕様記述を行う．何をもって単一の仕事と見なすかについては，自由度がある．それはまた，プロセス仕様記述に何を用いるかにもよる．たとえば状態遷移図をプロセス仕様記述に用いるなら，1 枚の状態遷移図に収まる程度に分解できたところで停めるという基準である．

どこまで分解するかはまた，データの流れモデルを何のために用いるのかという目的にも依存する．構造化分析では，プロセス仕様のレベルはプログラムの手続き（サブルーチン）レベルが想定されていたが，要求モデルとして用いる際は，もっと抽象度の高い段階で分解を停めることになろう．

5.5 モデルの検証

データの流れモデルが意図どおりのものを表現し，またプロセスの分解が正しく行われているかを見るには，主に 2 つの方法がある．これは 4.3.3 節で述べたグラフの性質と関連する．

1 つはデータの流れの経路に沿って追跡し，そこに現れるデータの一連の変換列に問題がないか検査するという方法である．たとえば図 5.9 では，外から入力された出庫依頼が「在庫あり」のケースで流れていく経路と,「在庫

なし」のケースで流れていく経路があるが，それぞれについてその一連のデータの変換が妥当かどうか検証する．

もう 1 つは，プロセスに注目して，そこへの入力データの組から出力データの組を正しく作ることができるかどうか分析する．入力データに不足するものや不要なものが見つかったり，出力データで欠落しているものやここで出力すべきでないものが見つかったりする可能性がある．

これらの検証は，通常は人手により非形式的に行われるが，それでもモデルの品質を高めるのに有効である．

その他に，以下のような点に注意するとよい．

- データの流れは，プロセスに入るデータあるいはプロセスから出るデータというプロセスとデータの関係のみを表したもので，処理の順序やタイミングを表したものではない．だから，制御の流れを表すフローチャートとは，本質的に異なる．
- モデル化を考える順序として，デマルコはまずプロセスを考えるのではなく，データの流れを考え，それを変換するものとしてプロセスを後から考えた方がよい，としている．
- プロセスにおけるデータ間の関係，たとえばどの入力を用いてどの出力が作られるか，複数の入力や複数の出力は互いに共存 (and) 関係にあるのか，選択 (or) 関係にあるのか，などは，直接には示さない．それらは，さらに下位の構造（最終的にはプロセス仕様）を見ることにより，分かる．
- 1 つの階層レベルには，理解しやすい範囲の複雑さを持ち（たとえばプロセス数で 3–7），概念としてもそろった水準のものを描く．
- 階層間のデータの流れのバランスをチェックすることで，検証ができる．
- データやプロセスには，必ず名前をつける．それも，意味をよく表す具体的な名前でなければならない．たとえば，データは名詞，プロセスは動詞.「なんとかデータ」とか「なんとか処理」という名前はよくない．
- ファイルは，そのレベルのプロセス間でやりとりがなされるもののみを記入する．下位レベルでのみ使うものは入れない．
- データの流れが入るだけあるいは出るだけのプロセスやファイルは，普通は存在しない．

演習問題

次の問題をデータの流れモデルを使ってモデル化せよ．

［書店の売り掛け管理］　ある書店では，特定の顧客に対して書籍の掛け売りを行なっている．

販売係は，書籍とともに納品伝票を顧客に渡し，すぐにその写し（売上伝票）を会計係に送る．会計係は，月末になるとその月の売上伝票を集計し売掛残高報告書を作成すると同時に，顧客に請求書を送ることになっている．

顧客は，現金や銀行振込などによって請求額を支払うが，一度に全額を支払うとは限らない．入金額は財務係から随時入金伝票として会計係に回送されてくる．

- 売上伝票： 顧客名；売上額
- 入金伝票： 顧客名；入金金額
- 売掛残高報告書： 前月末総未済残高；当月総売上高；当月総入金高；当月末総未済残高
- 請求書： 顧客名；前月ご請求額；当月お買上額；当月お支払額；当月ご請求額

会計係の仕事を分析し，システム化する．

（山崎利治「プログラムの設計」[117] より）

第6章　制御の流れモデル

制御の流れモデルは，世界を意味的にまとまりのある作業 (処理) が順次実行されているところと見る．その全体の作業の手順は制御され，ある作業の次にどの作業を行うかという順序，場合に応じた作業の選択，一定の手順の繰り返し，いくつかの作業の並行的な実行といった制御の構造が定められている．ここでの作業はデータの流れモデルのプロセスにほぼ相当する．しかしこのモデルでは，プロセス間にどのようなデータが受け渡されるかという点は捨象し，プロセスがどのような順序で実行されるかにのみ着目する．

6.1　フローチャート

制御の流れを記述する図で昔からよく使われているものは，フローチャートである．

フローチャート	
モード:	動的
対象:	視点の移動
頂点:	処理，判断
辺:	制御の流れ
部分グラフ:	サブルーチン

図 6.1 に例を示す．

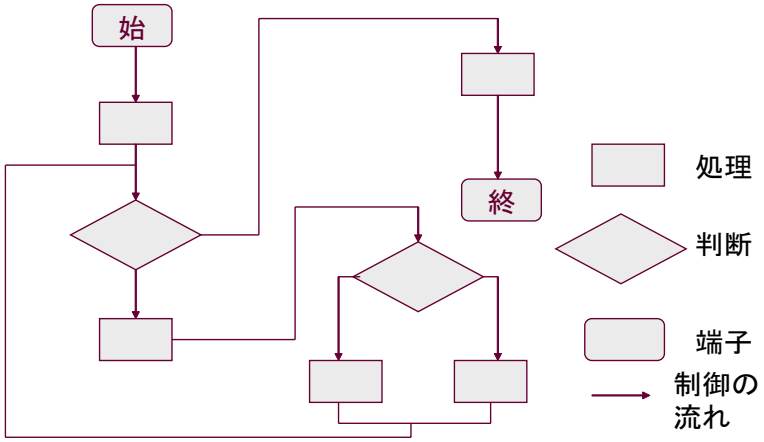


図 6.1: フローチャートの例

フローチャートでは，頂点には処理（長方形）または判断（菱形）が対応し，辺には制御の流れが対応する．さらに頂点の特殊なものとして処理の開始や終了点を表す端子（角の丸い箱）や，この図にはないがディスクなどの処理装置や媒体をさまざまなアイコンで描くが，あまり本質的ではない．

フローチャートでは処理を実行する計算機（人を含む他の情報処理機械でもよい）を想定し，その実行の制御がどのように移動するかを主要な記述対象とする．実行は逐次的であり，したがって現在，実行の制御がどの頂点にあるかがユニークに定まる．具体的な処理が行われる場所は箱で描かれた「処理」の中であり，辺は次に実行がどこに移るかを示すだけである．「判断」の後では制御の流れの分岐が起こり，また 2 つ以上の辺が流れ込む頂点は，分岐の合流点か反復実行の開始点かに対応する．

フローチャートの辺で表される制御の流れは，プログラミングレベルで言えば goto に対応する原始的なものである．1970 年代の構造化プログラミングの時代には，goto の使用によって 制御の流れが錯綜したプログラムは悪いとされ，したがってそれに対応するフローチャートも排斥された．それに代わり，分岐構造と反復構造を備えた制御の流れの記述法として，PAD, HCP, SPD などの多くの提案が出された．それらの記述を支援するツールも作られ，さらに図からプログラムを生成する機能も付加されて利用された．

6.2 動作図

UML の動作図 (activity diagram) は，制御の流れを表すものであり，フローチャートに近い．

動作図	
モード:	動的
対象:	視点の移動
頂点:	処理，判断
辺:	制御の流れ
部分グラフ:	下部手続き

ただし，フローチャートと異なる以下のような特徴をもつ．

1. フローチャートのようにプログラミングレベルのみに対応するものではなく，概念レベルのモデルに使用することが想定されている．
2. 並行動作の記述を可能としている．

その主な構成要素を，図 6.2 に示す．

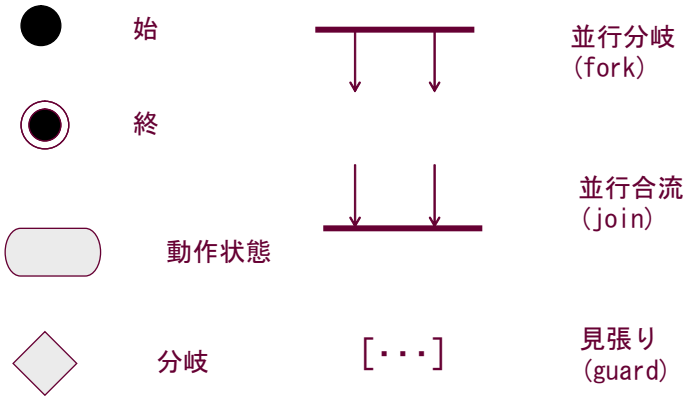


図 6.2: 動作図の構成要素

動作図はフローチャートと同様，基本的に手順を示すものである．面白いことに，動作図の制御の流れはフローチャートと同様に goto に相当する原始的な制御の移動のみで構成され，分岐構造や反復構造は用意されていない．共通問題の自動販売機を動作図として描いてみたのが図 6.3 である．

この動作図を 3.3.2 節のシナリオと比べてみると，利用者が途中でやめるケース，金を追加するケース，飲物を取り出した後さらに購入を続けるケース，その他の例外的なケースがいくつか含まれ，より一般的な状況が記述できている．一方で，その分だけ制御の流れが複雑になって，多少判りにくさが増している．

フローチャートの場合は，動作の主体は計算機（あるいはそれに相当するもの）で 1 つに決まっており，動作の主体が誰かということは問題とならなかった．しかし，動作図をオブジェクト指向の枠組みで用いる場合は，さまざまなオブジェクトが動作の主体となりうる．その動作主体を明示するために，プールのコースのような区分けをする記法が定められている．たとえば，図 6.4 のような描き方である．

もう 1 つの共通問題である酒屋問題を，この動作主体でコース分けするやり方で描いてみたのが，図 6.5 である．酒屋問題の動作は，コンテナが入庫する場合の動作と，顧客の注文に応じて酒を出庫する場合の動作とに大

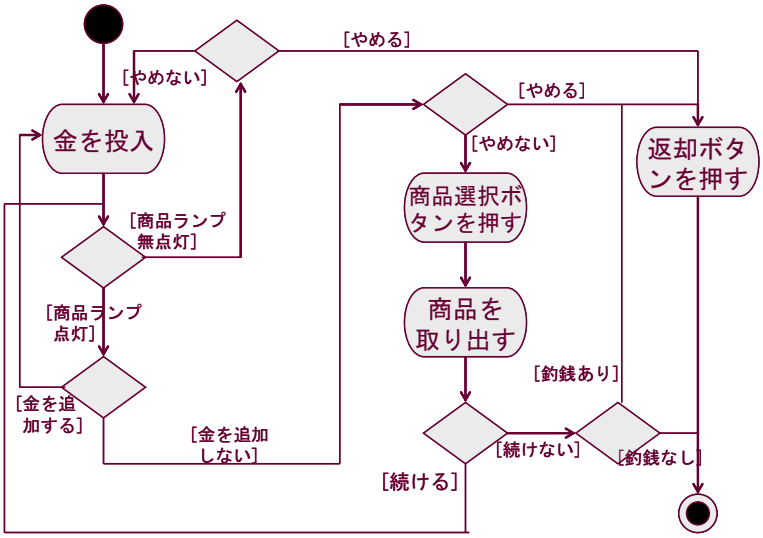


図 6.3: 自動販売機の動作図

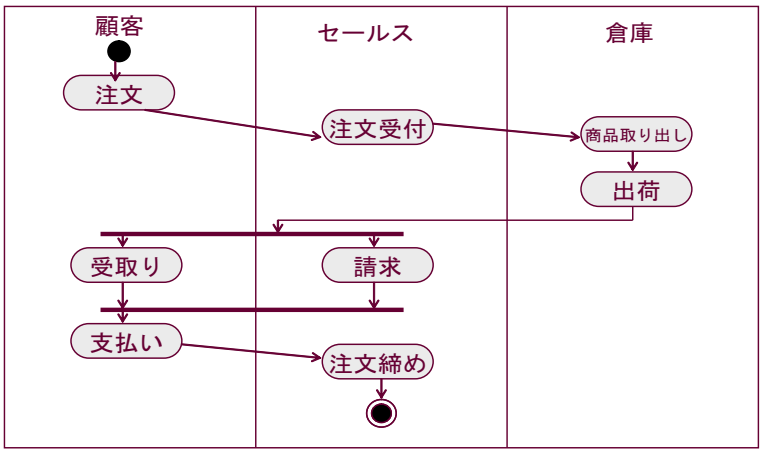


図 6.4: Swimlane: プールのコース

きく分けられる．図 6.5 は，それぞれの場合に対応した動作図を描いている．図 5.9 のデータの流れ図と比較すると，両者の特徴が判るだろう．

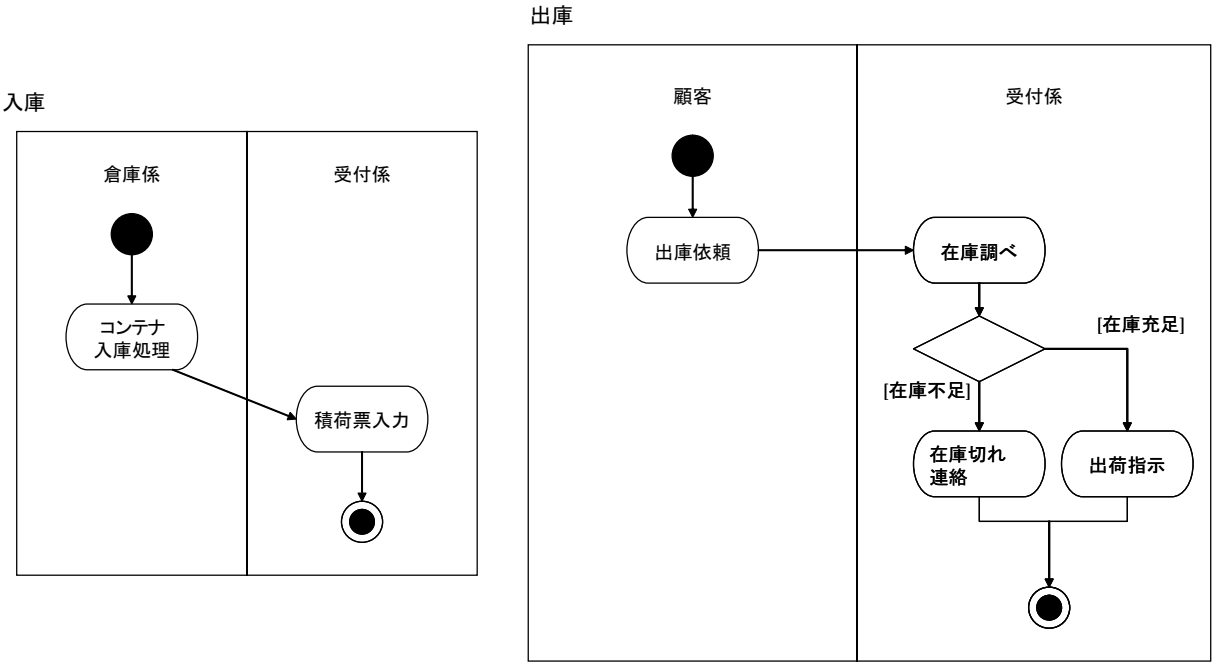


図 6.5: 酒屋問題の動作図

動作図をグラフとしてみると，その経路は実行の経路として解釈される．また分岐は OR（いずれかの経路の選択実行）として，並行分岐は AND（すべての経路の同時実行）として解釈される．部分グラフをまた動作図として扱うことが可能であり，それによりデータの流れ図と同じような階層構造が表現できる．プログラムレベルでは，下位の階層の動作は手続き (procedure) やサブルーチンに相当するものとなる．

6.3 動作図の使い道

動作図は使い道が多い．まず，システムの動作を全体として捉えるのに向いている．動作をモデル化するという点では，UML の振舞い関連の図である状態遷移図，系列図，協調図と相補的な関係をもつ．また，必ずしも直接システム化と結びつかないものとして，ワークフロー，ビジネスプロセス，開発プロセスなど，プロセスや手順に関するものを記述するのに役立つことが多い．

UML の動作図には，実はデータの流れも重ねて描けるような構成要素が導入されている．あるモデルでうまく表現できない対象がある場合に，モデル記法を拡張して対応しようとするのは，ある意味で自然の流れである．しかし，それによりモデルが本来目指していた抽象化の本質が歪んでくることは避けられない．また，記述する立場からも読む立場からも，記法が複雑化したモデル表現は使いにくい．そこで本書では，なるべく簡潔で本質を捉えたモデル表現に絞る方針を取っている．

あらためて動作図とデータの流れ図を比べてみると，データの流れ図は流れという動的な性質を対象にしながらも，静的なモデルに近いことが判る．動作図 では，現在どの頂点にいるかという視点が意味を持った．その意味で時間の進行が常に意識されている．しかしデータの流れモデルでは，現在どの頂点にいるかという質問は意味を持たない．つまり，データの流れモデルは，プロセスとプロセスの間にどのようなデータの流れが存在するかという構造を表しているので，その意味では静的なモデルに分類してもおかしくないものである．

第7章　協調モデル

協調 (collaboration) モデルでは，世界は動作主体の集まりで構成され，各動作主体は互いに情報をやりとりしながら振舞っていると見る．オブジェクト指向の枠組みでは通常，動作主体をオブジェクトと呼び，やりとりされる情報をメッセージと呼ぶ．また，プロセス代数（たとえば C. A. R. Hoare の CSP[49] や R. Milner の CCS[73]）や事象駆動の枠組みでは，動作主体をプロセスと呼び，やりとりされる情報を事象 (event) あるいは動作 (action) と呼ぶ．

7.1　系列図

UML の系列 (Sequence) 図は，複数のオブジェクトを想定して，それらの間でメッセージのやりとりが行われてまとまった意味を持つ動作が行われるとき，それを時間を追って記述するのに使われる．系列 (sequence) とは，時間軸に沿って進むメッセージの列を意味する．それによって表される一連の動作は，シナリオと呼ばれることもある．シナリオは，それに参加するオブジェクトの協調動作を記述していると考えることもできる．したがって要求をシナリオベースで記述した場合は (3.3.2 節参照)，その延長としてシステムの振舞いを系列図で記述することが自然になる．参加するオブジェクトは，その協調動作の中でそれぞれの役割 (role) を果たしている．

7.1.1　系列図の基本構造

図 7.1 は，自動販売機問題で想定される 1 つのシナリオを系列図 として描いたものである．

この図を描くには，まず動作主体としてのオブジェクトの集合を定めなければならない．これは系列図に限らず，協調型のモデルを構築する際に肝要なステップである．先にシナリオ記述があれば，そこに現れる動作主体を洗いあげてオブジェクトの候補を定め，そこから取捨選択，また必要に応じてオブジェクトの追加や分解・統合を行って一連のオブジェクト決定することは，ある程度自然に行えるだろう．もちろん，系列図を描くことで必要なオブジェクトが改めて明確になるということもしばしばある．すなわち，オブジェクトの同定と系列図の記述が並行的に進むといってもよい．

自動販売機の要求シナリオや動作図 (図 6.3）で想定されたオブジェクトは，客と自動販売機の 2 つであった．そこに現れる動作のうち，

- 金の投入
- 金額の表示
- 販売後の表示残高の減額

はすべて金銭の扱いに関することであり，それを担当するオブジェクトを考えてよい．それを「金庫」と呼ぶことにする．また，

- 販売ランプの点灯
- 商品ボタンに応じた販売

は商品の販売管理に関することであり，それを担当するオブジェクトを考えてよい．それを「販売」と呼ぶことにする．さらに商品の排出動作は「販売」に含めてもよいが，商品を配列し排出する物理的な装置が想定されるので，それを「ラック」という別オブジェクトとする．

こうしてオブジェクトを決定し，その間のやりとりを系列図に表している．

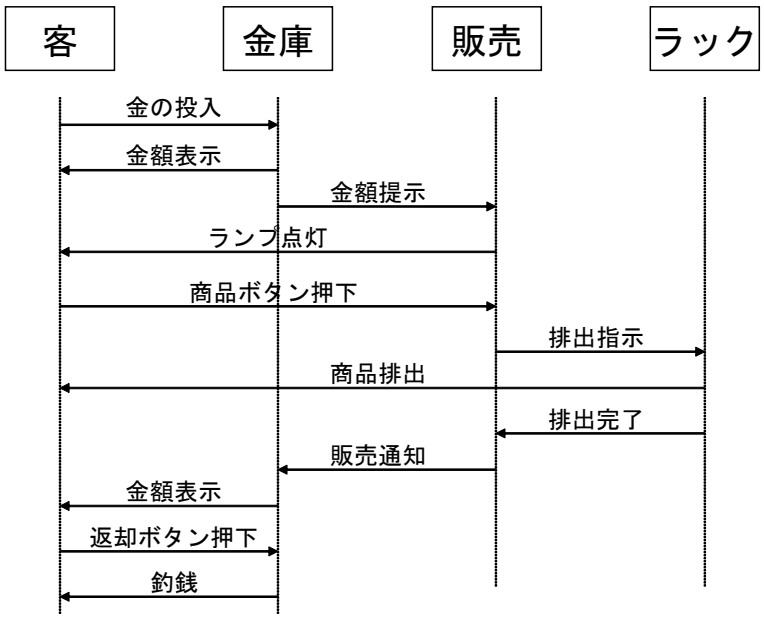


図 7.1: 自動販売機の系列図

系列図はグラフとしては次のような特徴をもつ.

系列図	
モード:	動的
対象:	情報の流れ
頂点:	(明示的でないが) オブジェクトが発信・受信した時点
辺:	メッセージ送信/メソッド起動
部分グラフ:	該当なし

また，各オブジェクトには縦方向の時間軸が 1 本ずつ割り当てられる．この形式は，動作図においてプールのコースで動作主体の受け持ち区分を示すのと類似する．

このように，系列図は単純なグラフとはやや異なる構造を持つ．また，部分グラフによる再帰構造は存在しない．なお，UML の系列図ではメッセージの発信条件を [条件] という記法で書いてもよいなど，他にいくつかの拡張がなされている．

7.1.2 並行プロセスの表現

系列図で並行プロセスを記述することができる．図 7.2 に例を示す．矢印の矢先が半分になっているのは，非同期メッセージ送信を表す．ここで非同期とは，メッセージを送信したオブジェクトが，それによって動作を開始した受信側のオブジェクトの動作の完了を待たずに，送信後も自分の動作を続けることを意味する．プロセスに対し，時間軸に沿って上から下へ描かれている横幅のある帯は，プロセスが起動中の区間を表している．

7.2 協調図

系列図もある意味でオブジェクトの協調動作を表すものであったが，オブジェクト間の協調関係を明示的に描くのが協調図である．図 7.1 に対応した自動販売機の協調図を，図 7.3 に示す.

グラフとしての特徴は次のとおりである.

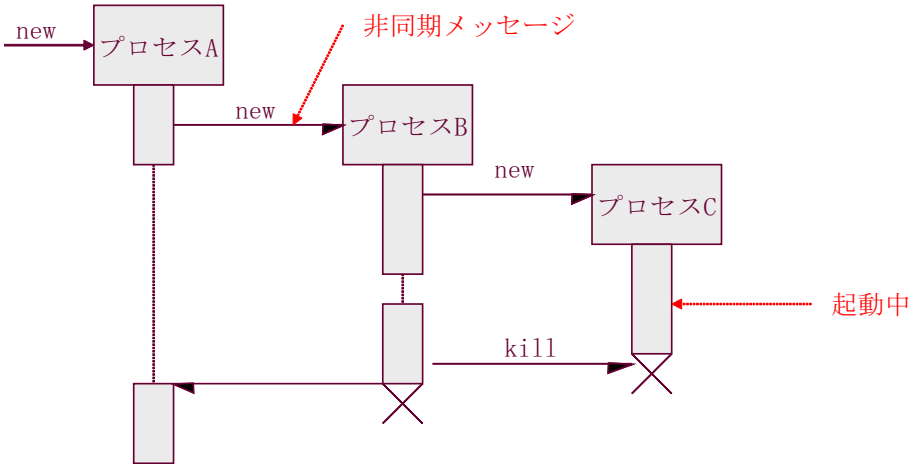


図 7.2: 並行動作のある系列図

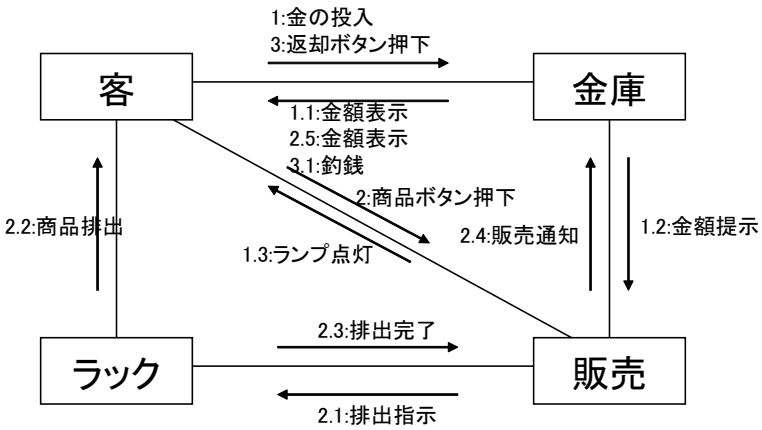


図 7.3: 自動販売機の協調図

協調図	
モード:	動的
対象:	情報の流れ
頂点:	オブジェクト
辺:	オブジェクト間の結合
部分グラフ:	下部オブジェクトの協調構造

協調図では頂点にオブジェクトを割りつけ，オブジェクト間にメッセージのやりとりがあるとき，対応する頂点間を辺で結合する．UML では協調図は意味的には系列図と等価であるとする．そのため，辺にラベルとしてメッセージをその送信方向とともに付ける際に，それらに 10 進式番号をつけて時間順序を表すようにしている．しかし，協調図は，オブジェクト間にどのような協調関係があるかを一覧するのに便利である．したがって，1 つのシナリオに対応した系列図と無理に等価にする必要はないように思われる．協調図には，複数のシナリオに基づく協調関係も一体として描けるからである．なお，系列図と同様に，UML の協調図ではメッセージの発信条件の記述などの拡張が定められている．

協調図をオブジェクト間の相互作用を表現した図として用いるなら，部分グラフによる階層構造は，1 つのオブジェクトを構成する下部オブジェクトが作る協調構造として想定できる．

多数のオブジェクトを対象として，オブジェクト間に協調関係があればその間に辺を引くことによって作られたグラフに対し，グラフの連結成分分解などの手法を用いると，関連するオブジェクトのグループ化ができる．このようなグラフ操作を活用することも，場合によっては有効であろう．

UML の協調図に類似した目的の図は他にも多く見られる．たとえば Statecharts を用いた Harel による応答型システムのモデル化技法 [47] では，この協調図に相当するプロセス間の相互作用を表した図を構造図と呼んで使用している．また，CSP を発展させたプロセス記述言語 FSP を中心とする Magee & Kramer の並行システムモデル化技法 [68] でも，やはり同様の図を構造図と呼んで使用している．これらの図は構造図という名前からもわかるように，動的な振舞いよりはオブジェクトあるいはプロセス間にどのような相互作用があるかという構造的な関係を表すことを目的としている．

7.3 Jackson システム開発法 (JSD)

Jackson システム開発法 (JSD) は，英国のコンサルタント Michael Jackson によって考え出されたシステム開発の手法である．Jackson はそれ以前に Jackson 構造的プログラミング (JSP) を提唱しているが [57]，JSD はそれをシステム開発全体に対して適用できるように拡張したものである．その内容は，著書 *System Development*[58] に詳説されている．

JSD はオブジェクト指向分析/設計方法論の提案に 7 ～ 8 年先行しているが，その考え方はとくに動的な振舞いに注目したオブジェクト指向方法論を先取りしているところがある．協調モデルの 1 つとしてここで JSD を取り上げるのは，そこで用いられるモデルが，実体ないしプロセスとそれらの動的な相互作用に着目して構成されているからである．

7.3.1 JSD モデルの基本構造

JSD のモデルの発想の基は， Hoare の CSP（通信逐次プロセス）モデル [49] にある．すなわち，対象とする世界を，たくさんのプロセスが互いにメッセージをやりとりしながら動的に動作する状況としてとらえる．

そこでモデルを記述するのに, 次の 2 つの基本要素を考える．

1. 行動 (action) または事象 (event)： 実世界で生起するもの．属性 (attribute）をもつ．
2. 実体 (entity)： 行動の主体．一連の行動（事象）を時間軸に沿って起こすもので，その意味では逐次的なプロセスとみなすことができる．

ここで実体は一連の行動を起こす主体と定義されていて，実体の構造や振舞いは，それに結びつけられた行動の形作る構造や，そこから生ずる振舞いの性質にのみ依存することに注意する．その意味で，まずオブジェクトが存在するというオブジェクト指向の世界のオブジェクトと，JSD の実体とでは異なる面がある．

7.3.2 開発手順

JSD によるシステムの開発手順は，次の 6 つのステップよりなる．

1. 実体行動ステップ
2. 実体構造ステップ
3. 初期モデルステップ
4. 機能ステップ
5. システムタイミングステップ
6. 実装ステップ

とくにモデル化技法という点では，1〜4 ステップが中心となる．

JSD の手順の説明のために，Jackson の本 [58] で用いられている例題を借用する．

銀行の預金業務 「銀行の預金業務では，顧客がまず口座を開設し，以降，金の預け入れや引き出しを繰り返す．口座は解約されると終了し，以降は預け入れも引き出しもできない．この口座は，貸し越しを許し，預金利息，貸し越し利息はない.」

機能要求

1. 顧客の残高をいつでも問い合わせできる．
2. 貸し越しが生じた時，レポートを出す．

7.3.3 実体行動ステップ

実体行動ステップでは，対象となる領域の行動 (action) と実体 (entity) を洗いだす．対象世界がなんらかの文章として記述されている場合，実体の候補となるのは文章に現れる名詞，とくに主語をなすものであり，行動の候補となるのは動詞である．行動に属性を付随させることがあるが，その候補となるのは目的語である．それらを洗い出した後，取捨選択する．さらに文章には直接現れないが，洞察により必要と判断される実体と行動を追加する．

預金業務の例題で洗い出される実体と行動は，次のようになろう．

実体
　　顧客
行動
　　開設　　属性（金額）
　　預け入れ　属性（金額）
　　引き出し　属性（金額）
　　解約　　属性（金額）

ここで実体と行動をリストアップする際，実体よりむしろ行動が先ということを，Jackson の弟子の Cameron[26] はとくに強調する．通常のオブジェクト指向と違う点の一つはここにある．De Marco がプロセスより先にデータの流れに注目しろと言ったことと対比すると面白い．

実際，行動の方はその内容を別にきちんと記述することが要求されるが，実体はそれらの行動の行為者という位置づけであり，行動の順序や，選択，繰り返しという構造そのものが，実体の定義となる．したがって，実体にはそれ以外の定義や付随する属性などを与える必要がない．

このようにして識別される実体は，動的 (dynamic) な性質を持つ対象をモデル化するのに適する．静的なデータ関係の記述には向かないことは，Jackson も認めている．たとえば化合物の成分比に関するデータを記述するようなことは，JSD の目的ではない．

7.3.4 実体構造ステップ

実体構造ステップでは，行動が形作る構造として実体を定める．行動の構造を表す時に用いる記法は，JSP でも使われたジャクソン構造図である．基本的に，連接 (sequence)，選択 (selection)，反復 (iteration) の 3 つの構成要素を用いた木構造で表す．1 つの頂点から出る辺はすべて同じ型（連接，選択，反復のいずれか 1 つ）の辺とする，という規則がある．なお，この図と等価なテキスト型の記法もある．

この記法による表現力は，正規表現と等価である．それにより，実体は逐次的な動作を行なう主体として捉えられたことになる．モデル化の能力として正規表現では不十分な場合もありうるが，それらに対して，複数の実体に分解するなどの，いくつかの対処法を Jackson は示している．

預金業務の例題で，前ステップで識別された顧客という実体の構造を表したのが，図 7.4 である．

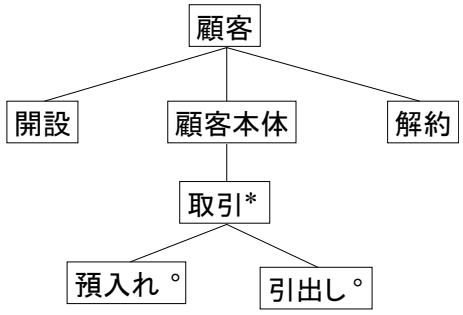
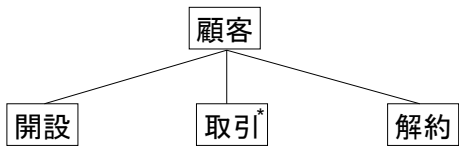


図 7.4: ジャクソン構造図

行動を表す箱の右肩に*があるのは反復を表し，◦があるのは選択を表す．1 つの頂点から分岐する枝の先の構造は，列，選択，反復のいずれか一つに限り，これらを混在させないというルールに注意．たとえば,「顧客」のすぐ下のレベルを



のように描いた方が簡潔のような気もするが，それは許さない規則になっている．

7.3.5 初期モデルステップ

初期モデルステップで，開発すべきシステムの仕様の記述が始まることになるが，その方法は実世界のシミュレーションを行うという発想に基づく．このシミュレーションを定義するために，実世界の実体に対応するシステム内のモデルプロセスというものを考える．モデルプロセスも逐次動作をするプロセスであり，実世界の実体をシミュレートするものであるから，通常は実体の動作の構造と同様の構造を持つ．JSD では，実体名にハイフンをつけて 0 を付加したもので実世界のプロセスを，1 を付加したものでモデルプロセスを表すという記法をとっている．たとえば,「顧客」という実体があった場合,「顧客-0」と「顧客-1」というプロセスが考えられる (通常は,これらのプロセスは個々の顧客のインスタンスごとに作られる).

実世界とシステムとの間には厳とした境界がある．そこで，その間にどのような結合を作るかがモデルの重要な要素となる．この結合は通常，実世界のプロセスが何か動作をし，その結果がシステム内のプロセスに伝わるという形をとるので，データストリーム結合と呼ばれる形態をとる．実際上は，これが人間機械間のインターフェースを定めていることになる．

銀行預金の例題でも,「顧客-0」と「顧客-1」というプロセスを考え，その間をデータストリーム結合結ぶこととなる．これは，顧客が銀行の窓口や ATM を通して，預け入れなり引き出しの行動を行なうことにより，データがシステム内の顧客に対応するプロセスに流れ込むことを意味する．

この関係を図示すると，ほぼ自明の図 7.5 となる．

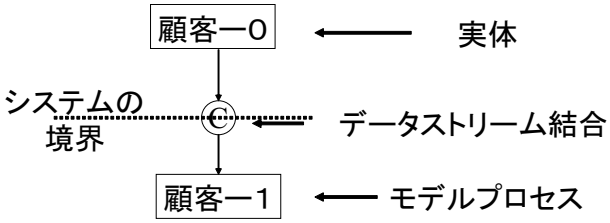


図 7.5: モデルプロセスの導入

ここで，丸印がデータストリームを表し，それに C という名前を付けている．矢印はデータストリームの流れる向きを表す.

結合はもちろん，モデルプロセス同士の間でも作られうる．また後で見るように，実世界に対応する実体を持たないシステム内のプロセスも徐々に追加されていくが，それらのプロセスの間でも結合が作られる．それにより対象となるシステム全体が，外部世界とのインターフェースも含めてネットワークとしてモデル化される．

プロセス間のネットワーク結合には，次の 2 種類がある．

- データストリーム結合
  データの流れ図におけるプロセス間の結合とほぼ同じ．ただし，データの送り手と受け手の間の同期という制御関係を意識している．データの流れを起こすのは送り手であり，受け手がそれを待つことで同期がとられる．この結合は，図では丸で囲って示す.

- 状態ベクトル結合
  この場合は，データの受け手が一方的に相手の保有するデータを見に行く．したがってこの状態データは，それを保有するプロセスの内部状態を表すもののうち外部に公開されているもの，という意味づけとなる．実際には，データベースや共有記憶として実現されることが多い．この結合は，菱形で囲って示す．この例は後で出てくる (図 7.6).

7.3.6 機能ステップ

機能ステップでは，システムの機能を果たすためのプロセスを必要に応じて導入し，それとモデルプロセスとの間や，機能プロセス間の結合を定める．機能を実現するための機能プロセスという概念が，実体を表すモデルプロセスと別に出てくるのは一見ややこしいが，よく考えるとモデルから設計への橋渡しとして重要であることがわかる．

預金業務の例題では，2 つの機能がある．そのうちの，貸し越しのレポートは，顧客-1 の内部状態で検出されるので，その機能を実現するための処理は顧客-1 の行動に追加するのが自然である．一方，残高報告の方は，外からの問い合わせに応じて行なわれる．これはそのためのプロセスを新たに導入するのが自然であるので，これを「照会機能」という名のプロセスとして導入する．このプロセスが機能を果たすには，顧客-1 の内部状態を見に行く必要がある．そこで，状態ベクトル結合が生じる．その結果，図 7.6 のようなネットワーク図となる．

ここで，菱形が状態ベクトル結合であることを示し，名前として CV を与えている．データは矢印の向き，すなわち顧客-1 から照会機能に流れるが，動作の主体は照会機能である．すなわち，照会機能が顧客-1 の内部状態を表す状態ベクトルを読みにいく．

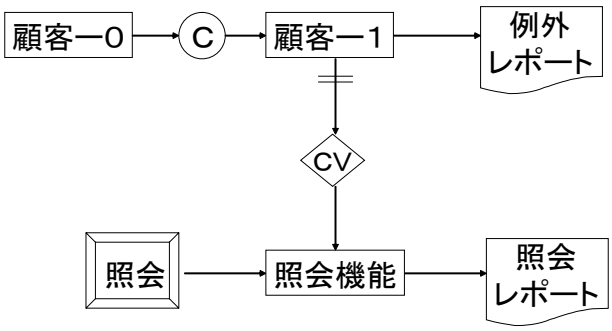


図 7.6: ネットワーク図

顧客-1 と CV との間の矢印に 2 本線が入っているのは，顧客-1 のプロセスが多数存在し，それが 1 つの照会機能プロセスと結合していることを示す．すなわち多対 1 の結合である．多対多の場合は，もう一方の矢印にも 2 本線を入れる．

このネットワーク図が，本章のテーマである協調モデルに対応するものである．グラフとしての性質を示せば，次のようになる．

JSD ネットワーク図	
モード:	動的
対象:	情報の流れ
頂点:	プロセス
辺:	データストリーム結合，状態ベクトル結合
部分グラフ:	下部ネットワーク

7.3.7 システムタイミングステップ

システムタイミングステップでは，プロセスの実行時間の考慮，プロセス間のタイミング・同期，などが扱われる．

7.3.8 実装ステップ

ここまでが問題を分析し，システムの仕様を作り上げる作業であり，実装ステップで具体的な設計を行なう．

1. データ設計
   実装時には，プロセスが大体モジュール（プロシージャ，サブルーチン，関数など）に対応するとして（しかし必ずしもそうとは限らない），状態ベクトルやストリームデータは，たとえばモジュールの引数とか，ファイルやデータベースとして実装される．どれを選ぶかは，個々の状況による．

2. プロセスの結合
   JSD で素直にモデル化すると，プロセスがたくさんできる．これをマルチプロセッサで直接動かすこともいまやそんなに非現実的ではなくなりつつあるが，プロセス数が莫大になるケースでは現実的でない．このプロセスを結合し，通常の環境で動くようにする技法には，いろいろなものがある．

   一つにはスケジューラをおく．これは多くの場合，いわゆるメインプログラムとして，入出力や実行管理のために，いずれにせよ必要になる．

   また，一つの実体クラスのインスタンスプロセスが多数できる場合には，その内部状態を配列化するなどの方法で，一つにまとめることができる．

メッセージをやりとりしている 2 つ以上のプロセスは，バッファを介して結合することもできるし，コルーチンの技法で実装することもできる．コルーチンを畳み込んで 1 つのプログラムにする技法も知られている．

逆にプロセスを分割した方がよい場合もある．いずれにせよ，このようにして同定されたプロセスをプロセッサに配分し，またスケジューリングするということが，実装の大きな作業である．

7.3.9 酒屋問題

酒屋問題を JSD で扱ってみよう．実体として自然に洗い出されるのは，コンテナ，酒を注文する依頼者，および受付係と倉庫係で になろう．

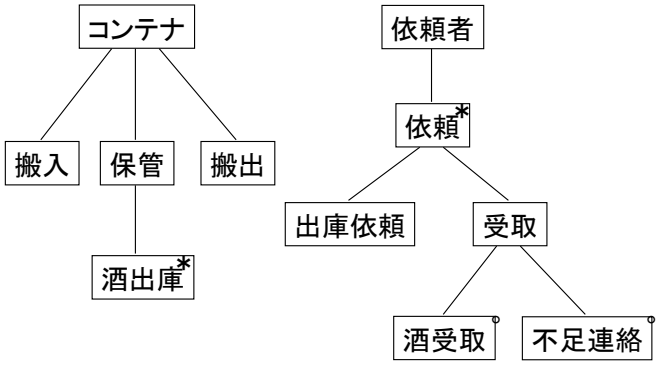


図 7.7: コンテナと依頼者の実体構造図

受付係は構築すべきシステムそのものに対応することになるが，実世界の受付係の仕事を素直に構造化すると図 7.8 のようになろう．

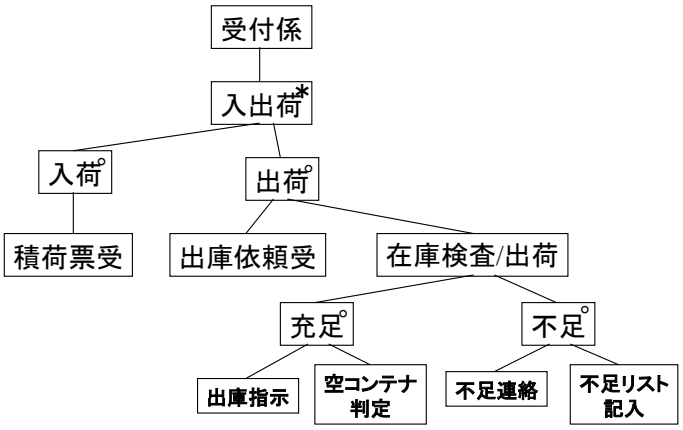


図 7.8: 受付係の実体構造図

倉庫係も同じように実体構造図で表すことができる．しかし，その先のモデル化を進めるのは，そう簡単ではない．ここでは大野佝郎氏の解答例があるので，それを紹介しよう [81]．まず，実体としてコンテナ，酒銘柄，注文主の 3 つが選ばれている．コンテナと注文主 (上では依頼者という名前を使っているが) はよいとして，酒銘柄

という比較的小さな概念を実体に選んでいるのは，奇異な感じがする．これらの実体構造図を見てみると，図 7.9 のように描かれている．

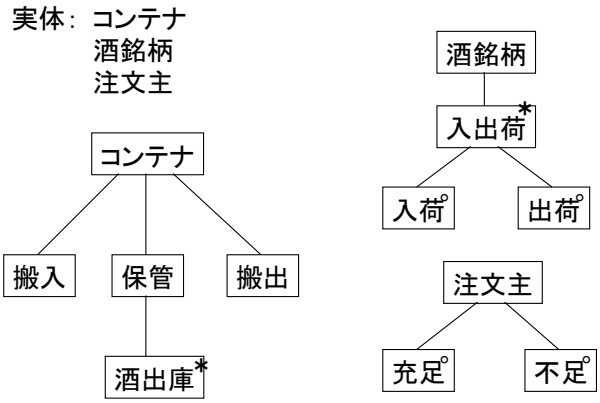


図 7.9: 大野氏の解における実体構造図

ここで，コンテナの実体構造図は図 7.7 と同じである．しかし，注文主の構造図は図 7.7 の依頼者とはずいぶん異なる．注文主が「充足」と「不足」という動作の主体というのは変ではなかろうか．それに注文主の構造図には繰り返しが現れないが，充足か不足かどちらかの行動を 1 回取るだけで消えてしまっていいのだろうか．酒銘柄に現れる「入荷」と「出荷」は図 7.8 の受付係にも現れる行動であるが，両者は同じ動きを表しているのだろうか．

これらの実体からモデルプロセスを生成して作られた初期モデルのネットワーク図（図 7.10）を見ると，これらの疑問にある程度答えることができる．

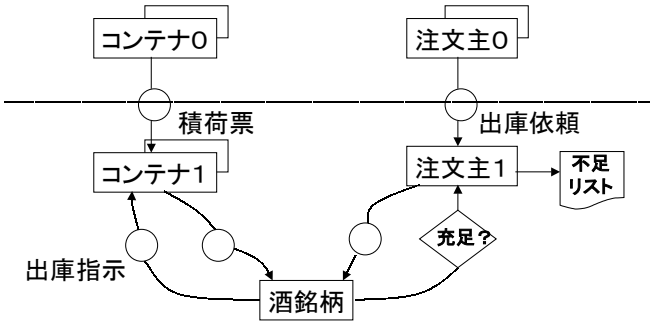


図 7.10: 大野氏の解のネットワーク図

もっとも注目すべき点は，酒銘柄がシステムの内部のプロセスとなっている点である．大野氏の論文における説明では，酒銘柄を実世界の実体として抽出したように述べられているが，この図から明らかなように，酒銘柄はシステムの構成上から役割を与えられたプロセスである．実際，この酒屋問題の本質は，倉庫への入庫がコンテナ単位で行われ，ある銘柄の酒は一般にいくつかのコンテナに分かれて入っているのに対し，酒の注文の方は銘柄単位で来るので，この両者のデータを関係づける必要があるところにある．経験の豊富なソフトウェア技術者は，このような問題の本質を直観的に捉えて，酒銘柄のような銘柄単位の情報を保持するプロセスが必要だと見抜くのであろう．しかし，JSD のような開発方法論の例題としては，そのようなプロセスを見出す過程の説明が必要だろう．

なお，注文主というモデルプロセスは，個々の注文に対応しているようだ．したがってその注文への処理が終われば，プロセスは消滅する．その意味では「注文主」という名前より,「注文」と呼んだほうがよさそうである．

7.3.10 自動販売機問題

自動販売機の問題を JSD で扱ってみよう．まず実世界の実体は，飲物を購入する客である．その実体構造図は，たとえば図 7.11 のようになろう．

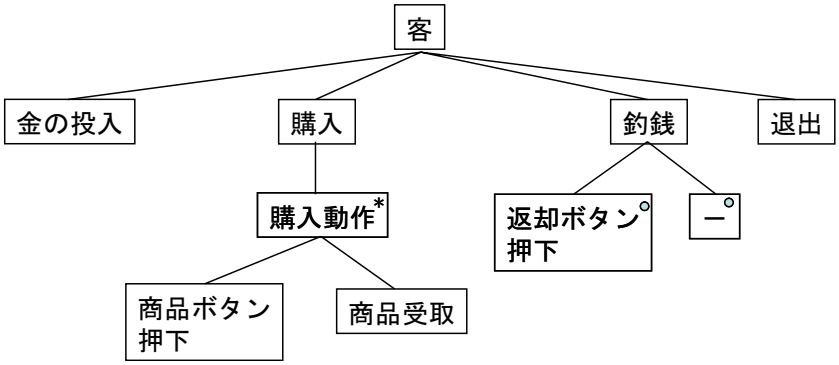


図 7.11: 客の実体行動図

次に，図 7.12 と 7.13 に金庫と販売プロセスの実体構造図を示す．本来はこのようなプロセスを見つける過程を JSD に従って追わなければならないが，すでに系列図や協調図で類似の分析をしているので，省略した．

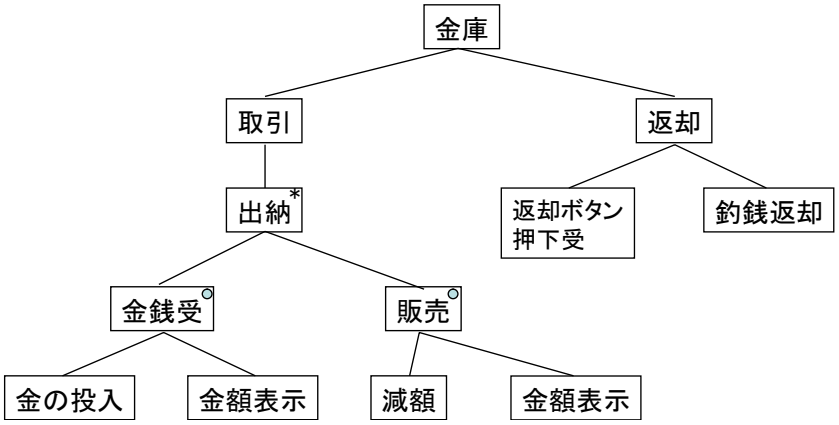


図 7.12: 金庫の実体行動図

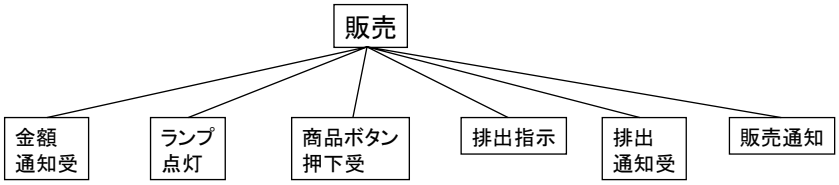


図 7.13: 販売の実体行動図

これらの作るネットワーク図も，ほぼ図 7.3 の協調図と同じになる．ただし，ここでは「客」はシステムの外部の実体で，JSD の流儀に従えば，客-0 と命名されるようなプロセスである．

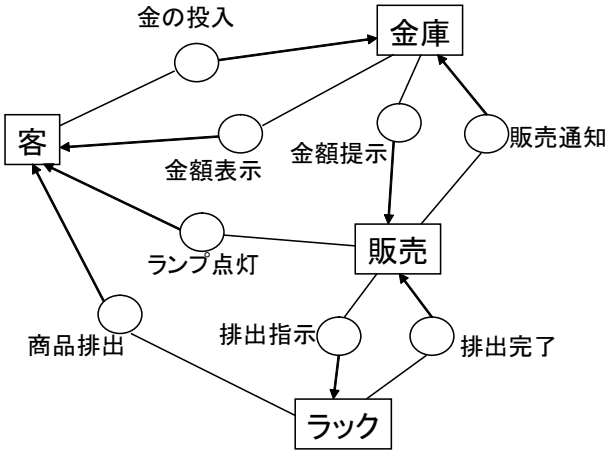


図 7.14: 自動販売機のネットワーク図

第8章　状態遷移モデル

状態遷移モデルでは，世界を 1 つの機械と見る．機械は内部状態を持ち，環境との間で事象をやりとりしながら状態を変えていく．

状態遷移モデルの歴史は非常に古いが，並行分散システム，応答型システムがますます普及し，その振舞いの検証に対する要請が高まるにつれて，改めて重要性が見直されている．

8.1　状態遷移モデルの基本的な性質

8.1.1　グラフとしての特徴

状態遷移モデルは，典型的な動的モデルである．そのグラフ表記である状態遷移図は，次のように特徴づけられる．

状態遷移図	
モード:	動的
対象:	視点の移動
頂点:	状態
辺:	遷移
部分グラフ:	下部状態遷移図

グラフとしての位相構造 (topology) に対応するモデルの構成要素は，この「状態」と「遷移」のみである．しかし，状態遷移モデルに意味 (semantics) を与えるための重要な概念として「事象 (event)」がある．各辺，すなわち遷移には事象がラベルとして結びつけられており，その事象が発生したときに対応する遷移が起こるというように解釈される．

状態遷移モデルで記述される対象は，内部状態を持つ機械，すなわち状態機械の振舞いである．以下では，状態機械の振舞いをプロセスと呼ぶ．事象は状態機械に外部から与えられる入力と見なしてよい．状態の集合の中に，通常はただ 1 つの初期状態が存在することを仮定する．また，状態の空でない部分集合として，終了状態が存在することも通常は仮定する．ただし，応答型 (reactive) のシステムを状態遷移モデルで記述する場合には，終了状態は存在しないか，特異な状態（deadlock) を指すことになる．

このような仮定のもとに，状態遷移モデルは次のような動的なプロセスを記述するものと解釈される．

1. プロセスは最初は初期状態にいる．
2. ある状態にいて，事象が発生した場合，その状態から出ている遷移の内で，その事象をラベルとして持つものがあれば，その遷移の先の状態に移る．
3. 終了状態に遷移したらそこでプロセスは終了する．

この説明から分かるように，状態遷移モデルでは「現在どこかの状態にいる」ことが仮定されている．現在いる状態は，必ず 1 つあり，かつ 1 つに限る[1]．ある状態にいて，ある事象が生じたとき，その事象をラベルとしてもつ遷移が常にちょうど 1 つある場合は決定的な状態遷移モデル，複数ある場合は非決定的な状態遷移モデルとな

[1]ただし，これは逐次プロセスの場合である．並行プロセスの場合は，後で述べるように「現在の状態」が状態の集合を指すことが一般的である．

る．また，1 つもない場合は，その事象は無視され，何の遷移も起こらないと解釈される場合と，何らかのエラーと解釈される場合とがある．

遷移は瞬時に起こるものと想定され，状態はある経過時間を持つものと想定される．もちろん，瞬時とか一定の経過時間という概念は相対的であり，モデルで考慮されている時間軸において，無視できる時間間隔を瞬時と見なすわけである．

状態機械を外から見た場合，その機械は記憶を備えたもののように振舞う．つまり，機械の動作は現在の入力によってのみ決まるのではなく，過去の入力の経過に依存し，それを「記憶」していると見えるからである．この「記憶」は実は状態に他ならないが，外からこの状態は見えない．外部と機械が共有するものは事象である．すなわち，状態遷移モデルの 2 つの構成要素である状態と事象のうち，状態は外からは見えず，事象は外から見える (あるいは外から与える) という区別を意識することは，重要である．

8.1.2 具体例

具体例を示そう．ボーリングの点数を計算するプロセスのモデル化を考えてみる．点数計算の規則は，次のようである．

1. ストライクの得点は，10 に次の 2 回の投球で倒れたピンの本数を加えたものである．
2. スペアの得点は，10 に次の 1 回の投球で倒れたピンの本数を加えたものである．
3. オープン・フレームの得点は，そのフレームの 2 回の投球で倒れたピンの本数で，加算はない．

実はさらに，最後の 10 フレーム目に特殊な処理が必要で，そのための規則があるが，ここではそれは考えないことにする．

図 8.1 は，この点数計算をするプロセスを状態遷移図でモデル化した例である．この図では，状態集合は { 始,

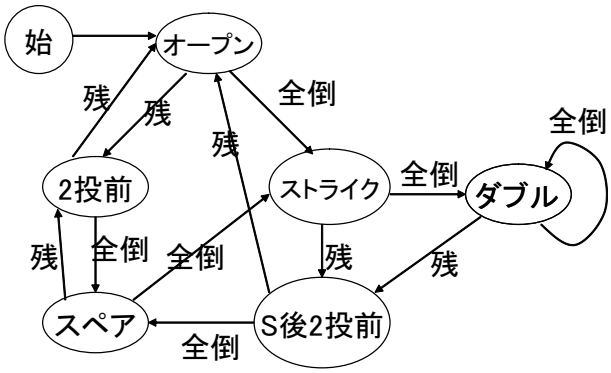


図 8.1: ボーリングの点数計算の状態遷移モデル

オープン，スペア，ストライク，ダブル，2 投前，ストライク後 2 投前 } である．なぜ，上に述べた規則には現れない「ダブル」という概念が現れているのだろうか．逆に，3 回連続ストライクのターキー（やフォースやそれ以上) という状態を入れていないのだろうか．あるいはフレームという概念に相当するものが直接現れなくてよいのだろうか．

また，事象集合は { 全倒，残 } である．舌足らずな言い方だが,「全倒」は立っているピンが全部倒れたという事象（1 投目ならストライクだし，2 投目ならスペアになる) を表し,「残」は 1 本以上ピンが残ったことを示す．なぜ，倒したピンの本数か残った本数によって事象を区別しないのだろうか．

これらはモデル構築の方針によって選択されたものである．モデルを作るには，目的がある．目的に必要なことがらが選択され，不要なものは捨象されるという「抽象化」がなされなければならない．ここでは，ボーリングの点数を計算するのに必要な場合分けを表すような状態を考えている．だから，事象は投球をしてピンが倒れた時点を取り，1 投目か 2 投目かを直接には区分せず，全部倒れたか否かという 2 つに 1 つの区別に抽象化しているのである．

この状態遷移図では終了状態を明示していない．上に述べたような抽象化をしているためにフレームという概念は表現されておらず，実際のボーリングのゲームが 10 個のフレームで終了するとしても，それはここで記述しているプロセスには反映されていない．

8.2　状態遷移モデルの拡張

状態遷移モデルは，さらに概念や記法を付加して拡張されることが多い．ここではとくに，出力事象，遷移条件，並行システム，状態の階層化という 4 項目を取り上げる．これらは D. Harel の Statecharts で採用されており，その Statecharts を採用している UML でも踏襲されている．

8.2.1　出力事象

これまで事象は状態機械にとっての入力であるとしてきた．機械といいながら，入力のみで出力がないのでは，役に立たないのではないだろうか．しかし，状態遷移モデルのもっとも簡素な形である有限状態オートマトンでは，出力がない．そこでは一連の入力事象の列にしたがってオートマトンが状態を遷移させていったとき，終了状態に達したらそれまでの入力列は受容されたとし，そうでなければ受容されないと判定することで，１つの言語を定めている，と解釈される．

しかし，やはり出力を扱えた方が，モデルの記述力は増す．そこで自然な拡張として，遷移に伴い状態機械から外界に向けて出力事象を発生するようなモデルが考えられる．出力事象を伴う状態遷移モデルは，有限状態オートマトンを拡張した Mealy 型の順序機械に対応する．

例として，図 8.1 のボーリングの点数計算を取り上げる．このモデルは本来，点数を計算するための情報を出力しないと役に立たない．そこで，出力事象として，次の 3 つを考えることにする．

1. 加算 1（倒したピンの数を加える）
2. 加算 2（倒したピンの数の 2 倍を加える）
3. 加算 3（倒したピンの数の 3 倍を加える）

ここで，括弧書きしたものは，この出力事象を受け取る外部の計算主体が行うはずの計算をコメントとして書き加えたもので，図 8.1 の状態遷移モデルにとっては，3 つの事象が区分されるということのみが意味を持つ．なお，通常のフレーム単位の点数計算に合わせるなら，

1. 加算 1（倒したピンの数を現在のフレームに加える）
2. 加算 2（倒したピンの数を現在と 1 つ前のフレームに加える）
3. 加算 3（倒したピンの数を現在と 1 つ前と 2 つ前のフレームに加える）

という方がよいかもしれない．とくに 10 フレーム目の処理を考えると，この表現の方が扱いやすい．

この出力事象を加えて図示したのが，図 8.2 である．遷移を表す辺は,「入力事象/出力事象」によってラベルづけされている．

8.2.2　遷移条件

ある状態にあって，そこからの遷移を起こす事象が発生した場合に，常に遷移が起こるのでなく，さらにある条件が満たされたときにのみ遷移が起こるという記述をしたいことがある．とくに後で述べる並行システムの場合，遷移条件を書けることで，状態の数の節約になることが多い．

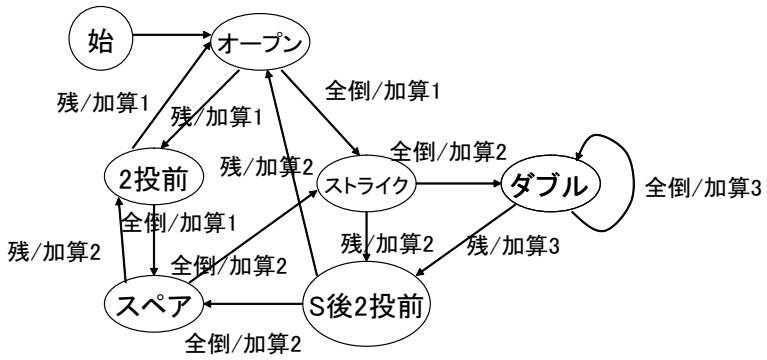


図 8.2: ボーリングの点数計算モデル (出力事象つき)

8.2.3 並行システム

並行システムは複数の状態遷移プロセスの共存という形で表現できる．複数の状態遷移プロセスを並べることで,「現在いる状態」は 1 箇所ではなく，プロセスの数だけあることになる．それぞれのプロセスがまったく独立に動作するのでは並行システムとしての面白さはないが，プロセス同士が事象を共有することで相互作用を起こす．事象を共有するとは，1 つの入力事象の発生でそれをラベルにもつ複数の遷移が同時に起こることや，あるプロセスの出力事象が別のプロセスの入力事象になること，などを意味する．

8.2.4 状態の階層化

状態遷移モデルをシステム開発のためのモデル分析に使用する際のもっとも大きな問題点は，実用規模の問題に対し，ほとんど常に状態数が爆発的に大きくなってしまうことである．この状態数の爆発に対処するにはいくつかの方法が考えられるが，有効なものの 1 つは，いくつかの状態を包含した親状態 (superstate) を導入することである．包含されている状態を子状態と呼ぶとすると，プロセスが親状態に含まれる任意の子状態にある時，またそれらの子状態の間でのみ遷移が起こっている間は，親状態が属する階層レベルでは，その親状態に留まって遷移が起こっていないものと見なす．親状態の外部から中の子状態に入る遷移や，子状態から親状態の外へ出る遷移が，親レベル上での遷移になる．これは，部分グラフを 1 つの頂点として階層構造を作るという一般的な方法の例である．

8.3 Statecharts

前節で述べたような状態遷移モデルの拡張を導入し，実用規模のシステムに対して起こる「状態数の爆発」に対処した手法の代表的なものとして，D. Harel による Statecharts がある．

8.3.1 Statecharts の基本的特徴

Statecharts の特徴を挙げると，次のようである．

1. 遷移には事象と，場合により条件や動作が結びつけられている．
2. 複数の状態をまとめたクラスタを導入できる．クラスタは内部を隠して抽象化された一つの状態とみなすことができる．逆に一つの状態を複数の状態からなる部分遷移モデルに詳細化して示すこともできる．
3. クラスタに対してデフォルトの初期状態を指定することができる．
4. 過去の遷移の「履歴 (history)」を使って，もっとも最近そのクラスタを離れた時にいた状態に入る，といった指定ができる．

5. And 独立な状態群が並行的に動作することを，記述できる．

6. 遷移の分岐 (fork) と合流 (join) を記述できる．

UML では，オブジェクトを状態機械と考え，その振舞いを状態遷移図で表す．記法としては，基本的に Harel の Statecharts を採用している．たとえば，図 8.3 は Fowler の本 [41] からとった．

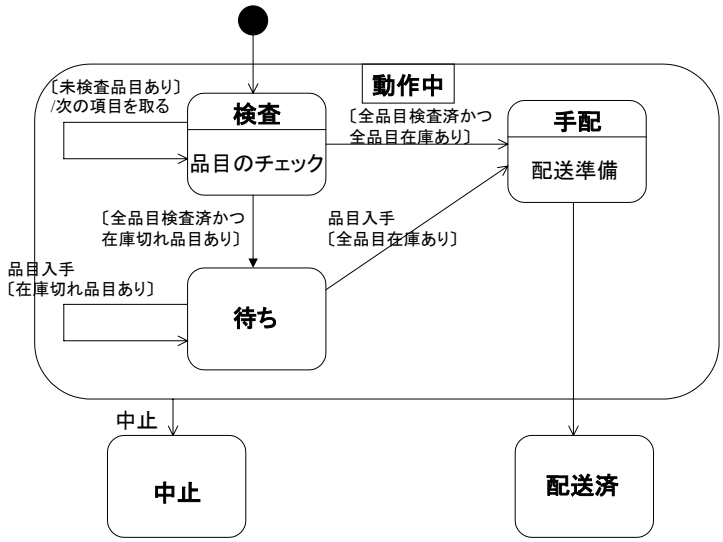


図 8.3: 親状態のある状態遷移図

図の「動作中」と名前を付けているのが親状態 (superstate) である．その中に 3 つの子状態があるが，それを隠して 1 つの状態と見なしたものが「動作中」である．黒丸は開始点を表す．したがって，最初に入る状態は「動作中」の中の「検査」となる．状態は角の丸い矩形で表し，中に状態名を入れる．状態を表す箱の真ん中に横線を引き，状態名を線の上に書き，下にはその状態にあるときに持続的に進行する作業を書くことができる．この作業は遷移に際して起こる出力動作のような瞬時のものではなく，持続的な性質を持つものである．検査状態における「品目のチェック」や手配状態における「配送準備」がその例である．

遷移は状態間を結ぶ矢印で表し，ラベルを持つ．ラベルのうち，[] で囲われた表現は遷移が成立する条件を記述するものである．単なる文字列は遷移を引き起こす入力事象を示し，/の後に書かれた文字列は遷移に際して出力として生じる事象を表す．

8.3.2 「アラームつき腕時計」の例題による Statecharts の詳細

Harel の論文 [46] をもとに，Statecharts の主な記法と意味を説明する．図 8.4 はボタンが 4 つついた，アラームつきの腕時計である．これを例題として用いる．

Statecharts では状態は角の丸い長方形で表し，その内部の左上に状態の名前を書く．遷移は矢印で表し，遷移を起こす事象の名前ををラベルとして付ける．図 8.5 の状態 A から状態 C への遷移は，ラベルとして事象 γ の後に (P) が付けられている．これはガードつき遷移を表し，事象 γ が起きてかつ論理値をとる式 P が真の時にのみ，遷移が起こる．

図 8.5 の右側の Statechart では，A と C を内部にもつ親状態 D が定義されている．この右の図と左の図は等価である．事象 β が親状態 D の縁から出ていることにより，β が起こると，D の中のどの子状態にいようが，すなわち A にいようが C にいようが，B への遷移が起こることを示す．

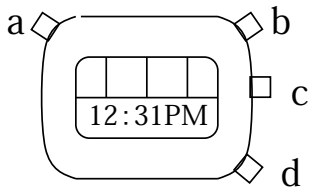


図 8.4: アラームつき腕時計

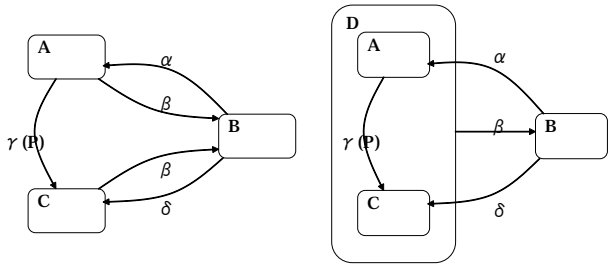


図 8.5: Statechart 説明図 (1)

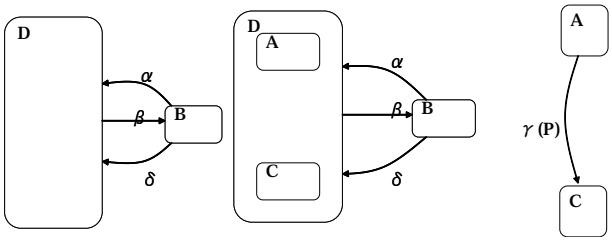


図 8.6: Statechart 説明図 (2)

この親状態 D は，図 8.6 の左図のように内部を隠して表示してもよいし，真ん中のように内部の状態だけを示してその間の遷移を省略して示してもよい．D の内部を独立の Statechart として，同右図のように示すこともできる．

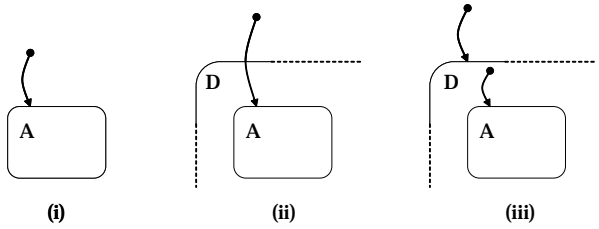
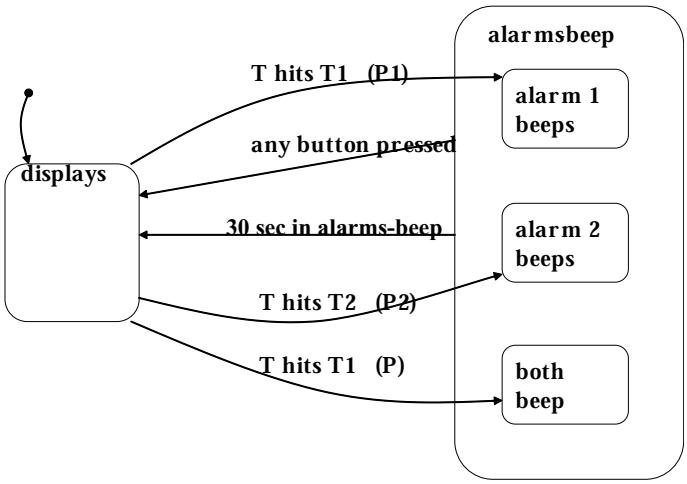


図 8.7: Statechart 説明図 (3)

この状態 A, B, C で構成される世界の中に，外から遷移したとき，最初に入る状態が A であることは，図 8.7 のように示す．黒丸から出る矢印の先が初期状態である．図 (iii) では初期状態が D であり，D の中ではさらに A が初期状態であることを示している．

さて，いよいよアラームつき腕時計の振舞いを記述しよう．

図 8.8 は，表示状態 (displays) からアラームが鳴る状態への移行を示している．アラームは 2 つあり，1 つのアラーム時刻が T1 に，もう 1 つのアラーム時刻が T2 に設定されている．現時刻 T とともに，T, T1, T2 はこの世界のいずれの状態からもアクセス可能な内部変数である．内部変数は，遷移に伴う出力事象として値を変更することができる．



P1: alarm1 enabled ∧ (alarm2 disabled ∨ T1≠T2)
P2: alarm2 enabled ∧ (alarm1 disabled ∨ T1≠T2)
P: alarm1 enabled ∧ alarm2 enabled ∧ T1=T2)

図 8.8: Statechart 説明図 (4)

図 8.9 は，表示状態 (displays) の内部を展開したものである．初期状態は時刻表示 (time) であり，ボタン d が押されると日表示に移る．またボタン a が押される度に，アラーム 1 設定，アラーム 2 設定，チャイム設定，ストップウォッチというように順に機能が切り替わる．

図 8.10 では，履歴 (history) という記法を導入している．左図のように外から Ⓗ に入ることが指定されていると，直近の過去にこの状態 (alarm1) から出た時にいた子状態 (on または off) に戻ることを意味する．履歴がなく初めてこの状態に入る時は，指定された初期状態，すなわち off に入る．右図も同じ意味を表す．

図 8.11 は Ⓗ の有効範囲を示す．左図では Ⓗ は K において指定されているので，履歴としては F にいたか G にいたかのみが記録される．その内部の A∼E のいずれにいたかの履歴は残されないから，それぞれの初期状態 (B と C) に入る．

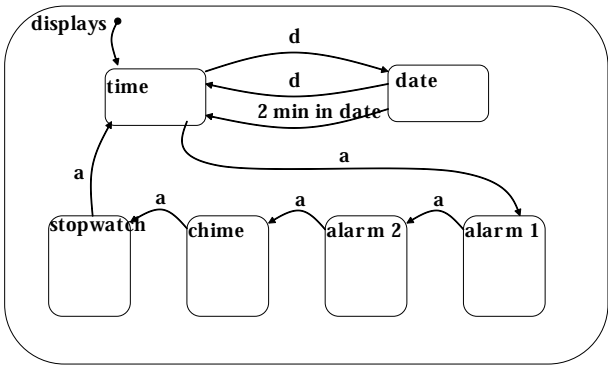


図 8.9: Statechart 説明図 (5)

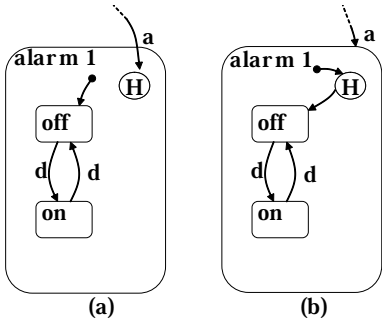


図 8.10: Statechart 説明図 (6)

一方，右図の Ⓗ* の意味は，履歴が再帰的に子状態にまで及ぶことを示す．すなわち，直近の過去にいた A∼E のいずれかの状態に戻る．

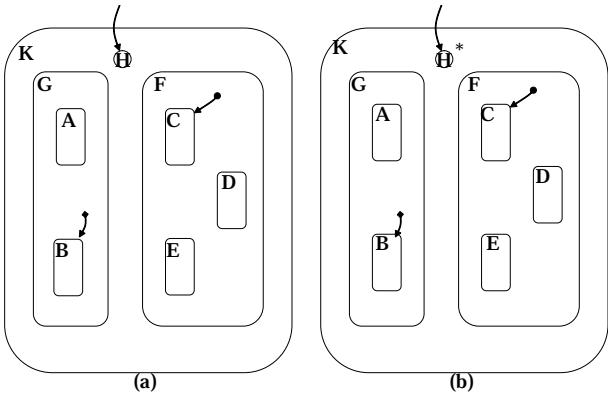


図 8.11: Statechart 説明図 (7)

図 8.12 の場合は，F の方には内部の履歴があるのでその記録に従うが，G には Ⓗがないので，A から出た直後であっても初期状態の B から始まる．

図 8.13 は，履歴を用いて displays を詳細化したものである．

図 8.14 は，displays の中の時刻更新状態 (update) をさらに詳細化した．この中では c ボタンを繰り返し押すことで，秒設定，1 分設定，10 分設定，時設定というように進んでいく．いずれの子状態にいても，ボタン b が押されると終了して外に出る．その後，d ボタンが押されると，設定の続きが可能になる．

図 8.15 のアラームの設定もほぼ同じであるが，ここでは c ボタンで，時，10 分，1 分と設定の順序が逆に進む．この内部の子状態を隠すと (a) のようになるが，これでは c ボタンと b ボタンの違いが分からない．そこで (b) のような記法を導入し，b が起こると内部のどこにいても外に出るが，c が起こって外に出るのは，内部のある特定の子状態にある時のみであることを示す．

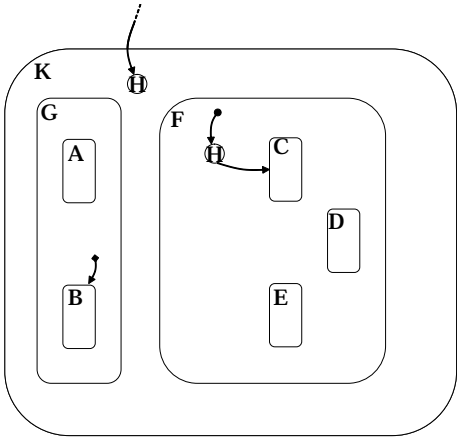


図 8.12: Statechart 説明図 (8)

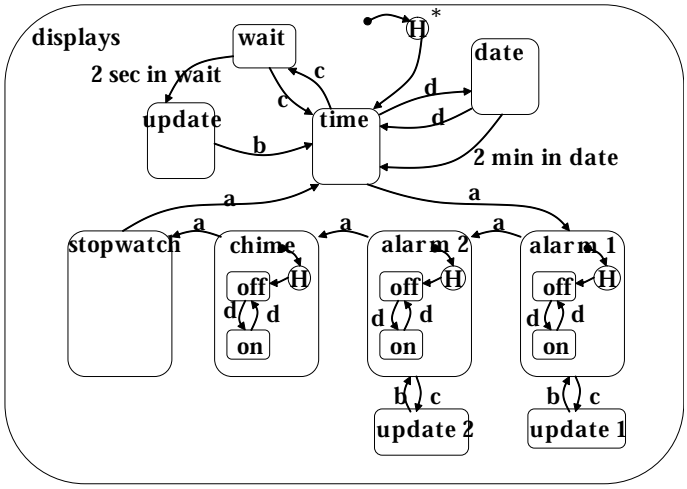


図 8.13: Statechart 説明図 (9)

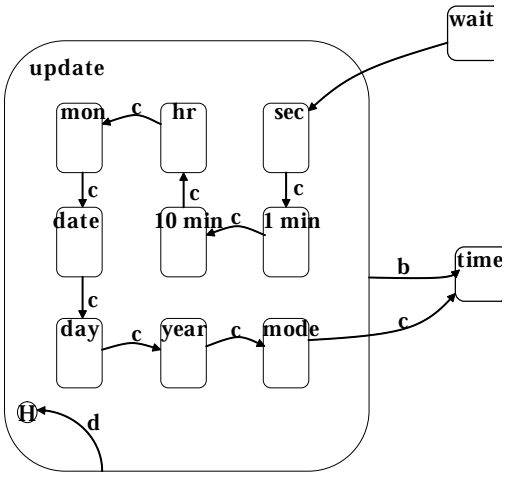


図 8.14: Statechart 説明図 (10)

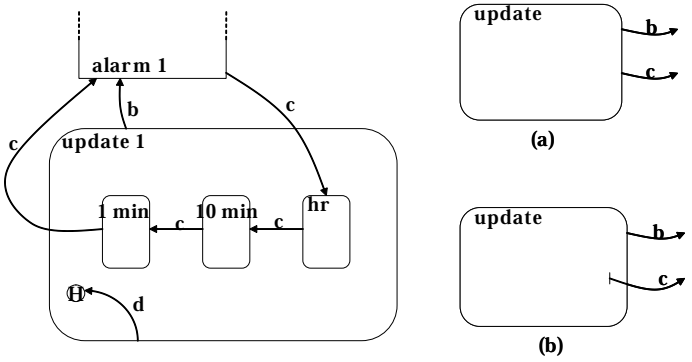


図 8.15: Statechart 説明図 (11)

次に並行プロセスの記法を導入する．図 8.16 で状態 Y は破線で A と D に分けられている．この A と D は AND 結合で，両者が並行的に動作する．A×D の状態を展開して描いた等価な Statechart が右図である．このように展開すると，状態数が積で増えるが，左図のような表現をとれば，状態数の爆発を防ぐことができる．

A と D はまったく独立に振舞うのではなく，相互作用がある．1 つは α という事象を共有することで，たとえば (B,F) という状態にある時に α が起こると，両者が同時に遷移し (C,G) に移る．これは一種の同期であるが，CSP や CCS と異なり，共有される事象は必ず同期をとって遷移しなければならないという制約はない．たとえば (B,E) にいて α が起こると，A の方にだけ遷移が起こり (C,E) に移る．CSP や CCS ではこのような遷移は許されない．

もう 1 つの相互作用は β(in G) という表現に見られる．これは A が C にいる時 β が起こった場合，D が G にいる時にのみ遷移するという条件を付加している．これも一種の同期を表現するものである．

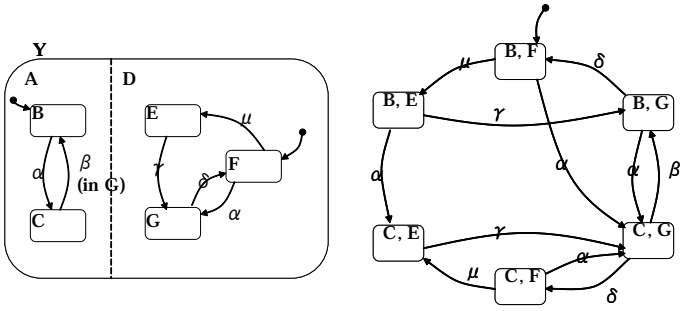


図 8.16: Statechart 説明図 (12)

このような並行プロセスは，とくに部品やサブシステムを合成して作られている機器やシステムの記述に有効である．

並行プロセスへの出入りはやや複雑になるが，論理的に考えれば自然である．図 8.17 にいくつかの場合を示している．遷移 ν では全体の縁に入っているので，A と D の初期状態，すなわち (B,F) に入る．δ では遷移先が明示されているので，(B,E) に入る．α では C のみが指定されているので，D 側は初期状態の F に入る．β では A 側は C に入り，D 側はその記録された履歴状態に入る．

出る方は，(B,G) にいて ω が起こると K に出る．A が C にいて θ が起こると，D 側はどこにいても K に出る．ε は縁からでているので，A も D もどの状態にいようが，L に出る．η は (in B) の指定があるので，(B,F) にいる時にのみ H に出る．

図 8.18 の左図は，図 8.17 の内部を隠した図である．右図は並行プロセスの記法を用いテストップウォッチ状態の内部を詳細化している．そこでは表示 (display) と計時 (run) が並行して動く．

これらを統合して全体の概要を示したのが図 8.19 である．

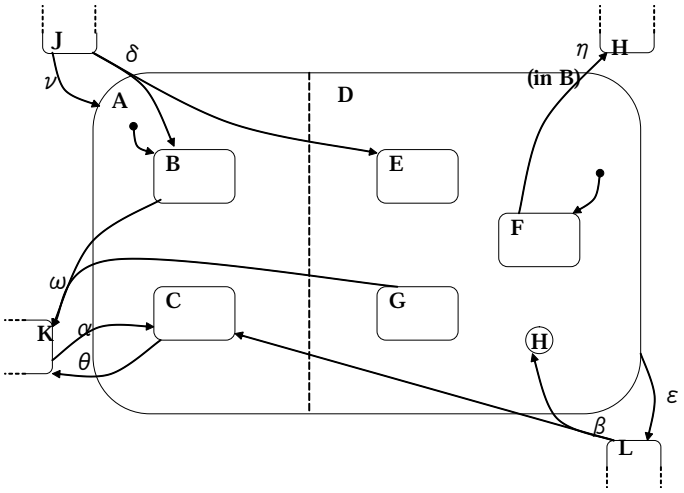


図 8.17: Statechart 説明図 (13)

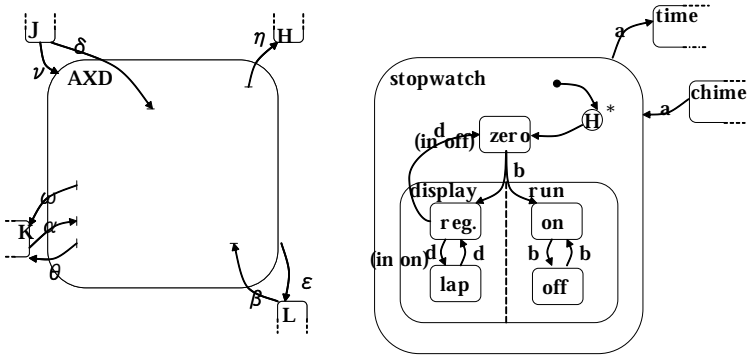


図 8.18: Statechart 説明図 (14)

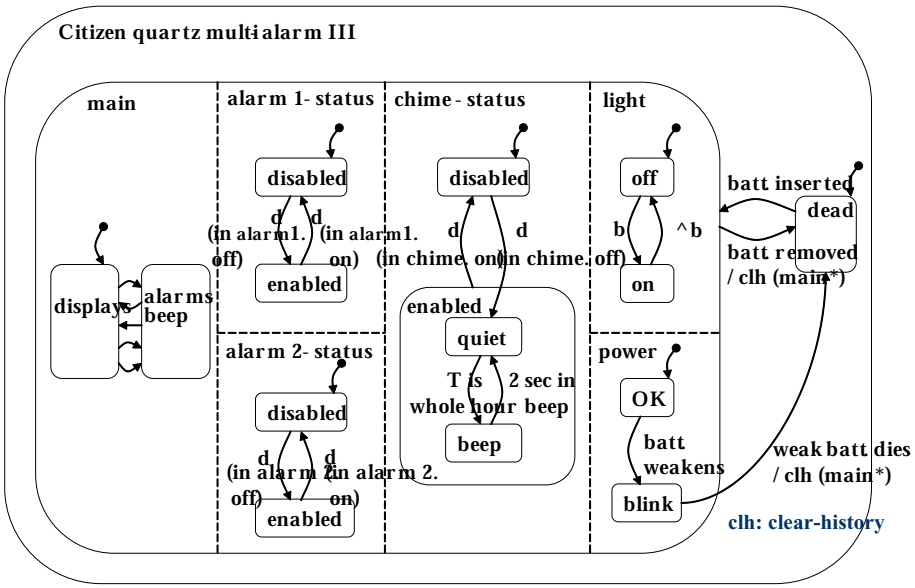


図 8.19: Statechart 説明図 (15)

8.4　自動販売機問題

系列図や協調図でオブジェクトとして役割を与えられた,「金庫」と「販売」について，その状態遷移図を描く．それぞれの状態遷移の振舞いは，系列図のそれぞれの時間軸上の系列と整合が取れているはずである．

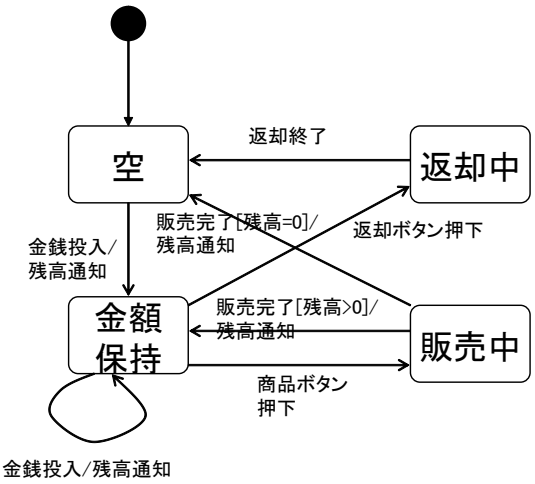


図 8.20: 金庫の状態遷移図

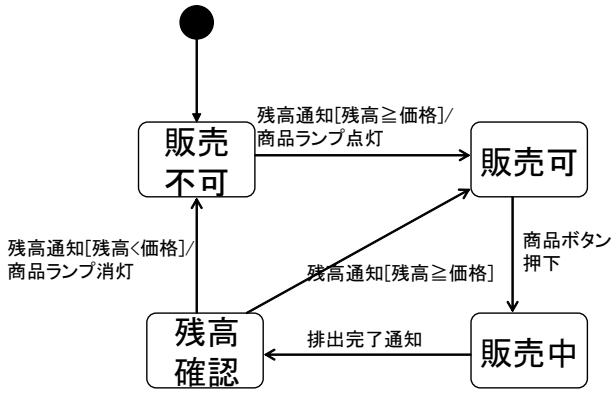


図 8.21: 販売の状態遷移図

Statechart における状態の子状態への分解の例として，販売の中の「販売中」状態を分解した見たのが，図 8.22 である．

8.5　状態遷移モデルの系譜

状態遷移モデルの歴史は古い．その系譜を振り返ると，大きく 2 つの流れがあると思われる．1 つは有限状態オートマトンに代表されるオートマトンの系譜である．もう 1 つは人工知能 (AI) やオペレーションズリサーチ (OR) でこれまた長く取り扱われてきた探索手法の系譜である．本章の締めくくりとして，この 2 つの流れを概観してみよう．

8.5.1　オートマトンの系譜

有限状態オートマトン

計算機科学の教科書なら必ず有限状態オートマトンと正規表現の関係について，かなりのページが割かれているはずである．また，コンパイラや言語処理の本でも，字句解析や形態素解析の方法として，有限状態オートマ

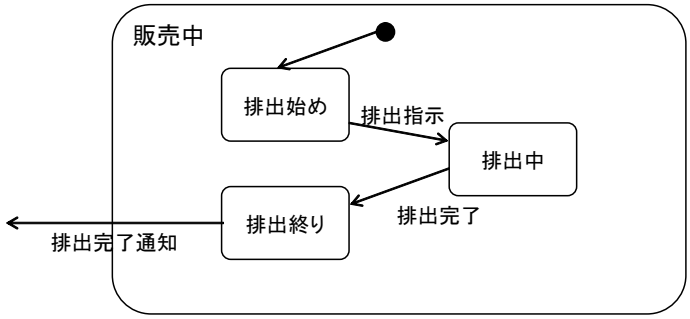


図 8.22: 販売中の状態遷移図

トンは避けて通れない.

有限状態オートマトンの定義は改めて述べるもでもなく，8.1.1 節で導入したモデルそのものである．有限状態という以上，状態集合は有限であることが仮定されている．正規表現の定義はどこででも見つけられるだろうが，ここでも煩を厭わず載せておくことにする．

アルファベット Σ 上の正規表現を考える．正規表現は，次のように構成的に定義される．

R1 ϵ(空列を示す)，$\varnothing$(空集合を示す)，および Σ の任意の要素は，それぞれ正規表現である．

R2 R_1 と R_2 が正規表現の時，$R_1 \cdot R_2$ および $R_1 \cup R_2$ は正規表現である．R が正規表現の時，R^* は正規表現である．

正規表現は，Σ を文字とする文字列からなる集合の部分集合（この部分集合を言語と呼ぶ）に写像される．その写像を与える規則は，次のようである．

R1' $\epsilon, \varnothing, a \in \Sigma$ は，それぞれ $\{\epsilon\}, \varnothing, \{a\}$ に写像される．

R2' 正規表現 R_1 と R_2 が写像された言語を L_1, L_2 とすると，$R_1 \cdot R_2$ は $L_1 \cdot L_2$ に，$R_1 \cup R_2$ は $L_1 \cup L_2$ に写像される．ただし，$L_1 \cdot L_2 = \{a \cdot b \mid a \in L_1 \ \& \ b \in L_2\}$．ここで文字列に対する演算 $\cdot$ は，連接である．

正規表現 R が写像された言語を L とすると，R^* は L^* に写像される．ただし，$L^0 = \{\epsilon\}, L^i = L^{i-1} \cdot L$ として，$L^* = \bigcup_{k=0}^{\infty} L^k$.

さて，有限状態オートマトンが受容する言語のクラスと，正規表現で記述できる言語のクラスとが同じであるというのが，Kleene による有名な定理である．証明は省略するが，直観的には明らかに見える．

ある言語を正規表現や有限オートマトンで表すことは，頭の体操としても面白い．以下の言語を正規表現ないし有限オートマトンで表してみよ．

1. 入力が 0 か 1 だとして，1 が偶数個ある文字列のみを表す正規表現，あるいはそれのみを受理するような有限オートマトンを作りなさい.
2. 入力が 0 か 1 だとして，それを順に左から右に並べて得られる 2 進数が 3 の倍数のもののみを表す正規表現，あるいはそれのみを受理するような有限オートマトンを作りなさい.

順序機械

順序機械は有限状態オートマトンに出力が加わったものである．主に，論理回路の設計や解析で用いられてきた．その形式的な定義は次のとおりである.

順序機械 M は，5 組 $(Q, X, Z, \delta, \omega)$ によって定義される．ここで

Q: 状態集合
X: 入力集合
Z: 出力集合
δ: 状態遷移関数，$X \times Q \to Q$
ω: 出力関数，$X \times Q \to Z$

状態および入力の集合が有限であれば，状態遷移関数と出力関数は表として表すことができる．なお，上の定義はいわゆる Mealy 型機械と呼ばれるもので，出力関数 ω を，入力に無関係に状態だけで決まるようにしたもの，すなわち ω: 出力関数，$Q \to Z$ という形のものは，Moore 型機械と呼ばれる．

順序機械をグラフとして表現する場合は，頂点には対応する状態をラベルとして与え，辺には，その遷移を起こす入力とその時に出される出力の対を，ラベルとして与える．たとえば，表 8.1 は $Q = \{a, b, c, d\}, X = Z = \{0, 1\}$ であるような状態遷移と出力の表であり，図 8.23 はそれに対応する状態遷移図である．

表 8.1: 状態遷移関数表

遷移関数表

	0	1
a	d	c
b	d	b
c	d	c
d	d	1

出力関数表

	0	1
a	1	0
b	0	1
c	1	0
d	0	1

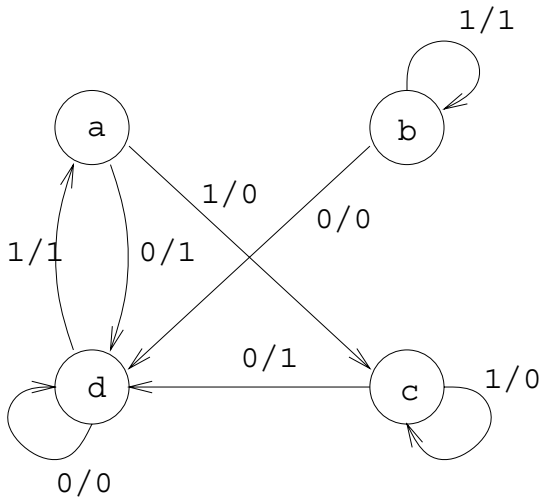


図 8.23: 状態遷移図

8.5.2 探索問題の系譜

AI や OR の分野で，古くから探索問題が研究されている．手法は解の空間を探索して目標を見つけるという形をとるが，通常はグラフ上の経路をたどって目標となる状態に達するという定式化がなされる．

1 つの典型は,「山羊と狼とキャベツ男」とか「宣教師と土人」といったクイズである．また，関連して，ゲーム，とくに 2 人の盤ゲームが類似の定式化で分析される．そこでは「盤面」が状態に対応し,「手」が遷移に対応する．

また，ロボットの行動計画 (planning) の問題も，状況とそこで可能な操作の選択を計画するという形で，状態遷移モデルに帰着する．さらにこれらを一般化した Simon & Newell による GPS(General Problem Solver) も同じ形式といってよい．

状態遷移の概念を使うときれいに解が求まるものの例を挙げる．

バケツのクイズ　容量 a リットルと b リットルの 2 つのバケツを使って，どちらかのバケツに c リットル残るようにする問題を考える．a と b は正の整数とする．実行できる操作は次のとおり．

1. 1 つのバケツを蛇口からの水でいっぱいにする．
2. 1 つのバケツに入っている水を全部捨てる．
3. 片方のバケツの水を，他方のバケツへ移せるだけ移す．全部移ることもあれば，移される側がいっぱいになった時点で終わる場合もある．

初めは，両方のバケツとも空とする．$a = 4, b = 5, c = 2$ の場合に，最低何回の操作で解に達するか．

第9章　オブジェクト指向モデル

オブジェクト指向モデルでは，世界はオブジェクトの集まりからなると捉える．オブジェクトとは，実世界の個物や観念世界の概念に対応するひとつの「もの」を指し，オブジェクト指向とはそのような「もの」を基本としてシステムモデルやソフトウェアを構築するという考え方や技術を総称するものである．

9.1　オブジェクト指向技術の歴史

オブジェクト指向の概念は，オブジェクト指向プログラミング言語の Smalltalk80 の仕様が公開された 1980 年代初めから，注目を集めるようになった．以降，オブジェクト指向プログラミング言語としては，C++や Java などが開発され普及した．また，80 年代の終わりから 90 年代半ばにかけては，ソフトウェアのライフサイクルをプログラミングから遡る形で，オブジェクト指向設計，さらにオブジェクト指向分析の方法論が次々と発表された．この経緯は，1970 年代に構造化プログラミングがもてはやされた後，構造化設計および構造化分析が登場してきたのと類似する．

このように現在のオブジェクト指向という考え方が普及するきっかけは Smalltalk80 が作ったが，その源流を遡ると，オブジェクト指向の基本となる概念はほとんどが 1970 年代までに提出されたものであることが判る．それらはさまざまな分野で独立に成長してきたが，以下で，それらを個別に見てみよう．

1. **シミュレーション言語**
   Smalltalk に大きな影響を与えた先行的なプログラミング言語に，1960 年代のノルウェーで研究開発された Simula というシミュレーション向きの言語があることは，よく知られる．Simula の最初の仕様は 1962 年に作られたが，その後改訂を重ね，67 年の Simula67 で一応の完成をみた．Simula67 にはクラスや継承の概念があるなど，現代のオブジェクト指向プログラミング言語が備えるべき性質を基本的に備えており，その先見性は敬服に値する．シミュレーション分野からオブジェクト指向の最初の着想が出てきたことは，オブジェクト指向の持つ優れたモデル化能力との関連を考える上で，注目すべきことであろう．

2. **抽象データ型**
   1970 年代には，D. Parnas による情報隠蔽の概念に基づいたモジュール仕様記述法の提唱をきっかけに，抽象データ型の研究が進み，また抽象データ型を文法要素として持つプログラミング言語としての CLU(1976), Alphard(1976), EUCLID(1977) などが提案された．抽象データ型は情報隠蔽の機構を持つ計算単位としてはオブジェクトとほぼ対応する．また，抽象データ型の厳密な意味を与える理論として，代数仕様の枠組みが提案され研究され始めたのも，70 年代である．

3. **知識表現**
   人工知能の分野では，やはり 70 年代に知識工学が提唱され，個別の問題領域の知識を収集記述し実践的に利用するという方法が精力的に進められた．その中で，知識表現の手段として 1975 年に M. Minsky が提唱したフレームに基づく表現法が注目され，それを基本要素とする知識表現言語の KRL(1977) や KL-ONE(1985) などが開発された．フレームは，構成要素としてスロットを持つ．スロットにはさまざまな種類のデータや関係，動作などを入れることができる．したがって，概念的にはオブジェクトとほぼ重なるものである．

4. **意味データモデル**
   データベース分野では，データベースの概念設計のためにデータモデルを構築するプロセスが必須である．P. P. Chen によって 1976 年に提案された実体関連モデル (Entity Relationship Model) は，対象となる世界を

実体と実体間の関連で記述するモデルで，広く使われている．実体はオブジェクトに対応するものと考えられ，このモデルはオブジェクト間のとくに静的な構造を表現するクラスモデルにきわめて近い．

5. **動的プロセスモデル**
   実体関連モデルが静的な構造を表すのに対し，動的な振舞いを示す対象を基本としてその動作を記述するモデルに，たとえば C. Hewitt が 1977 年に発表した ACTOR モデルなどがある．アクタはオブジェクトの動的な側面に対応し，オブジェクト指向計算モデルの理論的な根拠の一つを与えるものとなっている．

6. **マルチメディア**
   A. Kay を中心にゼロックス社パロアルト研究所で開発された Smalltalk は，もともと子供でも使える個人用のコンピュータのためのプログラミング言語という設計目標で開発された．Smalltalk を載せた Alto というパーソナルコンピュータでは，ビットマップディスプレイ，ウィンドウシステム，マウス，ポップアップメニューなど現在の GUI(Graphical User Interface) やマルチメディア環境の原型がすべて登場しているが，それらを操る基本概念としてオブジェクト指向が有効であることが示された．

このような多様な流れが合流し，オブジェクト指向という豊かな本流を形成したと見ることができる．

9.2 オブジェクト指向モデルの基本概念

オブジェクト指向モデルという場合，オブジェクトという計算主体が互いにどのような相互作用をすることで計算が進むかという計算モデルを指す場合と，オブジェクト指向の概念を用いてさまざまな分野の問題領域を表現したモデルを指す場合とがある．オブジェクト指向には豊かで自然なモデル化能力がある，という場合には後者のモデルを指しているが，そのモデル化能力はどこから来るかといえば，前者の計算モデルから導かれるので，ここではまず計算モデルとしての特徴を述べる．

オブジェクト指向モデルの基本要素は，もちろんオブジェクトである．オブジェクトとは対象となるものや概念で，ひとまとまりのものとして認識されるものを指す．オブジェクトには静的特性と動的特性がある．その静的特性を表すものが属性であり，動的特性を表すものが動作あるいはメソッドと呼ばれる振舞いの基本単位である．このように一連の属性と動作をひとつの単位にまとめることを，カプセル化 (encapsulation) という．

オブジェクト同士の間には，以下のようないくつかの種類の関係が考えられる．

1. **クラスとインスタンス**
   同じ種類のオブジェクトの集合をクラスと呼び，クラスに属する個々のオブジェクトをインスタンスという．オブジェクトは狭義にはインスタンスを指すが，クラスもオブジェクトの一種とする見方もある．たとえば Smalltalk では，クラスは独自の動作 (メソッド) を持つオブジェクトとして扱われる．

2. **汎化**
   クラス間の集合としての包含関係により，階層構造ができる．上位クラスは下位クラスを集合として含む．また，下位クラスは上位クラスの性質を継承する．このような階層構造は，古くから生物の分類などで用いられてきた．階層の上位のクラスは下位に対して汎化 (generalization) されたクラスと呼ばれる．逆に下位クラスは上位の特化クラスと呼ばれる．

3. **集約**
   オブジェクト B がオブジェクト A の部分を形作るとき，オブジェクト A はオブジェクト B の集約 (aggregation) と呼ばれる．たとえば自動車のオブジェクトはエンジンや車体や車輪のオブジェクトに対して集約となっている．部分全体関係とも呼ばれる．

4. **関連**
   関連 (relationship, association) は，クラス間の一般的な関係である．2 つのクラス間の関連は，それぞれのクラスを集合としてみた場合，それらの直積集合の部分集合として表される．具体例でいえば，学生というクラスと科目というクラスの間に履修という関連が考えられる．この場合は，いわゆる多対多の関連となるが，ほかに 1 対 1，1 対多という関連がありうる．汎化や集約も関連の特殊なものと見なすことができる．

5. **依存関係**
   あるオブジェクト (クラス)A の仕様を定める際，あるいは具体的に実現する際に，別のオブジェクト B を利用する場合，B の仕様を変更すると A は影響を受ける．この時 A は B に依存するという．

9.3 オブジェクト指向開発方法論

オブジェクト指向分析/設計技術の代表的なものに，Booch 法，J. Rumbaugh 等の OMT，I. Jacobson の OOSE, Coad & Yourdon, Shlaer & Mellor, Martin & Odell などがある．これらについては，多くの本が書かれてきた [20, 29, 37, 69, 60, 92, 96, 97, 116]．90 年代半ばになって，これらの手法を統合しようという動きが出てきた．とくに影響力の大きい J. Rumbaugh と G. Booch が手を組んで，OMT 法と Booch 法とを統合し，さらに I. Jacobson の OOSE をも取り込んだ Unified Method の構築が目指された．記法として，Unified Modeling Language が作られ，それにもとづくツールも市販されている．

そこでこれまでに提案された多くの手法も，少なくとも記法のレベルでは，UML を標準として採用する方向に進みつつある．しかし，方法論そのものは，記法との関連性が強いとはいえ，まだ統一された段階とはいえない．

オブジェクト指向に基づく開発方法論は，オブジェクト指向がもつ次のような特徴を活かそうとするものである．

1. **自然なモデル化能力**
   システム化の対象となる領域を自然に写像した分析・設計モデルが作りやすい．

2. **既存のすぐれたモデル化手法の取り込み**
   たとえばクラス類別，関連構造，状態遷移などモデル化手法の蓄積を活用できる．

3. **既存のすぐれた設計手法の取り込み**
   カプセル化，多相性，共通化／差分化，設計パターンなどの設計手法の蓄積を活用できる．

4. **並行システムの記述力**
   オブジェクトが並行して協調動作するというモデルに基づいた並行システムの記述・構築に，有効である．

5. **継ぎ目のない開発プロセス**
   オブジェクトという概念要素が分析段階，設計段階，実装段階を通じ一貫して用いられる．もちろん，作業工程としての分析，設計，実装というフェーズの区分はオブジェクト指向開発でも必要であるが，それらの間での記法や思考方法の違いが，たとえば構造化分析，構造化設計，構造化プログラミングの場合と比べて，はるかに小さい．

9.4 オブジェクト指向モデルの構築プロセス

Booch, Rumbaugh, Jacobson は，それぞれ UML に関する本を 1 冊ずつ出版している．Booch のものは UML の入門書 [21]，Rumbaugh のものは UML の言語仕様書 [93] であるが，Jacobson によるものは統合ソフトウェア開発プロセスと題され，OOSE を受け継いだユースケースに重きを置く開発プロセスを扱っている [59]．UML を単に記法としてだけでなく方法論として利用していくには，このようなプロセスの解説を参考にする必要があろう．以下では，Jacobson によるプロセスをごく簡単に述べる．

9.4.1 ユースケースの記述

ユースケース (use case) とは，利用者がシステムを使う時の典型的なシナリオを，システムへの要求を明確化する目的で記述したものである (「ユースケース」などと呼ばずに，使用事例とでもいえばよさそうなものだが，そのままで広く使われてしまったので，ここではそれに従う)．したがって，要求分析の章で述べたシナリオとほぼ対応する．

ユースケースにおいて，システムとやりとりする外部の人間の役割をアクタ (actor) という (これも「役者」とでもいえばよさそうだが，慣習に従う)．アクタとユースケースの関係を示した図を，ユースケース図という．たとえば，図 9.1 のようなものである．ここで楕円で表されているのがユースケース，人の形がアクタである．図の中で，<<include>>とあるのは，あるユースケースがその一部として別のユースケースをサブルーチンのように使うことを示す．また，<<extend>>は，あるユースケースが別のユースケースを拡張したものであることを示す．これは，たとえば例外的な事象が起こった場合の処理を，正常なユースケースを拡張して記述する場合などに用いる．

ユースケース図はグラフ構造をもった図としいう意味では，これまでに登場した UML の他の図と類似するが，図にしたことで得られる付加的な情報は少ない．ユースケースとそれに関係するアクタを並べ挙げたものにとくに優るものとはいえないだろう．

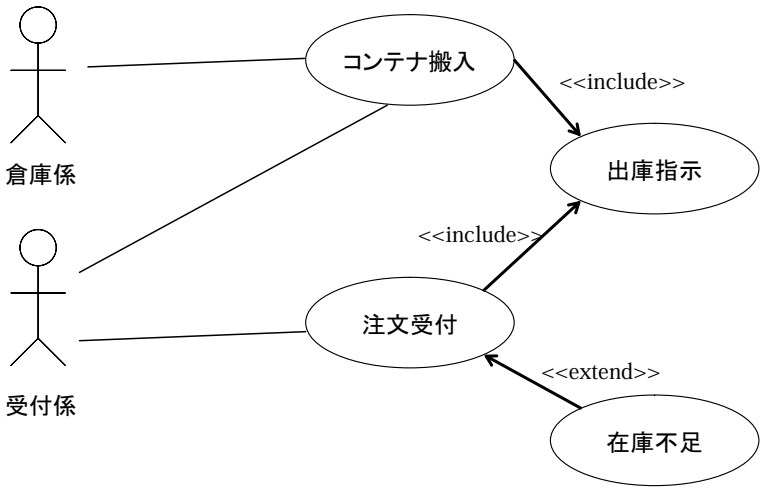


図 9.1: 酒屋問題のユースケース図

ユースケースを最初に並べ挙げるのは，主に OOSE が強調するやり方である．初期のオブジェクト指向分析/設計の多くは，まず対象となるモデルを構成するオブジェクトを定め，それらが形作る構造を明確にすることを最初の作業とした．システムと利用者のやりとりを考えることは，機能要求を定めることに相当するので，むしろオブジェクト・モデルができた後にすべき作業とされた．しかし，対象領域のオブジェクトとそれらの関連を明らかにしても，なかなかシステムのイメージが浮かび上がってこないという実践上の問題点が指摘される．また，そもそも意味的なまとまりがあるオブジェクトの過不足ない集合を定めるのに，何を手がかりとして作業すべきかという問題もある．ユースケースはシステムの動作イメージを早い段階で明確にすると共に，それらのユースケースを実現する責任を持ったオブジェクトを構築するというプロセスを経て，オブジェクトの同定に結びつくとされる．

ユースケースは OMT のうちのデータの流れ図で記述される機能モデルを代替するとも言えるわけで，実際 UML にデータの流れ図が採用されていないのは，ユースケースが導入されたからであろう．

ユースケースには，次のような項目を記述する．

1. ユースケース名
2. アクタ: このユースケースを利用するもの
3. 事前条件: このユースケースが実行される前に成立すべき条件
4. 基本系列: ユースケースにおけるシステムとアクタのやりとりを，時系列に沿って記述したもの
5. 事後条件: ユースケースの実行後に成立すべき条件
6. 代替系列: 基本系列と類似するが，条件によって途中から別の系列として分かれていくようなケースの記述

9.4.2　クラスの同定

収集されたユースケースから，分析レベルで想定すべきクラスを洗い出す．それには，アクタとシステムのインターフェースとなるようなクラス，ひとまとまりのデータを包含するような実体としてのクラス，ユースケースを実行するための制御的な役割をするクラス，の 3 種類を考えるのが，一般的である．クラスは意味的なまとまりを持つオブジェクトの種で，分析の段階におけるインターフェースクラスや実体クラスは，実世界の物や概念に対応するものであることが自然である．分析の段階でも必要な範囲で制御クラスが導入されるが，さらに設計の段階ではシステム固有のクラスが追加され，また分析段階のクラスの分割，変更，統合などが行われる．

クラスは，名前と属性および操作の集合を保有する．

9.4.3　クラス図の作成

クラスは互いにさまざまな関係を持つ．それを図示するのがクラス図である．クラス図で記述される関係は，通常は静的なものである．とくに汎化（逆に見れば特化あるいは継承），集約，関連という関係は区別して記述するのが通例である．

オブジェクト指向モデルで使用する他の図，すなわち協調図，系列図，状態遷移図などはすでに説明したが，クラス図はここで初めて登場するのでやや詳しく述べる．クラス図のグラフ構造は次のようである．

クラス図	
モード:	静的
対象:	ものとその間の関係
頂点:	クラス
辺:	汎化，集約，関連，依存
部分グラフ:	該当なし

具体的な例を，図 9.2 に示す．

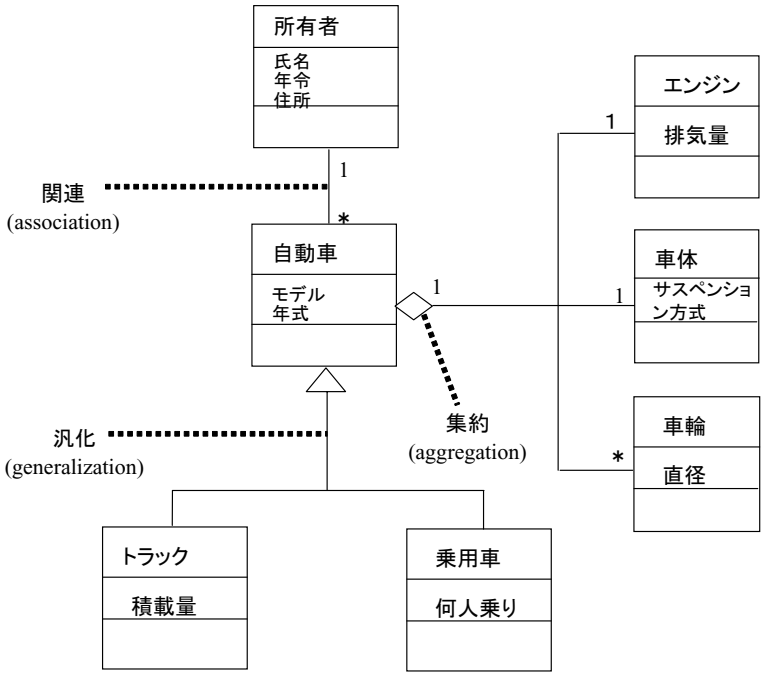


図 9.2: 車のクラス図

クラスは四角形の箱で表され，内部に縦方向に 3 つの区画を持つ．上段にはクラス名，中段には属性のリスト，下段には操作（メソッド）のリストが記入される．属性や操作を省略してもよい．

汎化の関係は，辺の親クラス側の端点に三角形をその 1 つの頂点が親クラスの箱の境界を指すように置いて表す．集約の関係は，辺の集約するクラス側の端点に菱形をその 1 つの頂点が集約クラスの箱の境界を指すように置いて表す．一般の関連は向きのない辺で表すが，その関連をたどる参照の仕方が一方向に限られている時，すなわち A と B が関連して，参照が常に A から B である場合は，参照先の B の側に矢印を置く．A が B に依存している場合は，A から B への矢印を点線で描く．ただし，依存先がインターフェースや抽象クラスの場合は，矢でなく汎化と同じように三角形を描く．

集約や関連については，多重度を示すことがある．図 9.2 で 1 や*が多重度の指定である．A と B が関連あるいは集約関係にあるとして，たとえば A 側に 1 と表示されていれば，1 つの関連に A はちょうど 1 つの個体が参加し，*なら 0 以上の任意の数が参加する．また，$m:n$ と書けば，m から n の間の区間の値を取ることを表す．

クラス図のグラフ構造の特徴の 1 つは，クラスの内部を展開するとまたクラス図になるという自然な再帰構造がない点である．クラス間の関係を汎化だけあるいは集約だけに限定した図なら，その部分グラフをクラスにまとめることができるが，一般的な関連を含むクラス図は，1 つのクラスとまとめて考えることができない．ただクラスを集めたパッケージ（後でも出てくる）は，それが公開している操作（メソッド）を持つクラスと考えれば，それは再帰的な関係といえる．その意味で，パッケージ図は再帰構造を持つ図といえよう．

9.4.4 協調関係の記述

クラス間の動的な関係は種々の側面によって記述されるが，もっとも基本的なものの一つが複数のクラスに属する個体同士による協調動作である．一つの協調に参加するオブジェクトは，そこである役割を果たす．したがって，協調を役割 (role) 間の相互作用として規定し，オブジェクトはその役を演ずることで協調に参加するという役割モデルを考えることが，オブジェクト群の振舞いをモデル化すること，またさらに遡ってオブジェクトの定義範囲を明確化することに繋がる．このような協調のモデルは，それぞれがユースケースと対応するものとして考えることができる．

UML ではすでに述べたように，このような協調動作を表すモデルとして，協調図と系列図を用いるものとしている．

9.4.5 状態遷移図の作成

オブジェクト間の動的な行動はたとえば系列図で記述されるが，個々のオブジェクトの動的な振舞いがこのような協調動作に整合的に参加できることをみるためには，そのオブジェクトの振舞いを個別にきちんと記述する必要がある．そのために状態遷移モデルを用いることが多い．旧来の状態遷移図をそのまま用いることは，状態数の爆発が起こることから現実的でないので，D. Harel による Statecharts の記法がよく使われる．

9.4.6 パッケージ化

大規模なシステムのモデルを作る際には，それをいくつかの部分に分割し，記述や理解をしやすくする工夫が必要となる．そのために一群のオブジェクトをグループ化した単位を，UML ではパッケージと呼ぶ．パッケージは先に多くのオブジェクトがあってそれらをまとめるというボトムアップ型のモデル化を想定するような用語であるが，トップダウンに分割することももちろん有効な手法であり，その場合はサブシステムなどという名称が適切かもしれない．

9.5 酒屋問題

酒屋問題をオブジェクト指向でモデル化した例を示す．

9.5.1 ユースケースの作成

ユースケースとして図 9.1 に示す 4 つを取り上げる．

コンテナ搬入

- アクター：倉庫係，受付係
- 事前条件：
- 基本系列：
  1. 倉庫係は搬入されたコンテナの内容と積荷票を照合し，積荷票をシステムに登録する．
  2. システムは，在庫不足として登録されている注文で，この入荷により在庫不足が解消されたものがあれば，それを受付係に知らせる．
  3. 在庫不足が解消された注文があれば，受付係はそれらにつき出庫指示ユースケースを実行する．
- 事後条件：搬入されたコンテナの持つ内蔵品のデータが，参照可能である．出庫があった場合は，それに応じて在庫データが更新されている．

注文受付

- アクター：受付係
- 事前条件：
- 基本系列：
  1. 受付係は，顧客から注文を受ける．
  2. 受付係は，その注文についての在庫状況をシステムに問い合わせる．
  3. システムは注文に応ずることができる在庫があることを受付係に伝える．
     [代替系列]： 在庫が不足している場合は，在庫不足ユースケースへ．
  4. 受付係は，この注文につき出庫指示ユースケースを実行する．
- 事後条件：出庫に応じて在庫データが更新されている．

出庫指示

- アクター：受付係，倉庫係
- 事前条件：注文が指定され，その注文に対し在庫が存在する．
  1. 受付係はシステムに指定された注文に応じた出庫指示書を作成するよう指示する．
  2. システムは出庫指示書と，空きコンテナが生じる場合はその空きコンテナ番号を，倉庫係に示す．
  3. システムは，在庫データを更新する．
- 事後条件：出庫に応じて在庫データが更新される．

在庫不足

- アクター：受付係
- 事前条件：注文が指定され，その注文の在庫が不足している．
- 基本系列：
  1. システムは指定された注文に応じた在庫がないことを受付係に伝える．
  2. 受付係は顧客に在庫不足連絡を行う．
  3. システムはこの注文を，在庫不足により未出荷であるものとして登録する．
- 事後条件：当該注文が在庫不足で未出荷であることが登録されている．

9.5.2　クラスの同定とクラス図

これらのユースケースから，分析レベルで想定すべきクラスを洗い出す．すでに述べたように，インターフェースクラス，実体クラス，制御クラスの 3 種類が考えられる．

インターフェースとしてまず考えられるのは，コンテナ搬入に伴う積荷票データを受け付けるオブジェクトである．これは倉庫係に対応するインターフェースとみなされる．このオブジェクトはまた出庫の指示を受ける役割も果たすと考えるのが自然である．このオブジェクトを「倉庫」と呼ぶことにしよう．次に，顧客からの出庫依頼を受けるインターフェース・オブジェクトが想定される．このオブジェクトは受付係に対応するインターフェースとみなされる．これを「受付」と呼ぼう．また，実体クラスとして,「コンテナ」,「顧客」,「注文」が自然に導かれる．コンテナの中身は品目別の酒であり，注文は品目ごとになされるので，コンテナの中にある品目ごとの酒の集合を「内蔵品」という名のオブジェクトとしよう．

それぞれのクラスの属性と動作 (メソッド) を数え上げる．その際，ユースケースに応じた処理が進められることを保障するようにする．次に述べる協調図や系列図を並行して記述すると，属性や動作がよりはっきりするので，両作業を並行して進めるのも実際的である．その上で，クラス間の関係を明確にし，クラス図を作成する．たとえば図 9.3 のようになろう．

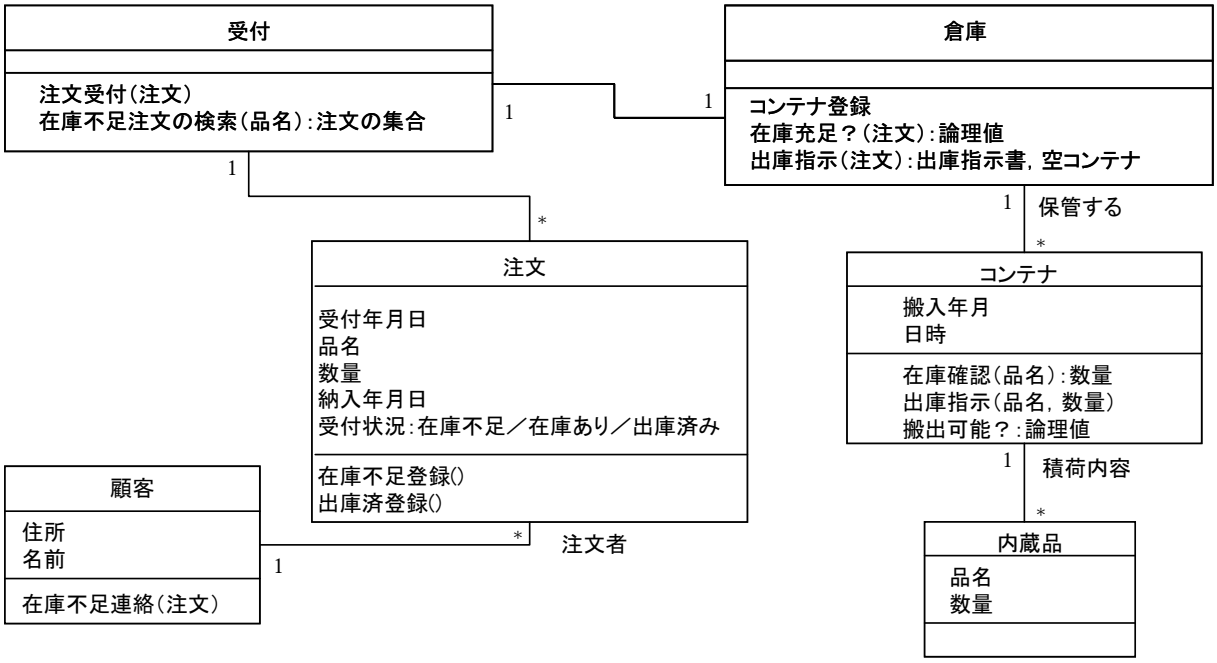


図 9.3: 酒屋問題のクラス図

9.5.3　系列図と状態遷移図

次は系列図である．系列図は少なくともユースケースに対応して同じ程度の数を描く必要があるが，ここでは代表的な在庫のある場合の注文受付に関する図 9.4 のみを示す．

最後に，状態遷移図は原則的にすべてのクラスに対して記述する必要があるが，ここでは例として,「注文」クラスの状態遷移を図 9.5 に示す．

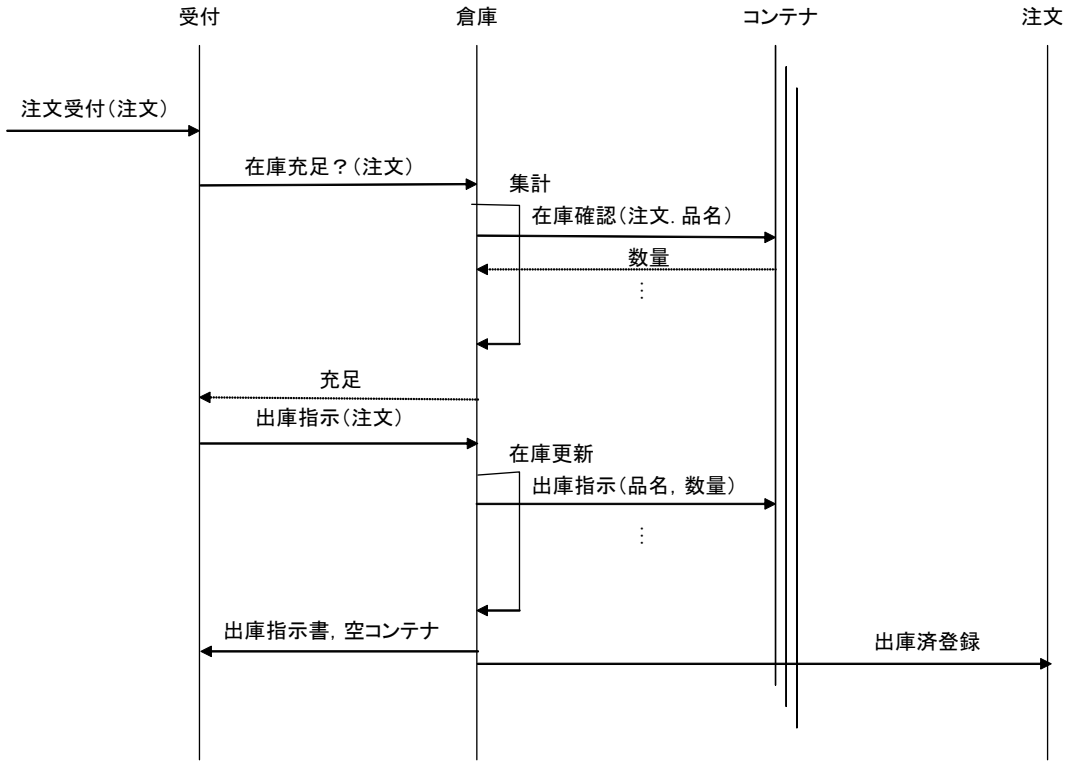


図 9.4: 注文受付 (在庫ありの場合) の系列図

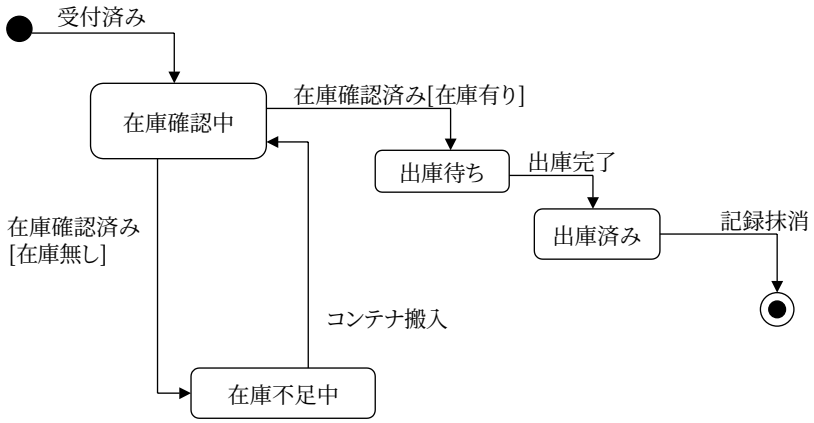


図 9.5: 注文の状態遷移図

第10章　形式手法

10.1　形式的抽象化の方法

モデル化のための方法の一つは，数学あるいは論理学で使われるような形式化 (formalization) による抽象化である．

これまで扱ってきた次のような概念が，形式化の対象となる．

- 機能の形式化
- データの形式化
- オブジェクト（もの）の形式化
- 事象生起の形式化

形式化に用いられる方法を，性質指向とモデル指向に分類する考え方がある．性質指向は，仕様化の対象となるシステムの外から見た性質を公理的な方法で定義するものであり，モデル指向はその性質を形成する基となる内部の構造を構成的に定義するものとされる．性質指向の代表は抽象データ型の代数仕様による意味定義の方法であり，モデル指向の代表は，Z や VDM のように，状態機械の仕様をその状態を構成するデータ構造と，それに対する操作の入出力仕様で定める方法である．

確かに，たとえばスタックの仕様を定めるのに，代数仕様なら push して pop すると元に戻るという等式で表された公理で基本的な性質を示すが，Z や VDM ではたとえば列 (sequence) を用いて構成的に定義するのが普通である．しかし，両者の境界はそれほど明確ではない．たとえば CCS や CSP はモデル指向の例に挙げられるようだが [115]，これまでの説明の仕方に従えばむしろ性質指向だろう．

形式化に用いられる数学的な道具としては，

- 形式論理
- 普遍代数
- 集合論

などがある．

10.2　形式的なソフトウェア開発技法

形式的な仕様化技法を，実際的なシステム開発にも応用するように適用範囲を広げる努力は，主にヨーロッパを中心として進められてきた．とくに産業的な適用を図る立場からは，厳密な仕様記述に重点を置き，開発されたソフトウェアの正当性の証明や自動的なプログラム生成は，必ずしも主眼とされていないところに特徴がある [45].

なぜヨーロッパでこのような活動が活発なのかという理由については，もちろん数学や論理学の伝統の強みということもあるだろうが，やはり Esprit プロジェクトの影響が大きいのではないかと思われる．ICOT プロジェクトに刺激された Esprit では，その対象分野を ICOT より広くとり，LSI の開発技術やソフトウェア工学を含めた．また，開発の体制は EU 内の国を跨ることはもちろんだが，産業界と学界との連係を意図している．その結果，もともと大学から生まれた Z などを，産業界に適用する試みが推進されたようである．

VDM[61] は，その名が Vienna Development Method に由来するものであることすら，いまやほとんど明示されなくなったが，もともとはプログラミング言語の操作的意味論を与えるウィーン定義言語 (Vienna Definition Language)

の流れを汲むものである．現在は操作的意味論という枠組からは完全に離れ，むしろ表示的意味論の影響を受けながら，述語論理表現に基づく仕様記述記法と，その上でのプログラム開発技法を総称したものとなっている．C. Jones を中心とするイギリスの流派と，D. Bjϕrner を中心とするデンマークの流派とがあり，両者で記法も多少異なっている．VDM に基づく仕様記述言語 VDM-SL 用の支援ツールも開発され市販されており，実際的なシステム開発に利用され始めている [40].

Z[101] はイギリスのオックスフォード大学を中心に開発された，形式的な仕様記述言語である．VDM の影響を受け，それと類似する点が多いが，記法としてより簡潔になるような工夫がなされている．実用化の例として，IBM が提供するデータ通信管理システム CICS の全体の仕様をこれで記述し，それにより保守性を高め，新しい版のエラー削減に画期的な成果を挙げた話や [87, 48]，浮動小数点演算の形式的仕様を与え，実際の演算素子開発に適用した例 [11] などが，よく引き合いに出される．前者は 2000 ページにわたる仕様を Z で書き，それに基づいて 50,000 行程度のプログラムを開発したという規模の大きさと，Z を用いた場合と用いなかった場合との比較を通じて，信頼性は数倍向上し，コストは 9%改善されたというようなデータを示したことで知られる．後者は，その成果を英国女王が表彰したことで有名になった．

RAISE(Rigorous Approach to Industrial Software Engineering)[22] はヨーロッパの Esprit プロジェクトの 1 つとして開発が進められたもので，ヨーロッパで VDM を指導してきた D. Bjϕrner を中心にしており，やはり数学的な形式性を重視している．仕様記述言語として RSL を定義し，それを用いたソフトウェアの開発方法論を展開している．RSL は，宣言的な記述だけでなく，命令的な記述や並列動作の記述のための基本要素を備えているところが，VDM などと異なる．

別の系譜に，代数的形式仕様記述言語がある．Joseph Goguen によって始められた OBJ の系列の言語が代表的で，その中でも日本の二木等による CafeOBJ は言語と処理系の完成度が高い [34].

その他，とくに通信プロトコルなどの並列動作システムの仕様記述用の言語に Lotos や Estelle があり，これらの仕様からプログラムを生成するシステムも作られている．一般に並列システムの仕様記述やシステム開発には誤りが入りやすいため，形式的な開発技法が効果をあげる可能性が高く，実例も多い．

10.3 機能の形式化

10.3.1 入力条件/出力条件に注目した機能の抽象化

対象とするシステム（ソフトウェアあるいはそのモジュール，プロセス）の機能を，入力時に成立すべき条件（入力変数やデータの条件および内部状態に関する仮定）と出力条件との対で表すことは基本的な方法である．入力条件と出力条件の記述には，述語論理などの論理的な表現が用いられることが多い．とくに，プログラムの正当性証明技術（プログラムの本体がその仕様と合致していることを数学的に証明する技術）では，通常，論理式による入出力仕様が用いられる．

対象とするシステム（モジュール，プロセス）には，通常内部状態を仮定する．

例 1

正の整数 x と y の最大公約数 z を求める．

入力条件: $integer(x) \cap integer(y) \cap x > 0 \cap y > 0$

出力条件: $integer(z) \cap divide(z, x) \cap divide(z, y) \cap \forall\, w.integer(w) \cap (divide(w, x) \cap divide(w, y) \supset z \geq w)$

例 2

配列 $a[1..n]$ をソートする．

入力条件: $integer(n) \cap intarray(a, 1, n)$

出力条件: $sorted(a') \cap permutation(a, a')$

ただし,a' は出力時の状態における配列 a を表す．$sorted(a)$ は，a の配列要素の値が昇順に並んでいること，$permutation(a, a')$ は，a' と a が置換の関係にあることを示す述語で，別にきちんと定義する必要があるが，ここでは省略する．$intarray$ も，同様．ただし，これらの例のようにデータの型もすべて述語で表すのはかなり面倒である．そこで，対象となるデータ集合を種類によって分け，それぞれの種類の集合をソートと呼び，その上で論理を展開する多ソート論理を用いるなどの方法も一般的である．

10.3.2 関数として機能を捉える方法

機能を入力（一般に複数のデータ）を変換して出力（1 つのデータ）を返す関数として捉えるもの．内部状態を仮定しない．

関数の定義は，通常，その関数が満たすべき性質を等式などの公理として与えることによって行う．たとえば次は，2 つの正の整数の最大公約数を与える関数を公理によって定義したものである．

例 1: 最大公約数を与える関数 $gcd(x, y)$

[公理]

```
gcd(x,y)  =  gcd(x,y mod x) = gcd(x mod y,y)
gcd(x,y)  =  gcd(y,x)
gcd(x,0)  =  gcd(0,x) = x
```

関数型プログラミングの場合は，公理としての等式に左辺を右辺に変換するという向きを与え，項書き換え計算の規則として用いる．次は整数のリストの要素の 2 乗和を与える関数の仕様を，関数型言語 Gofer で記述した例である．

例 2：2 乗和の計算—Gofer による記述例

[仕様]

```
sumsquares :: List Int -> Int
sumsquares = sum . squares      (関数の合成)
```

その sum と squares は，次のように定義される．

```
sum :: List Int -> Int
  sum [] = 0
  sum (n:x) = n + sum x
squares :: List Int -> List Int
  squares [] = []
  squares (n:x) = (n*n):(squares x)
```

::は関数の型を示す．

ここで a:l はリスト l の先頭に要素 a を加えたリストをあらわす．この仕様は，sumsquares を定義するのに，より簡単な sum と squares という 2 つの関数を定義して，それらの合成を用いている．

プログラム変換の考え方は，このような “仕様”（プログラムとみなせるが，そのままでは実行不可能か恐ろしく実行効率の悪いもの）を一定の規則を適用して変換していって，具体的なプログラムを得るという方法である．

10.4 データの形式化

10.4.1 抽象データ型

抽象データ型は 1970 年代に，プログラムの制御の構造化とともに，構造的プログラミングの中心概念として脚光を浴びた．抽象データ型を核とするプログラミング言語も，多く提案された．たとえば，

SIMULA, Euclid, CLU, Alphard, Modula, Ada, IOTA, ASL, LARCH

などの言語である．

これらの多くは研究的なもので広く普及したとはいえないが，後のオブジェクト指向言語に大きな影響を与えている．

抽象データ型の基本的な考え方は，次のようである．

- データ構造を，それに対する演算の組により定義づける．
- 演算には，データ生成のための演算と参照のための演算の 2 種類がある．

- 1 つの型に属するデータには，それに所属して外向けに公開されているいくつかの演算を通してのみアクセス可能である（データのカプセル化）.
- 外に公開する演算の仕様 (インターフェース) と内部の実装を分離し，内部構造は隠す (情報隠蔽，information hiding).
- データ型の具体化（実装）は，データ型間の写像として定式化される．たとえば,
  - スタックを配列で実装
  - スタックをリストで実装

10.4.2 代数的仕様記述

抽象データ型では，データ構造をそれに対する演算の組で定義するといったが，演算の意味を定義しない限りデータ型を定義したことにならない．演算の意味を与える方法として，代数的な仕様記述がある．例を示す.

例：スタックの代数的仕様—CafeOBJ による記述

```
module STACK {
 [ Stack Elt ]
 signature {
   op nilstack : -> Stack
   op push : Elt Stack -> Stack
   op empty : Stack -> Bool
   op pop : Stack -> Stack
   op top : Stack -> Elt
 }
 axioms {
   var s : Stack
   var v : Elt
   eq empty(nilstack) = true .
   eq empty(push(v, s)) = false .
   eq pop(push(v, s)) = s .
   eq top(push(v, s)) = v .
 }
}
```

この例では，スタック (厳密にいえばスタックがとる状態) を 5 つの演算で特徴づけ，その演算の意味を 4 つの等式で与えている．モジュール (module) 宣言に続いて `[ Stack Elt ]` により，ここで定義する 2 つのソートを宣言している[1]．抽象データ型としてのスタックは，演算の内の構成子 (constructor)，すなわち Stack を値域とする演算である nilstack, push, pop, を任意に組み合わせて作られる項をすべて集めた集合に対し，公理として与えられている等式で定められる同値類として，定義されることになる．記述は代数仕様言語 CafeOBJ[34] による.

10.5 形式仕様記述言語 Z

Z はオックスフォード大学で開発された形式的仕様記述言語である．Z はその基礎を集合論におき，すべてのデータは集合論に基づく型づけがなされる．そのようなデータの集まりで構成される情報単位とその性質，およ

[1]ただし，実際には Elt の方はこのモジュールで定義しているわけではない．これはスタックに入れられるデータの集合を表すが，具体的には整数でも文字列でもどんな集合でもよい．厳密には，他のモジュールで定義した構造のないソートとしての Elt を輸入 (import) する仕組みを使うべきだが，ここではそれを省略した.

びそれに対する操作を記述する枠組としてスキーマがあり，Z の言語の中で中心的な役割を果たす．

10.5.1 Spivey の例題

Z の教科書は普通，状態機械の仕様をどう記述するかという例題から始まっている．これは Z の一面的な理解に導くおそれがあると思うが，ます，記法に慣れるために，Spivey の本 [101] の最初にある例を引用する．

友人の誕生日を記録するという状況を考える．名前 (NAME) と日 (DATE) は基本的なデータ型として，所与と仮定する．それを Z では，次のように書く．

$$[NAME, DATE]$$

これを用いて，以下のようなスキーマを書く．

$BirthdayBook$

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$
$$known = \mathrm{dom}\, birthday$$

このスキーマでは，BirthdayBook という名前の構造データ型を定義している．構造データ型はその構成要素となる変数（フィールド）とその型の並びで定義される．ここでは known という変数が型 $\mathbb{P}\, NAME$ をもつものとして，また birthday という変数が型 $NAME \pfun DATE$ をもつものとして定義されている．$\mathbb{P}\, NAME$ は NAME 型 (集合）の巾集合である．つまり known の値は NAME の部分集合となる．birthday の型は NAME から DATE への部分関数である．つまり NAME の要素を与えると DATE が返ってくる関数であるが，必ずしも NAME の全域で定義されていない．

真ん中の線から下は，変数の known と birthday が満たすべき制約条件を表している．この場合の意味は，関数 birthday の定義域が known と一致するという条件である．つまり known に登録されている友達の名前のみに関して誕生日が与えられる．

$AddBirthday$

$$\Delta BirthdayBook$$
$$name? : NAME$$
$$date? : DATE$$
$$name? \notin known$$
$$birthday' = birthday \cup \{name? \mapsto date?\}$$

構造データ型を定義している BirthdayBook に対する操作を定義するスキーマである．この操作は BirthdayBook に 1 つの友人の名前と誕生日の対を登録している．ここで $\Delta BirthdayBook$ は BirthdayBook というスキーマを引用しているものであるが，Δ による引用の意味は，元の変数（known と birthday）とその間に成立すべき条件をそのまま引用するだけでなく，それらにプライム符号’ が付いたものもそっくり引用され，さらにそれらの間の制約条件もプライム符号なしの変数間の条件と同じように引用されるというものである．すなわち，明示されていないが

$$known' = \mathrm{dom}\, birthdy' \tag{10.1}$$

が条件として与えられている．

プライム符号’ が付いた変数は，ここで定義している操作の適用後の値を表すものと解釈される．これはあくまでも解釈であり，論理式は変数間の論理的な関係を表すだけである．

疑問符 “?” が付けられた変数は，これも解釈として操作に対する入力変数と見なされる．この例では $name?$ と $date?$ が入力変数である．操作による状態変化後の $birthday'$ の値は明示的に論理式によって与えられているが，

$known'$ に関する条件式がない．しかし，式 (10.1) によって $known'$ に $name?$ が加えられていることが判る．この点は Z 独特で，表現を簡潔にするが，慣れないうちは戸惑うかもしれない．

$FindBirthday$

$\Xi BirthdayBook$
$name? : NAME$
$date! : DATE$

$name? \in known$
$date! = birthday(name?)$

同じく BirthdayBook への操作を定義しているが，この操作はスキーマ BirthdayBook の状態を変えない．そのような場合は $\Xi BirthdayBook$ のように Ξ を付けて引用する．

ここで感嘆符 “!” 付きの変数は，出力変数と解釈される．

$Remind$

$\Xi BirthdayBook$
$today? : DATE$
$cards! : \mathbb{P}\, NAME$

$cards! = \{n : known \mid birthday(n) = today?\}$

FindBirthday と同様に，状態を変えない操作である．出力変数の型が NAME の巾集合である点に注意する．すなわちこの操作の出力は 1 つの誕生日ではなく一般に 0 個以上の誕生日の集合である．

$InitBirthdayBook$

$BirthdayBook$

$known = \varnothing$

スキーマで状態機械を表しているので，その初期状態を定義する必要がある．そのためのスキーマを InitBirthdayBook として定義している．これは初期化操作ではなく初期状態を宣言的に表しているので，BirthdayBook の引用に Δ も Ξ もつけられていない．

10.5.2 Z による酒屋問題の記述

以下で，Z の紹介を兼ねながら酒屋問題の仕様を Z で記述する．記述の順序はこの問題の仕様化の過程に沿っているが，節の見出しは Z の導入という意味合いでつけている．

10.5.3 Z における型

Z で表現されるすべての対象 (式) は，型を持つ．型は集合とみなしてよいが，基本的な集合としてあらかじめ与えられたものか，それらから構成された構造を持ったものかのいずれかである．基本的な集合の代表的なものは整数集合であるが (Z では $\mathbb{Z}$ と書かれる)，個々の問題領域に応じて定められるものも多い．

われわれの問題では，“コンテナ番号” と “品名” を基本的なデータの集合として所与とする．また，“数量” という名称を 0 を含まない (1 以上の) 自然数として用いる．このことを，次のように書く．

[コンテナ番号, 品名]

数量 == $\mathbb{N}_1$

一般に，

[識別子, . . . , 識別子]

により，基本となる型 (集合) の名前が導入される．以降の Z の記述では，ここで導入された型を，その内部構造には立ち入らず所与のものとして用いる．

また，

変数名 == 式

によって，右辺の式を左辺の名前で指すようにできる．

基本型から構成される構造的な型には，巾集合型，直積集合型，およびスキーマ型があるが，それらの具体例は以降の節に登場する．

10.5.4 スキーマ

さて，倉庫にはコンテナが積まれている．これを倉庫という名前のスキーマとして表そう．

倉庫

コンテナ : コンテナ番号 $\pfun$ (品名 $\pfun$ 数量)

スキーマは一般に，スキーマ名，宣言部と公理部とからなり，全体を右が開いた矩形の枠で括る．スキーマ名は枠の上部の横線の中に書く．宣言部と公理部は矩形の中に書き，両者の間に横線を入れて区切るが，この例は宣言部のみからなり，公理部がない．この宣言部では，倉庫の状態を表す 1 つの変数 “コンテナ” を，その型とともに示している．一般に，スキーマの宣言部には複数の変数宣言が並んでよい．それによって，このようなスキーマはいわゆるレコード型のデータ宣言とみなすことができる．しかし，変数の型には一般に関数なども含まれるから，それらをスキーマ内のデータへの操作と見れば，むしろオブジェクト指向でいうオブジェクトに近いともいえる．

このコンテナの型がまさに関数となっている．記号 $\pfun$ は部分関数を表すので，コンテナはコンテナ番号を定義域，(品名 $\pfun$ 数量) を値域とする部分関数であると読める (部分関数とは，必ずしもすべての定義域で関数値が定められていない関数をいう．これに対し，すべての定義域で定義された関数を全関数という)．また，値域の (品名 $\pfun$ 数量) 自身が部分関数で，品名を定義域に，数量を値域にしている．すなわち，コンテナは数量が定められた品名の集まりである．

このような関数も，集合論的に解釈するのが Z の流儀である．すなわちコンテナの集合としての型は，

コンテナ $\in \mathbb{P}$(コンテナ番号 $\times \mathbb{P}$(品名 $\times$ 数量))

となる．ここで，$\mathbb{P}$ は巾集合を作る演算である．すなわち，$\mathbb{P}\,X$ は X の部分集合の集合である．また，$X \times Y$ は X と Y の直積集合を作る演算である．すなわち，$X \times Y$ の要素は，X の要素 x と Y の要素 y の順序対 (x, y) である．

Z ではこのように，関数や関係を集合としてとらえるところに特徴がある．領域 X から Y への関数や，領域 X と Y との間の関係は，いずれも直積集合 $X \times Y$ の部分集合としてとらえ，その基本型は同じであるとする．関数は関係の一種であるが，そのうち定義域の任意の要素に対しては，それを含む対がたかだか 1 つしか関係として含まれないもの，という制約を満たすものである．

集合 X と集合 Y の間の関係という型は，$X \rel Y$ と書かれる．Z では，これを

$$X \rel Y == \mathbb{P}(X \times Y)$$

と定義している．これは，ある関係 R が $X \rel Y$ の型を持つ，すなわち $R : X \rel Y$ ということは，$R \in \mathbb{P}(X \times Y)$ であること，つまり R は集合として $\{(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\}$，ただし $x_i \in X, y_i \in Y (i = 1, \ldots, m)$，という形をしていることを意味している．

関係 $X \leftrightarrow Y$ に対して，dom と ran という，それぞれ $(X \leftrightarrow Y) \to \mathbb{P}\, X$ 型と $(X \leftrightarrow Y) \to \mathbb{P}\, Y$ 型の関数が標準的に定義されている．関係 R に対し，dom R はその定義域 (domain) を，ran R は値域 (range) を表す．

Z で関係や関数を集合として扱うのはそんなに分かりにくいことではないが，型と値，あるいは型の階層を混同しがちである．たとえば $R == \mathbb{P}\,\mathbb{P}\, S$ という型 R を考えた場合，R を型とする変数の宣言 $x : R$ や，x が R の要素であるという述語 $x \in R$ によって，x の値は S の部分集合の集合の一つとなる．さらに，$y \in x$ とすれば，y の値は S の部分集合の一つになる．またさらに，$z \in y$ とすれば，z の値は S の要素になる．このように，型宣言:や述語 $\in$ の左辺は，集合の集合という関係が作る階層で，右辺より 1 レベル下になる．一方，集合の階層レベルが上がるのは，$\mathbb{P}$ を直接適用した場合の他に，関係の定義を見れば分かるように，関係や関数をとることによっても生じるので，慣れないうちはそこに注意がいる．

スキーマの公理部とは，宣言部で定義された変数が満たすべき制約条件を，公理として与えるためのものである．この例で，問題文にある「1 つのコンテナには 10 銘柄まで混載できる」という表現を制約条件として与えるとすると，スキーマを次のように記述すればよい．

倉庫

コンテナ : コンテナ番号 $\pfun$ (品名 $\pfun$ 数量)

$\forall\, c$: dom コンテナ $\bullet$ $\#$(コンテナ c) ≤ 10

ここで，dom はすでに述べたように，関数 (一般には関係) を引数としその定義域を与える関数であり，$\#$ は有限集合を引数としてその要素数を与える関数である．いずれも Z では標準的な関数として定義されている．公理部には述語論理式が書かれる．論理演算子としては，以下が用いられる．

$\neg$	否定
$\land$	連言
$\lor$	選言
$\Rightarrow$	含意
$\forall$	全称
$\exists$	存在

全称束縛の論理式は，一般に次のように書かれる．

$\forall$ 宣言 [; 宣言; . . .] $\bullet$ 論理式

ここで，宣言とは

宣言 ::= 変数名 [, 変数名, . . .] : 型

ただし，型とは集合を表す式である．存在束縛の論理式も同様である．

なお，スキーマの公理部には，複数の論理式を改行によって並べて記述してもよい．その場合は，全体は各行の論理式の連言からなるものとみなされる．すなわち，改行を連言記号の代わりに用いることができる．

Z では，述語も集合論的に解釈される．すなわち，述語 $P(x)$ と書いた場合，P は集合を表し，その意味は $x \in P$ と等価である．したがって，新たな述語の定義も，集合を構成的に与えることでなされる．

コンテナを倉庫に入れる入庫処理は，データベースへのデータの追加という典型的な処理で，Z の種々の教科書にも必ず最初に出てくる例なので，ここでは後回しにする．

10.5.5 汎用構成子

高階関数

問題は，顧客の注文に応じた出庫の処理である．客は，ある品名の酒を何本というように注文してくる．これに応じるには，倉庫に入っているコンテナ単位の酒を，品名単位に検索する必要が生じる．そのために，酒在庫という概念を考えよう．

酒在庫 : 品名 $\pfun$ (コンテナ番号 $\pfun$ 数量)

これを，倉庫の宣言部に加える．酒在庫の型を集合の型として示すと，

$$\text{酒在庫} \in \mathbb{P}(\text{品名} \times \mathbb{P}(\text{コンテナ番号} \times \text{数量}))$$

である．

このコンテナと酒在庫が実質的に同じものであるというのが，この問題のみそである．そのために，少し一般化して考えてみよう．コンテナや酒在庫の型は，$(X \pfun (Y \pfun Z))$ という形をしているが，これは $((X \times Y) \pfun Z)$ に 1 対 1 の関係で変換することができる．この変換は，関数型プログラミングで，多変数関数を 1 変数関数の列に変えるカリー化の操作のちょうど逆になっている．そこで，型 $(X \pfun (Y \pfun Z))$ の関数を型 $((X \times Y) \pfun Z)$ の関数に変換する全単射の (高階な) 関数 $uncurry$ を定義してみよう．この関数を定義する際，対象となる型 X, Y, Z は，特定のものである必要はなく，一般に任意の型でよい．このように型をパラメータとして汎用的な関数を定義するのに便利な記述形式として，Z では汎用定義というものが用意されている．これは，次のように書く．

$$[X, Y, Z]$$

$$uncurry : (X \pfun (Y \pfun Z)) \bij ((X \times Y) \pfun Z)$$

$$\forall f : (X \pfun (Y \pfun Z)) \bullet$$
$$uncurry\, f =$$
$$\{ x : X ; y : Y ; z : Z \mid x \in \mathrm{dom} f \land y \in (f\ x) \land z = f\ x\ y \bullet (x, y) \mapsto z \}$$

これは，任意の型，X,Y,Z をパラメータとする大域的な関数 $uncurry$ を定義したものである．真中の線より下の公理部で，関数の性質を定めている．ここで，$\bij$ は全単射を表す[2]．また，$x \mapsto y$ は，関数の要素として値 x を値 y に写像する対を表している．すなわち，$x \mapsto y \in f$ であれば $f(x) = y$ である．集合としての表現と対応させると，$x \mapsto y$ は実は (x, y) と同じものである．なお，Z では関数型プログラミングの習慣に従い，関数適用を表すのに必要ない限り括弧は用いない．したがって，$f(x)$ は $f\ x$ と書くのが普通である．

この関数定義には，集合の内包記法を用いている．集合の内包記法は一般に次の形をしている．

$$\{D \mid P \bullet E\}$$

ここで，D は変数宣言，P は制約条件を表す論理式，E は式である．D で宣言された変数がとるその型の範囲内のすべて値のうち，論理式 P を満たすもののみを選び，それを E に代入して得られる値を集めた集合がこれによって表される．たとえば $\{x : \mathbb{N}_1 \mid x \leq 5 \bullet x^2\}$ は $\{1, 4, 9, 16, 25\}$ と等価である．全称あるいは存在束縛の論理式でも $\bullet$ を区切り記号として用いているが，集合の内包記法と混乱しないように適切な括弧を入れる必要がある場合があるので，注意がいる．

なお，E は省略可能で，その場合は P を満たす D の値そのものを集めた集合が表されることになる．たとえば $\{x : \mathbb{N}_1 \mid x \leq 5\}$ は $\{1, 2, 3, 4, 5\}$ と等価である．実際上は，この形式の方が使われることが多いかもしれない．

これで，述語論理式の書き方と集合の内包記法の書き方を見たことになるので，両者を使った関数の定義を見ておくことにする．すでに述べたように，関数は関係の一種で，定義域の任意の要素に対して，それを含む対がたかだか 1 つしか含まれないもの，であった．これは，Z で次のように定義される．

$$X \pfun Y == \{f : X \rel Y \mid (\forall x : X ;\ y_1, y_2 : Y \bullet$$
$$(x \mapsto y_1) \in f \land (x \mapsto y_2) \in f \Rightarrow y_1 = y_2)\}$$

(この式に現れる $\bullet$ は，集合の内包記法に出てくる $\bullet$ ではなく，全称束縛の $\bullet$ であることに注意．)

さて，$uncurry$ によってコンテナを 2 引数の関数とみなす変換ができたので，次に第 1 引数と第 2 引数を入れ換える変換を考える．

[2] 全単射は，全射でかつ単射という意味である．$f : X \pfun Y$ が全射とは，その値域が Y 全体を覆う場合，すなわち $\mathrm{dom} f = Y$ の場合で，X から Y の上への写像とも呼ばれる．単射は，いわゆる 1 対 1 の写像で，$f : X \pfun Y$ が単射とは，$\forall x_1, x_2 : \mathrm{dom} f \bullet f\ x_1 = f\ x_2 \Rightarrow x_1 = x_2$ を意味する．

$[X, Y, Z]$

$swap : ((X \times Y) \pfun Z) \rightarrowtail\!\!\!\!\!\!\rightarrow ((Y \times X) \pfun Z)$

$\forall f : (X \times Y) \pfun Z \bullet$
$\quad swap\, f = \{\, x : X;\ y : Y;\ z : Z \mid (x, y) \in \mathrm{dom}\, f \land z = f(x, y) \bullet (y, x) \mapsto z \,\}$

結局，一連の変換は $uncurry^{\sim} \circ swap \circ unurry$ と表すことができる．ここで，$\circ$ は関数の合成を，$uncurry^{\sim}$ は $uncurry$ の逆関数を表す (単射の関数には逆関数が定義できる)．この逆関数は $curry$ と名付けるのが自然だから，

$$curry == uncurry^{\sim}$$

としておく．

これらを用いて，倉庫のスキーマを改めて次のように書くことができる．

倉庫

コンテナ : コンテナ番号 $\pfun$ (品名 $\pfun$ 数量)
酒在庫 : 品名 $\pfun$ (コンテナ番号 $\pfun$ 数量)

酒在庫 $=$ $curry \circ swap \circ uncurry$ コンテナ

多重集合

出庫依頼があった場合，指定された銘柄の酒の在庫量が，注文に応じられるだけあるかという判断が必要となる．また，コンテナが空の場合にはコンテナを搬出するという要求があるので，コンテナに現在入っている酒の総量も必要かも知れない．これらを与える関数は，

銘柄在庫量 : 品名 $\pfun$ 数量
コンテナ在庫量 : コンテナ番号 $\pfun$ 数量

という形式となろう．この 2 つの関数の仕様を書いてみよう．

これらの関数と，酒在庫およびコンテナの定義を比べれば，それぞれの関係は明らかである．銘柄在庫量は酒在庫と同じく品名を定義域とし，その写像先の コンテナ番号 $\pfun$ 数量 に関して数量の総和をとったものである．同様に，コンテナ在庫量は，コンテナの値域の要素に対してやはりその数量の総和をとったものとみなせる．このことを形式的に記述すればよい．

そこで，たとえば 品名 $\pfun$ 数量，あるいは同じことだが，$\mathbb{P}$(品名 $\times$ 数量) の要素に対して，それぞれの第 2 成分である数量を集めるという操作をまず考え，次にその総和をとることにすればよい．それには一般に関係に対して定義されている関数 ran を使えばよいようだが，問題は第 2 成分を集めたものを集合とすると都合が悪いことである．集合は同じ要素を重複して考えないが，この場合は同じ数量の値が複数回現れても，それぞれ別のものとして扱う必要がある．そのような集まりを多重集合 (bag または multiset) という．

要素 $a_1, \ldots, a_n$ からなる多重集合を，Z では $[\![a_1, \ldots, a_n]\!]$ と書く．たとえば，$[\![1, 2, 2, 3]\!]$ と $[\![1, 2, 3]\!]$ とは異なる (しかし順序は関係ないから，$[\![1, 2, 2, 3]\!]$ と，たとえば $[\![1, 2, 3, 2]\!]$ は同じである)．Z では多重集合を，要素から 1 以上の自然数への関数として定義している．写像先の自然数が多重集合内の出現回数を表している．すなわち，一般に型 X の多重集合 bag X は，

$$\mathrm{bag}\, X == X \pfun \mathbb{N}_1$$

と定義される．たとえば $[\![1, 2, 2, 3]\!]$ は bag$\mathbb{N}$ の要素であるが，定義によれば $\{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 1\}$ と表される．

われわれは ran の多重集合版が欲しいので，参考のために dom と ran の定義を見てみよう．

$$
\begin{array}{|l}
\hline
[X, Y] \\
\mathrm{dom} : (X \leftrightarrow Y) \to \mathbb{P}\, X \\
\mathrm{ran} : (X \leftrightarrow Y) \to \mathbb{P}\, Y \\
\hline
\forall\, R : X \leftrightarrow Y \bullet \\
\quad \mathrm{dom}\ R = \{\, x : X;\ y : Y \mid x \mathrel{\underline{R}} y \bullet x \,\} \land \\
\quad \mathrm{ran}\ R = \{\, x : X;\ y : Y \mid x \mathrel{\underline{R}} y \bullet y \,\} \\
\hline
\end{array}
$$

ここで $\underline{R}$ は関係 R を中置記号の演算子として使う場合の記法で，$x \mathrel{\underline{R}} y$ は $(x, y) \in R$ と等価である．

これをまねて，多重集合用の dom と ran に相当する関数 $bdom$ と $bran$ の定義を次のように書いてみる．

$$
\begin{array}{|l}
\hline
[X, Y] \\
bdom : (X \leftrightarrow Y) \to \mathrm{bag}\, X \\
bran : (X \leftrightarrow Y) \to \mathrm{bag}\, Y \\
\hline
\forall\, R : X \leftrightarrow Y \bullet \\
\quad bdom\ R = [\![\, x : X;\ y : Y \mid x \mathrel{\underline{R}} y \bullet x \,]\!] \land \\
\quad bran\ R = [\![\, x : X;\ y : Y \mid x \mathrel{\underline{R}} y \bullet y \,]\!] \\
\hline
\end{array}
$$

残念ながら，現在の Z にこのような書き方はない．多重集合を記述する $[\![\cdots]\!]$ には，外延的な記述はできても内包的な記述は定められていない．

そこで，

$$
\begin{array}{|l}
\hline
[X, Y] \\
bdom : (X \leftrightarrow Y) \to \mathrm{bag}\, X \\
bran : (X \leftrightarrow Y) \to \mathrm{bag}\, Y \\
\hline
\forall\, R : X \leftrightarrow Y \bullet \\
\quad bdom\ R = \{\, x : \mathrm{dom}\, R \bullet x \mapsto \#\{\, y : Y \mid x \mathrel{\underline{R}} y \} \,\} \land \\
\quad bran\ R = \{\, y : \mathrm{ran}\, R \bullet y \mapsto \#\{\, x : X \mid x \mathrel{\underline{R}} y \} \,\} \\
\hline
\end{array}
$$

とすればよいだろう．

次に，自然数 $\mathbb{Z}$ の多重集合について，その要素の総和をとる関数 Σ を定義しよう．

$$
\begin{array}{|l}
\hline
\Sigma : \mathrm{bag}\, \mathbb{Z} \to \mathbb{Z} \\
\hline
\Sigma [\![]\!] = 0 \\
\forall\, x : \mathbb{Z} \bullet \Sigma [\![x]\!] = x \\
\forall\, B, C : \mathrm{bag}\, \mathbb{Z} \bullet \Sigma(B \uplus C) = \Sigma B + \Sigma C \\
\hline
\end{array}
$$

ここで用いた記法は，スキーマの外枠がない形をしているが，公理記述と呼ばれ，大域変数を宣言しその性質を公理として与えるものである．スキーマで宣言された変数はそのスキーマ名を引用しないと参照できないが，公理記述で宣言された変数は大域的なのでいつでも参照できる．その点では汎用定義で宣言される変数と似ているが，汎用定義はパラメータ化された大域変数 (通常は関数) を定めるのに対し，公理記述はコンスタントとしての変数 (関数) を定めているところが異なる．

なお，$[\![]\!]$ は空の多重集合を表し，記号 $\uplus$ は 2 つの多重集合の合併をとる演算を表す．

この Σ のような関数は，単位元を持ち可換で結合的な 2 項演算の定義された集合 X の上の多重集合 $\mathrm{bag}\, X$ に対して，一般化して定義することも可能だが，ここではそこまで一般化せず，自然数の上の加算に対して定義した．

以上の準備の元に，酒在庫およびコンテナからそれぞれ銘柄在庫量およびコンテナ在庫量をえる変換を $subtotal$ という (高階) 関数として定義すると，

$[X, Y]$

$subtotal : (X \pfun (Y \pfun \mathbb{Z})) \fun (X \fun \mathbb{Z})$

$\forall f : X \pfun (Y \pfun \mathbb{Z}) \bullet subtotal\ f = \{ x : dom\ f \bullet x \mapsto \Sigma \circ bran \circ f\ x \}$

これを用いて，

銘柄在庫量 $==$ $subtotal$ 酒在庫
コンテナ在庫量 $==$ $subtotal$ コンテナ

となる．

倉庫のスキーマを，この 2 つの関数を含めた形に拡張して定義し直しておく．

倉庫

コンテナ : コンテナ番号 $\pfun$ (品名 $\pfun$ 数量)
酒在庫 : 品名 $\pfun$ (コンテナ番号 $\pfun$ 数量)
銘柄在庫量 : 品名 $\pfun$ 数量
コンテナ在庫量 : コンテナ番号 $\pfun$ 数量

酒在庫 $=$ $curry \circ swap \circ uncurry$ コンテナ
銘柄在庫量 $==$ $subtotal$ 酒在庫
コンテナ在庫量 $==$ $subtotal$ コンテナ

10.5.6 抽象機械

これまでの Z による記述は，対象とする問題領域の，主として静的な側面をとらえていた．その方法としては抽象化された関数を利用し，対象のもつ論理的な構造をなるべく簡潔にとらえるというものであった．しかし，Z の大きな特徴の一つは，対象を状態を持つ抽象機械としてとらえ，その動的な動作を記述するところにある．スキーマは，そのために用いられることが多い．

操作の仕様　たとえば，倉庫というスキーマは，これまでのところはコンテナと酒在庫という構成要素を持つデータ型の値がとり得る値の空間を規定したもの，という扱いであった．これを抽象機械として考える場合は，任意の時点でそのような値の 1 つを状態として持つ機械 (またはプロセス，またはシステム) を想定することになる．機械は，外から受ける操作によって状態を変化させる．その状態変化を表すには，操作前と操作後の状態間の関係を記述する仕組みが必要である．

たとえば，コンテナの入庫処理を考えてみよう．

入庫

倉庫
倉庫′
$cn?$: コンテナ番号; $s?$: 品名 $\pfun$ 数量

$cn? \notin$ dom コンテナ
コンテナ′ $=$ コンテナ $\cup \{cn? \mapsto s?\}$

このスキーマ定義では，他のスキーマの引用と修飾という重要な記法が使われている．まずスキーマの引用として，ここでは“倉庫” スキーマが引用されている．この場合，引用されたスキーマの宣言部はそのままここで定義しているスキーマ “入庫” の宣言部に加えられ，またその公理部はそのまま “入庫” の公理部に加えられる．一方，“倉庫′” という記法はスキーマの修飾つき引用で，その意味は倉庫の宣言部で宣言されたすべての変数 (“コンテナ”，“酒在庫”，“コンテナ在庫量”，“銘柄在庫量”) にプライム符号 ′ を付け，またその公理部に現れるそれ

らの変数もすべて′付きに置き換えてえられるスキーマを引用するということである．この′付きの変数は，通常はここで定義しようとしている“入庫”という操作を表すスキーマの，操作終了後の状態を指し示すと解釈される．

状態機械の操作を定義するスキーマでは，このように状態を表すスキーマと，それにプライム符号′を付けたスキーマの両者を引用することが普通なので，それをいっぺんに表す記法も定められている．それには Δ(ギリシャ文字デルタの大文字) を引用するスキーマ名の頭につける．つまり，上の例では

入庫

Δ倉庫

$cn? :$ コンテナ番号; $s? :$ 品名 $\rightarrow$ 数量

$cn? \notin$ dom コンテナ

コンテナ′ $=$ コンテナ $\cup \{cn? \mapsto s?\}$

と書くことができる．

入力/出力条件　このスキーマでは別に，$cn?$ と $s?$ という 2 つの変数を導入している．このように後ろに?を付けた変数は，この操作に対する入力データを表す変数と解釈される．また，この例では現れないが，出力を表す変数は後ろに!を付ける決まりである．これらはしかし，あくまでも状態機械の操作を表しているという解釈のための便宜上の約束で，論理式としては通常の変数とまったく同様の変数にすぎない．

このような操作のスキーマの公理部は，操作の入力条件と出力条件を表している．Z 以外の形式的仕様記述言語では，入力条件と出力条件を構文的に区別して記述するようにしているものが多いが (たとえば VDM)，Z ではそのような区分をしない．しかし，公理部を構成する論理式 (全体はそれらを連言で結合したものと解釈される) のうち，′あるいは!を接尾辞としてもつ変数を一つも含まないものは入力条件であり，それらを一つでも含むものは出力条件である解釈できるので，あえて入力/出力条件を区分する構文要素を導入しない方針をとっているわけである．

この′記法は強力で，とくに引用されたスキーマの公理部も自動的に取り込まれることで，操作によって不変な条件を明示的に書く必要がない．上の例で，“コンテナ”に関する出力条件のみを記述して，“酒在庫”について記述していないのは，修飾つきのスキーマの引用により，暗に

$$\text{酒在庫}' = curry \circ swap \circ uncurry \ \text{コンテナ}'$$

が成立している (入庫のスキーマの公理部に加えられている) からである．このやり方は Z 独特で便利ではあるが，時として操作の仕様を分かりにくくすることもある．

参照型の操作　操作の中には状態を変化させないで，状態を参照するためのものもありうる．その場合は，引用するスキーマの頭に Ξ(ギリシャ文字クシーの大文字) を付けると，Δ を付けた場合と同様に，そのスキーマと′付きのスキーマとを同時に引用するだけでなく，公理部には暗にスキーマで宣言されているすべての変数 x に対し，$x = x'$ という等式が追加される．(しかし，実際に公理部に記述する論理式には，′付きの変数は現れないはずである．)

たとえばすでに定義した関数“銘柄在庫量”を使って，銘柄を指定し在庫量を調べる参照操作を定義してみよう．

銘柄在庫確認

Ξ倉庫

$b? :$ 品名, $v! :$ 数量

$v! =$ 銘柄在庫量 $b?$

初期化　状態機械ならば，その初期状態をなんらかの方法で定める必要がある．それも一つのスキーマとして定義すればよい．倉庫に対しては，たとえば

初期倉庫

倉庫
コンテナ = ∅

これを初期化の操作として定義するなら，Δ 倉庫を用いて コンテナ′ = ∅ とすることになるが，上の定義は倉庫の初期状態を宣言的に記述したものである．

エラー処理 “入庫” 操作では，入庫されるコンテナのコンテナ番号が，すでに倉庫にあるコンテナのコンテナ番号と一致しないという入力条件をおいている ($cn? \notin$ dom コンテナ)．この条件を満たさないような例外事象を，仕様としてどのように記述したらよいだろうか．Z ではそのような例外処理の仕様は，別のスキーマとして分けて記述するのが，通常のスタイルである．

まず，エラーを識別するための記号からなる型，“エラー条件” を定義する．

エラー条件 ::= 正常 | コンテナ番号重複

ここで ::= は自由型の宣言子で，右辺で | で区切られて列挙された要素で構成される型を宣言している．この例では単純な列挙型だが，再帰的な定義が可能で，それにより木のような再帰的に定義される型を導入できる．

次に，正常な場合を表すスキーマを，“正常終了” として定義する．

正常終了

結果! : エラー条件
結果! = 正常

これはつまらないスキーマに見えるが，スキーマ式という仕組みを使うことで役に立つようになる．スキーマ式とはスキーマ同士を演算で結んだもので，その演算子にはいくつかの種類があるが，ここでは論理型スキーマ演算子の ∧ と ∨ のみをあげておく．どちらも 2 つのスキーマをとり，新たなスキーマを構成する．2 つのスキーマの宣言部は，どちらの場合も合併がとられる．その際，同じ名前の変数が両者にある場合は，それぞれのスキーマにおけるその変数の型 (集合) の共通部分をとって，構成されるスキーマでは，それを型とする一つの変数が宣言される．また，公理部は ∧ の場合は 2 つのスキーマの公理部の連言が，∨ の場合は 2 つのスキーマの公理部の選言がとられる．

そこで，“入庫 ∧ 正常終了” とすれば，さきほど定義した入庫に正常という結果を返すという機能が付加されることになる．

次にコンテナ番号の重複が起こるケースは，次のようなスキーマで記述すればよい．

番号重複

Ξ倉庫
$cn?$: コンテナ番号
結果! : エラー条件
$cn? \in$ dom コンテナ
結果! = コンテナ番号重複

これを用いて，例外条件を含めた入庫操作は，

一般化入庫 ≙ (入庫 ∧ 正常終了) ∨ 番号重複

と書ける．ここで ≙ は，左辺のスキーマを右辺のスキーマ式で定義するものである．

10.5.7 仕様の完成

仕様の残りの部分を完成させよう．具体的には，酒の出庫の操作と在庫不足処理についての仕様を記述する必要がある．

出庫

出庫処理は，受付係が出庫依頼を受けて出庫指示書を作り，それに基づいて倉庫係が出庫する，という一連の処理となる．在庫不足の場合は，在庫なし連絡をして，在庫不足リストを作成する．

この出庫処理全体の仕様を一つのスキーマとすれば，在庫がある場合の出庫というスキーマと在庫不足の場合の処理を記述するスキーマに分け，

出庫処理 $\widehat{=}$ 出庫 $\vee$ 在庫不足処理

のような構造とすればよいであろう．また，出庫は受付係の仕事と倉庫係の仕事に分け，その順に処理するように記述するのが自然だろう．それには，

出庫 $\widehat{=}$ 出庫指示書作成 $>>$ 出庫実施

と書けばよい．ここで，$>>$ はパイプ結合を表すスキーマ演算子で，$A >> B$ は直観的には A の実行の後，その出力を B の入力として B を実行することを意味する．より正確な意味は後で示す．

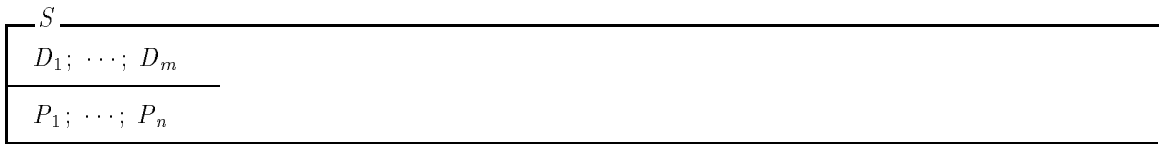
以下で，これらのスキーマを記述していこう．まず，基本的なデータとして，出庫依頼と出庫指示書を定義する．そのために，基本型として，“送り先名” を追加しておく．

[送り先名]

これを用いて，出庫依頼と出庫指示書を次のように定義する．

出庫依頼 $\widehat{=}$ [b : 品名; a : 数量; to : 送り先名]
出庫指示書 $\widehat{=}$ [to : 送り先名; b : 品名; f : コンテナ番号 $\pfun$ 数量]

今までは，スキーマを定義するのには常に箱を用いてきた．しかし，この場合のように，単に組 (tuple) のデータ構造を定義するために導入するような短いスキーマ定義には，箱の表記は大げさ過ぎる．そこでこのような「横書き」の表記法がある．一般に，箱を用いた縦書き表記，



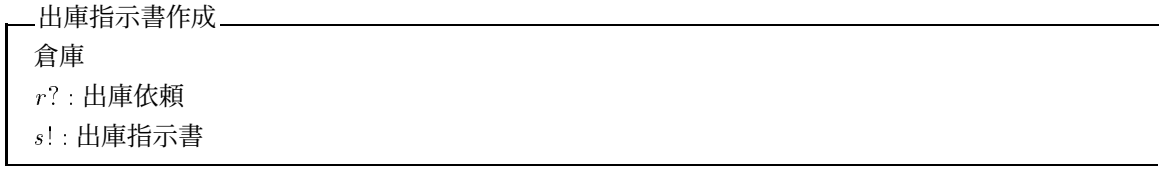
と，横書き表記

$S \mathrel{\widehat{=}} [D_1; \cdots; D_m \mid P_1; \cdots; P_n]$,

とは等価である．ただし，縦書き表記では，セミコロンの代わりに改行を用いてもよい．

上の定義で，問題文にある出庫指示書の構造定義から「注文番号」を省き，また品名は 1 つの送り先について共通なので，コンテナに関するデータの繰り返しの外側に括り出した．さらに，ここではとりあえず空コンテナに関する仕様は後回しにすることにし，省いている．

出庫指示書作成の宣言部は，次のような形式になろう．



ここでスキーマを型として用いている．スキーマの宣言部は，変数名と型の対の並びであり，公理部は宣言部で宣言された変数の間に成り立つべき制約条件を与えるものであった．したがって，スキーマはいわゆるレコード型を，それが保つべき不変条件とともに定義しているものとみなすことができる．

公理部には，出庫依頼にある品名の数量が現在の在庫で充足されるという入力条件と，出庫指示書にある数量の合計がその依頼数量と一致するという出力条件とを規定する必要がある．そのためには，すでに準備した銘柄在庫量という便利な関数があるので，それを使えばよい．すなわち，

出庫指示書作成

$$
\begin{array}{l}
\text{倉庫} \\
r? : \text{出庫依頼} \\
s! : \text{出庫指示書} \\
\hline
r?.b \in \mathrm{dom}\ \text{銘柄在庫量} \land\ r?.a \leq \text{銘柄在庫量}\ r?.b \\
s!.to = r?.to \land s!.b = r?.b \land \Sigma \circ bran\ s!.f = r?.a \land \\
\forall\, cn : \mathrm{dom}\ s!.f \bullet cn \in \mathrm{dom}\ \text{コンテナ} \land f\ cn \leq \text{酒在庫}\ r?.b\ cn
\end{array}
$$

ここですでに定義した Σ と $bran$ を用いた．

この仕様は，非決定的である．すなわち，この仕様を満たす出力 $s!$ は一般に一意には定まらない．しかし，仕様は決定的である必要はまったくない．むしろ本質的でない性質についてはあえて自由度を残して非決定的に記述した方が，仕様として簡明で分かりやすいことが多い．ただし，あまりに自由度が大きな仕様は，設計者の負担が大きくなるという面もあり，その兼ね合いに配慮がいる．

出庫実施のスキーマを記述するには，少し準備がいる．次のような総称関数 $ransum$ を用意する．これは同じ定義域を持ち，値域が整数である 2 つの関数から，同じ定義域の要素に対してそれぞれの関数の適用結果の和を与えるような関数を作る高階関数である．

$[X]$

$$
\begin{array}{l}
ransum : (X \pfun \mathbb{Z}) \to (X \pfun \mathbb{Z}) \to (X \pfun \mathbb{Z}) \\
\hline
\forall\, f, g : X \pfun \mathbb{Z} \bullet \\
ransum\ f\ g = \{x : (\mathrm{dom}\, f) \cap (\mathrm{dom}\, g) \bullet x \mapsto ((f\ x) + (g\ x))\} \cup \\
\qquad (\mathrm{dom}\, g) \ndres f \cup (\mathrm{dom}\, f) \ndres g
\end{array}
$$

ここで，$(\mathrm{dom}\, g) \ndres f$ は，f の内定義域が g と重なるものを除くことを表す．これを用いて，出庫実施は次のように書ける．

出庫実施

$$
\begin{array}{l}
\Delta\text{倉庫} \\
s? : \text{出庫指示書} \\
\hline
s?.b \in \mathrm{dom}\ \text{酒在庫} \land \\
\forall\, cn : \mathrm{dom}\ s?.f \bullet cn \in \mathrm{dom}\ \text{コンテナ} \land f\ cn \leq \text{酒在庫}\ s?.b\ cn \\
((s?.b \in \mathrm{dom}\ \text{酒在庫}' \land \text{酒在庫}\ s?.b = ransum\ (\text{酒在庫}'\ s?.b)\ s?.f) \lor \\
\qquad (s?.b \notin \mathrm{dom}\ \text{酒在庫}' \land \text{酒在庫}\ s?.b = s?.f)) \\
\{s?.b\} \ndres \text{酒在庫} = \{s?.b\} \ndres \text{酒在庫}'
\end{array}
$$

この記述も非決定的である．とくに $ransum$ という 2 つの項の “和” を取る関数を，操作後の状態値である酒在庫’ と入力変数とに対して用いている点に注意しよう．一般に，状態変数 S と 2 項演算 $\oplus$ があって，

$$S' = S \oplus A$$

のように書ける時は決定的であるが，

$$S = S' \oplus A$$

のように書ける場合は，$\oplus$ が逆演算をもたない限り決定的ではない．これは，S を S' と A に “分割” するが，分割の方法が一通りとは限らない場合と解釈することができる．しかし，あえて条件を付加して決定的な形に書き換えるよりも，この形式の方が仕様らしい記述となる場合がある．なお，非決定的な仕様に条件を付加して決定性を高める方法については，後で述べる．

出庫指示書作成と出庫実施をパイプ結合でつなげる．

出庫 $0 \mathrel{\widehat=}$ 出庫指示書作成 $>>$ 出庫実施

この結合は，出庫指示書作成の出力 $s!$ と出庫実施の入力 $s?$ を結合して同一の変数とし，さらにその変数を存在限量子で束縛して外部から隠すというものである．

在庫不足処理

在庫不足で出庫できない依頼については，それを在庫不足リストとして登録する．そのために在庫不足管理というスキーマを定義する．

在庫不足管理 $\mathrel{\widehat=}$ [在庫不足リスト : seq 出庫依頼]

ここで seq X は X の要素からなる列 (sequence) を表す．Z では列は自然数から X への関数として定義される．すなわち，

$$\text{seq}\, X == \{f : \mathbb{N} \pfun X \mid \text{dom}\, f = 1..\#f\}$$

在庫不足処理 0

倉庫
Δ在庫不足管理
$r?$: 出庫依頼
$r?.b \notin \text{dom}\, 酒在庫 \lor (r?.b \in \text{dom}\, 酒在庫 \land r?.a > 銘柄在庫量\ r?.b)$
$在庫不足リスト' = 在庫不足リスト^\frown < r? >$

ここで $^\frown$ は 2 つの列の連接演算子である．

在庫不足の場合は，依頼元に報告することになっているので，その分を常套的な方法で付け加える．

報告 ::= 充足 | 不足
不足報告 $\mathrel{\widehat=}$ [r : 報告 | $r =$ 不足]
充足報告 $\mathrel{\widehat=}$ [r : 報告 | $r =$ 充足]

結局，

出庫 $\mathrel{\widehat=}$ 出庫 $0 \land$ 充足報告
在庫不足処理 $\mathrel{\widehat=}$ 在庫不足処理 $0 \land$ 不足報告

となる．

これで仕様がようやく完成した．ただし，この仕様は酒屋問題の第 1 版に対応するものであり，在庫不足で応じられなかった出庫依頼に対して，後に入荷されたコンテナにより出庫が可能となった場合の出庫処理については記述していない．

第11章　設計技法

設計は工学の華である．これまでの章で見てきたモデル化技法は，対象となる問題領域のモデルを構築して何を作ればよいかという「要求」を明確にする目的に用いるとともに，いかに作るべきかという「設計」の基盤を与えるものとして使うことができる．すなわち要求分析で作ったモデルに基づいて，設計さらに実装まで一貫したプロセスをとることが，重要である．

11.1　アーキテクチャ

設計の仕事は，分析によって明らかになった領域モデルの上で，満たすべき要求，解くべき問題を正確に捉え，その解となるべきシステムの構造を構築することである．その作業には，全体の構造，すなわちアーキテクチャを定める部分と，より具体的なアルゴリズムやデータ構造を設計する部分とがある．

11.1.1　アーキテクチャとは

アーキテクチャとはもちろん「建築」を意味する．建築物を指す場合もあれば建築術あるいは建築学を指す場合もある．これを他の工学分野の設計概念として用いる場合は,「基本設計思想」と訳されることもあり，基本的な設計構造というほどの意味で使われる．

工学分野の中でも，とくにコンピュータ分野で比較的古くからアーキテクチャという用語が使われてきた．たとえば

コンピュータ・アーキテクチャ コンピュータ・ハードウェアの基本構造を表す用語として古くから定着している．狭い意味では命令セットを表す．命令セットに設計の基本思想が如実に現れるからであろう．

ネットワーク・アーキテクチャ ネットワークのプロトコル階層モデルをネットワーク・アーキテクチャと呼んでいる．典型的なものに ISO/OSI モデルがある．

ソフトウェアのアーキテクチャという表現も歴史は古い．たとえば 1975 年に出版された F. Brooks の “The Mythical Man-Month”[23] にも，ソフトウェア・アーキテクチャという語がしばしば登場する．その意味するところは要求仕様に近く，現在の使い方と若干のずれがあるが，基本的な要求仕様がソフトウェアの基本構造を決めるという意味では違和感はない．命令セットがコンピュータの基本構造を決めているのと類似の関係といえる．

しかし，ソフトウェア工学分野でアーキテクチャという語が華々しく復活したのは，1990 年代中頃である．第 2 章で述べたように，ソフトウェアプロセスの研究は 1980 年代の後半から 1990 年代の前半に盛んとなったが，プロセスへの関心がある程度続くと，その反動としてプロダクトへの関心に回帰するというのが，ソフトウェア工学の歴史である．1990 年代半ばからのアーキテクチャへの関心の高まりは，プロセスからプロダクトあるいはプロダクトの設計に重心が移行したものと見ることもできよう．

11.1.2　アーキテクチャの役割

ソフトウェア・アーキテクチャの探求は研究界が先行した．それを象徴するのが，1996 年にカーネギーメロン大学の M. Shaw & D. Garlan が出した “Software Architecture” という本である [95]．その後，産業界でもとくに Web 上で動く応用システムをコンポーネントベースで開発するにあたって分散アーキテクチャを採用し，コンポーネントを配備するために EJB(Enterprise JavaBeans) などの標準的なフレームワークを使用するという状況から，アーキテクチャに意識的にならざるを得なくなった．

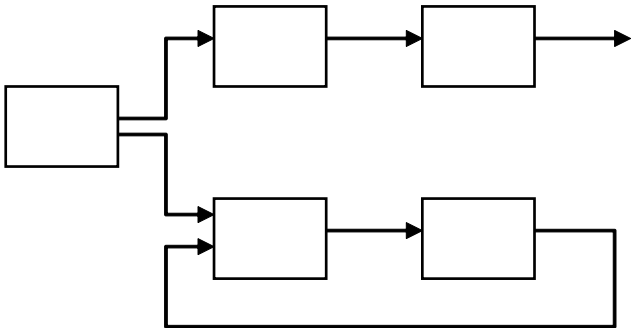


図 11.1: パイプとフィルタ・アーキテクチャ

アーキテクチャはシステム要求仕様を受け，それを実現するためのシステム全体の構造を定めるものである．システム要素によって構成されるシステム構造は，要求の構造と基本的に対応する．システム要素の機能と振舞いと，それを合成して作られるシステム全体の機能と振舞いが，アーキテクチャによって定まる．その際，次のようなシステム全体にわたる性質を考慮する．

- 処理規模と性能
- 大域的制御構造
- 物理的分散配置
- 通信プロトコル
- データベースとデータアクセス
- 将来の発展方向の見通し
- 入手可能なコンポーネントとその適合性

これらを考慮した上で，アーキテクチャの代替案を候補の対象とし，取捨選択する．

11.1.3 建築様式 (Architecture Style)

建築史の上では，ギリシャ様式，ローマ様式，ロマネスク様式，ゴシック様式，ルネッサンス様式，バロック様式，ロココ様式，アールヌーヴォー，モダン，ポストモダンなど，さまざまな建築様式の流れがある．その様式という見方を借用して，さまざまなアーキテクチャ・スタイルを類別しアーキテクチャ設計に利用しようという考え方がある．

ソフトウェア・アーキテクチャの様式として挙げられる代表的なものに，以下がある．

- パイプとフィルタ：1 つ以上のデータ流 (data stream) を入力とし，それを変換して 1 つのデータ流を出力する計算部品をフィルタという．1 つのフィルタの出力を別のフィルタの入力にパイプでつないで，全体の計算流を構成するのがパイプとフィルタによるアーキテクチャである．UNIX のパイプとフィルタが典型的である（図 11.1 参照）．
- 層別 (layers)： 計算部品は上位から下位に並べられた層に分かれて配置される．各層は同様なレベルの機能を提供する部品群からなり，それぞれの機能を実現するために呼び出すことが可能な部品は，自分の直接下位にある層の部品だけである．逆に，自分の直接上位にある層の部品にのみ機能を提供する．層の構造は図 11.2 のように同心円状に描かれる場合もあれば，水平方向の厚みをもった平面からなる層を，垂直方向に重ねるように（あるいはむしろその断面図として）描かれる場合もある（図 11.2 参照）．

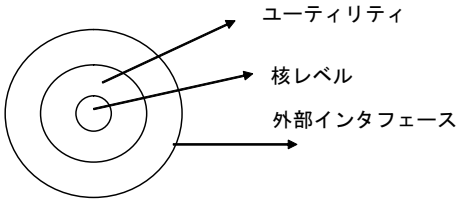


図 11.2: 層別アーキテクチャ

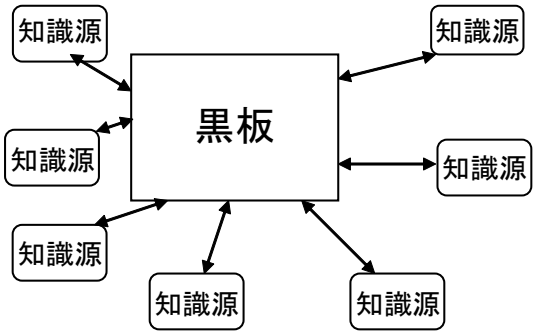


図 11.3: 黒板アーキテクチャ

- 事象駆動： 計算部品を手続き呼び出しで明示的に起動するるのではなく，それぞれの計算部品が特定の事象を見張っていて，それが発生すると一定の動作をするというアーキテクチャである．事象を発生させるのも計算部品である．事象を発生させる側が事象の受け取り手を指定する場合は，手続き呼び出しに近くなるが，不特定多数への同報通信 (broadcasting) で事象を伝達する場合は，典型的な事象駆動型となる．
- MVC： システムをモデル (Model)，表示 (View)，制御 (Control) という 3 つの側面に分け，それを合成して全体のシステムとするアーキテクチャを MVC と呼ぶ．GUI による対話式の応用システムでは，古くから使われてきた．制御は対話式に利用者の指示や入力を受け取ってモデルに伝え，モデルは計算やシミュレーションをモデル化し，表示はモデルの状態変化に応じて表示結果を変える．同じモデルに対して，表示を異なるものに取り替えたり，異なる制御と組合わせたりすることが比較的容易にできる．
- ルールベース： 条件と動作の対で記述された規則の集合を核とするシステムのアーキテクチャである．各規則は条件部が成立したときに動作部が実行されるという形で解釈される．古くは Simon & Newell の生成 (production) システムや知識工学がもてはやされた時代の知識ベースシステムなどが，このアーキテクチャの典型例である．知識を規則の集合として表現し，知識を追加したり変更したりして全体の動作を進化させていくような応用システムに向いている．
- 黒板システム： やはり AI の分野で分散的な知識システムの実現のために考案されたアーキテクチャである．独立性の高い知識源が多くあり，それらは大域的な共有メモリである「黒板」を介して相互作用する．各知識源は黒板から情報を読み取り，また黒板に情報を書き込む（図 11.3 参照）．音声認識システムなどに適用された．

11.1.4 アーキテクチャに関連する概念

ソフトウェア・アーキテクチャはオブジェクト指向とは独立の概念であるが，ソフトウェア・アーキテクチャがさまざまに検討された 1990 年代半ば以降はオブジェクト指向技術が定着した時でもあるので，関連する多くの問題がオブジェクト指向と結びつけて議論された．その中でとくにフレームワークとパターンについて簡単に触れる．

フレームワーク　特定の領域向けにアーキテクチャを具体化した骨組みを，フレームワークという．フレームワークに用途に応じた特定のモジュールを付け加えることで，具体的な応用システムが開発できる．たとえば金融商品の販売管理システムのフレームワークがあるとすると，A 銀行がそれを利用し，自社固有の新しい金融商品向けに特殊化したシステムを構築する，といった使い方が想定される．共通性の高い部分はすでにフレームワークの内部に作りこまれている点が，基本構造のみを定めるアーキテクチャとは異なる．特殊化すべき部分は高温部 (hot spot) などと呼ばれ，それを特殊化するのに継承によるサブクラス化などの技法が用いられる．

設計パターン　アーキテクチャよりは小さい単位で，繰り返し現れる設計上のパターンを設計パターンという．パターンを集めたライブラリを活用することにより，質の高い設計を効果的に行うことができる．とくに有名なのが，Gamma 等による 23 の設計パターンを集め詳しく解説した書物 [43] である．これ以降，多くの設計パターンの収集・公開活動が続けられて来ているだけでなく，より上流工程の分析パターンや下流工程のプログラミング・レベルのパターン (定石集のようなもの）も改めて注目されるようになった．

Gamma 等のグループが設計におけるパターンという考え方の提唱者として崇拝するのが，建築家の Christopher Alexander である．今や，Alexander は計算機科学界でもっとも著名な建築家といえよう．その著書 “A Pattern Language”[7] や “The Timeless Way of Building”[6] などが大きな影響を与えている．本節のアーキテクチャという主題に対し，Alexander が建築家 (architect) であるのは示唆的である．

11.1.5　酒屋問題のアーキテクチャ

Z による酒屋問題の仕様記述のところ (10.5 節) で述べたように，この問題の本質は，酒の入庫の情報はコンテナ単位でくるのに対し，酒の注文は品目単位でくるので，その間を仲介する仕組みが必要なところにある．1 つのコンテナには複数の品目の酒があり，1 つの品目の酒は一般に複数のコンテナ内に存在する．アーキテクチャの判断の中心は，この仕組みをどう実現するかといういうことになろう．

考えられる方法は次の 3 つである．

1. コンテナ単位にデータを保持し，品目単位のデータは必要に応じてそこから抽出する．
2. 品目単位にデータを保持し，コンテナが入庫するたびにそのデータを品目単位に変換する．
3. コンテナ単位と品目単位の両者のデータを保持する．

この内，2 の品目単位のデータのみを保持する方法は，コンテナが空になったことを検出して搬出指示を出すという要求に対応しにくい．少なくともコンテナごとの酒の総本数というデータを保持し，品目単位のデータから出庫指示で酒の本数を減らすたびに，対応するコンテナの総本数を減らすという処理が必要になろう．そこで，以下ではコンテナ単位のデータのみを保持する方式と，コンテナ単位と品目単位の両者のデータを保持する方式とを検討する．

両者の違いは，データか手続きかという典型的な設計上の選択の問題に帰着する．たとえば n 番目のフィボナッチ (Fibonacci) 数を求めたいとする．フィボナッチ数列とは，最初の 2 項が $F_0 = 0, F_1 = 1$ で始まり，反復公式

$$F_n = F_{n-2} + F_{n-1}$$

で定まる数列である．すなわち，0, 1, 1, 2, 3, 5, 8, 13, $\cdots$ のように続く．整数 n を読み込んで，フィボナッチ数列の第 n 項を求めるプログラム F(n) は，上の定義式を用いて手続きとして書くことができる．もちろん，この再帰式をそのまま使って再帰的なプログラムにすると恐ろしく効率は悪いが，単純な繰り返し計算に変えれば $O(n)$ のオーダーで計算できる．工夫すれば $O(\log n)$ のオーダーにすることもできる．

しかし，もし一定の n の範囲で頻繁にフィボナッチ数を計算する必要がある場合には，$n \to F_n$ の対応表をデータとして持てば，F(n) を得る手間は $O(1)$ ですむ．あらかじめ数表を作って保存しておく手間が十分にもとをとれるほど，範囲内の種々の n に対して繰り返し F(n) を求める必要がある場合には，このように手続き方式の替わりにデータ方式をとる方がよい．逆にデータ方式よりも手続き方式の方が記憶域が少なくてすみ柔軟性も高いという点で，優れている場合も多い．

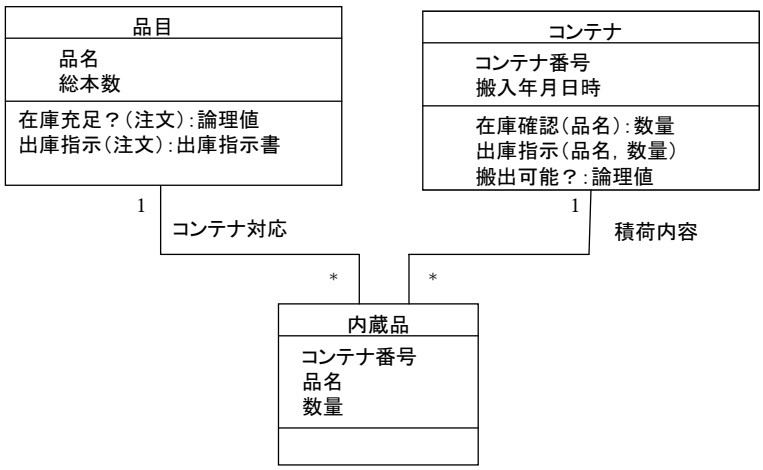


図 11.4: 品目クラスの導入

コンテナのデータのみを保持する方式をアーキテクチャ1，略して A1 と呼び，コンテナと品目の両方のデータを保持する方式をアーキテクチャ3，略して A3 と呼ぶことにする．A1 と A3 の違いは，品目単位の注文に対し在庫を判断し，在庫があれば出庫指示書を作り，在庫がなければ顧客に在庫不足を連絡し，その注文を在庫不足リストに付け加える，というシステム要求に対し，在庫情報をコンテナデータに手続きを適用して求めるか，品目データから直接求めるかの違いであるから，この点について A1 を手続き方式，A3 をデータ方式と見ることができる．

ここでコンテナデータから品目単位の情報を得る手続きは，それほど単純ではない．注文に応ずる機能を実現するためには，コンテナを順に見て，注文の品目の在庫を洗い出すという反復計算が必要になる．しかも，単純に考えると，在庫があるかないかを確認するために 1 回の反復計算をし，さらに在庫がある場合は出庫指示書を作成するためにもう 1 回反復計算をすることになる．

情報処理学会でこの酒屋問題を共通問題として種々の設計技法による解法が競作された際，対象をなるべく自然にモデル化しようとする方式で作られた解は，A1 の形をとるものが多かった．問題領域に直接現れる実体はコンテナで，品目単位にそれを含むコンテナ番号とその数量を持つようなものは，少なくとも物理的実体としては存在しないからである．実世界を素直に写したモデルでは，具体的な計算レベルの詳細は隠蔽されるはずであるのに，A1 のアーキテクチャをとったために，コンテナは品目ごとに並べられた酒瓶の列 (sequence) であり，注文に応じてそれらを探索する繰り返しを行う，というような低レベルな記述をモデルに持ちこまざるをえなくなるというやや皮肉な結果が見られた．

一方で，A3 を選択したものは，7.3.9 節に紹介した大野氏の解のように，その設計方針の選択理由の説明が天下り的なものが多かった．品目に相当する実体が問題記述から自然に見つかるという議論には無理があり，やはりここで要求モデルからアーキテクチャ設計に移るという意識的な切り替えが必要であろう．

品目クラスを導入しそのコンテナとの関係を表すと，図 11.4 のようになろう．

倉庫クラスのメソッド

- 在庫あり？(注文)：論理値
- 出庫指示（注文)：

を注文に応じた処理の API として考えれば，品目クラスのメソッド

- 在庫あり？(数量)：論理値
- 出庫指示（数量)：

に委譲する形で実現されていようが，コンテナを検索する手続きとして実現されていようが，構わないともいえる．すなわちこれまでの話の流れでは，品目というクラスを導入するのは，それをデータとして実現するアーキテクチャの選択によるものということになるが，これを手続きかデータかという判断は後回しにして，要求モデ

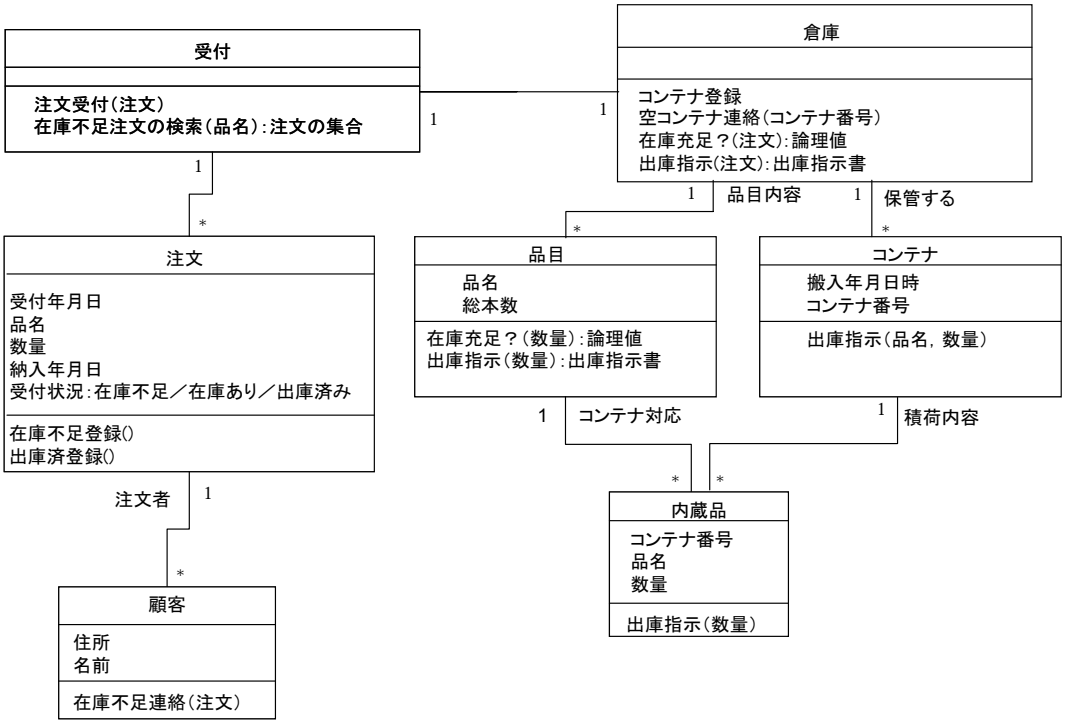


図 11.5: 酒屋問題のクラス図：品目クラス導入版

ルの記述を簡潔にする方法と見なすこともできる．このような考察をするのが，ちょうど要求分析とアーキテクチャ設計の境目にあって，面白いところである．

図 11.4 では，品目とコンテナがそれぞれより小さい単位の「内蔵品」というクラスと，1 対多の関係をもつものとして関連づけられている．これは品目とコンテナを直接関連づけると多対多の関係になるので，それを避けるための典型的な手法と見ることもできる．そもそも入荷のコンテナ単位の情報を出荷では品目単位に見なければならないという元の問題そのものが，データモデルの方ではしばしば現れる多対多の関係の解消問題なのだと言ってしまえば，とくに目新しい話ではないともいえる．データベース上の実現方法としては，コンテナのデータに対し，品目を第 2 検索キー (secondary index) すればよいということになるかもしれない．

そのようによく知られた問題のパターンであるはあるが，要求モデルからアーキテクチャを考える際に現れた設計判断として意識的に考慮し，またその結果を文書化することは重要である．最後に，品目クラスを導入した酒屋問題のクラス図を図 11.5 に (図 9.3 に対応するもの)，それに応じた系列図を図 11.6 に (図 9.4 に対応するもの) を示しておく．

11.2 アルゴリズムの設計

アーキテクチャ設計の次には，それより詳細なレベルの設計に移る．アーキテクチャはモジュールによって構成されるから，それらのモジュールをいかに設計するかという問題となる．

モジュールの基本的な切り分けとそれぞれの機能や振舞いは，アーキテクチャでほぼ定められている．そこでモジュールの設計は，たとえば次のような作業となる．

1. 既存のモジュールを再利用する．とくにそのまま利用できる形で，商用化されあるいは自由に入手できるものとして流通しているモジュールを，コンポーネントという．必要な機能と振舞いを持つコンポーネントを探し選ぶことは，通常の設計作業のイメージとは異なるかもしれないが，だからといって簡単な作業というわけではない．さらに多くの場合，入手したコンポーネントがそのまま使えるとは限らず，適応化のためのなんらかの設定や変換を必要とする．

2. 新たなモジュールを自分で設計する．多くの場合，アーキテクチャから定まる機能や振舞いを個々のモジュー

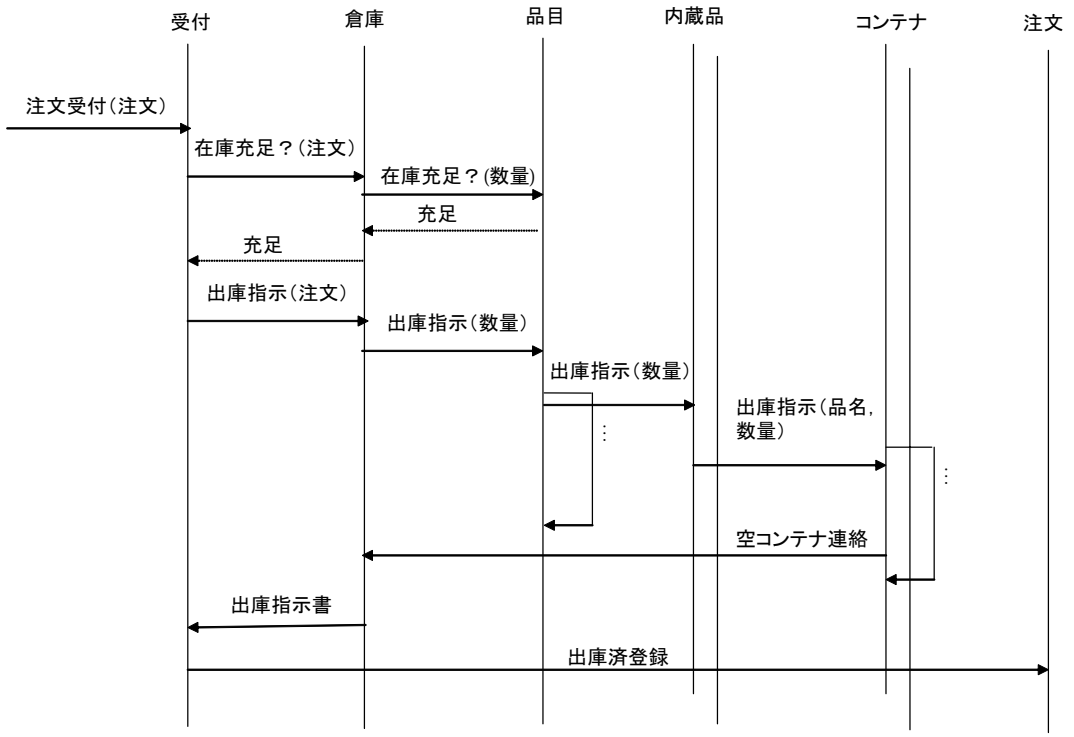


図 11.6: 注文受付 (在庫ありの場合) の系列図：品目クラス導入版

ルとして実現する設計は，さほど技術的に困難というわけではない．しかし，規模が大きくなるとその作業量だけでも膨大となり，さらにモジュール同士のインターフェースが煩雑となるため，その設計に多くの労力を費やすことになる．もともとのモジュールの切り分けをやり直す必要も頻繁に生じる．

3. データ構造やアルゴリズムを設計する．上記 2 のケースも，単純とはいえデータ構造やアルゴリズムを設計していることにはなるが，多くの場合はそれほど高度なものを必要としない．また，ある程度高度なデータ構造やアルゴリズムを必要とする場合でも，すでによく知られ教科書やハンドブックにまとめられているものを利用すればよいことが，ほとんどである．しかし，技術的に高い新規性を求められるソフトウェアの設計においては，新たなデータ構造やアルゴリズムを工夫する必要があることがある．また，既知のデータ構造やアルゴリズムを用いる場合でも，その適用に工夫が求められることは多い．

1970 年代のソフトウェア工学では，データ構造やアルゴリズムをいかに設計するかは重要なテーマの 1 つであった．その後，ソフトウェア工学の専門分化が進み，データ構造やアルゴリズムという分野は独立なものとして別れていった．ソフトウェア工学側からいえば，プログラムレベルより上位の分析設計や管理に重点を移したといえよう．

しかし，アルゴリズムやデータ構造の設計は，それ自身がソフトウェア設計の重要な一部であるだけでなく，より上位のアーキテクチャレベルの設計を考える上でも，実りの多い概念や手法を与えるものである．その意味で，ソフトウェア工学がこのレベルの設計を軽視するとすれば，不幸なことといわなければならない．幸い，アルゴリズムとデータ構造というテーマについては，すでに多くの優れた本が書かれている（たとえば Aho, Hopcroft & Ullman[4], 石畑 [54] など）．体系的な叙述はそれらの教科書に譲るとして，ここでは反復型や再帰型のプログラムを構成する方法を，いくつかの例で見ることにする．

第 10 章で扱った形式的方法は，仕様を形式的に記述するものであるから，その仕様から形式的な手法でプログラムを導出することが原理的には可能である．それを完全に自動化できたとすると自動プログラミングとなる．自動化ではなく人間が行なうとすれば，それを設計作業とみなすこともできる．その際にも，自動化と方向が重なる系統的なアルゴリズム/プログラムの導出が考えられる．このように形式仕様と結びついたデータ構造やアルゴリズムの設計は，最初から高い信頼性を確保するものといえる．また，第 12 章で扱う検証技術とも関連性が強く，設計技術と検証技術が融合することで高品質が期待できる．

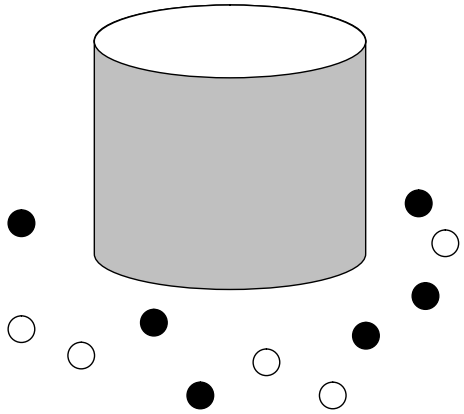


図 11.7: コーヒー缶の問題

以下の主な題材は，D. Gries[44] と J. Bentley[16, 17] からとっている．

11.2.1 不変条件

繰り返し（あるいは再帰）型のプログラムの設計では，次のように考えることが重要である．

1. プログラムの事前条件 P と事後条件 Q を明確にする．
2. 繰り返しの中で，変わらない性質（不変条件 (invariant condition)）I を定める．意味のある不変条件を発見する手段の一つは，事後条件を適当に緩和することである．
3. 繰り返し構造に入る前の事前条件によって，不変条件が満たされなければならない．すなわち，$P \supset I$ が成立することを確かめる．通常は，繰り返しに入る前の初期設定でこの条件を満たすようにする．場合によっては，不変条件の方を変更する．
4. 繰り返しを続ける条件 R を明確にする．
5. 繰り返しを脱けたときに成り立つ条件 $I \wedge \neg R$ から，求めるべき事後条件が導かれることを確認する．すなわち，$I \wedge \neg \supset Q$．あるいはむしろそうなるように，I と R を定める．
6. 繰り返し処理の中で，繰り返しの脱出条件が成立する方向（それは通常また，事後条件が成立する方向）へ向かっていることを確認する．それにより繰り返しが終了することが保証される．

不変条件ということの意味を考えさせるのが，次の問題である．

[問題 1] コーヒー缶の問題
缶に白い豆と黒い豆が入っている．缶から 2 つの豆をランダムに取り出す．2 つが同じ色ならそれを捨て，代わりに黒い豆を 1 つ戻す（外には黒い豆が沢山あるとする）．2 つが違う色なら，白い豆を戻し，黒い豆は捨てる．
これを繰り返すと，缶には最終的に 1 つの豆が残る．その豆の色が黒か白かについて，何がいえるか．

解答を見る前に，しばらく考えてほしい．不変条件を考えるということが，ヒントになっている．

解答
1 回の操作で缶の豆は 1 つずつ減る．その時，白の豆の数，黒の豆の数はどう変化するだろうか．問題の記述にしたがって場合に分けてみると，

取り出した 2 つの豆	戻す豆	白の数	黒の数
白白	黒	-2	+1
黒黒	黒	± 0	-1
白黒	白	± 0	-1

すなわち，黒の数は毎回 1 ずつ増減するが，白の数は 2 減るか変わらないかである．そこで，缶の中の白豆の数の偶奇性 (parity) は，不変であることが判る．つまり白の数が最初に偶数なら常に偶数，最初に奇数なら常に奇数という訳である．したがって，白豆の数が最初に偶数だったら最後に残るのは黒，奇数だったら最後に残るのは白になる．

次に，プログラムの問題である．

[問題 2] 最長の台の長さ

整数の配列が与えられ，その数列が単調非減少であるとする．すなわち，配列を a，その大きさを n としたとき，

$$a[0] \leq a[1] \leq \cdots \leq a[n-1]$$

台とは，同じ値をもつ数の列をいうものとする．このとき，最も長い台の長さを求めよ．ただし，$n > 0$ である．

なお，数列が単調非減少に並んでいるという条件を外したらどうなるか．

考え方

以下，Gries による．

プログラムの結果得られる最大の台の長さを p としたとき，出力条件をきちんと書くとどうなるか？

$$\begin{aligned} R: \ & (\exists\, k : 0 \leq k < n-p : a[k] = a[k+p-1]) \wedge \\ & (\forall\, k : 0 \leq k < n-p-1 : a[k] \neq a[k+p]) \end{aligned}$$

これは，配列 a に長さ p の台はあるが $p+1$ の台はないことを表したものである．

このプログラムを書くのに，繰り返しが必要なことはほぼ自明．そこで不変表明をどう選ぶか．1 つの常套手段は，出力条件 R を弱めることである．そのまた常套手段は，問題の大きさを決めている定数を変数に変えることである．この場合，区間 $[0 : n-1]$ の最大の台の長さが p という代わりに，変数 $i : 1 \leq i \leq n$ を導入して，区間 $[0 : i-1]$ に対し最大の台の長さが p であるとする．

初期値は $p = 1;\ i = 1$ とすればよく，繰り返しの終了条件は $i = n$．繰り返しで $n - i$ が単調に減少するようにすれば，停止が保証される．

不変条件

$$\begin{aligned} P: \ & 1 \leq i \leq n \wedge \\ & p \text{ は } a[0 : i-1] \text{ の最長の台の長さ} \end{aligned}$$

を満たすようにループを書くと次のようになる．

```
i=1; p=1;
while (i!=n)
  if (a[i]==a[i-p]) {i++; p++;}
  else i++;
```

11.2.2 最大公約数

最大公約数の問題は形式的仕様記述の例としてすでに述べたが，ここではその仕様から不変条件に基づくプログラムを導出する例として，再び取り上げる．

[問題 3] 最大公約数

2 つの自然数 x, y が与えられ，ユークリッドの互除法により最大公約数を求める関数 $gcd(x, y)$ を作る．

1. 事前条件： $x = \alpha \wedge y = \beta \wedge \alpha > 0 \wedge \beta > 0$
2. 事後条件： 返り値 $= gcd(\alpha, \beta)$

不変条件は，次のようにとるのが自然.

$$x > 0 \wedge y > 0 \wedge gcd(x, y) = gcd(\alpha, \beta)$$

［方法１］減算を用いる．この時，次の公理を用いる．ただし以下で，$x > 0, y > 0$.

$$\begin{aligned} gcd(x,y) &= gcd(x,y-x) = gcd(x-y,y) \\ gcd(x,x) &= x \\ gcd(x,y) &= gcd(y,x) \end{aligned}$$

プログラム例

```
/* gcd by subtractions */
gcdSub(int x, int y) {
  while (x != y) {
    if (x > y) x = x-y;
    else       y = y-x;
  }
  return x;
}
```

［方法２］除算を用いる．用いる公理は

$$\begin{aligned} gcd(x,y) &= gcd(x,y\%x) = gcd(x\%y,y) \\ gcd(x,0) &= gcd(0,x) = x \end{aligned}$$

この場合，不変条件は次のように微妙に変わる．

$$x \geq 0 \wedge y > 0 \wedge gcd(x,y) = gcd(\alpha,\beta)$$

プログラム例

```
/* gcd by divisions */
gcdDiv(int x, int y) {
  int t;
  while (x!=0) {
    t = x;
    x = y%x;
    y = t;
  }
  return y;
}
```

11.2.3 探索

[問題 4] 2 分探索

数が昇順に並んだ配列 x と，任意の数 q が与えられて，q が配列の中にあればその位置，なければ -1 を返す．

問題をもう少し厳密に定義する．

1. 事前条件： 大きさ n の整数配列 x, 整数 q が所与．

$$n \geq 0 \wedge x[0] \leq x[1] \leq \cdots \leq x[n-1]$$

2. 事後条件： 返り値を p として，x の中に q があれば $x[p] = q$，なければ $p = -1$

手法としては，2 分探索 (binary search) を用いる．この問題の，Bentley による解法（特に不変条件の概念を用いたアルゴリズムの導出の説明）は大体次のとおり．

1. 最初のスケッチ

   区間を $0:n-1$ に初期化
   以下の繰り返し
   { 不変条件：あるとすれば区間の中 }
   もし区間が空なら
   見つからなかったとして終了
   区間の真中を求め m とする
   m を使って区間を縮める
   もし区間縮小に際して q がみつかれば，その位置を返す

2. 区間を具体化

```
l=0; u=n-1;
以下の繰り返し
    { 不変条件：あるとすれば l から u までの間 }
    もし l > u なら
        p=-1 として繰り返しをぬける
    m=(l+u)/2
    m を使って区間 l:u を縮める
        もし区間縮小に際して q がみつかれば，その位置を記録して繰り返しをぬける
```

3. q と $x[m]$ の比較によるケース分け

```
l=0; u=n-1;
以下の繰り返し
    { 不変条件：あるとすれば l から u までの間 }
    もし l > u なら
        p=-1; break
    m=(l+u)/2
    case
        x[m] < t:  l = m+1
        x[m] == t:  p = m; break
        x[m] > t:  u = m-1
```

Knuth によれば，2 分探索の最初のアルゴリズムは 1946 年に発表され一般に知られていたが，データ数 N が 2^n-1 ではない一般の場合に適用可能で完全に正しいアルゴリズムは，長い間発表されなかったという．最初にきちんとした形でアルゴリズムが印刷されたのは 1960 年で，D. H. Lehmer によるだそうだ．それだけこのアルゴリズムは間違いやすいということだろう．そのたたりが残っているのか，これを題材に使った有沢氏の本 [10] でもプログラムに虫があるし，筆者自身もこれを講義に使ってこのように虫のないプログラムができますとスライドで説明して，後でそのスライドに虫のあることを発見した経験がある．

[問題 5] 2 次元探索

$m \times n$ の 2 次元の数値配列 a と，任意の数 x が与えられて，問題 4 と同様に x の配列中の位置を返す．簡単のため，x は必ず a の中にあるとする．さらに，a は列方向と行方向にそれぞれ昇順に並べられているものとする．

1. 事前条件：
   任意の $k \in [0:m-1]$ に対し：
   $a[k,0] < a[k,1] < \cdots < a[k,n-1]$
   任意の $k \in [0:n-1]$ に対し：
   $a[0,k] < a[1,k] < \cdots < a[m-1,k]$
   $x \in a[0:m-1,0:n-1]$

2. 事後条件：
   $0 \leq i < m \wedge 0 \leq j < n \wedge x = a[i,j]$

不変条件として，次式を考える．

$$0 \leq i \leq p < m \wedge 0 \leq q \leq j < n \wedge x \in a[i:p, q:j]$$

つまり，2 分探索と同様に x があるとしたら，縦は $i:p$，横は $q:j$ の矩形の範囲内にある，ということを不変条件とする．

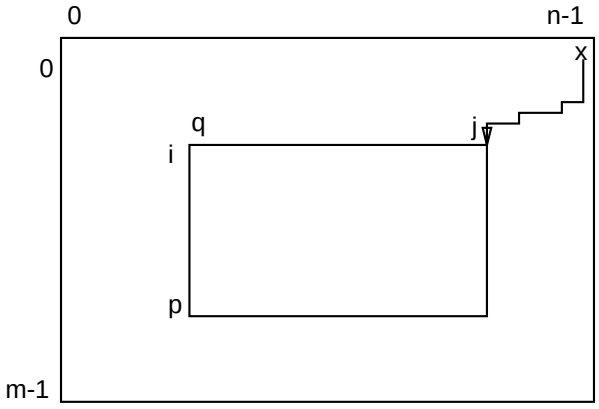


図 11.8: 2 次元（鞍の背）探索

これからプログラムは次のような繰り返しで書けそうである．

```
i=0; p=m-1; q=0; j=n-1;
while (a[i,j]!=x)
  if      (a[i,j] < x) i++;
  else if (a[p,q] > x) p--;
  else if (a[p,q] < x) q++;
  else if (a[i,j] > x) j--;
```

止まる条件を考えると，上の 4 ケースの内，1 番目と 4 番目のみを考えれば十分なことが分かる．したがって，だいたい次のようなプログラムになる．

```
i=0; j=n-1;
while (a[i,j]!=x)
  if (a[i,j] < x) i++;
  else            j--;
```

このアルゴリズムによる 2 次元探索は，図 11.8 のように進む．横方向は低い方を目指し，縦方向は高い方を目指して進むので,「鞍の背」探索と呼ばれる．

[練習問題] 共通人物の検出

1. 2 ファイルからの共通人物の検出
   2 本のファイルにそれぞれ人の名前が五十音順に並んでいる．両方のファイルに出てくる名前を少なくとも一つ見つけて出力する．

2. 3 ファイルからの共通人物の検出
   上と同じ問題を，ファイルが 3 本として考える．

いずれも，問題の構造を分かりやすくするために，ファイルの代わりに配列を用い，名前でなく数字が入っているとしてもよい．

なお，類似のものとして， Dijkstra & Feijen[35], 野木 [79], 有沢 [10] などが関連する題材を豊富に提供している．

第12章　検証技術

12.1　検証の基本概念

検証とは，ソフトウェアが要求される品質を満たし，信頼できることを確かめるための作業をいう．

12.1.1　用語

検証に関する用語は数多く，また混乱している．米国では Verification & Validation，略して V&V という用語がよく用いられる．しかし，Verification と Validation がどう違うのかについて，必ずしも共通な理解があるわけではない．比較的広く受け入れられているのは，前者がソフトウェア開発の各工程ごとに品質を確かめる作業であるのに対し，後者は全工程に対して，とくに最終製品に対して品質を確かめる作業であるというものである．

これをやや異なる観点から，Verification はプログラムが個々の要求仕様を正しく満たしていることを示し，Validation は製品として顧客の期待するものになっているかどうかを検査するものである，と説明されることもある．B. Boehm によれば，前者は「正しく製品を作っているか」，後者は「正しい製品を作っているか」，という違いだという．

この Verification にも Validation にも，また結合形としての V&V にも日本語の定訳がない．ここでは V&V をまとめて検証と呼ぶことにする．

いずれにせよ，検証はソフトウェアの品質を確保することを目的としている．ソフトウェア工学の分野ではおもに検証という言葉を用いるが，品質管理の分野では同じことを品質保証活動という．ただ，品質保証という言い方には，利用者の満足度を高めるという目標がとくに強く意識されている．

また，検証が検出すべき誤りに関する用語も混乱している．システムの構成要素がその機能を失うことを，通常，fault（故障）といい，それが error （誤り）として顕在化し，その誤りが訂正されないと failure（障害）となる，という用語の使い分けが，かなり一般的ではある（「岩波情報科学辞典」）．しかし，JIS では fault に障害という訳語を当て，逆に failure に故障という訳語を当てている．他に，欠陥 (defect) や機能不全 (malfunction)，異常 (anomaly) などの語も，若干のニュアンスの違いを持ちながら，それぞれ故障や障害とほぼ同義に用いられる．

12.1.2　要求と検証

検証とは，ソフトウェアが要求されている品質を満たすことを確認する作業である．したがって，検証技術は要求技術と密接な関係にあり，検証の計画は要求の確定段階で要求の内容に応じて定められることが望ましい．

要求を機能要求と非機能要求に分けるという考え方が，一般的である（3.3.3 節参照)．機能要求は，ソフトウェアがどのように動作しどのような仕事を果たすべきかを記述するものである．したがって機能要求の検証は，実装されたシステムの動作が要求された機能を実現するものであることを実証することである．テストを代表とする古くからの検証技術の多くは，主に機能要求の検証に用いられてきた．

非機能要求には，実行速度，必要とする記憶容量，セキュリティ，インタフェース，使いやすさ，移植性，保守性など，多様なものが含まれる．これらの一部は，テストでも確かめることができるが，機能検証のためのテストとは用意すべきデータや評価の方法が自ずから異なってくる．実行速度や記憶容量といった性能の検証についてはシミュレーション技術が，またセキュリティについてはモデル検査などの形式検証技術が有効な場合もある．

12.1.3　さまざまな検証技術

検証は何を相手にするのかによって，要求や設計の仕様を対象にするものとプログラムを対象にするものとに分けられる．

仕様の検証　仕様を対象とする検証の方法としては，仕様記述が形式言語で書かれている場合は，なんらかの（自動的な）解析方法が考えられる．とくに，振舞いに関する仕様については，その安全性（デッドロックなどの望ましくない状態に陥らないこと）や活性（望ましい振舞いは必ずいつかは実現すること）の検証にモデル検査が有効で，ハードウェアの論理回路や通信プロトコルの検証だけでなく，一般のソフトウェアに対しても適用がひろがりつつある．

より実践的な方法としては，チームによる見直し (review)（徒歩検査 (walkthrough)，査閲 (inspection) など多少のニュアンスの違う用語と方法があるが，人間の目によるチェックに基づくという点は共通）が，多くの企業で実施されている．

プログラムの検証　プログラムに対する検証の方法も，いろいろある．これを動的検証と静的検証 とに分けて考えてみよう．

動的検証とは，プログラムを実際に実行して行う検証である．テストは，その代表的な方法の１つである．他には，モジュール別や原始プログラムの実行文別にみた実行回数の頻度分布（これをプロフィールという）解析や，実行済みの文の比率（これを網羅度という）の計測，プログラム中への表明（そこで成立すべき論理条件）の挿入によるチェック，などがある．

静的検証とは，プログラムを実行せずにその特性を解析するものである．コンパイラが行う構文エラーの検査の延長として，静的な型検査がある．型検査の技術はとくに関数型言語を対象に研究が進展し，現在では一般の多くの言語のコンパイル時に，強力な検査が可能となっている．さらにプログラムの静的解析技術では，原始プログラムの制御の流れやデータの流れを解析し，それに基づいてプログラムの構造や意味の抽出，あるいは不適切と思われる部分の指摘を行う．実際，種々の言語を対象とする静的解析ツールが実用に供され，利用されている．別の静的検証方法として Hoare 論理などを用いた正当性証明技法がある．これはプログラムが形式的に記述された仕様と一致することを数学的手段で証明するものである．また仕様の検証の一方法として挙げたモデル検査を，プログラムの検証に直接用いることも研究レベルではさまざまな取り組みがある．

以上をまとめて図示したのが，図 12.1 である．

12.2　プログラムの検証技術

プログラムの検証技術として，テスト技術を中心に紹介する．テストはソフトウェアの品質を検査・検証する方法の１つで，検証技術の中でももっとも古くから実践されてきたものでありながら，現在でもその中心的な役割を譲っていない．

12.2.1　テストの基本的な性質

テストは，対象となるプログラムを一定の条件下で実行し，結果を分析する．その意味で動的な技術であり，プログラムのコードを解析して品質を検証する静的技術と対照的な位置を占める.「一定の条件下」とは，テストデータを選択し実行環境を制御して実行することを意味する．１つのテストケースを実行する際に，その実行でプログラムがどのように振舞い，どのような結果が得られるかについてあらかじめ予想できることが重要であり，予想と実際が異なれば対象プログラムに欠陥が存在することが示されたことになる．

E. W. Dijkstra による「テストではプログラムの虫の存在は示せても，虫が存在しないことは示しえない.」という名言がある．プログラムの考えられる実行のケースは通常無限に存在するか，有限でも膨大ですべてを網羅することは実質的に不可能である．そこでテストは本質的にサンプル調査にならざるをえない．完全なテストは，ごく例外的な場合を除いて存在しない．

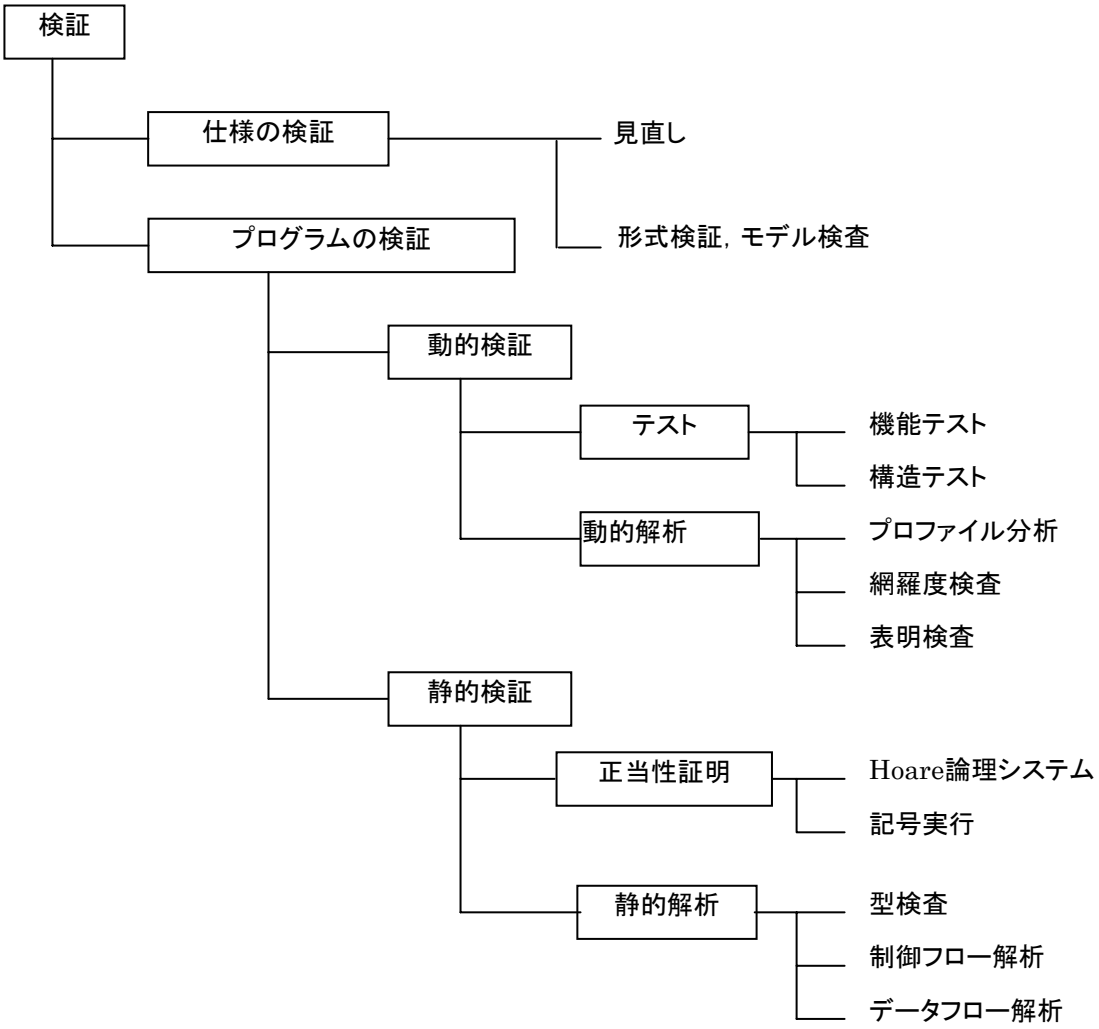


図 12.1: 検証技法の種類

もちろん，だからテストは無意味だということにはならない．テストで虫が見つかれば，その原因を取り除くことによって運用時に起こりうる障害を，少なくとも発見された問題については未然に防ぐことができる．テストで問題が起こらなければ，より信頼性は高まる．しかし，テストをどの程度行えばよいかという課題や，テストケースをどのように選択したらよいかという課題がテストにつきものである根本的な要因は，このテストが完全たりえないという性質に帰着するものである．

12.2.2　テストケースの選定

テストの技術的な課題には，次のようなものがある．

- テストケースをどう選ぶか．
- テストを実行する環境をどう作るか．
- テスト結果をどう評価するか．

テストケースとは，1 つのテストの実行を定める条件と，その実行により想定される結果とを合わせたものである．テストの実行条件は，テストデータを決めることによって定まることが多いが，応答型のソフトウェアの場合は，外部から与えるデータだけでなくシステムの内部状態が実行条件を定める．

テストケースを選択する方法に基づいて，機能テストと構造テストを区別するという考え方がある．機能テストとは，プログラムの機能仕様のみからテストケースを決めるもので，ブラックボックステストとも呼ばれる．構造テストとは，プログラムの内部構造に依存し，その構造をなるべく隈なく網羅するようなテストケースの集合を作り出そうとする方法で，ホワイト（またはグラス）ボックステストとも呼ばれる．

機能テスト

1. **同値分割**
   入力空間をテストに対して同値な部分空間に分割して，各部分から１つのテストケースを選ぶもの．テストに対して同値とは，１つのテストデータで実行しエラーがなければ，同じ類の他のデータに対してもエラーがないことが保証されるというものである．そのイメージを図示すれば，図 12.2 のようになる．図で，

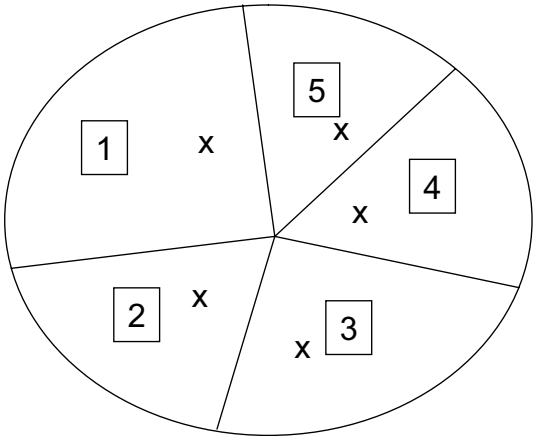


図 12.2: 同値分割のイメージ図

   1 から 5 の部分がそれぞれ同値クラスだとしたら，そこから適当な点をそれぞれ 1 点とれば（図でイメージを x で示している），適切なテストデータの組が得られる．

   しかしこれは理想的な話で，現実にはそのような意味での同値クラスは決定できないか，決定できたとしても数が多すぎて実際的ではないことが普通である．そこで，入力がある範囲で示されれば，その範囲の中と外とか，離散的な値を取れば，各離散値を１つの類にするなどの便宜的な方法が取られる．

```
/* binary search */
int bsearch(int x[], int n, int q) {
/* pre   n >= 0,
         x[0]<=x[1]<=...<=x[n-1]
   post  x[p]=q, if q is in x,
         p=-1,    otherwise,
         where p is the returned value. */
  int l, u, m;
  l = 0; u = n-1;
  while (l<=u) /* if q is in x, q is in [l:u] */ {
    m = (l+u)/2;
    if (x[m]==q) return m;
    else if (x[m]<q) l = m+1;
    else u = m-1;
  }
  return -1;
}
```

図 12.3: 2 分探索のプログラム

11.2.3 節に挙げた 2 分探索の例で考えよう．その仕様は,

(a) 事前条件： 大きさ n の整数配列 x と 整数 q が所与.

$$n \geq 0 \wedge x[0] \leq x[1] \leq \cdots \leq x[n-1]$$

(b) 事後条件： 返り値を p として，x の中に q があれば $x[p] = q$，なければ $p = -1$

であった．この仕様を満たす C によるプログラム例を図 12.3 に示す.

この問題の入力データは，n と配列 x と q である．それぞれについて自然な区間の分割を考えたのが，表 12.1 である．ここで無効同値クラスとして仕様の条件を外れるものも考慮している．これは想定外の入力に対するプログラムの頑健性を見るために重要な方法である.

表 12.1: 2 分探索の同値分割例

入力	有効同値クラス	無効同値クラス
n	1) $n > 0$ 2) $n = 0$	3) $n < 0$ 4) 配列の大きさ $< n$
x	5) $x[0] < x[1] < \cdots < x[n-1]$ 6) 少なくとも 1 つの i につき $x[i] = x[i+1]$ 7) $x[0] = x[1] = \cdots = x[n-1]$	8) ある i に対し $x[i] > x[i+1]$
q	9) x に q が含まれる 10) x に q が含まれない	

同値クラスという意味では，これら 3 種類のデータのクラスの組合せになる．すべての組合せを尽くそうとするとテストケースが多くなりすぎる．データ間の関連が強く，組合せによって振舞いが大きく変わる場合を除けば，各データのクラスが少なくとも 1 つのテストケースに含まれていればよいとする方が，実際的であろう.

```
/* test 'bsearch' */
main() {
  int x[11], n, i;
  for (i = 0; i <= 10; i++) x[i] =2*i;
  for (n = 0; n <= 10; n++) {
    printf("n= %d\n", n);
    for (i = 0; i < n; i++) {
      assert(bsearch(x,n,2*i) == i);
      assert(bsearch(x,n,2*i-1) == -1);
      assert(bsearch(x,n,2*i+1) == -1);
    }
    assert(bsearch(x,n,-2) == -1);
    assert(bsearch(x,n,2*n) == -1);
  }
}
int assert(int a) {
  if (!a) printf("Assertion failed!\n");
}
```

図 12.4: 2 分探索のテストプログラム

2. **限界値分析**
   入力空間を同値分割の場合と同様に何らかの基準で部分空間に分割し，それらの部分空間の境界上やその近傍からテストデータを選ぶ．また，出力空間にも注目し，その境界上の結果を出すようなテストケースを工夫する．エラーが起こりやすいのは例外的な事象が起きた場合で，それは入力部分空間の境界上やその周辺で生じることが多いという知見に基づく．また，正常な入力値の外にあるデータも意識的に選ぶことにより，プログラムの頑健さを検査するという目的にも対応している．

   同値分割で取り上げた 2 分探索の例を用いると，表 12.1 の内，区間として与えられているものに対し，たとえば次のように境界付近のものを選ぶことになる．

   1) $n > 0$ の内，$n = 1, n = 2$ など

   3) $n < 0$ の内，$n = -1$ など

   4) 配列の大きさ $= n + 1$ など

   9) $q = x[1], q = x[n]$ など

   10) $q < x[1], q > x[n]$, ある i に対し $x[i] < q < x[i + 1]$ など

   図 12.4 の Bentley による上の 2 分探索プログラムのテストに手を加えたものである．ここにはこのような境界上のテストケースがかなり含まれている．

3. **原因結果グラフ**
   原因として異なる入力条件を，結果として異なる出力条件を洗いだし，それらの間の論理的な関係を，一定の規約にしたがったグラフ（原因結果グラフ）で表す．原因結果グラフは決定表に変換することができ，その決定表から具体的なテストデータを生成する．これは入力と出力の組み合わせ複雑で，しかも場合わけが爆発的に多くならないようなプログラムのテストに有効である．たとえば状態遷移モデルで表されるような対象が，この方法に向いている．

   例として状態遷移モデルで取り上げたボーリングの点数計算（8.2 節参照) を考えてみよう．原因となるのは次のような事象である．

- 前のフレームの結果：{ オープン，スペア，ストライク，ダブル }
- 現フレームの何投目か：{1 投目，2 投目 }
- 倒れたピンの数：{ 全倒，残 }

結果となるのは次のような事象である．

- 点数の計算法：{ 倒したピン数の加算，倒したピン数の 2 倍の加算，倒したピン数の 3 倍の加算 }

原因結果グラフとは，原因の各項目を頂点にし，それらの論理的結合を and 結合の辺や or 結合の辺で表し，それと結果の頂点とを結びつけるものである．このボーリング点数計算の原因結果を表したのが，図 12.5 である．

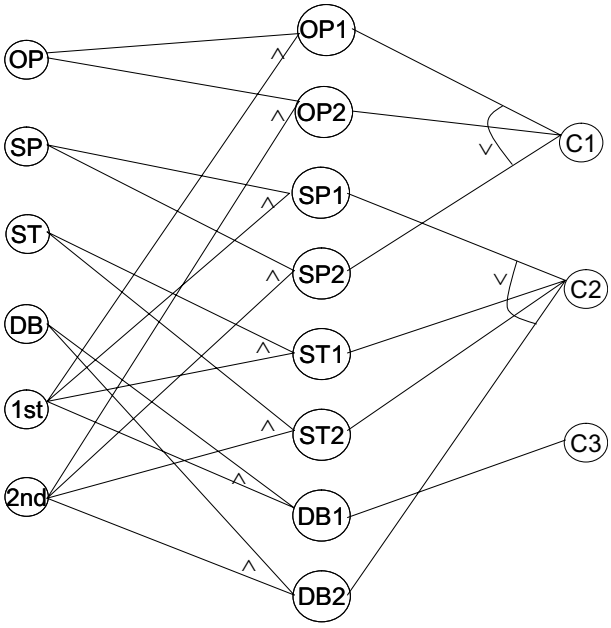


図 12.5: 原因結果グラフの例（ボーリングの点数計算）

図で頂点の OP, SP, ST, DB はそれぞれ原因のオープン，スペア，ストライク，ダブルに対応し，1st, 2nd はそれぞれ原因の 1 投目，2 投目に対応する．全部倒れたか残ったかという原因事象は点数の結果事象に直接影響しないので，グラフから省かれている．また，頂点の C1, C2, C3 はそれぞれ結果の，倒したピン数の加算，倒したピン数の 2 倍の加算，倒したピン数の 3 倍の加算という事象に対応する．

図の頂点に入る辺の集合に対し，それらが and で結合される場合は ∧ 記号を，or で結合される場合は ∨ 記号を付けている．論理記号として他に，否定，排他的選言，なども使えることになっているが，ここでは必要ないので説明を省略する．そもそもこの問題の場合には，前のフレームの 4 状態と現フレームの何投目かという 2 状態の単純な積で原因の場合が尽くされるので，わざわざ原因結果グラフを描くほどのこともないかもしれない．

結果の事象を点数の 3 種類の加算法の区別としたが，次にどの状態に移るかということも，別の結果事象である．これを考える際には，先ほど無視した倒したピンの数（とくに全部倒れたか，それとも 1 本以上残ったか）という事象を，原因事象に加える必要がある．そうなると原因の場合数が大幅に増えるが，それをなるべく小さな状態数にまとめたのが，実は図 8.2 の状態遷移図である (同図を図 12.6 として再掲する）．そこでは前フレームの結果と現在何投目かで状態を定め,「全倒」か「残」かで遷移事象を定めていた．また，フレームの 1 投目でピンがすべて倒れれば 2 投目はないこと，2 投目がある場合，その点数計算はオープンの後とスペアの後は同じであり，ストライクの後とダブルの後でも同じであることを利用して，状態の数を減らす工夫をしていた．

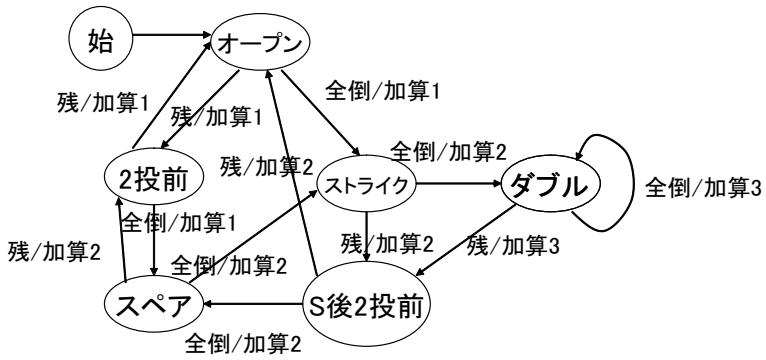


図 12.6: ボーリングの点数計算モデル (再掲)

表 12.2: ボーリング問題のテストのための決定表

状態	オープン	1 1 0 0 0 0 0 0 0 0 0 0
	2 投前	0 0 1 1 0 0 0 0 0 0 0 0
	スペア	0 0 0 0 1 1 0 0 0 0 0 0
	ストライク	0 0 0 0 0 0 1 1 0 0 0 0
	S 後 2 投前	0 0 0 0 0 0 0 0 1 1 0 0
	ダブル	0 0 0 0 0 0 0 0 0 0 1 1
遷移事象	全倒	1 0 1 0 1 0 1 0 1 0 1 0
	残	0 1 0 1 0 1 0 1 0 1 0 1
結果	加算 1	1 1 1 1 0 0 0 0 0 0 0 0
	加算 2	0 0 0 0 1 1 1 1 1 1 0 0
	加算 3	0 0 0 0 0 0 0 0 0 0 1 1
次状態	オープン	0 0 0 1 0 0 0 0 1 0 0 0
	2 投前	0 1 0 0 0 1 0 0 0 0 0 0
	スペア	0 0 1 0 0 0 0 0 1 0 0 0
	ストライク	1 0 0 0 1 0 0 0 0 0 0 0
	S 後 2 投前	0 0 0 0 0 0 0 1 0 0 0 1
	ダブル	0 0 0 0 0 0 1 0 0 0 1 0

この状態遷移図を用いて，原因，すなわち現在の状態と遷移事象と，それに対する結果，すなわち点数の加算と次の状態を対応させた決定表 (decision table) を作ってみると，表 12.2 のようになる．

ここで値が 0 と 1 からなる 12 個の列があるが，それぞれがテストケースに対応する．状態と遷移事象でテストの入力を定め，結果と次状態が期待されるテスト結果を表す．1 が対応する事象が起こること，0 が起こらないことに対応している．

4. **ユースケースに基づくテストケース**

要求分析がユースケースに基づいて記述されている場合には，それに応じたテストケースを考えることが自然である．これも仕様からテストケースを選定するので機能テストの 1 種と見なすことができる．

ユースケースは一般にいくつかのシナリオ（1 つの実行系列）からなるので，シナリオごとに少なくとも 1 つのテストケースを生成することになる．例外的なケースはそれぞれ別シナリオとして記述されているはずであるから，限界値分析のような方法を独立に取る必要はない場合もあるが，1 つのシナリオに応じた入力データ空間が広い場合は，その中でどのようなテストデータを選ぶかについて，同値空間や限界値の考え方を併用した方が望ましい場合もある．

```
int bsearch(int x[], int n, int q) {                    \\1
  int l, u, m;                                          \\2
  l = 0; u = n-1;                                       \\3
  while (l<=u) {                                        \\4
    m = (l+u)/2;                                        \\5
    if (x[m]==q)                                        \\6
      return m;                                         \\7
    else if (x[m]<q)                                    \\8
      l = m+1;                                          \\9
    else u = m-1;                                      \\10
  }                                                    \\11
  return -1;                                           \\12
}                                                      \\13
```

図 12.7: 2 分探索プログラム（図 12.3 の別表現）

構造テスト

1. **網羅度に基づくテストケース選定**
   プログラムの構造に基づいて，プログラムを構成する実行単位（実行文や分岐判定）を，なるべく広く覆うようなテストの実行を実現しようとするものが構造テストである．

   一連のテストケースによるテスト実行の累積によって，テストがどれほど達成されたかを測るテスト網羅度 (test coverage) という指標がよく用いられる．具体的な指標には，文網羅度や分岐網羅度がある．

   - **文網羅度**　全実行文の何%が少なくとも 1 度は実行されたかを示す指標である．C_0 という記号がよく使われる．
   - **分岐網羅度**　if 文，case 文，while 文などによるあらゆる実行の分岐のうち，何%が少なくとも 1 度は実行されたかを示す指標である．C_1 という記号がよく使われる．

   構造テストは，文網羅度や分岐網羅度をなるべく高くするようなテストデータを選んで実行するというのが，基本方針である．そのために現在までまだ 1 度も実行されていない文や分岐を抽出し，それを実行するようなテストデータを生成することが必要となる．そのためには実行されていない文や分岐を含む実行経路を定めて，その経路を通るようなテストデータを用意することになる．

   たとえば図 12.8 は，図 12.7 の 2 分探索プログラム（図 12.3 の注釈を省き，行番号を入れるなど若干の修正を行ったもの）のフローグラフである．フローグラフとはプログラムの 1 つづきの実行断片を頂点とし，実行制御の移動を辺で表したグラフである．制御の流れグラフの 1 種であり，フローチャート（6.1 節参照）とほぼ等価であるが，頂点を 1 種類に簡素化している．

   この例ではたとえば 3-4-5-6-8-9-11-12, 3-4-5-6-8-10-11-4-5-6-7 という 2 つの実行経路を実現するテストデータが用意できれば，文網羅度も分岐網羅度も 100%に達する．もちろん，これはプログラムが短く構造も比較的単純なために，たった 2 つのテストケースで 100%の網羅度を得たもので，実際には次のような問題が生じる．

   (a) 実行経路を指定しても，それを実際に実行するテストデータが見つからないかもしれない．一般に経路を満たす入力データは，その経路上の分岐条件の組み合わせからなる論理条件式を満たす解として求めなければならないが，その解が存在しない場合もあり，存在したとしてもその解を求めるのは一般に非線形連立不等式を解くことになるので，実際的でないことが多い．

   (b) 大きなプログラムに対し統合テストを行う場合は，実際の運用で使われるデータや，運用時の入力データの統計的な分布に従って生成されたテストデータを用いてテストし，網羅度を計測することが一般的

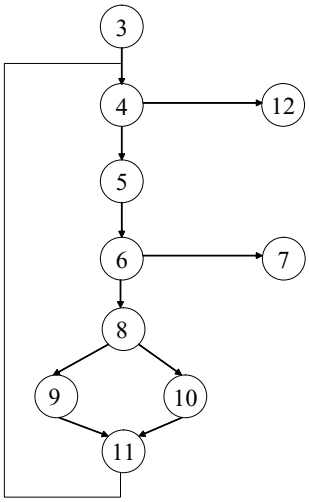


図 12.8: 2 分探索プログラムのフローグラフ

であるが，そのようなケースでは網羅度が 100%に達することはまずなく，高くても 80%から 90%程度にしか達しないことが通常である．

(c) 逆に，文網羅度や分岐網羅度が 100%になっても，テストとして十分とはいえない．実際，上の 2 分探索プログラムの例では 2 つのテストケースで 100%に達したが，どちらも繰り返しを 1 回しか行っておらず，機能テストにおける同値分割や限界値分析と比べても不十分である．一般に，プログラムの構造からなるべく網羅的なテストケースを作るには，可能なあらゆる実行経路を網羅するという**経路網羅度**を高くすることが理想的であるが，このように繰り返しのあるプログラムでは一般に経路の数は無限になる．ループの反復回数を有限範囲に限定したとしても，プログラムの制御構造が少しでも複雑になれば，反復回数を 1 とか 2 とかきわめて小さい値に限定しない限り，経路の数は爆発的に増えてしまう．

(d) 構造テストはプログラムの構造に基づいてテストケースを作るので，プログラムが仕様の一部を実現していない場合に，それを検査するようなテストケースを生成できない．

このような問題点はあるものの，テスト達成度が数値的に把握できることは有効であり，そのため網羅度を計測するツールはさまざまなものが開発され，実際に使用されている．また，実行経路から機械的にテストケースを作るという意味での構造テストは実際上難しいとしても，網羅度を上げるためのテストケースを工夫することは重要であり，テストを担当する組織ではそのためのさまざまな努力が行われている．

また，このような制御の流れに基づく網羅度を基準とするのではなく，データの値の定義と参照関係を表したデータの流れグラフに基づいて，網羅度を測る方法もある．一般にこの方が，制御の流れに基づくものよりテストケースが細かくなる．

2. **状態遷移図からのテストケース選定**

状態遷移図は一般にプログラムより抽象化されているが，対応するプログラムの振舞いを表している．状態遷移図からその遷移を網羅するようなテストケースを生成することは，プログラムのフローグラフから文網羅や分岐網羅の基準でテストケースを選定するのと同様な方法が考えられる．

再度，ボーリングの点数計算の例を使おう．図 12.6 で，すべての遷移を少なくとも 1 回は実行する経路を考えてみよう．すべての遷移をちょうど 1 回だけ実行する経路があるだろうか．それはこのグラフを一筆書きできるかという問題に帰着する．これについては有名なオイラーの定理がある．連結グラフの任意の頂点を出発点として同じ点に戻る一筆書きが存在する必要十分条件は，すべての頂点につきそこに入る辺の数とそこから出る辺の数が等しいことである．図 12.6 のグラフでは，開始点を除いた連結グラフで，このオイラーの条件が成立している．だから，たとえば次のような経路が，各遷移をちょうど 1 度だけ実行するものの例として存在する．

始 → オープン → 2 投前 → スペア → オープン → ストライク → ダブル → ダブル → S 後 2 投前 → スペア → ストライク → オープン

遷移条件のない状態遷移図の場合，任意の経路を実現する遷移列は必ず存在するので，対応するテストデータを作ることは簡単である．その点では，プログラムのフローグラフから適当な経路を選び，それを実現するテストデータを生成する方法よりも，扱いやすいといえる．

12.2.3 テスト環境と結果の評価

テストの環境の作り方は対象に依存する部分が多いが，テスト環境生成の支援ツールとしてテストベッドと呼ばれるものがあり，一部で用いられている．その主な機能は，テスト対象のモジュールを起動するためのドライバと，対象モジュールが呼び出す下位モジュールの動作を模擬するスタブとを生成し，テスト実行環境を作ることである．さらにテストの網羅度を計測するツールを組み込んだりたり，テストのロギング機能を持つなどが標準的である．

テスト結果の評価は，個々のテストケースに関するものと，テスト工程全体に関するものがある．予想されるテスト結果と実際の結果との照合機能をテスト環境に組み入れることは，比較的簡単である．それでも評価結果を保存し管理することにより，保守の際の回帰テキスト (後述) などにも応用できる．

テスト全体の評価は，テストを適切な時点で打ち切るための基準を与えるという意味で重要である．テスト網羅度のほか，テスト工程で発見されたエラーの出現履歴などに基づく種々の信頼性モデルが用いられるが，それについては項を改めて説明する．

12.2.4 テストのプロセス

ソフトウェアライフサイクル上のテストプロセス

テストの種類として，単体テスト，統合テスト，システムテストの区分をすることが一般的である．

- 単体テスト： プログラム単位としてのモジュールを，それぞれのモジュール仕様に対して独立にテストする．
- 統合テスト： 単体を結合し，ひとまとまりの処理を実行する単位にまとめてテストする．設計仕様に対するテストといえる．
- システムテスト： システムとして，要求仕様を満たしているかどうかを判定するために行うテストである．

このように，この 3 種類のテスト区分は，ソフトウェアライフサイクルの要求仕様，設計仕様，モジュール仕様を作成するプロセスの逆順に，それぞれの仕様に対して実施される (図 2.2 の V 字型ライフサイクルモデル参照)．したがって各テストの計画もその対応する仕様の作成時に作られることが望ましい．

別に保守プロセスで実行されるテストのうち，既存の機能に変更の悪影響が及んでいないかどうかをテストするものを，回帰テスト (regression testing) という．それには旧版のシステムに対して実施されたテストを再実行することになるが，そのテストケースの管理や影響範囲を考慮した選択が重要な課題となる．

12.2.5 信頼性モデル

テスト工程の進行に伴ってエラーが逐次発見され修正されて，プログラムの信頼性は向上していく．しかし，テストはエラーを完全に除去することを保証できる手段ではなく，そのためテストの終了を判断するにはある程度経験的なものに頼らざるをえない．テスト終了の基準として考えられる基準には，次のようなものがある．

1. テスト期間による基準： プロジェクト計画時に開発システムの規模や性質に応じて必要とされるテスト期間を定め，その期間が終わればテストを終了する．
2. 工数による基準： 期間による基準と類似で，計画時にテストに投入する工数を定め，実績の工数が計画値に達したらテストを終了する．

3. テスト項目数・テストケース数による基準： テスト計画時に対象システムの規模や性質に応じて必要とするテスト項目数あるいはテストケース数を定め，その数だけのテストを実施したら終了する．

4. テスト網羅度による基準： たとえば文網羅度が 90%を超えたらテストを終了する．

5. 発見エラー数による基準： たとえば原始プログラム 1000 行あたりのエラー数の見積もりを過去のデータから推定し，その数に達したらテストを終了する．

6. 発見エラー数の収束による基準： テスト工程の進行に従って発見されたエラーの累積数の状況を観測し，それが収束過程にあると判定されたらテストを終了する．

テスト期間や工数による基準は危険である．実際，次のような例を筆者は経験している．ある大企業のシステム部門では，開発プロジェクトの工程別の生産性を計測し保存している．生産性は対象システムの規模（原始プログラム行数）を工程に投入した工数で割った値を用いている．そのデータを見ると，多くのプロジェクトでテスト工程の生産性がきわめて高い．その理由は，それらのプロジェクトの進捗が予定より大幅に遅れ，一方でシステムの運用開始を後に延ばせないために，テスト工程にしわよせがきて期間が短縮され，見かけの生産性が上がった，というものであった．

この状況はテストの終了基準を期間や工数以外のものとしても解決しないかもしれないが，テストを終了した時点で対象プログラムにどの程度エラーが残存しているか見当がつかないのであれば，どこでやめても同じだという考え方を反映していることが予想される．

上に挙げた 6 つのテスト完了基準は，リストの下にいくほどより「科学的」な根拠がありそうだという順で並べている．つまり，発見数の収束状況によってテストの終了を判断するのが，もっとも正当であると思われる．エラーの収束状況を判断するには，テスト工程でのエラー発見過程をモデル化する必要がある．すなわちテストの進行を時間軸で測り（経過時間でもテスト実施の累積時間でも累積テストケース数でもよい），それに応じて発見されたエラー数の分布をモデル化するのである．このようなモデルは信頼性モデルと呼ばれる．

多くのモデルが提案されているが，もっとも基本となるのは，非斉次ポアソン過程をモデル化した次式によるものである．

$$\mu(t) = V_0(1 - exp(-\frac{\lambda_0}{V_0}t)) \tag{12.1}$$

ここで $\mu(t)$ は時間 t までに発見された累積エラー数を表し，V_0 は初めにあったエラーの総数を意味する．

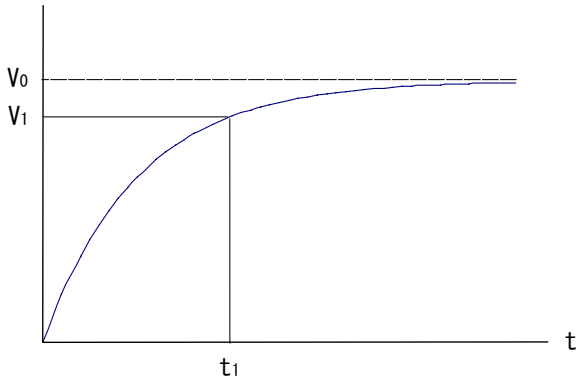


図 12.9: 非斉次ポアソン過程

図で現在時刻が t_1 でその時点での累積発見エラー数が V_1 であるとして，推定残存エラー数は $V_0 - V_1$ となる．

テスト工程の途中段階で，それまでの累積エラー数を時間に対してプロットし，モデルに当てはめてパラメタ λ_0 と V_0 を推定する．V_0 と現在までに発見されたエラー数の差により，残存エラー数が推定できる．また，これから発見すべき目標エラー数を定めれば，それに達するまでのテスト時間が推定できる．式 (12.1) を微分した形が次式である

$$\lambda(t) = \lambda_0\, exp(-\frac{\lambda_0}{V_0}t) \tag{12.2}$$

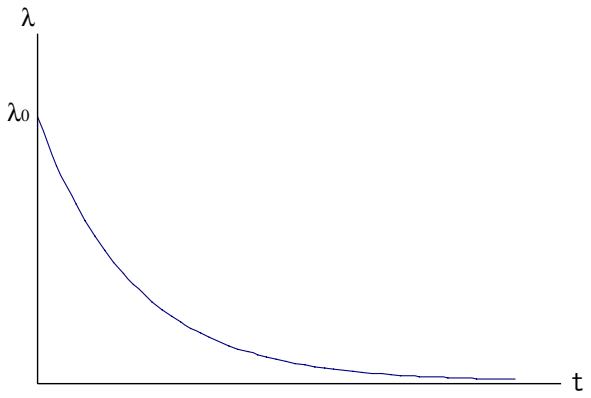


図 12.10: エラー強度

$\mu(t)$ の導関数 $\lambda(t)$ はエラー強度，すなわち時間 t にエラーが発見される確度の強さを意味する．

信頼性モデルには他にも多くの提案がある．たとえば上限値に漸近するモデルは現実的ではないとして，上に有界でない対数ポアソン過程を用いるものや，実際のデータの多くは，初期にはエラー発見率が低く，中間でエラー発見率が高くなり，後半にまたエラー発見率が低下して，信頼度曲線は S 字のカーブを描くとして，その形を実現するようなモデル，などが提案されている．また，現実にはエラーが発見されるとそれを修正するコードが挿入されるが，そこに新たなエラーが潜入するとして，総エラー数が増加するようなモデルも考案されている．

ただ，モデルをいかに精緻化しても，開発組織に適合するかどうかが問題である．簡便なモデルを用いてその使用経験を積み，テスト開始前の段階からある程度妥当性のあるパラメタの値が推定できるような利用方法が望ましいといえよう．

12.2.6 テスト重視の開発プロセス

開発プロセスや方法論の中で，テストにとくに重点を置いているものがある．

クリーンルーム開発法 (cleanroom software engineering) [72, 90] クリーンルーム開発法は，数学的に厳密な設計法と統計に基づいたテスト手法を組み合わせて，信頼性の高いソフトウェア開発を目指すものである．テストの担当者とシステムの開発者を厳密に分け，開発者には，開発したプログラムをコンパイラにかけることすら許さない．テストは，運用時の入力データ分布に基づく統計的なサンプリングによって生成されたテストデータで，実施される．

極端プログラミング (extreme programming) [13] XP とも略称される極端プログラミングはいくつかの要素から成り立っているが，とくにまずテストケースを書くことを強調する．すなわち，想定される動作をするテストプログラムを最初に書く．機能を実現するプログラムはまだできていないから，それを「実行」すると不具合が起こる．その問題を解消するようにプログラムに手を加える．その操作を続けて，テストが思い通りの結果となれば，その部分のプログラムは完成である．

12.2.7 対故障性

検証技術によってエラーを除去し故障の発生を防ぐとともに，稼動時に万一故障 (fault) が発生しても障害 (failure) には至らないようにする技術も重要である．これは一般に対故障性 (fault torelance) と呼ばれる．しかし，対故障性はもともとハードウェアに対して工夫されてきた技術であり，ソフトウェアに対しては独自の発達がなされてきたとは必ずしも言い難い．代表的なものとして，回復ブロック法と N バージョン・プログラミングを紹介する．これらはある意味で，テスト技術の応用と見なすことができる．

回復ブロック法　回復ブロック法では，同じ機能を果たす N 個の独立に開発されたモジュールを用意する．その機能が正しく動作しているかを実行時に判定できる受容テストを用意できることも，前提となる．N 個のモジュールに順序を定め，実行時にまずモジュール 1 を実行しその結果を受容テストにかける．その結果が成功であれば，そのまま先に進む．失敗であれば次のモジュール 2 を実行し，やはり受容テストにかける．モジュール N までいってもまだ成功しない場合は，異常と判定しそのための特別処理をすることになる．

N 個の独立したモジュールを用意することが簡単でないだけでなく，実行時の余分な負荷も大きいので，実際に実装されているケースはさほど多くないようである．

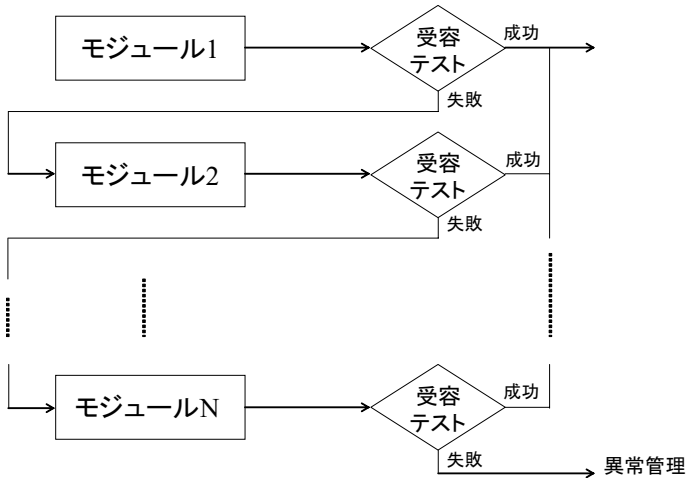


図 12.11: 回復ブロック法

N バージョン・プログラミング　回復ブロック法と同様に，同じ機能を果たす n 個の独立に開発されたモジュールを用意する．その結果が一致すればそれをそのまま出力とするが，一致しない場合はその中の多数決をとって結果とする．スペースシャトル搭載の制御システムで，これに相当する処理が行われていることで有名である．

スペースシャトルの場合は実行時に多数決を取るが，テスト時に多数決をとって正しいモジュールを判定するという方法もある．もっとも簡単な場合，2 つの独立したモジュールを作って動作を比較する．$N = 2$ では多数決にならないが，両者に違いが出た場合，どちらが正しいかを別の方法で判定することになる．N を 3 以上にして多数決をテスト時に持ち込むことも原理的には可能である．また，結果が正しいかという比較だけでなく，性能などの非機能要求について比較し，優れた方をとるということもしばしば行われる．

回復ブロック法と同じく，独立した N 個のモジュールを用意することは難しい．独立の開発組織で作成しても，エラーが起こる場所には相関があるという分析が報告されている．

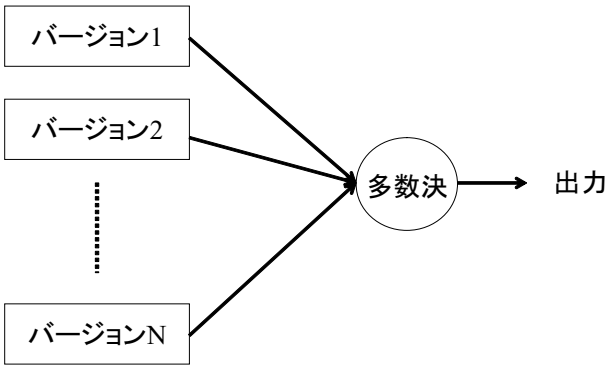


図 12.12: N バージョン・プログラミング

12.2.8 正当性の証明

Dijkstra の言を引用して述べたように，テストではプログラムが完全に正しいことは検証できない．その意味での「完全性」を求めるなら，形式仕様を記述し，プログラムの形式的意味論と組み合わせて，対象プログラムが仕様を満たすことを数学的に証明するしかない．そのような研究は 1970 年代以降続けられており，一部のとくに高い信頼性が求められるプログラム，たとえば原子力発電プラントの制御システムの核部分などには適用されてきた．一般のプログラムに対しての適用例はまだ少ないが，この技術がプログラミング言語の設計やプログラミング方法論に与えた影響は大きい．

プログラムの正当性を形式的に証明する手順は，概略次のようである．

- 仕様を事前条件と事後条件で記述する．
- ループに対しては不変表明 (invariant assertion) を置く．その他，任意に表明を挿入してよい．
- 事前／事後条件や表明を前後に持つプログラムの実行経路につき，経路の先頭の条件から，最後の条件が導出されることを証明する．

  その際，プログラムの各要素の意味については，Hoare 流の公理を利用する（あるいは，言語要素が述語で表されたプログラム状態を “変換” すると解釈してもよい）．
- 以上は結果が正しいことの証明だが，同時に必要なのが，プログラムが正しく停止することの保証である．

  停止を含む正しさを，強正当性と呼ぶ．通常，有限下降鎖性を持つ順序，すなわち単調に減少する列は必ず有限であるという性質を持つ順序，を有する何らかの値（たとえば非負整数）が，ループを回る度に単調に減少することで示す．

この方法は，11.2 節で述べた形式仕様から不変条件 (表明) を用いて反復型のプログラムを構成する手法と対応している．実際，もともと形式仕様から展開するという発想で作られていないプログラムの正当性を証明するの一般にきわめて困難であり，むしろこの考え方をプログラム設計時に用いるほうが前向きであり効果的でもある．

正当性の証明を人手で行うのでは，その証明に誤りがないことを保証できない．そこでさまざまな定理証明系 (theorem prover) が作られ，試用されてきている．定理証明系による証明自身は一般に自動化できない．人が証明の戦略を立て，必要な補助定理を導入するなどの手間をかけなければならない．また，証明に失敗した場合，プログラムが悪いのか，仕様が悪いのか，証明の戦略が悪いのか，証明系の能力に問題があるのか，直ちには判らないという問題もある．

しかし，定理証明系の技術も進歩し，とくにプログラムの検証に向いた証明系も種々開発されている（たとえば J. Rushby による PVS[83] など）．より軽量な形式技法である型検査やプログラム解析，あるいはモデル検査 (12.3.2 節参照) と組み合わせて，それぞれの特徴を活かせば有効であろう．

事前条件，事後条件，不変条件という考え方は，オブジェクト指向開発方法論の 1 つとして提唱されている「契約に基づく設計（design by contract)」でも採用されている [70]．オブジェクトのメソッドを設計するに当たり，その事前条件と事後条件をまず記述してそれを契約とみなし，設計はその契約を守るように行う．またクラスで常に成り立つ条件を不変条件として記述し，それもクラス設計上の契約とする．利用者はその契約を見てオブジェクトがそれに従って動作するものと信用して使う．

事前条件，事後条件，不変条件を実行時に検査可能な表明として記述しておくと，定理証明系のような大道具を使わなくても，実行時にその表明が成り立つかどうかを検査し，成り立たない場合には警告を出したり割り込みを起こすようにすることができる．その代わり事前事後不変表明として一般的な論理式は書けず，実行時に論理値として定まるもののみが有効になるという制約はある．

UML でも契約に基づく設計を支援するために，事前条件，事後条件，不変条件を記述する言語として OCL(Object Constraint Language)[114] を導入し，クラス図などと組み合わせて用いることが勧められている．

12.3　仕様の検証技術

12.3.1　見直し（レビュー），査閲（インスペクション），徒歩検査（ウォークスルー）

設計文書やプログラムコードを査閲するインスペクションという方法は，IBM の Fagan によって提唱され，IBM で実践されて成果を上げ有名になった [39].

Freedman と Weinberg の見直し，査閲，徒歩検査について具体的に述べた本 [42] は，よく参考にされている．それによると，例えば，コード査閲は，

- 議長，プログラムの設計者，テストの専門家，と書いた当人の４人でチームを構成
- 資料は事前に配布
- 発見されたエラーは漏れなく記録
- プログラマはコードを読み上げる
- チェックリストを利用してチェック
- 結果は形式の定められた文書とする
- かける時間は１～２時間
- エラーの修正はこの中では行わない

という特徴を持つ.

徒歩検査（ウォークスルー）もよく似ているが，簡単なテストケースを定め，コンピュータの実行を模擬的に追跡してみるところが異なる．また，見直し (レビュー) という用語は，査閲や徒歩検査の総称として使われることが多い.

12.3.2　モデル検査

モデル検査とは

モデル検査は，システムの仕様を表現したモデルがシステムに要求される性質を満たすかどうかを自動的に検査する手法である．とくに並行性を含む応答型システムを対象に，モデルはラベル付き遷移システム (Labeled Transition System) で記述し，性質は時相論理 (temporal logic) で表すものを，狭い意味でモデル検査と呼ぶことが多い [28]. この手法は，1980 年代の前半に提唱され，とくにハードウェアの論理回路や通信プロトコルの正しさの検証に用いられてきたが，その後，一般のソフトウェアを対象としたモデルの検証にも用いられるようになった.

ラベル付き遷移システムとは，基本的に第 8 章で述べた状態遷移モデルと同じであるが，遷移に動作 (action) のラベルが付いている，という意味で LTS と呼ばれる．また伝統的な状態機械と異なり，終了状態を仮定せず，応答的な状態遷移による振舞いの過程をモデル化することに主眼を置くという視点の違いもある．時相論理としては LTL(Linear Temporal Logic) や CTL(Computation Tree Logic) が多く用いられる.

手法の中心はモデルで定まるシステムの振舞いの遷移状況を表すグラフ上の経路探索に帰着され，有限な探索で結果が得られることが特徴となる．とくに性質が満たされない場合，その反例となる具体的なケースを示すことができる点が役に立つ.

検査対象となる性質

検証の対象となる性質は，通常次の 2 種類に分けられる.

- **安全性 (safety)**　望ましくない事象がどんな遷移経路をたどっても決して起こらないという性質．望ましくない事象の典型例はデッドロックである．また，資源アクセスの違反（相互排除に違反するなど）も並行システムでは典型的な望ましくない事象である．時相論理式では通常次のようなパターンで記述される．

  AG $\neg P$

  ここで **A** は「どの計算経路を通っても」を意味する限量子，**G** は「いつでも」を意味する時相演算子，P は望ましくない性質を表す述語である．

- **活性 (liveness)**　望ましいことがいつかは必ず起こるという性質．何が望ましいことかはシステムの機能要求に依存するが，たとえばクライアント/ サーバー・システムでクライアントからのサービス要求に対し，いつかは必ずサービスが提供されるというような性質が典型的である．時相論理式では通常次のようなパターンで記述される．

  AF Q

  ここで **F** は「いつかは」を表す時相演算子．Q は望ましい性質を表す述語である．

安全性や活性をより具体的に把握するために，これまで使ってきた酒屋問題や自動販売機の例題で，どのような性質が想定されるか考えてみよう．

まず安全性であるが，自動販売機の問題記述そのものに，ある種の安全性に対する考慮が見られる．

- 販売ランプが点灯している商品の購入ボタンが押されると，当該商品の販売ランプは点滅し，商品が取り出し口に 1 つ出される．同時に，残高は商品価格分が減額される．**この間は，金銭の投入はできない．**
- 返金ボタンを押すと，その時点での残額がつり銭取り出し口に返金される．**返金動作中は，金銭の投入はできない．**

上記の太字部分で，購入ボタンが押された際や返金ボタンが押された際の処理中には，金銭の投入をブロックするように要求している．これは投入金額の管理が，金額の減額操作と増額操作が同時に起こることによって不整合に陥ることを配慮したものだろう．この問題記述では金銭の投入を物理的にブロックするという要求と読めるが，金額データの相互排除をソフトウェア的に実現して，利用者からは金銭投入がブロックされているようには見えないようにすることも，可能ではある．また具体的にブロックするように実現した場合は，デッドロックにならないことに注意し，それをきちんと検査しておく必要がある．

活性であるが，酒屋問題で顧客の注文は必ずいつか出荷されるという要求を考えるとすれば，これは活性の例となる．これを満たすには，システムの設計を工夫するだけでは不十分で，注文を満たすような酒の入荷が必ずいつかはあるという，システム外部の業務上の条件を前提としなければならない．この条件を正確に記述し，かつそれに応じたロジックの設計を行うことは，案外難しい．たとえば在庫切れの注文が発生した場合，それ以降に不足分を上回る注文品目の入荷が累積で必ずある，という条件を想定したとする．それに対し，出荷は先入れ先出し，つまり時間的に早い注文を優先して出荷するという戦略で，注文に対する出荷は必ずいつかあるという活性が保証されるだろうか．考えてみると，これではだめなケースがある．不足分を上回る入荷が累積であったとしても，1 回の入荷では常に不足分を下回っている場合，後から来たより少量の同品目の注文に先を越される．その事態が常に続けば，いつまでも待たされることになる．1 回で不足分を上回る入荷が必ずあるという条件にすればよさそうだが，今度はその条件が強すぎるという議論もありうる．

このようにシステムの仕様のモデルに対して望ましい性質を検証するという考え方は，12.2.8 節で述べたプログラムの正当性の証明とはアプローチが若干異なる．しかし，最近はモデル検査の手法を直接プログラムの検証に用いるという研究が盛んに行われている．そこではプログラムからその振舞いモデルを抽出して，そのモデルに対し満たすべき性質を検査するという方法を取る．

EJB アーキテクチャのモデル検査

モデル検査の事例として，Enterprise JavaBeans(EJB) のアーキテクチャの妥当性をモデル検査の手法で分析したものを紹介する (中島・玉井 [76, 75]).

EJB コンポーネント・アーキテクチャ　EJB は分散ビジネス・アプリケーション用のコンポーネント・アーキテクチャである．EJB アーキテクチャは図 12.13 に示すような構成を持つ．

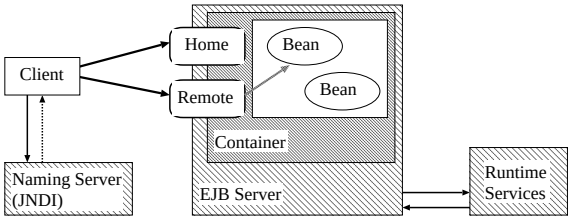


図 12.13: EJB アーキテクチャの概要

EJB ではコンポーネントをビーン (bean, つまり豆) と呼ぶ単位として扱う．クライアントは JNDI という名前サーバを用いて必要なビーンを要求すると，Home というオブジェクト参照が得られる．Home はビーンオブジェクトを生成し，その代理オブジェクトとなる Remote を作ってその参照をクライアントに渡す．その後の処理でクライアントが出す処理要求は，Remote が受けてビーンに委譲することになる．ビーンは Container という動作環境の下で動く．Container は EJB サーバの機能を用いて，ビーンの退避処理，永続化処理，自動消去処理などの実行時サービスを適宜行う．

ビーンには実体ビーンとセッションビーンとの 2 種類がある．実体ビーンは永続記憶装置内に常駐し複数のクライアントに共有される．セッションビーンは単一のクライアントに対応して生成され，一般に複数の実体ビーンにアクセスしながら一連のトランザクション処理を実行する．図 12.14 はその関係を図示したものである．

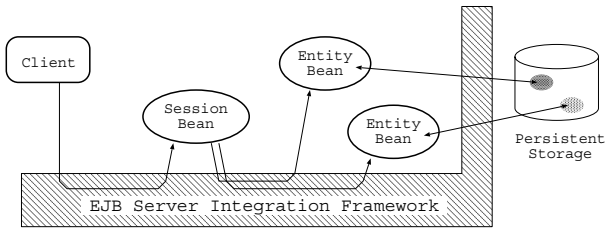


図 12.14: セッションビーンと実体ビーンの関係

実体ビーンは図 12.15 のように振舞う．ここで遷移に付けられているラベルはいずれも実体ビーンのメソッドで，メソッド起動に応じて図のように状態遷移が起こる．ejbCreate でビーンが作られ，ejbRemove で消去される．ejbStore と ejbLoad は永続化処理に関わるメソッドで，それぞれ永続記憶装置への書き込みとそこからの読み出しを行う．ejbPassivate と ejbActivate は退避処理に関わるメソッドで，それぞれ主記憶からの退避と復帰に対応する．ビジネスメソッドはこのビーン固有のビジネス機能を実現したメソッドである．

モデル検査系 SPIN　SPIN は AT& T のベル研究所で G. Holzmann によって開発されたモデル検査系である [50]. EJB のコンポーネントアーキテクチャを分析するのに，この SPIN が用いられた．SPIN ではモデルは Promela という仕様記述言語で記述する．また，検査の対象となる性質は線形時相論理 LTL で記述される．

Promela による記述の構成要素は，プロセスとプロセス間のチャネルである．実体ビーンの記述例を示す．

```
proctype EntityBean ()
{
endLoop:
  do
    :: mthd?ejbActivate -> if :: retv!Void :: excp!SysError fi
    :: mthd?ejbPassibate -> if :: retv!Void :: excp!SyssError fi
    :: mthd?BM -> if :: retv!AppError :: excp!SysError fi
    ...
  od
```

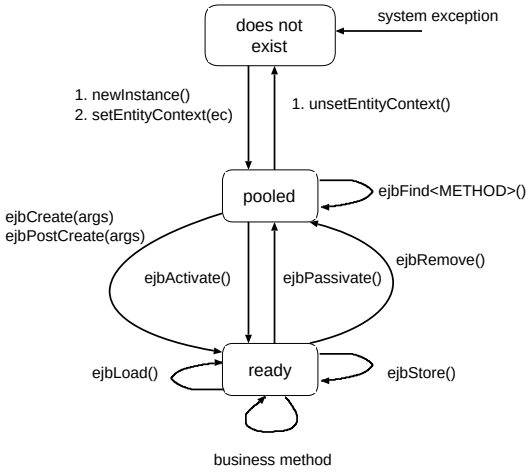


図 12.15: 実体ビーンのライフサイクル

}

Promela では Dijkstra 流のガードつき命令を制御構造として用いる．ガード付き命令の do 文は，次のような構文を持つ．

do $B_1 \to S_1 :: B_2 \to S_2 :: \cdots :: B_n \to S_n$ od

ここで，B_i は論理式で示される条件を表し，S_i は実行可能な文を表す．その意味は，条件 $B_1, B_2, \cdots, B_n$ の内，真と評価されるものがあれば，その内の一つを非決定的に選び，それを B_i とすれば対応する文 S_i を実行するということを繰り返す．もしすべての条件が偽となったら繰り返しを終了する．

また，ガード付き命令の if 文は，同じように

if $B_1 \to S_1 :: B_2 \to S_2 :: \cdots :: B_n \to S_n$ fi

という構文を持ち，その意味は，条件 $B_1, B_2, \cdots, B_n$ の内，真と評価されるものがあれば，その内の一つを非決定的に選び，それを B_i とすれば対応する文 S_i を 1 回実行して終了するというものである．

なおガードがない場合は $true \to S$ と等価である．EntityBean のプログラムでは if 文でガードがないものを使っているが，その意味は「非決定的にいずれかを実行する」ということになる．

プロセスはチャネルを介したメッセージ交換によって相互作用する．ここでいうチャネルの概念は Hoare による CSP および Milner による CCS や π 計算におけるチャネルと同じである．すなわち c!e はチャネル c に式 e の値を送信することを意味し，c?v はチャネル c から受信した値を変数 v にしまうことを意味する．なお，EntityBean のプログラムで `mthd?ejbActivate` などをガードに使っているが，その意味はチャネル mthd の待ち行列の先頭に ejbActivate というメッセージが存在するというパターン照合を表し，照合すれば真となる．ここで ejbActivate, ejbPassivate, BM はいずれも定数である．BM はビジネスメソッドを指す．

次に EJBObject を Promela のプロセスとして定義したものを示す．EJBObject は Remote で与えられるビーンへの代理オブジェクトである．

```
proctype EJBObject()
{
  chan returnValue; chan exceptionValue; short value;
progressLoop:
endLoop:
  do
    :: remote?remove,returnValue,exceptionValue
       -> { request!reqRemove; retvalFC?value
            -> returnValue!value; goto endTerminate }
```

```
          unless { exceptFC?value -> exceptionValue!Error; goto entTerminate }
    :: remote?BM,returnValue,exceptionValue
       -> { request!reqBM; retvalFC?value -> returnValue?value }
          unless { exceptFC?!value -> exceptionValue!Error }
  od;
endTerminate:
  skip
}
```

ここでチャネル remote の定義を略しているが，このチャネルは 3 つの値からなる組をメッセージとして受け取る．1 番目がメッセージの内容であり，後の 2 つは値を送り返すべきチャネルである．また受け取るメッセージの remove は，ejbActivate などと同じ意味の定数で，ejbActivate などが実体ビーンのメソッド名であったのに対し，EJBObject に定められているメソッド名である．

実体ビーンの入れ物である Container の動作は複雑である．図 12.16 にその状態遷移図を示す．

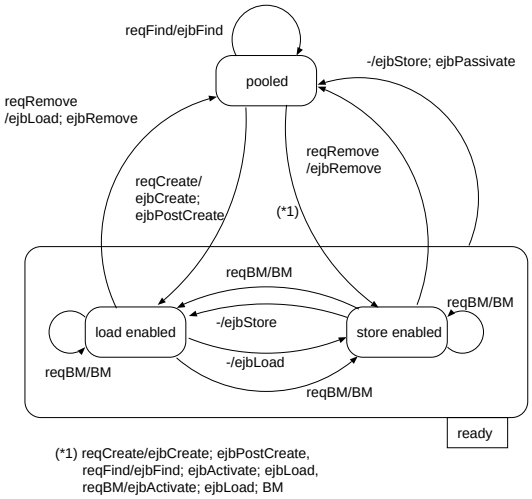


図 12.16: Container の振舞い

この振舞いに従って，Container も Promela でモデル化する．約 250 行の Promela 記述になっている．

性質の検証　検証したい性質を LTL で記述する．LTL では計算経路の分岐が起こらないので,「どの計算経路を通っても」という限量子 A や「ある計算経路が存在して」という限量子 E は用いる必要がない．時相演算子としては，SPIN における LTL では以下を用いる．

- **G** P　常に P が成り立つ．
- **F** P　いつか P が成り立つ．
- Q **U** P　P がいつか成り立ち，それまでの間は常に Q が成り立つ．

たとえば実体ビーンの振舞いについて，検証の対象として考えられる性質の自然言語記述と LTL 記述の例を示す．

1. pooled 状態の実体ビーンにクライアントから BM 起動の要求があった場合，ejbActivate がいつか起動される．

$$\mathbf{G}\,(pooled \wedge BM\_Client \rightarrow \mathbf{F}\; ejbActivate) \tag{12.3}$$

2. ejbActivate の後，BM が実行される場合には，その前に必ず ejbLoad が実行される．

$$\mathbf{G}\,(ejbActivate \wedge \mathbf{F}\; BM \rightarrow (\neg BM\; \mathbf{U}\; ejbLoad)) \tag{12.4}$$

3. EJB サーバーは BM の後 ejbPassivate をする場合は，その前に ejbStore を実行しなければならない．

$$\mathbf{G}\,(BM \wedge \mathbf{F}\; ejbPassivate \rightarrow (\neg ejbPaasivate\ \mathbf{U}\; ejbStore) \tag{12.5}$$

SPIN ではこのような LTL の式を Büchi オートマトンに変換する．Büchi オートマトンは有限状態オートマトンの 1 種で，通常の正規表現を受理するものとほぼ同じであるが，無限長の文字列を受理対象とするために，終了状態の替わりに受理状態を定義する．すなわち受理状態に至ったときに，そこで必ずしも文字列の入力を打ち切らず遷移を続けることができる．

たとえば，最も簡単な活性の式

AF Q

を Büchi オートマトンに変換すると，図 12.17 のようになる．モデルを記述している LTS は，そのままオートマ

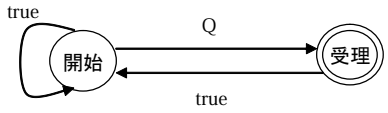


図 12.17: LTL 式の Büchi オートマトンへの変換

トンと見なすことができる．これに検証の対象となる LTL 式の否定を変換して作られた Büchi オートマトンを合成した積オートマトンを作る．その積オートマトンが受理言語を持たない場合は，検証したい性質が満たされたことになる．受理する文がある場合は，その文が検証したい性質に違反する反例を表すことになる．これが LTL 式の否定を合成していることの効果である．

EJB のモデルにクライアントのモデルを想定して合成し，さらに検証対象の LTL 式の否定から作られたオートマトンを合成して，上の 3 つの性質を SPIN で検証 すると，いずれも満たされることが確認される．

一方，クライアントが要求を出すと，それに応じた実体ビーンの動作がいつか必ず実行されるというタイプの活性の性質は，単純には満たされるとは限らないことが判る．たとえば

4. クライアントが remove を要求すると ejbRemove がいつか起動される．

$$\mathbf{G}\,(remove \rightarrow \mathbf{F}\; ejbRemove) \tag{12.6}$$

を検証すると，満たされないという結果を得る．その反例を見ると，ejbStore と ejbLoad が交互に無限に繰り返すライブロック (livelock) が生じ，進行を妨げることがありうることが判る．このライブロックはある種の公平性 (fairness) の仮定で取り除くことができるが，それでも式 (12.6) はまだ満足されない．図 12.15 から判るように，実体ビーンが ready 状態から pooled 状態に遷移するのは，ejbRemove による場合と ejbPassivate による場合とがある．ejbRemove はクライアントの remove 要求に応じて起動されるが，ejbPassivate は実行時の退避サービスとして Container により起動される．クライアントが remove を発行した後，それと独立に Container が ejbPassivate を起動してしまうと，ejbReomve は起動される機会を失う．この状況は EJB1.1 仕様書では想定されておらず，仕様書の不備と考えられる．

このようにモデル検査の手法を用いて，モデルがどのような性質を満たすか満たさないか，満たさないのはどのようなケースか，が解析できる．

第13章　ソフトウェアの保守・発展

13.1　ソフトウェア発展の基本的特徴

ソフトウェアは本来的に変化しつづける性質を持つ．この変化をここでは発展（evolution）と呼ぶ．文脈によっては「進化」という語を用いることもある．

ソフトウェアが「発展する」あるいは「進化する」という言い方は，実質的にはソフトウェアの保守と同じ現象を指しているが，より前向きな響きを持つ．さらに「進化・発展」は自動詞であるが,「保守」は他動詞である．すなわち，ソフトウェアは進化・発展するが，人によって保守されるのである．前者の見方はソフトウェアの発展を客観的に観測する手法を示唆するが，後者はソフトウェアが人工物であり，保守は人間によって行われる作業であることを常に意識させる．

ソフトウェアを発展させる要因には，要求の拡大や変更，システム環境（たとえばハードウェア，OS，ミドルウェア，他の関連システム）の変化，利用環境の変化といった外部要因と，保守性・信頼性・性能などの向上のための内部構造の再編といった内部要因とがある．このように多様な要因がある上に，ソフトウェアはハードウェアと比べて簡単に手を加えることができるという性質を持つために，継続的に発展を続けていく．

発展の形態は様々である．とくに発展の規模と時間について見れば，

1. 規模の多様性：１つのシステムを対象としても，部分的な修正では，1 文，さらには 1 文字の変更という場合がある．大規模な発展では，システム全体の数 10%に対して変更が加えられる．その割合がある程度以上となると，修正というより作り直しと呼ぶべきものとなるが，作り直しも発展の１つの形態と見ることができる．また，発展の対象となる単位はシステムとは限らない．コンポーネントが，ライブラリ全体としてとしてあるいは個別に発展することもある．

2. 時間の多様性：ソフトウェアの寿命には数日のものから数 10 年のものまである．一般にソフトウェアは，その一生の間発展を続ける．また，１つのソフトウェアの寿命が尽きても，それが新たに作り直される場合には，世代をまたがって発展が続くと見ることができる．個々の変化を実現するための期間も，たとえば応急的に 1 文を変更する場合にはごく短時間で終了することが要請されるだろうし，大規模な変更の場合には，その作業は新規の開発と同等であり，数年を要することもまれではない．

   また，変化を起こすタイミングについても，システムをいったん停止してシステムを再構成し起動しなおす場合と，運用を停めずに動的に変化させる場合とがある．近年のソフトウェアシステムは応答型で常時運用されるものが多く，後者の必要性が高まっている．時間的な発展形態としては，前者を離散的，後者を連続的と見ることもできる．

13.2　ソフトウェア発展のモデル

13.2.1　システムの進化モデル

ソフトウェアの進化プロセスを実証的に示したものとして，Belady & Lehman によるよく知られた研究がある [15, 66]．彼らは IBM の OS/360 の進化過程を分析し，次のようなソフトウェアの進化法則を提言した．

1. 継続的変化の法則
   使用されるシステムは，変更を凍結し作り直す方がコスト的に有利になるまで，継続的な変化を続ける．

2. エントロピー増大の法則

システムのエントロピー（構造の無秩序性）は，その増大を防いだり減少させたりするための意図的な努力がなされない限り，時間と共に増大する．

3. 統計的に滑らかな進化の法則
   システム属性の進化は，局所的にはランダムな過程に見えるが，統計的に見れば，規則的な変動と円滑な長期傾向とが合成されたものとなる．

そして彼らは，ソフトウェアは一般に機能的には進化を続けるが，構造的には劣化していき，やがては死に至るという動的特性を持つと主張した．

Belady & Lehman と異なり，アプリケーションソフトウェアの発展過程について行われた玉井・鳥光による調査もある [106, 108, 112]．そこでは，よく使われるソフトウェアは，一定の期間の後作り直されること，すなわちソフトウェアが世代交替により長期的な発展を続けていくというプロセスに注目し，次のような知見をまとめている．

1. ソフトウェアの一世代の寿命，すなわち実用に供されてから作り直しによって新しいものにとって代わられるまでの期間は，平均約 10 年である．しかし，寿命のバラツキは大きい．
2. 小規模のソフトウェアは寿命が短めである．
3. 管理的業務用のシステム（たとえば経理や人事システム）の方が現業的業務用システム（たとえば生産管理や販売管理システム）より寿命が長い．
4. ソフトウェアの規模は作り直しによって増大する．
5. 作り直しの決定要因は複合化している．利用者の機能変更・拡張要求を満たすという要因を作り直しの要因として挙げる事例は，全体の過半数を占める．
6. 保守性の悪化は，作り直し理由として重要な要因ではあるが，支配的なものではない．またそれを理由として作り直されるソフトウェアの寿命は，平均的に長い．

13.2.2 オブジェクトの進化モデル

オブジェクト指向技術が広い範囲で適用され，遺産ソフトウェアは徐々にオブジェクト指向型のシステムで置き換えられつつある．また，古いソフトウェアを置き換えたオブジェクト型のシステムも，発展を続ける．

オブジェクトレベルの進化　そもそもオブジェクトは動的に変化する性質を持つ．もっとも基本的なレベルでは，オブジェクトはまず生成され，互いにメッセージの交換をしながら自らの状態を変化させることにより動作し，やがて消滅する．さらに，より長期の時間軸で考えた場合，オブジェクトの内部構造や外部から見た役割，およびオブジェクトによって構成される組織が，成長・進化を続けていく．

ここではシステムの進化とオブジェクトの進化は必ずしも同期しない．システムが生命を終えても，その中のオブジェクトの一部は別のシステムに移って進化を続けるかも知れない．これを生物の種 (個体) の進化と DNA の進化の違いとの間に，類比することもできる．このような観点からオブジェクトの進化プロセスを実証的に分析するとともに，進化プロセスモデルを構築する試みが，行われている．

ダーウィンの進化論の本質は，2 つの仕組みに帰着できる．1 つは複製でありいま 1 つは自然淘汰である．この両者を含むプロセスは，自然現象であろうと人為現象であろうと，進化とみなしてよい．そこで R. Dawkins はミーム (meme) という言葉を，種々の人工概念が普及する社会現象を説明するための遺伝子に対応するものとして提唱した [32]．Dawkins によれば，ミームは文化複製子の単位で，例としては曲，アイディア，惹句，ファッション，壷の作り方，アーチの架け方などがあるという．多くの人がこのミーム概念を発展させた．代表的な例として，S. Blackmore による大胆な展開がある [18]．ソフトウェアコンポーネントの進化は，明らかにミームに基づく進化の特徴を有している．

観測事例　オブジェクトレベルの進化・発展を実証的に分析した結果が中谷・玉井によって報告されているので [78, 77, 104]，以下でその概略を述べる．

以下の 3 システムを第 1 段階の分析対象システムとしている．

1. **熱交換シミュレーションシステム**
   多様な熱機器からなるシステムの熱流と温度分布を，シミュレートするシステム．
2. **入金管理システム**
   サービス会社向けの入金管理システムで，顧客からの入金と請求伝票をつき合わせ，消し込み処理を行うもの．
3. **証券管理システム**
   企業の保有証券データ，すなわち額面価格，買取価格，金利，償還などに関する情報を管理し投資判断を支援するシステム．

各事例につき，4 版以上のデータがある．

観測された進化パターン　各システムの各版につき，システム，クラス，メソッドの 3 つの層にについて計測が行われている．システム層についてはクラス数やクラス木の深さ，クラス層についてはメソッド数，インスタンス変数の数，サブクラスの数，メソッド層については原始コード行数，など．それぞれの層の 1 段下の層のデータの集計値や平均はまた，上の層の計測量となる．たとえば，メソッドの行数の合計や平均は，クラス層の計測量である．

その結果得られた観測結果は，次のようにまとめられる．

1. 基本統計量と分布形は，時間の推移に対し相対的に安定している．
2. 一方で，例外的に大きな特異値を持つデータが存在することがある．これらは設計上の不具合か例外的な設計判断を反映していると見られる．
3. 多くのデータは時間軸上で右上がりに増大する傾向を示すが，その変化は連続的ではない．急激な上昇の期間と変化の遅い期間とがある．非連続的な変化は，アーキテクチャレベルの変更が行われたことを示す場合が往々にしてある．
4. クラス木を特徴づける固有の計量の存在が発見された．

統計モデルの安定性　オブジェクトシステムの規模に関しては，伝承的なデータが知られている．Smalltalk プログラマとして 100 万行以上のコードを書いてきたという A. Aoki は [9]，自ら開発したあらゆるシステムやライブラリの実績から，クラス当たりの平均メソッド数は 20，メソッド当たりの平均行数は 10，したがってクラス当たりの平均行数は 200 だといっている．さらにこの値は，自分の開発したプログラムのみならず，提供されている標準ライブラリや他の組織で開発されたコードでも変わらないという．

この 4 事例でもこの値が確認された．表 13.1 に示すデータは，この観測を裏付けるものである．これらの値は時間 (版) によらず，またシステムによらずほぼ一定である．

図 13.1 と図 13.2 は，典型的な規模データの度数分布図を示したものである．これを見ると，異なる版だけでなく異なるシステム間でも平均値がほぼ同じであるばかりか，分布形も共通していることがわかる．

クラス木の特徴づけ　予想されるように，クラス当たりのコード行数とメソッド数は強い相関を持つ．実際，3 つのシステムについてクラスのコード行数とメソッド数の相関を統計的に検定した結果，有意と認められた．

図 13.3 の上段左に，熱交換シミュレーションシステムの関して，この 2 つの値を両軸にとった散布図が描かれている．ここでサンプルは，このシステムの 4 つの版すべてのものを集めて表示している．すぐに気づくように，この図では相関を示す直線が複数本識別される．調べてみると，これらの直線はそれぞれ同じクラス木に属する

表 13.1: 熱交換シミュレーションシステムの基本データ

クラス当たりのメソッド数

版	1	2	3	4
平均	15.1	19.4	19.7	18.3
標準偏差	10.3	16.5	19.4	19.9

メソッド当たりの行数

版	1	2	3	4
平均	8.1	8.5	9.1	9.4
標準偏差	10.8	16.0	19.5	21.5

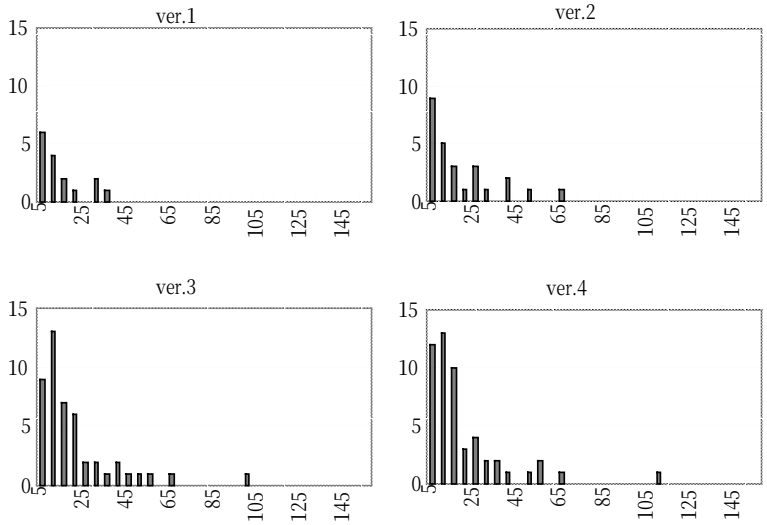


図 13.1: 熱交換シミュレーションシステムのクラス当たりメソッド数の度数分布図

クラス集合に対応したものであることが分かる．図 13.3 の他の 3 つの散布図は，これを 3 つのクラス木に分けて表したものである．

きわめて注目すべき現象が，編集木（上段右) の散布図に見られる．図中の矢印は，この木に属するあるクラスの第 3 版における値から第 4 版における値への移動を示している．このクラスの第 3 版における行数とメソッド数に対応した座標点は，直線から大きく外れた例外値であるが，第 4 版では「正常な」値に戻っている．この現象は，同じクラス木の行数/メソッド数の値，あるいは両測定値間の回帰係数の値が，かなり強力な拘束力を持つものであることを示唆するものである．

この木に固有な値の持つ意味をさらに探求するために，以下の 3 つの仮説が立てられた．

1. **異なるクラス木の回帰係数の値は異なる．**
   この仮説は，異なる木の回帰係数が等しいという帰無仮説が棄却されることにより，検証された．3 システムすべてでこの仮説が支持された．

2. **クラス木の回帰係数は進化の過程で不変である．**
   この仮説は，異なる版における同じクラス木の回帰係数が等しいという帰無仮説が，棄却できないことにより検証された．

3. **クラス木の回帰係数は異なるプログラマ間でも安定している．**
   このデータでは，1 つのクラス木に属するクラスを，複数のプログラマで分けて開発している例は，1 例しかなかった．そのケースでは，両者の間での回帰係数に統計的に有意な違いは認められなかった．ただ，これは 1 例でしかないので，確定的な結論とはいえない．

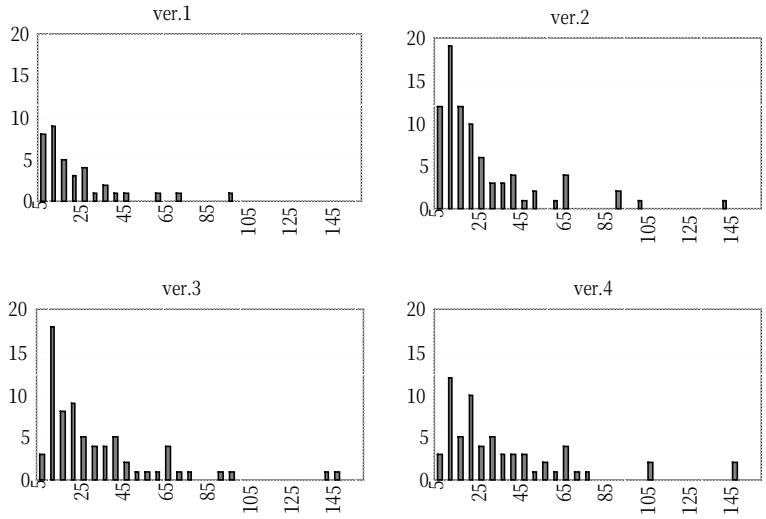


図 13.2: 入金管理システムのクラス当たりメソッド数の度数分布図

これらの知見から，各クラス木には，開発者が暗に仮定しているある種の設計基準が存在することが，示唆される．

負の 2 項分布　オブジェクト指向以前の昔から，モジュール別のコード行数のようなプログラム規模を表すデータは，もっとも基本的な統計量として収集されてきた．それらの度数分布図も度々描かれてきたが，そのようなデータの分布がなんらかの統計モデルに従うのではないかという議論はこれまでほとんどなかった．

図 13.2 が示すように，オブジェクトのクラスやメソッドの規模データは，以下のような共通の特徴を持つ．

1. 度数分布の山は 1 つで，全区間の左側にある．
2. 平均値は山よりは右，区間の中央よりは左にある．
3. 分布形右側は，長い裾野を持つ．

いくつかの統計モデルの当てはめを試した結果，多くのケースで負の 2 項分布が計測値とよく一致していることが観測された．負の 2 項分布は次のように定義される．事象 S が起こるかどうかを観察する繰り返し試行において，各試行が互いに独立で，S の起こる確率 p が時間に依らず一定であるとき，それはベルヌーイ試行と呼ばれる．S がちょうど k 回起こったときの試行系列の長さを x とすると，x の確率関数は，

$$P(x) = \left(\begin{array}{c} x-1 \\ k-1 \end{array} \right) p^k (1-p)^{x-k}, \tag{13.1}$$

で与えられる．

この x の確率分布を負の 2 項分布と呼ぶ．この確率モデルは 2 つのパラメータ，p と k で定められる．

負の 2 項分布に従う x の期待値は，

$$E(x) = k/p, \tag{13.2}$$

で与えられ，分散は

$$V(x) = k(1-p)/p^2, \tag{13.3}$$

で与えられる．

負の 2 項分布に従うことが予想されるサンプルデータが与えられたとき，パラメータの推定値 $\hat{p}$ と $\hat{k}$ はそれぞれ，

$$\hat{p} = \bar{x}/(s^2 + \bar{x}), \tag{13.4}$$

と

$$\hat{k} = \bar{x}^2/(s^2 + \bar{x}), \tag{13.5}$$

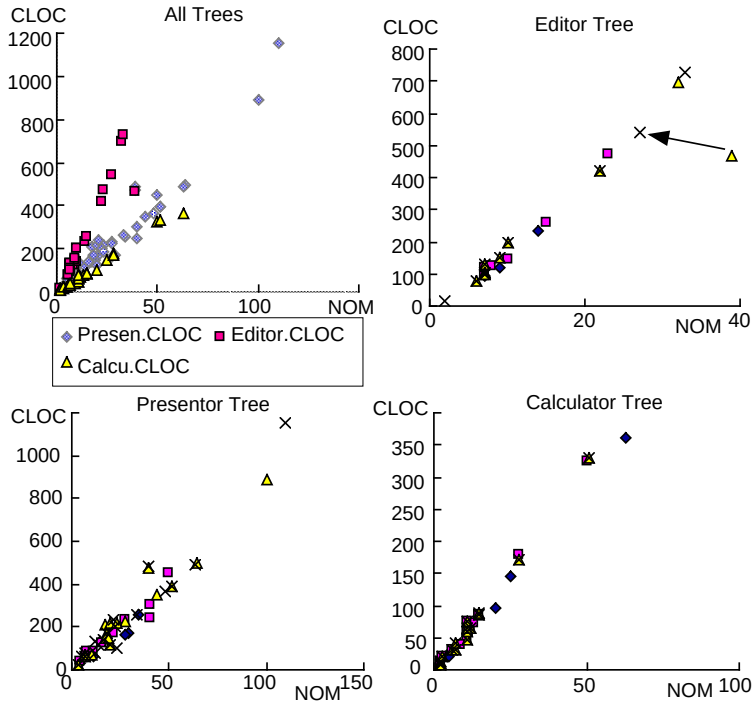


図 13.3: 熱交換シミュレーションシステムの行数対メソッド数の散布図

で与えられる．ここで，$\bar{x}$ はサンプルデータの平均値，s^2 は不偏分散である．

負の 2 項分布は実データとの適合性がよいだけでなく，ソフトウェア開発プロセスのモデルとしての解釈が可能な点に意義がある．コードの長さが決定されるプロセスは，次のような確率プロセスとして解釈することができる．プログラマによるプログラミング作業を，第 3 者が観察しているとする．観察者の目には，プログラミングは，文（または行) あるいはメソッドの無作為の選択の繰り返しに見える．ある定まった数 (パラメータ k に対応) の特定の性質を持った文が選ばれると，メソッド（あるいはクラス) が完結する．ランダムに選ばれた文（あるいはメソッド）が特定の性質を持つ確率 (p に対応) は一定である．

この解釈に従うと，k が大きければ，問題領域や開発環境・組織で規定される慣例，スタイル，制約などによって要求される特定の文（あるいはメソッド) の集合から，より多くを選ばなければならない．また，p が大きければ，そのような特定の文（あるいはメソッド) を，そのような制約から自由な文との対比で，相対的により多く選ばなければならないことを意味する．したがって，一般的に，ソフトウェア設計あるいはプログラミングがより自由な場合，すなわちプログラマの裁量の余地が大きい場合，k と p は小さくなる傾向にあり，設計やプログラミングががよりパターン化，規律化されている場合は，それらが大きくなる傾向にあるといえよう．

モデルのパラメータの進化　2 つのパラメータ p と k でモデルが完全に決定されるので，この両者の値は平均値と分散よりも豊かな情報を含んでいる．大量のデータの有する構造が，(p, k) の作る 2 次元空間の 1 点で表されることになり，複数の版に渡るデータの動きを追跡するのに都合がよい．

図 13.4 は，熱交換シミュレーションシステムのクラス木ごとのパラメータ推定値をプロットしたものである．図の中で矢印は，版の進行に沿った推移方向を表す．

他の 2 つのケースに対しても同様なグラフが描かれる．これらのグラフから，以下のようなことが判る．

1. 両パラメータの間には，強い線形関係がある．各点がのることが推定される直線は (p, k) 平面の原点を通過するので，関係

$$k = mp, \tag{13.6}$$

の成立が予想される．ここで，m は定数係数である．

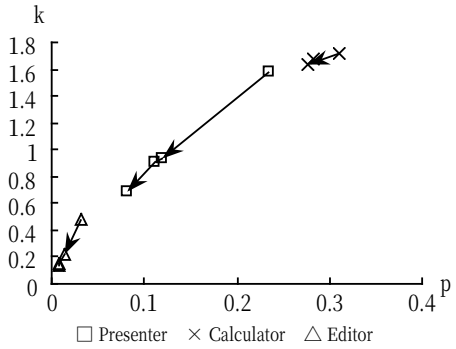


図 13.4: 熱交換シミュレーションシステムのパラメータ (p,k) の動き

等式 (13.2) の $E(x) = k/p$ という関係を思い起こすと，m は x の期待値にあたることが分かる．すなわち，メソッド数の平均は，版に依らず一定であることを意味する．

2. この線形関係により，k の値が大きくなれば，p の値も大きくなり，逆に一方が小さくなれば他方も小さくなる．k と p が大きいことは，パターン化されたコーディング，強い規約，構造の一様化されたプログラムを意味し，k と p が小さいことは，プログラミングのより大きな自由度を表す．

3. 図の矢印の進行方向を見ると，熱交換シミュレーションシステムでは k や p は時間とともに減少している．一方，同様なグラフを入金管理システムについて描くと，両パラメータが時間とともに増大していることが観測される．また，証券管理システムではどちらともいえない．熱交換シミュレーションシステムでは，版を改めるごとに利用者の要求に応じて新たなモジュールを追加してきており，一方，入金管理システムでは版が進むにつれてソフトウェア開発者がシステムを再構築してきた．この事実は，上に述べた k と p の意味づけとよく対応する．

13.3 ソフトウェア発展と保守

ソフトウェア発展は伝統的な用語である「保守」を概念として含み，それをより前向きに捉えたものと見ることができる．この節では，従来からの保守という見方から，ソフトウェアの運用開始後のプロセスを概観する．

13.3.1 ソフトウェアの保守の特性

ソフトウェアに対する保守という言葉は，他の多くの用語と同様にハードウェア製品に使われてきた言葉を転用したものである．しかし，ソフトウェアの保守はハードウェアと比べて異なる点が多く，同じ言葉を使うのが不適切という議論も成り立ちうる．

ソフトウェアの保守の大きな特徴は，次の３点である．

1. ソフトウェアを納入あるいは出荷した時点で，なんらかの欠陥が残存することは珍しくないばかりか，製作者も使用者も欠陥がありうることをある程度想定しており，その欠陥を修正する作業を保守と呼んでいる．ハードウェア製品でも出荷後に欠陥が発見されることがまれにあるが，例外的な事態とされ，その欠陥を修正することはリコールなど別の呼び方が使われて，保守とは区別される．

2. ソフトウェアには部品の摩耗・損傷といった経年的な変化は存在しない．したがって，部品の摩耗・損傷が原因となる保守作業は発生しない．

3. ソフトウェアは機能の改善，変更，拡張がハードウェアに比べてはるかに容易である．そのため，保守の中に，機能の変更や拡張も通常含めて考えられる．ハードウェア製品の場合は新製品の開発と呼ばれるような作業も，ソフトウェアの場合は往々にして保守と呼ばれるわけである．

すなわち，ハードウェアの場合保守といえば部品の経年的な劣化に基づくものを指すのに対して，ソフトウェアではそのような保守はありえない代わりに，もっと単純な潜在する欠陥の除去や，逆にもっと高度な機能の変更・拡張を，保守と呼ぶのである．

13.3.2 保守作業の大きさ

ソフトウェアの保守のこのような特性から，保守の重要性は高く，それに要するコストがシステム総費用のうちで占める割合は大きい．その比率は，Boehm による 70%という予想値がよく引用されるが，調査に基づいた根拠ある数値としては，Lientz & Swanson による 50%というものがある．それにしてはソフトウェア工学における保守の扱われ方は比較的軽く，実践上も開発ほどには戦略性が意識されず，計画性に欠け，能力のある人材が投入されにくい．しかし，開発がソフトウェアのフローを作るものであるのに対し，保守はストックに依存する．しかも，このストックは他の資産と異なり，常に変化が要請され，また実際変化していくという特性を持つ．したがって，企業というミクロベースでも，社会というマクロベースでも，ソフトウェアストックの蓄積が進むにつれ，保守作業がますます大きな問題となるのは明らかである．

13.3.3 保守のプロセス

従来のソフトウェアのライフサイクルモデルは開発中心であり，落水型のモデルが典型的にそうであるように，保守フェーズはライフサイクルの最後に付け加えられた形で考えられることが多かった．しかし，ソフトウェアのより長期的な発展プロセスをとらえることに目を向ける必要がある．

実際，次のような要因は，開発と保守をさらに融合させたライフサイクルモデルを考慮すべきことを示唆している．

1. 開発プロジェクトといっても，まったく未経験でゼロからすべてを作るというケースは，非常に少ない．
2. 保守作業も,（変更）要求の分析定義，設計，プログラミング，テストと開発と同様の工程をたどる．すなわち，通常の開発プロセスに既存システムの理解という作業がさらに加わったものが保守プロセスと見ることもできる．
3. 長期に渡るソフトウェア開発では，開発の途中でユーザの要求変更や設計変更がしばしば起こる．これは開発工程の中で，要求仕様や設計仕様の保守作業が発生しているのと同等である．逆に，継続的なソフトウェアの保守による進化プロセスは，開発が引き延ばされて続いている状態と見ることもできる．

たとえば逐次進化型モデルは，このような開発と保守を融合させたライフサイクルモデルの一つと見てもよい．さらに極端プログラミング (XP) とか軽快プロセス (agile process) という新たなプロセスモデルは，永続的に発展するプロセスを意図していると見ることもできる．

保守の中には故障やユーザからの要求により，稼働中のシステムに迅速に手を加え，サービスレベルを保つという，長期の開発にはない特徴を持った作業も存在する．これらは開発よりむしろ運用に近い作業である．

さらに，開発/保守環境と運用環境の境界が喪失しつつあることにも，注目すべきであろう．すでに述べたように，分散環境で動作している応答型のシステムでは，それに「保守」を加え，また新たな機能を付け加えることも，運用と並行して行なわざるをえなくなる．すなわち，これまでのようにまず開発/保守環境でプログラムを作成し検証して，その後，運用環境へ移行するという段階を踏んだ作業は現実的でなくなってくる．コンポーネントウェアのような開発環境は，その意味で運用環境と同化しつつあるといえよう．

13.3.4 保守の分類

保守の形態分類は種々のものが提案されているが，もっともよく参照されるのが Swanson による次の３分類である [67].

1. 修正保守：　エラーを修正する保守

2. 適応保守：　ハードウェア，ＯＳ，周辺装置，データ仕様などの変更に応じてシステムを適応させるための保守

3. 完全化保守：　機能の拡張や効率の向上を目的とする保守

この他に，信頼性や保守性を高めるためにシステムの構造をよくするというような予防保守を４番目の種類として挙げることもある．また，機能拡張のための保守は利用環境の変化に応じるものであるとしてこれを適応保守と呼び，機能を変更しない効率の向上や保守性の向上のための保守を完全化保守と呼ぶべきだとする意見もある．

13.3.5　保守の体制

保守をどのような組織体制で行ったらよいかについては，議論の分かれるところである．基本的には，保守専用の組織を開発部隊と独立に作るか，それとも保守は開発組織が並行して行うかという選択がある．実際には，保守を完全に独立した組織としている例は，少ない．しかし，組織を独立させることによる効果は大きいという報告もある [102].

保守の組織を別にしていなくても，要員を開発担当と保守担当とに分けている例は少なくない．担当区分を長い期間固定する場合と，ある程度短い期間でローテーションする場合とがある．同じ要員が開発作業と保守作業を同時期に並行して持つ方が，むしろ少ないであろう．

このように保守を独立させる程度は，組織をまったく別にするものから要員レベルでも保守と開発を並行するというものまで，さまざまな段階がある．保守の独立性が高い方が，

1. 保守と開発の予算，要員，資源などが明確に区分でき，計画と管理の質が高まる．

2. 保守は利用者からの不定期な要求に対応するという面があり，作業負荷が一様でないため，開発担当者が兼任すると開発プロジェクト進捗への影響が大きく出るが，独立しているとその問題を避けられる．

3. 利用者の保守要求に対して，迅速できめの細かい対応がとり易い．

などの利点がある．一方，

1. 開発と保守との間に知識を受け渡すための負荷が高く，またその移行がうまく行かないためのトラブルが起こりがちである．

2. 保守の要員の士気や技術力の低下が起き易い．

といった欠点もある．

とくに，要員の動機づけの問題は管理者にとって頭の痛い問題である．保守に新人を割り当てるという比較的よく行われる方法は，本来開発より難しい面がある保守作業の生産性の悪化をもたらしているという批判がある一方で，能力のある人間に保守を担当させると士気低下によりせっかくの能力が生かされなくなるという危険がしばしば指摘される．

13.3.6　保守の技術とツール

保守作業の多くの部分は，開発と同様のものである．したがって分析，設計，プログラミング，テストというプロセスは，新規開発で用いられるものと共通した技術とツールが使われる．一方で，保守に固有の技術やツールもある．ここではそれらの代表的なものを挙げる．

余波分析

保守は既存のソフトウェアの一部に手を加えるものであるから，その変更あるいは追加の影響が及ぶ範囲を把握することは，基本的な要件である．ミクロレベルでは，たとえば１つのプログラム変数の値をある場所で変えた場合，その変数を参照しているプログラムの他の部分に影響が及びうる．もう少し大きな単位でいえば，一つ

のモジュールを変更した時，そのモジュールを呼び出している他のすべてモジュールが影響を受ける．このような影響を分析することを，余波分析という．

余波分析には，制御の流れ分析やデータの流れ分析といった手法が応用でき，それらに基づくツールも作られている．しかし，分析できる範囲に限界もあり，ツールの利用はそれほど進んでいない．

回帰テスト

保守作業の結果を検証するためのテストは，ます変更した部分が正しく作られているかを確かめるために行なわれる．同じく重要なのは，変更されなかった部分がもと通り正常に動くことを確かめることである．そのために保守を行なう前のシステムに対して適用されたテストの一部を再度実行し，前と結果が変わらないことを確かめる必要がある．このようなテストを，回帰テストという．

回帰テストでは，もとのすべてのテストケースを繰り返す必要はない．そこで上に述べた余波分析の結果を用いて，必要なテストを選択することが考えられる．ただ，必要最小限のテスト範囲を正確に求めることは，それほど容易ではない．回帰テスト用のツールは，過去のテストデータを管理し，それを再実行するのを支援する．テスト結果を過去のものと比較することも，ツールで自動化しうる．

プログラムスライシング

プログラムのある文の中のある変数について，実行時にその変数が持つ値に影響を与える文は，プログラム内のすべての文ではなく限定される．影響の与え方は 2 通りあり，データの定義参照関係の流れによって影響するものと，if 文や while 文などの制御の流れによって影響を与えるものとがある．これらの文を切り出した断片をスライスといい，それを切り出す手法をプログラムスライシングという．プログラムスライシングには，プログラムを静的に解析して得る方法と，動的に実行して得る方法とがある．

スライシングの結果は，保守のためのプログラムの変更によって影響を受ける部分を取りだし，それに関するテストを行なうというような形で利用される．

逆構築

逆構築 (reverse engineering) とは，エンジニアリングすなわち通常の製造開発の入力と出力の関係を逆転させたプロセスである．ソフトウェアでは，プログラムから設計仕様を回復することが，典型的な逆構築になる．機械語コードから原始プログラムコードを得ることや，詳細設計から概要設計を得ること，また設計から要求仕様を得ることも逆構築として考えられる．しかし，機械語コードから原始プログラムコードを得ることは現在では重要性が低く，一方，設計から要求仕様を得ることは技術的に難しいため，どちらもそれほど行なわれていない．

保守作業に際して起こりがちな問題は，原始プログラム以外に信頼できる文書が存在しないという事態である．開発時に作られた要求仕様や設計仕様という文書が残っていても，それらが現在のシステムの状況を正しく反映し，原始プログラムと対応するものであるかどうかが確かでない．この問題に対しては，開発時及び保守時に，常に仕様書類を最新の状態を表すように保つ管理を励行することが本来的な解決策であるが，現実には逆構築によってプログラムから仕様が逆生成できれば便利である．

しかし，プログラムから仕様を生成することは技術的に困難な点が多い．本質的には，プログラムに書かれていない設計情報を付加しないと，完全な設計の回復は無理であろう．ただ，仕様を完璧に生成しないまでも，プログラムの構造や意味を理解するための情報を提供する方法は，いろいろなものが考えられる．そのような手法は一般にはプログラム理解と呼ばれ，研究と一部で実用化が進められている．後で述べる再構築（リエンジニアリング）も，プログラム理解と関連が深い．

13.3.7　保守の戦略

保守はソフトウェアの長いライフサイクル全体にかかわる行為であり，そのためある意味では開発以上に戦略的な計画管理を必要とする．保守の戦略といった場合，次の２つが大きな意思決定対象となろう (V. Basili[12]).

1. 保守を続けるか，現在のシステムを廃棄して新たに作り直すかの判断
2. 個々の保守作業をどのような基準で行うかの判断．具体的には
    - 簡便型： 当面の保守作業の負荷を最小化するという戦略．すなわちパッチ式の保守
    - 構造保全型： ソフトウェアの構造をできる限り整った形に保つことを最重点においた保守

簡便型は短期的な費用は最小となるが，長期的にはソフトウェアの構造劣化により保守費用が加速度的に増大する可能性がある．構造保全型は短期的な費用と時間がかかるが，長命のソフトウェアでは後の保守費用を大いに軽減する効果がある．戦略としては，この２つの極端なものの適当な中間を選ぶこともできる．

この判断は，1. の保守か作り直しかという判断と相互に関連する．一般に，簡便型の保守を続ければ構造劣化により早めにつくり直さざるをえない．構造保全型の保守を実施していればソフトウェアの寿命は長くなり，作り直す必要が生じるのを先の延ばすことができる．しかし，ソフトウェアを作り直す必要が生じるのは，構造の劣化だけが要因ではない．たとえばハードウェアや OS の変更，あるいは利用者による機能変更や拡張の要請，法制や業務環境の変化への対応などにより，作り直さざるをえなくなることがしばしば起こる．その場合は，どうせ作り直すのだから簡便型の保守の方が総コストが安くすむということがありうる．すなわち，構造の劣化以外の要因で寿命が比較的短いと予想されるようなソフトウェアに対しては簡便型の保守を行い，寿命が長いと予想されるものには構造保全型の保守を行うことが，適切な戦略になってくる．

13.4　再利用

再利用技術がソフトウェア開発の生産性を高める鍵となると言われて久しい．しかし，実際に多くの実践例が積み重ねられ，またそのための方法論の議論や支援ツール・実例の報告が学会などの場で活発になされるようになったのは，1990 年代に入ってからである．

再利用の対象としてまず考えられるのはプログラム・コードであり，報告されている例ももっとも多い．しかし，要求仕様や設計も再利用の対象となりうるし，さらにソフトウェアの開発や保守のプロセスそのものも重要な再利用対象である．設計や開発プロセスの再利用は，これまでも個人やチームの経験・知識という形では当然のことながら行われてきた．しかし，それを明確に記述し計画的に再利用するという考え方は比較的新しく，プログラムの再利用と比べて未発達の状況である．

13.4.1　再利用のための技術

部品化

プログラムの再利用をする場合，既存のシステム全体を対象にするのであれば，それはシステムを「利用」しているのにすぎないから，再利用という以上なんらかの部分を再利用単位として想定していることになる．この再利用単位を一定の基準のもとに標準化したものを部品と呼び，それを集めたものを部品ライブラリという．

部品を整備するための部品化とその利用技術には，次のような課題がある．

1. どのような部品を用意すべきか.
2. 部品を検索するのはどのようにしたらよいか.
3. 検索した部品を必要に応じてどのように変換し，さらに組み立てればよいか.

ここではとくに 1 のどのような部品を用意すべきかという点について，考えてみよう．この問題は，部品の分類法にかかわる．まず事務処理，科学技術計算，さらに細分して会計処理，在庫処理，構造計算，グラフィックスと

いった分野による分類が考えられるが，これらは適用分野が定まれば自ずから決まるものであるから，一般に大きな問題とはならないであろう．実際，数学ソフトウェアなどの古くからあるライブラリは，分野別につくられた部品集合の例である．

次に，部品となるプログラム単位の大きさ，あるいは意味的な階層レベルで分類することが考えられる．サブシステムレベルをもっとも大きい単位として，モジュール（といってどのくらいのものを指すのかは設計法やプログラミング言語に依存してさまざまであるが），手続きや関数，マクロなどが想定されよう．これらのうちのどの単位で部品を構成すべきかは，一つのかなり大きな判断である．

また，部品の完成度，あるいは逆に再利用に際して必要とされる加工度で分類することが考えられる．ここで完成度というのは，部品をそのまま使う場合が完成度がもっとも高く，加工する割合が多ければ完成度が低いという意味であるから，完成度が高いということは，個別の再利用の要求に対して適応しにくく，柔軟性に欠けるという面があることも注意する必要がある．再利用に際して行われる加工は，パラメータを設定するというものから，指定された部分を個別の条件に合わせて書き込むものまで考えられる．パラメータを設定する方式も，マクロのような置き換えで行うものや，パラメータを与えられた生成系 (generator) が作動して特殊化されたプログラムを作り出すものまでがある．また，指定部分を書き込む方式は，部品として処理のパターンのみを与えているという場合には，骨格部品などと呼ばれる．

オブジェクト指向における再利用

オブジェクト指向技術はソフトウェア工学の観点からは，とくに再利用に対する効果が大きく期待されてきた．

オブジェクト指向の場合，オブジェクト（あるいはクラス）というまさに部品とするのに格好の単位がある．オブジェクトを部品とすることにより，上に述べたいくつかの問題のかなりの部分が解消する．たとえば，部品の大きさやレベルをどうするかという問題は自然に消える．部品を使う場合の加工度については，そのまま使うこともできるし，クラス継承という手段によって既に定義されているデータ（属性）や処理（メソッド）を利用しながらさらに特殊化したり変更したりすることも可能である．再利用されるクラス部品では仕様のみを定義して，具体化は個別のクラスやインスタンスを作るときに行うような，部品の用意の仕方も有効である．またクラスからインスタンスを生成する際（さらにはメタクラスからクラスを生成するという仕組みも考えられる），パラメータを指定して用途に応じたものを作り出せるようにすることも，自然にできる．すなわち，上に挙げた様々なケースが同じオブジェクト指向の枠組みの中で，無理なく実現できる．

オブジェクト指向はさらに，部品の検索と組み立てという問題に関しても，完全ではないが部分的な解を与えている．検索にはクラス階層の構造が役に立つし，部品同士の結合はオブジェクト間のメッセージ交換によって行われるので，必要な部品（オブジェクト）さえそろえれば，組み立てはある意味では自然に可能であり，機能の追加のために新たにオブジェクトを導入するにも，メッセージ交換のインターフェースに従えばよい．

オブジェクト指向に基づいたライブラリは，種々のものが作られている．汎用性の高い部品ベースだけでなく，個別の適用分野に対してもそのようなライブラリが作られ流通し始めている．しかし，一方で次のような問題も明らかになってきた．

1. クラス単位の再利用では単位が小さすぎて，再利用の効果が限定的である．
2. クラスライブラリが作る継承構造が，開発しようとする応用システムから導かれる構造に適合しない場合，個々のクラスを再利用できる可能性があっても，効果的に利用できない．

前者については，より大きな単位としてのフレームワークや設計パターンの再利用が促進されてきた．後者に関しては，一定の協調動作をするオブジェクト群や，たとえばセキュリティとかプライバシーとかいう特定の関心事を切り離し，それらを独立に記述した単位を再利用するという研究が進められている．さまざまな方法が提案されているが，それらを大きく局面指向ソフトウェア開発 (Aspect Oriented Software Development) と呼んで，目標を共有化し研究を進展させる動きが盛んになっている．

13.4.2　再利用を進めるための体制

個人が過去に作ったソフトウェアを自分で再利用するというのではなく，組織的な再利用を図るのであれば，部品ベースを整備するだけでは不十分で，再利用を進めるための組織的対応がとられなければならない．実際，再利用の普及に熱心な企業では，比較的早くから部品を登録し，利用するための体制作りに力を注いできた．そのような体制のもと，システムを開発した際の再利用率，グループや個人別の部品登録数，部品利用数，登録した部品の被利用数，といったデータがとられ，それらに基づいた表彰制度を導入するなどの方法で，再利用を促進するといった動機づけが有効であるようだ．

13.5　再構築

再構築 (re-engineering) は，システムを作り直す作業である．

13.5.1　再構築と逆構築の関係

再構築と逆構築の関係を描いた Chikofsky による図を図 13.5 に示す [27].

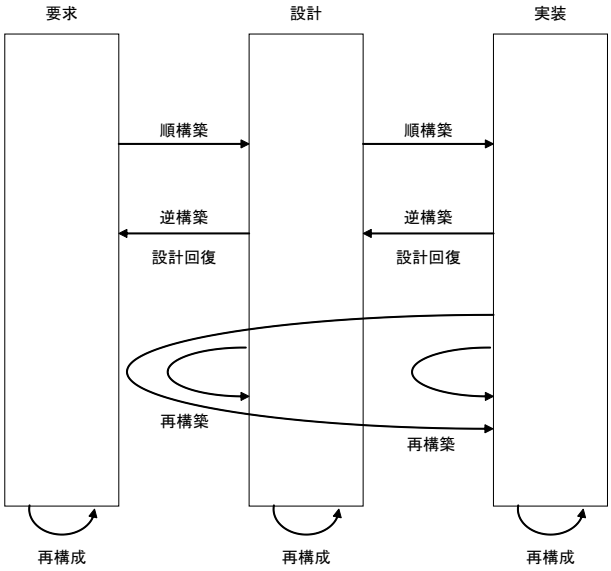


図 13.5: Chikofsky によるリエンジニアリングの位置づけ

CASE ツールに対する期待が一時非常に高まったが，その唱い文句の一つは保守に効果を発揮するということであった．しかしそれは，新たに行なうシステム開発に CASE を用いると，そこで作られるリポジトリ（要求仕様，設計仕様，データモデル，プログラム，テストケースなど，開発で生じる生成物を，それらの構造や関係についての情報と共にしまっておく倉庫）が，保守で役に立ちますということで，すでに存在し運用されていて，原始プログラム以外には確実な情報がなくなっているソフトウェアに対しては，直接適用できるものではなかった．そこで，既存のソフトウェアを解析してリポジトリに登録し，その構造や振舞いを分析したり，変更や再生を支援するという再構築ツールへの需要が生まれた．

13.5.2　再構築ツール Refine

再構築ツールの中で，商業的に成功したとは必ずしもいえないが，優れた技術に基づいたものとしてして Reasoning Systems 社の Refine を挙げることができる．

Refine は長い研究開発の歴史の所産として産まれた．一貫して指導的な立場にあったのは C. Green であり，歴

史的には次のような順序でプロジェクトが進められてきた.

PSI　C. Green が Standford 大学にいた時代，1970 年代の半ばに行なわれたプロジェクトである．PSI は自然言語による仕様記述からプログラムを自動的に合成するという，壮大な目標をもったプロジェクトで，いくつかのサブシステムが試作された．このプロジェクトから，David Barstow(現 TeraQuest Metrics 社), Elaine Kant(現 SciComp 社) などが育った.

CHI　C. Green は 1978 年に Systems Control Technology という組織を作り，1981 年にそれを改組して Kestrel Institute とした．この両研究機関を拠点として 1978-1984 に行なわれたプロジェクトが CHI である.

CHI の成果の中心は，DKB という開発対象プログラムに関する知識ベースの仕組みと，広範囲言語 (wide-spectrum language)V である．V は仕様記述にも使えるし，手続き的な記述も書ける.

Kestrel Institute からはその後，D. R. Smith により開発された KIDS(Kestrel Interactive Development System) という成果も出ている [99, 98]．KIDS は Reasoning Systems が提供する商用の知識ベースプログラミング開発環境 Refine の上で開発された研究システムで，探索型のアルゴリズムを用いる効率的なプログラムの生成などに成功している.

Reasoning Systems　C. Green はさらに 1984 年に，Reasoning Systems という会社を作った．ねらいは，CHI で開発された技術を，産業界に適用しようというものである．そのためのツールとして Refine が作られた.

Refine の当初の目的は，仕様記述とプログラム開発の自動化にあった．ただ，ツールとして販売するというより，顧客企業の生産性向上などを目標としたコンサルティング業務を請け負い，その際に道具として Refine を利用するという形態だったようだ.

しかし，やはりそのようなアプローチは産業界になかなか受け入れられにくく，1990 年代に 2000 年問題への対処を念頭において，再構築への適用に方向転換して，ある程度の成功を収めたようである．ツールとしての Refine も，当初はトップダウンにソフトウェアを開発する自動プログラミングの正統的なツールとして使われることを目指しながら，ボトムアップの再構築という分野に実用的な利用法を見い出すに至ったことになる．それを可能にしたのが，Dialect という構文解析系生成系である．これを用いて原始プログラムから分析操作の対象となるオブジェクトベースを作り，それに操作を施すことで，自由な “リエンジニアリング” が可能となる．その際，とくに言語としての Refine がもつプログラム変換機能が有効に働く.

Refine の構成　Refine は次のような要素から構成されている.

- **Refine**　同じ名前の言語 Refine で記述された仕様やプログラムをコンパイルする．また，それによって作られるオブジェクトベースに種々の操作を行なうことができる．言語としての Refine は CHI プロジェクトで作られた V の直接の後継言語で，論理式，集合や写像を直接記述でき，しかもある程度の効率で実行ができる．さらにオブジェクトベースへの操作を変換規則として記述でき，とくに対象オブジェクトが Refine プログラムの構文解析木であれば，プログラム変換が実現できる.
- **Dialect**　言語の構文を記述すると，それをもとに構文解析系 (parser) が生成できる．C, Cobol, Ada などは，既にできあいのものが提供される.
- **Intervista**　会話的なユーザインターフェースをつくるための道具を提供する.

この 3 つをまとめて，Refinery と呼ぶ.

またこの上に作られた各種言語用の CASE ツールがある．Refine/C, Refine/Cobol, Refine/Ada, Refine/Fortran など．これらはそれぞれの言語で書かれた原始プログラムを解析したり，またカスタマイズすることにより必要な形式に変換することを支援するツールで，まさに再構築用の機能を持つものといえる.

関連図書

[1] Unified Modeling Language 1.5. http://www.omg.org/technology/documents/formal/uml.htm.

[2] IEEE Transactions on Software Engineering, Vol.SE-2, No.1, 1977.

[3] Guide to the software engineering body of knowledge: SWEBOK. http://www.swebok.org, 2001.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[5] 鰺坂恒夫, 池田健次郎, 中谷多哉子, 野呂昌満. OO'97 オブジェクト指向モデリングワークショップ報告. 情報処理学会研究報告 97-SE-116, pp. 33–43, 9 月 1997.

[6] C. Alexander. *The Timeless Way of Building.* Oxford University Press, 1979.

[7] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[8] V. Ambriola, P. Ciancarini, and C. Montangero. Software process enactment in oikos. In *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*, pp. 183–192. ACM, 1990.

[9] A. Aoki. Smalltalk textbook. http: //www.sra.co.jp /people /aoki /SmalltalkTextbook /index.html.

[10] 有沢誠. アルゴリズムとその解析. コロナ社, 1989.

[11] G. Barrett. Formal Methods Applied to a Floating-Point Number System. *IEEE Trans. Softw. Eng.*, 15(5):611–621, 1989.

[12] V. R. Basili. Viewing maintenance as reuse-oriented software development. *IEEE Software*, pp. 19–25, January 1990.

[13] K. Beck. *eXtreme Programming eXplained*. Addison-Wesley, 2000.

[14] L. A. Belady. The Japanese and software: Is it a good match? *IEEE Computer*, pp. 57–61, June 1986.

[15] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[16] J. Bentley. *Programming Pearls*. Addison-Wesley, 1986. 邦訳：野下浩平 訳「プログラム設計の着想」, 近代科学社.

[17] J. Bentley. *More Programming Pearls*. Addison-Wesley, 1988. 邦訳：野下・古郡 訳「プログラマのうちあけ話—続・プログラム設計の着想—」, 近代科学社.

[18] S. J. Blackmore. *The Meme Machine*. Oxford University Press, 1999.

[19] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, N.J., 1981.

[20] G. Booch. *Object-Oriented Analysis and Design with Applications 2nd Editon*. Benjamin/Cummings, 1994.

[21] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, 1999.

[22] S. Brock and C. George. *RAISE Method Manual (RAISE/CRI/DOC/3/V1)*. Computer Resources International, 1990.

[23] F. P. J. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.

[24] F. P. J. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Software*, pp. 10–19, April 1987.

[25] F. P. J. Brooks. *The Mythical Man-Month: Essays on Software Engineering Anniversary Edition.* Addison-Wesley, 1995.

[26] J. R. Cameron. An overview of JSD. *IEEE Transactions on Software Engineering*, SE-12(6):222–240, 1986.

[27] E. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pp. 13–17, January 1990.

[28] E. M. J. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[29] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Prentice Hall, 1990.

[30] C. S. Committee. Computing curricula 2001 software engineering volume. http://sites.computer.org/ccse/, 2003.

[31] A. M. Davis. *Software Requirements: Objects, Functions and States*. Prentice-Hall, 1993.

[32] R. Dawkins. *The Selfish Gene*. Oxford University Press, 1976.

[33] T. DeMarco. *Structured Analysis and System Specification.* Prentice Hall, 1978. 邦訳，高梨智弘，黒田純一郎（監訳）：構造化分析とシステム仕様，日経マグロウヒル，1986.

[34] R. Diaconescu and K. Futatsugi. *CafeOBJ Report — The Langauage, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. World Scientific, 1998.

[35] E. W. Dijkstra and W. H. J. Feijen. *A Method of Programming*. Addison-Wesley, 1988. 邦訳：玉井浩 訳「プログラミングの方法」，サイエンス社.

[36] A. Dorling. Spice: Software process improvement and capability determination. *Information and Software Technology*, 35(6/7):404–406, 1993.

[37] D. E. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1999.

[38] A. P. Ershov. Aesthetics and the human factor in programming. *Communications of ACM*, 15(7):501–505, 1972.

[39] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[40] J. Fitzgerald and P. G. Larsen. *Modelling Systems*. Cambridge University Press, 1998. 邦訳: 荒木啓二郎 他訳「ソフトウェア開発のモデル化技法」，岩波書店，2003.

[41] M. Fowler and K. Scott. *UML Distilled: Applying the standard object modeling language*. Addison-Wesley, 1997.

[42] D. P. Freedman and G. M. Weinberg. *Handbook of Walkthroughs, Inspections, and Technical Reviews*. Little, Brown and Company, Boston, Mass, 1977.

[43] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[44] D. Gries. *The Science of Programming*. Springer-Verlag, 1981. 邦訳：筧捷彦 訳「プログラミングの科学」培風館.

[45] A. Hall. Seven myths of formal methods. *IEEE Software*, September 1990.

[46] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[47] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts*. McGraw-Hill, 1998.

[48] I. Hayes. *Specification Case Studies, 2nd ed.* Prentice-Hall, 1992.

[49] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[50] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):1–17, 1997.

[51] K. E. Huff and V. R. Lesser. A plan-based intelligent assistant that supports the process of programming. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 97–106. ACM, Nov. 1988.

[52] 飯塚悦功. ソフトウェアの品質管理と品質保証. 情報処理, 33(8):922–933, 1992.

[53] K. Inoue, T. Ogihara, T. Kikuno, and K. Torii. A formal method for process descriptions. In *Proc. 11th International Conference on Software Engineering*, pp. 145–153, May 1989.

[54] 石畑清. アルゴリズムとデータ構造, 岩波講座 ソフトウェア科学, 第 3 巻. 岩波書店, 1989.

[55] 伊東暁人. 大規模システム開発におけるプロジェクト管理問題. ソフトウェアシンポジウム'91 論文集, D9-D16, 1991.

[56] M. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudice*. Addison-Wesley, 1995. 邦訳. 玉井哲雄, 酒匂寛 訳: ソフトウェア博物誌 — 世界と機械の記述, トッパン, 1997.

[57] M. A. Jackson. *Principles of Program Design*. Academic Press, 1975. 邦訳: 鳥居宏次 訳「構造的プログラム設計の原理」, 日本コンピュータ協会, 1980.

[58] M. A. Jackson. *System Development*. Prentice Hall International, 1983. 邦訳： 大野・山崎 監訳, システム開発法—JSD 法, 共立出版, 1989.

[59] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, Reading, 1999.

[60] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM press, 1992. 邦訳, 西岡, 渡辺, 梶原, 監訳, オブジェクト指向ソフトウェア工学 OOSE, アジソン ウェスレイ・トッパン, 1995.

[61] C. B. Jones. *Systematic Software Development using VDM, 2nd ed.* Prentice Hall, 1990.

[62] G. Kaiser, P. Feiler, and S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, 5:40–49, May 1988.

[63] T. Katayama. A hierarchical and functional software process description and its enaction. In *Proc. 11th International Conference on Software Engineering*, pp. 343–352. IEEE, May 1989.

[64] M. Kellner. Software process modeling support for management planning and control. In *1st International Conference on the Software Process*, pp. 8–28, Redondo Beach, CA, Oct. 1991.

[65] B. W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, New York, 1974.

[66] M. M. Lehman and L. A. Belady. *Program Evolution: Processes of Software Change*. Academic Press, 1985.

[67] B. P. Lientz and E. B. Swanson. *Software Maintenance Management*. Addison-Wesley, 1980.

[68] J. Magee and J. Kramer. *Concurrency – State Models & Java Programs*. John Wiley & Sons, 1999.

[69] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall International, 1988. 邦訳，二木 監訳，酒匂 訳, オブジェクト指向入門，アスキー，1991.

[70] B. Meyer. *Object-Oriented Software Construction 2nd Edition*. Prentice Hall International, 2000.

[71] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psycological Review*, 63:81–97, 1956.

[72] H. Mills, M. Dyer, and R. Linger. Cleanroom software engineering. *IEEE Software*, pp. 19–25, Sept. 1987.

[73] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[74] 村上憲稔. ソフトウェアのライフサイクル管理. 情報処理, 33(8):912–921, 1992.

[75] 中島震，玉井哲雄. EJB コンポーネントアーキテクチャの SPIN による振舞い解析. コンピュータソフトウェア, 19(2):2–18, 2002.

[76] S. Nakajima and T. Tamai. Behavioural analysis of the enterprise javabeanstm component architecture. In *Model Checking Software–Proceedings 8th International SPIN Workshop*, pp. 163–182, Toronto, Canada, May 2001.

[77] T. Nakatani and T. Tamai. Empirical observations on object evolution. In *Asia-Pacific Software Engineering Conference (APSEC'99)*, pp. 2–9, Takamatsu, Japan, Dec. 1999.

[78] T. Nakatani, T. Tamai, A. Tomoeda, and H. Matsuda. Towards constructing a class evolution model. In *Asia-Pacific Software Engineering Conference*, pp. 131–138, Hong Kong, December 1997.

[79] 野木兼六. 最適化に基づくアルゴリズムの発見. コンピュータソフトウェア, 11(4):20–43, 1994.

[80] 落水浩一郎. ソフトウェアプロセスモデルに基づくソフトウェア開発支援環境 vela. 日本ソフトウェア科学会第 7 回大会論文集, pp. 205–208, 1990.

[81] 大野佝郎. ジャクソンシステム開発法. 情報処理, 25(9):955–962, 1984.

[82] L. Osterweil. Software processes are software too. In *9th International Conference on Software Engineering*, pp. 2–13, Apr. 1987. Monterey CA.

[83] S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. PVS: an experience report. In D. Hutter, W. Stephan, P. Traverso, and M. Ullman eds., *Applied Formal Methods—FM-Trends 98*, Vol. 1641 of *Lecture Notes in Computer Science*, pp. 338–345, Boppard, Germany, Oct. 1998. Springer-Verlag.

[84] D. L. Parnas. Software aspects of strategic defense systems. *Communications of ACM*, 28(12):1326–1335, 1985.

[85] B. Peuschel and W. Schäfer. Concepts and implementation of a rule-based process engine. In *Proceedings of the 14th International Conference on Software Engineering*, pp. 262–279. IEEE, 1991.

[86] S. L. Pfleeger. *Software Engineering – Theory and Practice*. Prentice-Hall, 1998.

[87] M. Phillips. CICS/ESA 3.1 experiences. In J. E. Nicholls ed., *Z User Workshop: Proceedings of the Fourth Annual Z User Meeting, Oxford 1989*. Springer-Verlag, 1990.

[88] C. Potts, K. Takahashi, and A. I. Anton. Inquiry-based requirements analysis. *IEEE Software*, 11(2):21–32, 1994.

[89] R. Pressman. *Software Engineering : A Practitioner's Approach – 5th edition.* McGraw-Hill, 2001.

[90] S. J. Prowell, R. C. Linger, C. J. Trammell, and J. H. Poore. *Cleanroom Software Engineering: Technology and Process*. Addison-Wesley, 1999.

[91] W. W. Royce. Managing the development of large software systems. In *Proceedings WESCON*, Aug. 1970. reprinted in the *Proceedings 9th International Conference on Software Engineering*, 1987.

[92] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lonrensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991. 邦訳，羽生田監訳，宮追，中谷，桜井 他 訳，オブジェクト指向方法論 OMT，トッパン，1992．

[93] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[94] M. Shaw. Prospects for an engineering discipline of software. *IEEE Software*, 7(6):15–24, November 1990.

[95] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an emerging discipline*. Prentice-Hall, 1996.

[96] S. Shlaer and S. J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Prentice Hall, 1988. 本位田真一，他訳,「オブジェクト指向システム分析」，啓学出版，1990．

[97] S. Shlaer and S. J. Mellor. *Object Lifecycles: Modeling the World in States*. Prentice Hall, 1992. 本位田真一，他訳,「続・オブジェクト指向システム分析: オブジェクト・ラ イフサイクル」，啓学出版，1992．

[98] D. R. Smith. Kids: A semiautomatic program development system. *IEEE Trans. Software Engineering*, 16(9):1024–1043, 1990.

[99] D. R. Smith. Kids: A knowledge-based software development system. In M. R. Lowry and R. D. McCartney eds., *Automating Software Design*, pp. 483–514. AAAI Press/MIT Press, 1991.

[100] I. Sommerville. *Software Engineering (6th edition).* Addison-Wesley, 2001.

[101] J. M. Spivey. *The Z Notation — A Reference Manual, Second Edition*. Prentice Hall, 1992.

[102] E. B. Swanson and C. M. Beath. Departmentalization in software development and maintenance. *Comm. ACM*, 33(6):658–667, 1990.

[103] T. Tamai. Current practices in software processes for system planning and requirements analysis. *Information and Software Technology*, 35(6/7):339–344, 1993.

[104] T. Tamai. Process of software evolution. In *2002 International Symposium Cyber Worlds: Theory and Practices (CW2002)*, pp. 8–15, Tokyo, Japan, Nov. 2002. IEEE.

[105] T. Tamai and A. Itou. Requirements and design change in large-scale software development: Analysis from the viewpoint of process backtracking. In *15th International Conference on Software Engineering*, pp. 167–176, Baltimore, Maryland, U.S.A., May 1993.

[106] T. Tamai and Y. Torimitsu. Software lifetime and its evolution process over generations. In *Proc. Conference on Software Maintenance – 1992*, pp. 63–69, Orlando, Florida, November 1992.

[107] 玉井哲雄. 要求定義とプロトタイピングの現状. 情報処理学会「プロトタイピングと要求定義」シンポジウム, pp. 103–110, 4 月 1986. 招待論文.

[108] 玉井哲雄，鳥光陽介. 世代をまたがるソフトウェア進化プロセス — ソフトウェアの寿命と作り直しに関する考察 —. 日本ソフトウェア科学会第 9 回大会論文集, pp. 53–56, 1992.

[109] 玉井哲雄. ソフトウェアの語源. *bit*, 33(1):54–57, 2001.

[110] F. C. Taylor, R. N.and Belz, L. A. Clarke, L. J. Osterweil, R. W. Selby, J. C. Wileden, A. Wolf, and M. Young. Foundations for the arcadia environment architecture. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 1–13. ACM, Nov. 1988.

[111] K. Torii. Analysis in software engineering. In *Proceedings of the 1994 Asia-Pacific Software Engineering Conference*, pp. 2–6, Tokyo, 1994.

[112] 鳥光陽介, 玉井哲雄. ソフトウェア・システムの寿命とその要因についての考察. ソフトウェアシンポジウム '92 論文集, pp. E–2–E–10, 長野, 1992. ソフトウェア技術者協会.

[113] A. van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings RE'01, 5th IEEE International Symposium on Requirements Engineering*, pp. 249–263, Toronto, Aug. 2001.

[114] J. Warmer and A. Kleppe. *The Object Constraint Language : Precise Modeling with UML*. Addison-Wesley, 1998.

[115] J. M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, pp. 8–24, September 1990.

[116] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall, Englewood Cliffs, 1990.

[117] 山崎利治. プログラムの設計. 計算機科学／ソフトウェア技術講座 3. 共立出版, 1990.

索引