ソフトウェア工学 74－1
（1990. 9. 11）

ｄｍＣＡＳＥを用いた リフト問題の設計プロセスについて

大林 正晴

㈱管理工学研究所

最近、ワークステーション上の高度なＧＵＩを利用したＣＡＳＥ環境が、広く研究されているが、筆者らは、概念による設計法ＤＭＣに基づくＣＡＳＥ環境ｄｍＣＡＳＥを開発している。ｄｍＣＡＳＥでは、図的モデル（概念構造図）と数学的モデル（ＭＬ：ラムダ計算）を組合せてモデリングする手法を採用している。

この環境上で、リフト問題の記述実験を行なった。まず、設計方法論ＤＭＣに従って机上で設計作業を行ない、その時の設計プロセスを記録し分析した。つぎに、ｄｍＣＡＳＥ上で同様の作業を行ない、設計プロセスの観点からツールを評価した。また、設計プロセスを工学的な立場から考察し、独創的で、よい設計を行なうための要件や原則をまとめた。

Design Process of Solving the Lift Problem on dmCASE

Masaharu OBAYASHI

KANRI KOGAKU, LTD.
2-2-2, Sotokanda Chiyoda-ku, Tokyo, 101 JAPAN

We have been developing the software design tool: dmCASE that is based on our proposed methodology: DMC. There are two modeling facilities on dmCASE. One is the conceptual diagram as a graphical model and the other is the lambda calculus: ML as a mathematical model.

We have solved the lift problem on dmCASE. We took a note of the activities of the designing process to show how to solve the problem with DMC. Finally, we discuss the creative activities and the principles of the software design process.

１．はじめに

今日の科学技術の進歩は、めざましいものがある。工学分野全般について言えば、より複雑で高度な構造物を創り出すための基盤となる技術が確立されつつある。

例えば、超高層ビルを建てるには、様々な要素技術が使われている。それらは、設計技術や工法などの長年に渡る技術改良の蓄積に裏付けられている。また、電子工学は、超ＬＳＩの発達により、飛躍的に進歩し、多くの複雑な構造物を微細な空間に創り出すことができるようになった。

ところで、ソフトウェア工学は、ハードウェアの利用価値を高める上からもますます重要になってきている。ソフトウェアを目で直接認識することは難しいが、その実体は、情報を操作する巨大な構造物と見なすことができる。このような複雑なソフトウェアの信頼性を高めることが、高度情報化社会には不可欠であり、そのためのソフトウェア開発の工学的な基礎技術の確立が望まれている。

筆者らは、より工学的で科学的にソフトウェアを開発するための設計方法論ＤＭＣとその支援環境ｄｍＣＡＳＥを研究している。その一環として、リフト問題の記述実験を行ない、設計プロセスについて考察した。

２．ＤＭＣとｄｍＣＡＳＥ

ＤＭＣやｄｍＣＡＳＥの概要については、文献［７］を参照されたい。ここでは、背景となっている基本的な考え方を述べる。

２．１ モデリング

システムとは、いわば組織であり、システム設計は組織をつくることである。われわれは、日常、職場の組織の中でチームを組んで仕事を行なっているが、ソフトウェアで作るシステムもそのような組織と類似している。そのためには、中心的な事柄を擬人化して考えると都合がよい。

例えば、会社では、部長のもとに何人かの課長がいて、さらに、係長や個々の仕事の担当者が複数人いると言った具合に役割分担がされている。形式的には、単純な階層構造をしているが、実際の業務を遂行していくには、それらの役職にふさわしい役割を果すだけではなく、相互にかかわり合いをもちながら複雑な仕事をこなしていかなければならない。

現実には、必ずしも期待した役割を果さなかったりコミュニケーションがうまく行かなかったり、外部の要因などでいろんな問題や障害が生じるものである。この点は、われわれの作成するソフトウェアに関しても似たようなことがいえるのではないかと思う。作成したプログラムは、設計者の意図通りにはなかなか動いてくれない。蛇足だが、ソフトウェアのバグは、人間の組織と同様に人間が関与している限り宿命的なものなのかもしれない。

ところで、システム設計作業は、そのような擬人化した人の配置や役割分担を決定する過程であると言換えることができる。いきなり、そのような人の配置や役割分担が分かるわけではない。そのようなときは、処理方式あるいはアルゴリズムが明確になっていない場合が多い。そこで、まず、要求や目的をはっきりさせ、処理方式（アルゴリズム）を大まかに決める必要がある。

要求や処理方式がある程度固まってきてから、それを機械化するためのモデル化を始めることになる。

２．２ 設計指針

ＤＭＣでの設計は、分析した要求や処理方式から、擬人化した担当者を決めていくのである。当然、それぞれの人にふさわしい名前をつけなければならない。その人の役割を連想できる言葉を選択し名前とする。そのような言葉は、設計に関係する者の共通認識を得るための手段として重要な役割を果たす。言葉が指し示す共通の認識がつまり概念であり、言葉を用いて構造を組み上げていく手法がＤＭＣである。

概念に着目して、仕事を分担することにより、同種の事柄は、１つの概念としてまとめられたり、近い位置関係に配置されることになるであろう。

ＤＭＣでは、概念分割の指針として、つぎのようなガイドラインを考えている。

・役割、機能を果す部品を概念として想像し、全体を、概念の集まりとして構成する。
・概念相互の関連性に着目し、概念間の関連性が強いか弱いかの度合い（距離）を意識する。また、上位の概念から下位の概念の順に大まかな、自然な配置を見つけ、それを基準とする。
・さらに、処理アルゴリズムを考えながら相互依存性をより明確にし、概念の構成や配置を修正していく。
・概念相互のコミニュケーションの仕方を決定する。相手の概念に仕事を依頼する場合に、必要な情報を受け渡し、仕事の成果を受け取る。
・仕事の内容を連想させる名前を、依頼メッセージ（仕事名、関数名）とする。
・情報は、いくつかのかたまりに分類（引数）して引渡す。つまり、そのように分類した情報に適切な名前をつけたものをデータ型として認識する。
・ある仕事をどの概念に担当させるかは、構造を決める重要な作業である。一般には、つぎのことに留意して、それらを決めるとよい。
－共通な情報を集中化する。
－１つの概念があまり多くの仕事をやりすぎない。
－極端に、仕事の負荷が集中しないようにする。
－負荷が大きい場合には、部下の担当者を増やし、仕事を分担させる。
－概念から連想される内容にふさわしいことだけをその概念で行なう。
－与えられた仕事がその組織で効率よく実行可能であるかを考える。

２．３ ＣＡＳＥツールとしてのｄｍＣＡＳＥ

ｄｍＣＡＳＥ［７］は、方法論として概念による設計法ＤＭＣと形式言語としてＭＬを採用したＣＡＳＥツールである。

筆者らは、ＣＡＳＥ環境として大事な要件は、つぎの点であると考えている。

① 設計方法論が明確であること
② モデル化のための支援機能を備えていること
③ 設計したモデルを種々の角度から解析、評価できること
④ 一連の設計作業を統合的な操作で行なえること

設計の中心的な作業は、モデリングである。ｄｍＣＡＳＥでは、対象を表現するために２つのモデルを採用している。全体の関連を大局的に表現する図的モデルとして概念構造図を導入し、局所的な詳細を表現するための計算モデルとして形式言語ＭＬ（ラムダ計算モデル）を用いている。

その他の特性をまとめるとつぎのようになる。

用語、辞書管理	単語表、ｲﾝｽﾀﾝｽ定義表
図的モデル	概念構造図（ﾁｯﾌﾟ、ﾎﾞｰﾄﾞ）
数学的モデル	ＭＬ（ﾗﾑﾀﾞ計算）
品質管理	ｶｰﾄﾞ、ｶｰﾄﾞﾊﾟﾈﾙ
構成管理	ﾒﾓ帳
データフロー解析	回路図、回路図ﾏｯﾌﾟ、ｽｺｰﾌﾟ
コード生成	ｺﾝﾋﾟﾈｰﾀｺｰﾄﾞ

工程管理や状態遷移解析などは、現在はサポートしていない。

３．リフト問題

ここでは、まず、リフト問題［８］を示す。
リフト問題（原文は英語）は、つぎのように与えられている。下線は、特に注目した単語である。

ｎ台のリフトシステムがｍ階立のビルに設置される。リフトと制御機構は、製造元によって供給される。それらの内部機構は、あたえられると仮定する。問題は、リフトをつぎの制約のもとで各フロア間を移動させるロジックに関してである。

① 各リフトは、各フロアに対応するボタンのセットをもつ。それらのボタンは、押されたときに明るくなり（点灯し）対応するフロアにリフトを向わせる（移動）要因となる。対応するフロアにリフトが到着したら暗くなる（消灯する）。
② １階と最上階を除いて、各フロアには２つのボタンがあり、１つはリフトを上に、もう１つはリフトを下に移動するように要求するためのものである。それらのボタンは、押されたとき明るくなる（点灯する）。リフトがフロアに到着し、望んだ方向に動き出したときか、未対応の要求がない場合に暗くなる（消灯する）。後者の場合は、フロアの両方の要求ボタンが押されていれば、１つだけが無効（消灯）になる。どのサービスを先に行なうか（サービス順位）を決めるアルゴリズムは、両方の要求に対し待ち時間が最小になるようにする。
③ サービス要求がないとき、リフトは最後の目的地で停止しドアを閉じて、つぎの要求をまつ。（または、定められた待機フロアで停止するようにせよ）。
④ フロアからのリフトに対する要求は、すべてのフロアで同じ優先度で、最後にはすべてがサービスをうけなければならない。（このことを証明あるいは実演できるか？）
⑤ リフト内のフロアに対する要求は、移動の方向順に各フロアがサービスを受け、最後にはすべてがサービスを受けなければならない。（このことを証明あるいは実演できるか？）
⑥ 各リフトは、非常用のボタンをもち、それが押されたときは、警告の信号が管理者に送られる。そのとき、リフトはサービス中止になる。各リフトは、サービス中止状態を無効にする機構をもっている。

４．実験の概要

実際の設計プロセスについて述べる。できるだけ思考過程が分かるようにするために、記述の最終結果ではなく途中の履歴を設計作業の流れにしたがってできるだけ記録に残すことにした。実験は、まず、机上でワープロを使って行ない、後にその結果をｄｍＣＡＳＥ環境で記述してみた。

４．１ 机上での記述実験

（１）最初に何を考えたか
まず、問題を理解するために、与えられた問題文（英文）を日本語に翻訳した。その過程で、システムの全体像を頭の中で想像した。具体的には、５０階建てぐらいのビルに数十台のリフトが、スケジューリングされて、フロアなどからのリアルタイムな要求に応じてサービスするシステムを考えた。
つぎに問題文から、言葉を概念として取り出す。このときには、この問題を他人に説明するときに、必要な言葉が何かを考えて選択する。
一般にＤＭＣでは、それぞれの概念は、その概念に特有の状態が付属していると考える（ｄｍＣＡＳＥでは、最終的に関数型言語ＭＬで記述する）。また、擬人化して考えるので、それぞれの概念がいくつかの機能（メッセージをもつ）を果たし、それらを起動するのが別の概念や外からの呼び出し、つまり、イベントである。
例えば、リフト問題では、サービスと言う概念を考えた。それには、サービスの状態として稼働状況というものがあり、イベントとしては、要求、中止、解除といったメッセージを受け付けるものと考えた。

（２）スケジューリング問題を意識したのはいつか
問題文を読んだ段階で、スケジュール問題を含んだ最適制御問題として捉えた。つまり、ｎ台のリフトを効率よく動かすことが目的であると考えた。
しかし、スケジュールの詳細は、最後に検討することにし、それ以外の要素の構造から決定していこうという方針を立てた。

（３）記述過程でどのようにバックトラックしたか
実際の記述では，設計のプロセスをできるだけ忠実にトレースするようにした。
まず、最初の単語の抽出、概念構造図の作成を行ない。大まかな構成を決めた。つぎに、最も上位の概念の役割を詳細に考えた。要求される仕事を行なうには、下位の概念にどのような役割が必要かが分かってくる。
単語の整理を行ない、最初の概念構造図、インスタンス定義表を作成したのち、つぎからは、２、３のインスタンスの仕様記述を行なった。その後、概念構造図を変更、修正するという作業を繰り返した。最後の概念構造図は、初期のものからかなり変化したが、仕様記述の後戻りは、２、３の機能の追加や引数の変更程度ですんだ。
また、概念構造図で全体の構成や動きを頭の中で常に考えながら設計を進め、詳細を上位から下位のインスタンスの順に決めていくことが容易にできた。
これらの点は、形式言語ＭＬの特性も反映していると考えられる。つまり、上位の記述では、下位のデータ型の詳細を意識せずに、抽象化したデータ型で処理することが可能だからである。

（４）スケジューリングのアルゴリズムをどう考えたか
最適なリフトを選択する規則としては、例えば、以下の規則を順に適用するものと考えた（記述はしなかった）。
① 階から逆方向で一定距離内にあり、同方向に移動中のもので停止予定がないリフト
② 階から逆方向で一定距離内にあり、同方向に移動中で停止予定があるリフト
③ 階から逆方向で一定距離内にあり、停止中のリフト
④ 階から逆方向で、移動中のものがないとき、最も近い場所に停止しているリフト
⑤ 階から順方向で、移動中のものがないとき、最も近い場所に停止しているリフト
⑥ 階から逆方向で一定距離だけ前（方向とは逆）の場所を最も先に通過したリフト

（５）問題の範囲をどこまで考えるか
実際のリフトの制御では、駆動装置やセンサなどとのリアルタイムな処理が問題となる。問題文には、そこまでの詳しい記述はない。まじめにやると現実のリフト制御の専門的な知識が要求され、そうしないと本物のリフトシステムは設計できない（現実の業務では、そのへんの要素技術を十分に調査し研究を行なってから設計する）。
ここでは、あくまで記述実験として、与えられた問題文の要求を満足する抽象的なモデルを１つ与えたにすぎない。

（６）どこがやりやすく、どこがやりにくかったか
◎やりやすかった点
ＤＭＣでは、全体と細部を、常に意識しながら構造を作っていくので、設計の見通しが立てやすい。つぎに考えるべき

ことが、自然に決ってくるので、結果として思考過程を誘導することになる。この点から、設計作業の能率が向上したように思える。

◎やりにくかった点

リアルタイムな処理をまじめに記述しようと考えると、今の枠組みでは不十分である。特に、外からのイベントのタイミングに関する制約などを記述するのが難しい。

（７）例外処理をどのように考えたか

非常ボタンなどは、一種の例外処理とも解釈できるが、ボタンの種類とそれに応じた処理が他のボタンと異なるだけであり、本記述では非常ボタンということで記述上の特別な扱いはしなかった。本来は、電源停止や、装置の故障、ドアの開閉異常、重量オーバなどが例外処理に当たると思われるが、それらに関しては問題文にも触れられていないので記述しなかった。

４．２ ｄｍＣＡＳＥ上での記述実験

ｄｍＣＡＳＥでは、最も単純な部品が単語であり、それらを用途別に分類整理したものが単語表である。チップ部品は、単語をもとに仕様記述されたインスタンス定義の実体である。さらに、ボード部品は、概念構造図上に配置されたチップ間を配線したものである。これらの部品の構成は、図４．１のようになっている。

設計の初期段階における問題の整理や分析に単語表の操作機能が便利であった。部品の組み立て作業は、チップ部品をボード上に配置し、全体の構成を頭に描きながら構造を決定していく。これらの作業を、単語表、インスタンス定義表、概念構造図を見ながら操作することができ、机上の場合に比べて本来のモデル構築のための思考に集中できることが確かめられた。

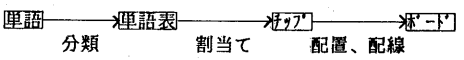


図４．１　部品の構成

仕様記述は、インスタンス定義表から構文誘導型エディタをよびだして行なう。構文パターンを埋め込むことにより段階的に記述を詳細化していく。必要な単語は、単語表から選択し、関数呼び出しは、その呼び出しパターンをマウスで選択し埋め込むことができる。また、カット＆ペーストを基本にした編集ができ、類似の記述を行なうときには、大変便利であった。

一般に、この種の構造エディタでは、構文木に依存した構造をしているため操作のステップ数が増加する傾向があり操作上問題になることが多い。本エディタでは、パターンの埋め込み操作を連続的なメニュー選択で行なえるようにしているので、通常はそう気にならない。ただし、カット操作に関連して、意図した構文要素レベルまで戻す際に、ステップを繰り返す必要が生じる場合があり多少不便さを感じた。

同時に平行して複数の記述の編集が可能であり、それらが、図４．２のように統合化された環境で作業できることは、試行錯誤しながら行なう設計環境として有効であることが確かめられた。

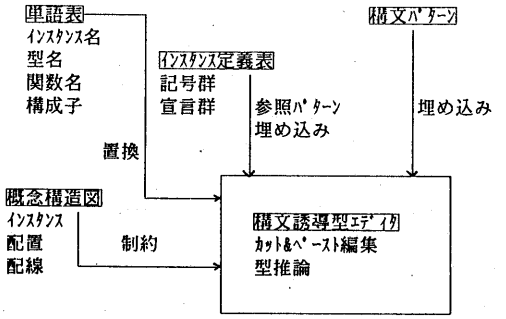


図４．２　仕様記述支援環境

５．設計プロセスの考察

５．１ 創造的な設計

いずれの工学にも共通する事柄は、新しいものを産み出すこと、つまり、人間の創作活動をともなうことであろう。工学だけでなく、小説や絵画などの創作活動とも通じるものが何かあるのではないかと考える。また、子供は工夫しながら、積み木やブロックを並べたりつないだりして様々な構造物の模型（ミニチュア）を作って遊ぶ。このような活動が工学の根底にある物作りの出発点であると考える。

積み木やブロックによる工作を観察してみると、それぞれの断片をいじくりまわしながら組み立てていく。このような、ものを組み合わせて形、つまり、構造を作っていくことは、人間の自然な思考過程と考えられる。特に、目で見て構造を確かめたり、手で触れて操作したりすることで、つぎのステップに進んだり、後戻りしたりする。つまり、頭の中で試行錯誤を繰り返しながら、想像しているものを作り上げていくのである。

これらの状況は、回路設計で部品となる素子やＬＳＩを選択し組み合わせて１つの回路を作る場合の思考過程とよくにている。もちろん、回路設計の場合には、電子回路に関する専門的な知識と経験や洞察力などが要求されることはいうまでもない。いずれにせよ、構造物を作る時には、その人の経験、知識をフルに活用し、人間の創造力を引出すことが大切である。

５．２ 設計の思考過程

では、人間の創造力を助長するためには、どうすればよいのであろうか。その１つの手がかりは、具体的なイメージを連想したり、想像することである。人間の頭の中では、あいまいなイメージのままでも、大雑把に操作してみることができる。そのようにして、頭の活動を活発にすることが、創造力を生むことにつながると考えられる。

実際の設計では、まず、ラフなイメージを描き、それを手がかりにして、システム全体の要件や、動きを考える。つぎの段階として、より具体的なイメージを描く。具体化することで、つぎのステップの思考活動や意志決定が行なえるようになる。

さらに、そのモデルが適当か問題点がないかなどを検討しなければならない。このとき、具体的な対象物を見ながら操作できることが望ましい。しかし、それができない場合には、できるだけ繰り返し頭の中でシミュレーションしてみるので

ある。このようなことが可能であるのは、やはり、人間の頭脳がすばらしいからにほかならない。

５．３ 設計の目標

人間が新しいアイデアに到達するには、さまざまなケースがあるであろうが、つぎのような過程をたどることが多いのではないかと思われる。

① 問題を理解する
② 問題の解決法を模索する
③ あたためる
④ 新しいアイデアがひらめく

設計の場合でも、初期の設計作業を終えた後、数日間、あるいは、もっと長い期間、別の仕事をしていて、何気なく、ふと新しいアイデアが思い浮かぶといったことがよくある。このような、一見無駄であるような「あたためる期間」が、独創的なものを創る場合にはどうしても必要な気がする。

ところで、設計の各場面には、多様な選択肢が存在する。例えば、

・全体の構成の決定
　アルゴリズムやデータ構造などの選択
・細部の構成の決定
　要素となるモジュール（概念）の選択
・機能の詳細の決定
　関数の実現の仕方

などがある。

設計の目標は、そのシステム全体として、よりふさわしいと思われる解を１つ決めることである。その際、多くの代替案を考えその内容を吟味し、適切な判断基準で評価し取捨選択する。このことが、よりよい設計かどうかの分かれ道となる。

５．４ 設計の原則

回路や機械などの工学的設計では、一般に、物理的な拘束条件などの設計ルールがある。また、よい設計を行なうための原理原則といったものが知られている。ここでは、ソフトウェアが主体となるシステムの場合の原則について議論する。

（１）機能に対する原則

一般に、実現されるシステムには、機能の独立性が要求される。つまり、操作や制御の対象となる機能が互に影響を及ぼさずに独立にコントロールできることが大切である。例えば、リフト問題では、機能的要求として

①フロア、または、リフトから要求があればサービスを行なう。
②非常用のボタンが押されたらサービスを中止する。

これらの２つの要求は、同時には満足できないのは明らかである。もし、非常用のボタンが押されて、サービスが中止されている時、フロアからの要求に対しサービスを再開するような設計は機能の独立性が守られてないことになるので良くない。この場合には、問題文にもあるように、サービス中止状態を無効にする機能をつけ、互に独立に制御できるように設計すべきである。

（２）包括的な設計

設計を行なう状況には、ゼロからシステムを設計する場合や、既存のシステムを拡張する場合など種々のケースがある。

いずれの場合にも、要求されている機能全体を包括的に考慮して最適な構成、構造を設計することが原則であり、重要である。

このような包括的な設計に対し、付加的な設計、組合わせ設計というものが考えられる。付加的な設計は、既設計の部分には手を付けずに機能を次ぎ次に付加していくやり方である。一方、組合わせ設計は、既存の部品を再利用することだけを優先したやり方である。ソフトウェア開発の現場では、このような設計が少なくない。

付加的な設計も組合わせ設計も、全体を見渡した最適な設計からは、次第に離れていくことは明らかである。その結果、システムの物理的な規模が大きくなり、保守コストの増大をまねく。

６．おわりに

リフト問題の記述実験を通して設計プロセスの一面を分析できた。今後は、シミュレーションなどによるモデルの妥当性の検証などを行ないたい。

ｄｍＣＡＳＥについては、方法論に従った記述支援機能の有効性が確かめられた。他の解析支援機能など全体的な評価は、今後の課題である。

また、設計プロセスに関する上述の考察は、ＤＭＣやｄｍＣＡＳＥの開発思想やゴールと重複するものである。

謝辞

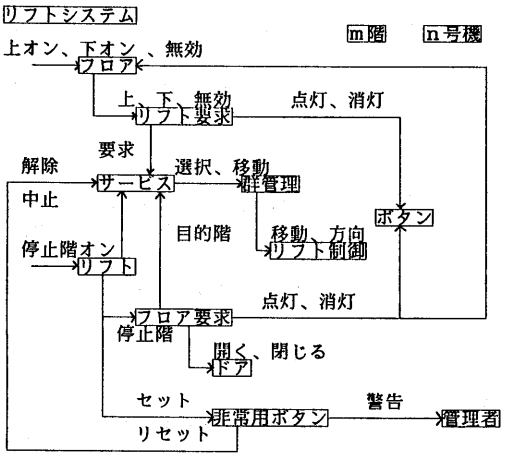
リフト問題の記述実験は、情報処理振興事業協会（ＩＰＡ）の技術センターにおける「ソフトウェアプロトタイピング技術研究」ＷＧでの共同研究の成果である。古宮誠一氏をはじめメンバーの方々の貴重な助言に心から感謝する。

参考文献

１） 関根,大林.： 新しいプログラム設計法－概念による設計法（ＤＭＣ）,bit 1982
２） 大林.:標準ＭＬを用いたプログラム設計支援環境,ソフトウェア工学 51-5 1986
３） 大林.:概念による設計法（ＤＭＣ）による在庫管理システムの記述，ソフトウェア工学 58-5,1988
４） 松浦,中里,大林.:概念による設計法（ＤＭＣ）に基づくプログラム開発環境の実現 ,ソフトウェア工学 58-6,1988
５） 松浦,大林.:プログラム開発環境ｄｍＣＡＳＥにおける解析環境,CASE環境シンポジウム,1989
６） 松浦,大林.:ソフトウェア開発環境ｄｍＣＡＳＥ,第39回全国大会,1989
７） 松浦,大林.:ソフトウェア開発環境ｄｍＣＡＳＥ,情報処理学会論文誌,vol 31, No 7,1990
８）Problem Set for the 4th International Workshop on Software Specification and Design,pp.ix-x(1987)

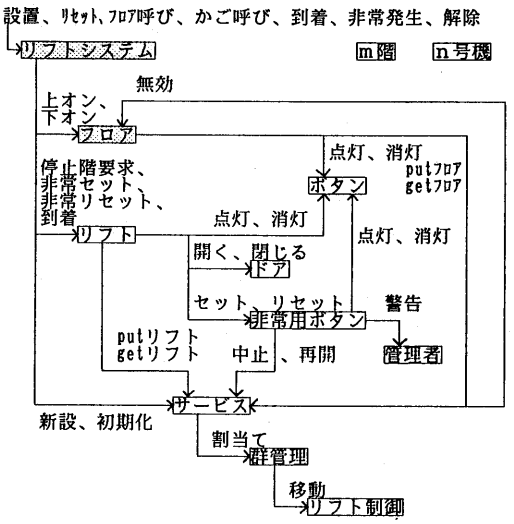
付録：記述結果

［概念構造図：第１版］



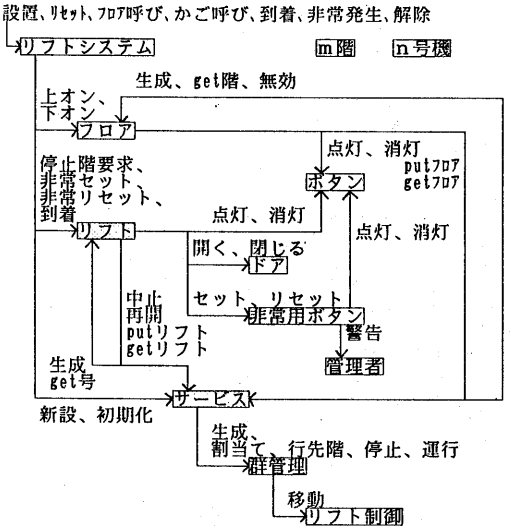
［概念構造図：第２版］

で示すインスタンスの仕様記述を行なった。



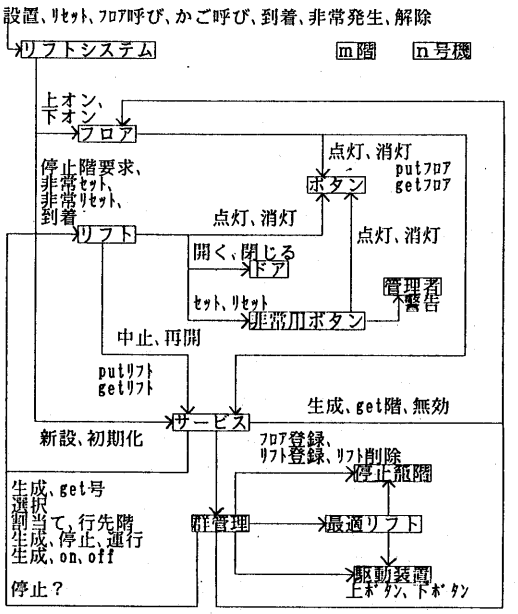
［概念構造図：第３版］

で示すインスタンスの仕様記述を行なった。



［概念構造図：第４版］

で示すインスタンスの仕様記述を行なった。



［仕様記述］

```
signature　リフトシステム
type システム
val 設置　　：階数 * 台数 -> システム
val リセット　：システム -> システム
val フロア呼び：階 * 方向 * システム -> システム
val かご呼び　：階 * 号機 * システム -> システム
val 到着　　：階 * 号機 * システム -> システム
val 非常発生　：号機 * システム -> システム
val 解除　　：号機 * システム -> システム
end

module リフトシステム-Mod
instance　フロア、リフト、サービス
type システム = 系 of(階数 * 台数 * 稼働状況)
val 設置(階数,台数) =
                系(階数,台数,{サービス}新設(階数,台数))
val リセット(系(階数,台数,稼働状況)) =
                系(階数,台数,{サービス}初期化(稼働状況))
val フロア呼び(階,上,系(階数,台数,稼働状況)) =
                系(階数,台数,{フロア}上オン(階,稼働状況))
  | フロア呼び(階,下,系(階数,台数,稼働状況)) =
                系(階数,台数,{フロア}下オン(階,稼働状況))
val かご呼び(階,号機,系(階数,台数,稼働状況)) =
                系(階数,台数,
                  {リフト}停止階要求(階,号機,稼働状況))
val 到着(階,号機,系(階数,台数,稼働状況)) =
                系(階数,台数,
                  {リフト}到着(階,号機,稼働状況))
val 非常発生(号機,系(階数,台数,稼働状況)) =
                系(階数,台数,
                  {リフト}非常セット(号機,稼働状況))
val 解除(号機,系(階数,台数,稼働状況)) =
                系(階数,台数,
                  {リフト}非常リセット(号機,稼働状況))
end

signature フロア
instance
type フロア
val 生成　　：階 -> フロア
val 上オン　：階 * 稼働状況 -> 稼働状況
val 下オン　：階 * 稼働状況 -> 稼働状況
val 無効　　：階 * 方向 * 稼働状況 -> 稼働状況
val 上ボタン：フロア * 階 -> ボタン
val 下ボタン：フロア * 階 -> ボタン
end

module フロア-Mod
instance ボタン、サービス
type フロア = 床 of(階 * ボタン * ボタン)
val 生成(階) = 床(階,{ボタン}生成,{ボタン}生成)
val 上オン(階,稼働状況) =
      let val 床(階,上ボタン,下ボタン) =
                        {サービス}ｇｅｔフロア(階,稼働状況)
      in {サービス}ｐｕｔフロア
            (階,
             床(階,{ボタン}点灯(上ボタン),下ボタン),
             稼働状況)
      end
val 下オン(階,稼働状況) =
      let val 床(階,上ボタン,下ボタン) =
                        {サービス}ｇｅｔフロア(階,稼働状況)
      in {サービス}ｐｕｔフロア
            (階,
             床(階,上ボタン,{ボタン}点灯(下ボタン)),
             稼働状況)
      end
val 無効(階,上,稼働状況) =
      let val 床(階,上ボタン,下ボタン) =
                        {サービス}ｇｅｔフロア(階,稼働状況)
      in {サービス}ｐｕｔフロア
            (階,
             床(階,{ボタン}消灯(上ボタン),下ボタン),
             稼働状況)
      end
  | 無効(階,下,稼働状況) =
      let val 床(階,上ボタン,下ボタン) =
                        {サービス}ｇｅｔフロア(階,稼働状況)
      in {サービス}ｐｕｔフロア
            (階,
             床(階,上ボタン,{ボタン}消灯(下ボタン)),
             稼働状況)
      end
end

signature リフト
instance
type リフト
val 生成　　：号機 -> リフト
val 停止階要求：階 * 号機 * 稼働状況 -> 稼働状況
val 非常セット：号機 * 稼働状況 -> 稼働状況
val 非常リセット：号機 * 稼働状況 -> 稼働状況
val 到着　　：階 * 号機 * 稼働状況 -> 稼働状況
val 停止？　　：階 * リフト -> bool
end

module リフト-Mod
instance　ドア、非常用ボタン、サービス
type リフト = 籠 of(号機 * ボタン list *
                                ドア * 非常用ボタン)
val 生成(号) = 籠(号,{ボタン}列生成,
                        {ドア}生成,{非常用ボタン}生成)
val 停止階要求(階,号,稼働状況) =
      let val 籠(x,ボタンリスト,y,z) =
                        {サービス}ｇｅｔリフト(号,稼働状況)
          val 新ボタンリスト=
                        更新(階,ボタンリスト,点灯)
      in {サービス}ｐｕｔリフト
            (x,
             籠(x,新ボタンリスト,y,z),
             稼働状況)
      end
val 非常セット(号,稼働状況) =
      let val 籠(x,y,z,非常用ボタン) =
                        {サービス}ｇｅｔリフト(号,稼働状況)
      in {サービス}ｐｕｔリフト
            (x,
             籠(x,y,z,
               {非常用ボタン}セット(非常用ボタン)),
             {サービス}中止(号,稼働状況))
      end
val 非常リセット(号,稼働状況) =
      let val 籠(x,y,z,非常用ボタン) =
                        {サービス}ｇｅｔリフト(号,稼働状況)
      in {サービス}ｐｕｔリフト
            (x,
             籠(x,y,z,リセット(非常用ボタン)),
             {サービス}再開(号,稼働状況))
      end
val 到着(階,号,稼働状況) =
      let val 籠(x,ボタンリスト,ドア,y) =
                        {サービス}ｇｅｔリフト(号,稼働状況)
          val 新ボタンリスト=
                        更新(階,ボタンリスト,消灯)
      in {サービス}ｐｕｔリフト
            (x,
             籠(x,新ボタンリスト,
                {ドア}開く(ドア),y),
             稼働状況)
      end
val 更新(1,b::bs,f)= f(b)::bs
  | 更新(n,b::bs,f)= b::更新(n-1,bs,f)
end

signature サービス
instance
type 稼働状況
val 新設　　：階数 * 台数 -> 稼働状況
val 初期化　　：稼働状況 -> 稼働状況
val ｇｅｔフロア：階 * 稼働状況 -> フロア
val ｐｕｔフロア：階 * フロア * 稼働状況 -> 稼働状況
val ｇｅｔリフト：号機 * 稼働状況 -> リフト
val ｐｕｔリフト：号機 * リフト * 稼働状況 -> 稼働状況
val 中止　　：号機 * 稼働状況 -> 稼働状況
val 再開　　：号機 * 稼働状況 -> 稼働状況
end

module サービス-Mod
instance
type 稼働状況=稼 of{フロア list * リフト list * 群状態}
val 新設(階数,台数) =
```

```
      let val make(0,f)=nil
            | make(n,f)=
                        f(n)::make(n-1,f)
      in 稼(make(階数,{フロア}生成),
            make(台数,{リフト}生成),
            {群管理}生成(階数,台数))
      end
val 初期化(稼(フロアリスト,リフトリスト,群状態))=
      let val map(f,nil,s)= s
            | map(f,a::l,s)=
                        map(f,l,f(a,s))
      in 稼(フロアリスト,リフトリスト,
            map({群管理}行先階,リフトリスト,
                  map({群管理}割当て,
                        フロアリスト,群状態)))
      end
val getフロア(階,稼(フロアリスト,_,_))=
                  get(階,get階,フロアリスト)
val putフロア(階,フロア,
            稼(フロアリスト,x,群状態))=
      稼(put(階,get階,フロア,フロアリスト),
         x,
         {群管理}割当て(フロア,群状態))
val getリフト(号,稼(_,リフトリスト,_))=
      get(号,get号,リフトリスト)
val putリフト(号,リフト,
                  稼(x,リフトリスト,群状態))=
      稼(x,
         put(号,get号,リフト,リフトリスト),
         {群管理}行先階(リフト,群状態))
val get(n,f,a::l)=
      if f(a)=n
      then a
      else get(n,f,l)
val put(n,f,b,a::l)=
      if f(a)=n
      then b::l
      else a::put(n,f,b,l)
val 中止(号,稼(x,y,群状態))=
                  稼(x,y,{群管理}停止(号,群状態))
val 再開(号,稼(x,y,群状態))=
                  稼(x,y,{群管理}運行(号,群状態))
end

signature 群管理
instance
type 群状態
val 生成  : 階数 * 台数 -> 群状態
val 割当て: フロア * 群状態 -> 群状態
val 行先階: リフト * 群状態 -> 群状態
val 停止  : 号機 * 群状態 -> 群状態
val 運行  : 号機 * 群状態 -> 群状態
end

module 群管理-Mod
instance  停止籠階、駆動装置、リフト、最適リフト
type 群状態 = 群 of (駆動装置 * 停止籠階)
val 生成(階数,台数) = 群({駆動装置}生成(階数,台数),
                        {停止籠階}生成(階数,台数))
val 割当て(フロア,群状態 as 群(駆動装置,停止籠階))=
      更新(階,下,{フロア}下ボタン(フロア,階),
         更新(階,上,{フロア}上ボタン(フロア,階),
               群状態))
val 更新(階,方向,オン,群(駆動装置,停止籠階))=
      群(駆動装置,
         {停止籠階}フロア登録
               (階,方向,
                  {最適リフト}選択(階,方向,駆動装置,
                                    停止籠階),
                  停止籠階))
  | 更新(階,方向,オフ,群(駆動装置,停止籠階))=
      群(駆動装置,
         {停止籠階}フロア登録(階,方向,空号,停止籠階))
val 行先階(リフト,群(駆動装置,停止籠階))=
      if {リフト}停止？(階,リフト)
      then 群(駆動装置,
               {停止籠階}リフト登録(階,号,停止籠階))
      else 群(駆動装置,
               {停止籠階}リフト削除(階,号,停止籠階))
val 停止(号,群(駆動装置,停止籠階))=
                  群({駆動装置}off(号,駆動装置),停止籠階)
val 運行(号,群(駆動装置,停止籠階))=
                  群({駆動装置}on(号,駆動装置),停止籠階)
end

signature 停止籠階
instance
type 停止籠階
val フロア登録 : 階 * 方向 * 号機 *
                              停止籠階 -> 停止籠階
val リフト登録 : 階 * 号機 * 停止籠階 -> 停止籠階
val リフト削除 : 階 * 号機 * 停止籠階 -> 停止籠階
end

module 停止籠階-Mod
instance
type 停止籠階 = 停止籠 list
type 停止籠 = 停 of (フロア呼び * かご呼び)
type フロア呼び = 上下 of (号機 * 号機)
type かご呼び = 号機 list
val フロア登録(1,方向,号,
                  停(上下(x,y),z)::残リスト)=
      case 方向 of
        上 => 停(上下(号,y),z)::残リスト
      | 下 => 停(上下(x,号),z)::残リスト
  | フロア登録(n,方向,号,停止籠::残リスト)=
            停止籠::フロア登録(n-1,方向,号,残リスト)
val リフト登録(1,号,停(x,y)::残リスト)=
                  停(x,登録0(号,y))::残リスト
  | リフト登録(n,号,停止籠::残リスト)=
                  停止籠::リフト登録(n-1,号,残リスト)
val 登録0(号,[])=[号]
  | 登録0(号1,号2::残リスト)=
      if 号1=号2 then 号2::残リスト
      else 号2::登録0(号1,残リスト)
val リフト削除(1,号,停(x,y)::残リスト)=
                  停(x,削除0(号,y))::残リスト
  | リフト削除(n,号,停止籠::残リスト)=
                  停止籠::リフト削除(n-1,号,残リスト)
val 削除0(号,[])=[]
  | 削除0(号1,号2::残リスト)=
      if 号1=号2 then  残リスト
      else 号2::削除0(号1,残リスト)
end

signature 駆動装置
instance
type 駆動装置
val 生成  : 階数 * 台数 -> 駆動装置
val on  : 号機 * 駆動装置 -> 駆動装置
val off : 号機 * 駆動装置 -> 駆動装置
end

module 駆動装置-Mod
instance
type 駆動装置 = 駆動 list
type 駆動 = 駆 of (電源 * 階数 * 状態)
type 電源 = オン | オフ
type 状態 = 状 of (方向 * 位置 * 速度)
val 生成(_,0)=[]
  | 生成(階数,n)=
      駆(オフ,階数,状(上,1,0))::生成(階数,n-1)
val on(1,駆(_,x,y)::残リスト)=
                  駆(オン,x,y)::残リスト
  | on(n,駆動::残リスト)=
                  駆動::on(n-1,残リスト)
val off(1,駆(_,x,y)::残リスト)=
                  駆(オフ,x,y)::残リスト
  | off(n,駆動::残リスト)=
                  駆動::off(n-1,残リスト)
end
```