ソフトウェア工学 77－1
（１９９１．２．７）

円卓：オブジェクトベースを中心とした業務システム開発支援

横山岳浩

ソニー（株） スーパーマイクロ事業本部

オブジェクトベースを中心とした業務システム開発支援「円卓」を開発している。業務システムで扱う情報と相性のよい関係データベースを基底においた。開発支援系では、入出力部品を組み合わせて目的システムを定義していくが、「人事構成」のような、構造を持つオブジェクトをも入出力部品として扱うことができる。単に目に見える入出力画面を定義するにとどまらず、情報の論理的な性質をも定義できる。オブジェクトベースで業務情報の整合性を保つ仕組みを管理できるため、業務の流れや手順をよく知っている、業務の専門家がシステム開発を行なえるようになる。

ENTAKU (Round Table) : Object-base oriented system development support system

Takehiro Yokoyama
Sony SuperMicro Systems Group

This paper presents an object-base oriented system development support system ENTAKU(Round Table). This system is based on the relational database, which fits in business system informations.

Developer can define the target system combining input-output parts. He can also use "structured" objects as input-output parts. By constructing the application system in object-base oriented manner, the system keeps the integrity between data and display as well as among data. ENTAKU supports the staff of non-EDP department, who knows his own business well.

1．はじめに

ソフトウェア開発のためのさまざまなトゥールが開発されている。その多くは、要求分析からプログラミング、テストまで、いわゆる落水モデルの上流から下流までを一貫して支援しようとしている。

残念ながら、こういったトゥールは、期待したほど普及し、効果を上げているとは言い難い状況である。いろいろ原因が考えられるが、一番大きいのは、ソフトウェアに関する高度な知識・技術がないと使いこなせないからであろう。特に Upper CASE と呼ばれる上流工程のトゥールで顕著である。

一方で、ソフトウェアの需要は大きくなっており、開発人員の不足が問題となっている。

このような状況を考えると、ソフトウェアの専門家にしか使えない開発支援トゥールではソフトウェア危機に対する何の解決にもならないことがわかる。誰のための開発トゥールが求められているのか、を考え直す必要がある。

筆者は、業務知識はあるが計算機に関する知識はあまりない、という人を主な対象として、その業務のために使うプログラムを開発するための支援トゥール「円卓」を構想し、試作している。

「円卓」では、業務の実行手順に関する事項以外のさまざまな処理について、できるだけ考えずに開発できることを目標にしている。情報を保存しておく方法、適当な加工を施して表示する方法などがこれに当たる。また、情報間の整合性を保つための機構を備える。

表示の形式については、データの性質に応じてさまざまに変えることができる。文字で表示する形式のほか、結果に応じて絵で表示したり、分類して階層構造で表示したりすることができる。この場合でも、階層構造を表現するために必要なさまざまな処理を全く意識せず、単に階層構造にしたい旨を指示するだけで済むのである。

2．特徴

(1) できるだけ直感的に仕様の指定ができること。

従来のトゥールでは、ソフトウェア構造、データフローなど、いわば計算機の言葉で問題を記述する必要があった。これに対し、データの入出力を行う画面を、できるだけ目的システムに近い形で記述するところから仕様の指定を始める。テキスト、ボタンなどの、入出力部品を貼り付けることに加え、表、階層などの「構造を持った」入出力部品も用意されており、表示要素として指定できる。

インターフェース・ビルダーと呼ばれるものと違い、インターフェース仕様だけでなく、扱う情報の論理的な性質をも、入出力部品を手掛かりとして指定する。

(2) データ間、データと表示間の整合が保証されること。

専門のプログラマーでもそうだが、プログラムの誤りの多くは、関連を持ったデータが正しく同期しない、という形で発生する。例えば、ある計算結果を正しく求めたのに、それを画面に反映し損なったとか、あるデータの更新に伴って当然更新しなければならないデータをそのままにしてしまったとかいう具合である。

画面とデータとの整合を保つために、論理データを参照しながら画面を作る構成の画面作成部を用意し、論理データと独立な画面の直接更新を禁止している。また、データ間の整合を保つために、「データに付随する手続き」を「目的システムに付随する手続き」と区別し、前者をデータベース側で管理する。

3．目的システム開発の手順

目的システム(開発しようとするアプリケーション)が利用者にとってどう見えて欲しいか、を指定する手順は次のようになる。

(1) 静的な見え方の指定：ある時点で、画面上にさまざまな入出力部品がどのように配置されるか、を指定する。
(2) 動的な見え方の指定：ある入力操作を施したときに、画面がどのように変化して欲しいか、を記述する。

開発者は、直接目に見え、把握しやすい画面の変化という形で指定しているが、実は、論理的なデータの動きも定義したことになっている。

3．1．静的な見え方の指定

目的システムのユーザー・インターフェース(ＵＩ)部分を開発／管理するためのシステムを、ＵＩＭＳ(User Interface Management System)などと呼ぶのが通例である。しかしここでは、ＵＩの定義・開発段階と、目的システムの実行時におけるＵＩ管理機能とを明示的に区別するために、[1]に従って、ＵＩＤＳ(User Interface Development System)と呼ぶことにする。

静的な見え方の指定は、さまざまな入出力部品を画面に貼り付けていくことによって行う。定義の様子を図１に示す。入出力部品として、プルダウンメニューやボタンなどがあるが、これに加え、クラス部品と呼ばれるものも、入出力部品の一種として扱っている。図１左上はクラス部品のメニューになっている。

代表的なクラス部品として、テキスト、図形・イメージ、表構造、階層構造、グラフ構造を用意している。更

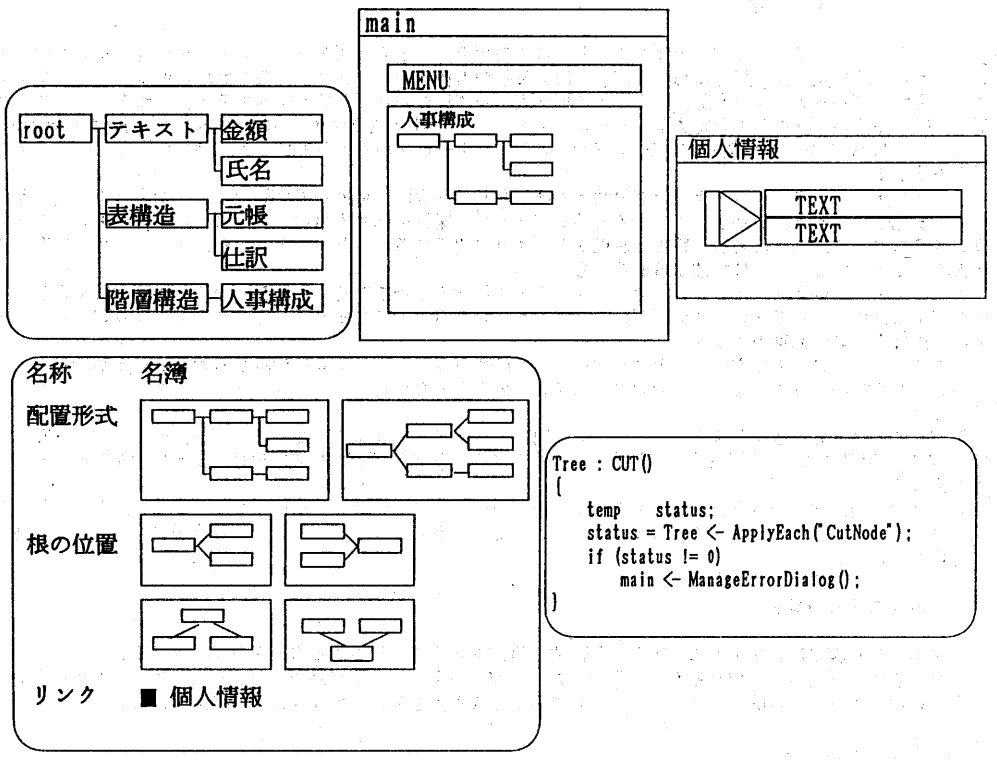

図1　目的システム開発の様子

に、これらのサブクラスを追加することも可能である。

例えば階層構造は、各節の形は様々だが、節の追加や部分階層の削除などの一連の操作が許される、型構成子と考えることができる。節の形状は別に定義し、階層構造と関連づけることにより、型構成子に具体的な基底型が与えられ、一つの型の定義が完成する。なお、複数種類の節を定義すれば、内容によって形の異なる節を持った階層構造を実現できる。図1中上の例では、階層構造のサブクラスである「人事構成」を使っている。

節は、これもいくつかの入出力部品の組み合わせとして定義することになる。図1右上の例では、ボタンと2つのテキストが組み合わされて1つの節を構成している。論理的には、複合オブジェクトの定義に相当する。ここで、2つのテキストは、「氏名」「身分」など「人事構成」クラスと関連づけられたものでなければならない。「在庫量」はおそらく置くことができないであろう。このことを、後述のメタベースを参照して確認し、論理的な正当性を確保しようとする。

入出力部品には、その種類に応じて、属性を指定することができる(図1左下)。次のようなものがある。

(1) 純粋に表示に係る属性：例えば、ボタンについて、選択状態にあることを示すための色がこれに当たる。また、階層構造で、根を上側に置くか左側に置くか、という選択もこれに当たる。利用者が好みによって変更しても、目的システムの機能に影響を与えないものである。

(2) 表示に係る属性であるが、間接的に論理的な性質を規定するもの：例えば、矢印ボタンについて、その向きが可変である旨がこれに当たる。これは、表示が変化しうることを指定するだけでなく、これによって論理的な処理が影響を受けるものであるから、論理的な性質をも規定していることになる。

(3) 論理的な性質を規定するもの：後述のオブジェクトベースの構成ともかかわる重要な属性であるので、項を改めて述べる(4．2)。

３．２．動的な見え方の指定

ＵＩＤＳの中で、各入出力部品を介して何らかの入力があったときに何を行うべきか、を記述することができる。これは動的な見え方の指定に当たるが、入出力部品の属性の一種でもある。最も簡単な例は、プルダウンメニューを使ってあるメニューを選択したときに何が起こって欲しいか、という定義である。これをスクリプトの形で、図１右下のようにして指定する。これは、メニューの CUT が選択された場合の動作で、選択された各節に対して、Tree クラスに付随している CutNode メソッドを適用する旨を示している。結果が不正ならば、その旨を表示する。

スクリプトは、一種のプログラミング言語とも考えられ、非専門家にとっては確かにとっつきにくいところもでてくる危険性がある。しかし、ここでの記述レヴェルは、業務に関する事項に限られ、スプレッドシートの計算式定義のような感覚で扱える。特に、画面をどのように更新するか、とか、データをどのようにすれば保存できるか、などといったことは記述する必要がない。

スクリプトも、後述のメソッドも、オブジェクトを操作する手順を記述しているという点では同じである。スクリプトは常にメソッドを介してオブジェクトにアクセスすることになるが、一旦スクリプトが登録されてしまえば、外部から見て区別がつかない。

スクリプトとメソッドの違いは、それがデータに付随するか目的システムに付随するか、という点である。同一のデータを操作するのでも、一つの目的システムでのみ有効な処理は、オブジェクトベースに登録しておく必要がない。

例えば在庫管理を行う場合を考えてみる。「出庫」メソッドは、在庫量の確認処理を行う。もし在庫がある一定数(この数は商品によって異なる)を割り込んでしまうようならば、「発注」処理を起動し、その旨を状態コードの形で通知する。

ところで、発注が行われた旨を表示するのに、ある目的システムではメッセージウィンドウで表示し、別の目的システムでは、ログファイルに記録するというようなことが起こりうる。こういった事項は、スクリプト側の記述内容である。

４．目的システムの構成

これまでの定義の内容をもとに、実際に動作する目的システムを実現する方式を以下に述べる。
しかしそれに先立って、できあがった目的システムがどのような環境で動作することになるか、を述べておく。
図２に示すように、目的システムは、３つの部分からなる。なお、この構成は[2]を参考にした。

(1) イヴェント操作部
利用者の入力その他のイヴェントを取り込み、適当なスクリプトを起動する部分である。

(2) オブジェクトベース部
目的システムが処理の対象とするデータを永続的オブジェクトとして管理する部分である。オブジェクトベー

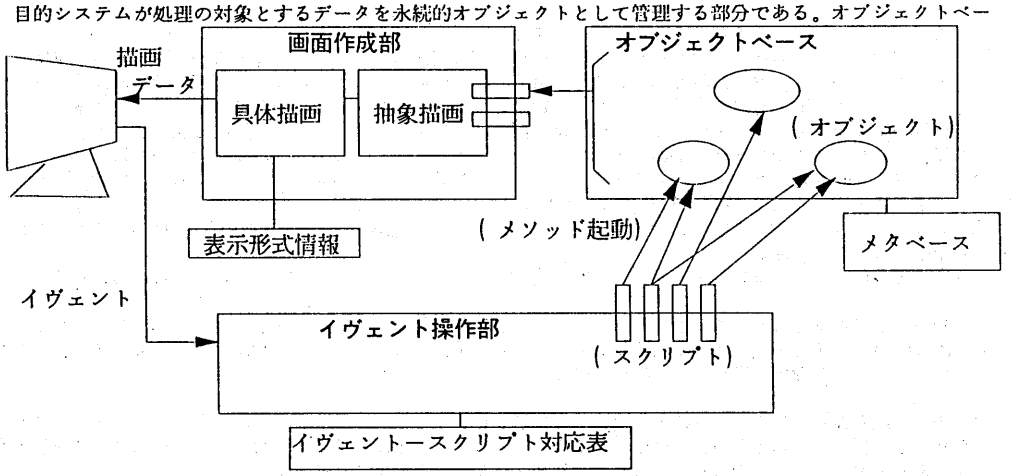

図２　目的システムの構成

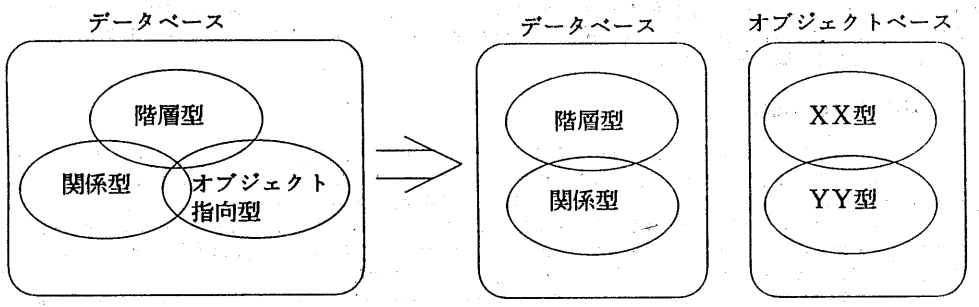

図3　オブジェクトベースの概念

スの管理のためにメタベースがある。データに論理的な操作を施すためのメソッドはここで管理されている。

(3) 画面作成部
オブジェクトベースと整合を取りながら、画面を更新する部分である。

このうち(2)(3)の部分について、以下、詳しく述べる。

4．1．オブジェクトベース部

永続的オブジェクトを管理し、データの論理的な操作を実行するための部分である。
オブジェクトベースは、通常の意味のデータベースとしての機能はすべて備えている。これに加えて、データを論理的に操作するための手続きを、メソッドとして保持している。例えば階層構造を表現するクラスでは、節の親子関係を表現するために必要なデータと、その親子関係に矛盾が生じないようにしながら節を追加したり部分木を削除したりするメソッドが登録されている。メソッドは、通常のデータベースの言葉で言うと、シェーマ定義の一部として管理される。
現在、オブジェクト指向データベースの確定した定義はまだないようである。筆者は次のように解釈している。
昔、プログラムとデータを分離してデータベースという概念が創造されたとき、その目的とするところは、プログラムと独立にデータを保守・管理することであった。この結果、一つのデータを多くのプログラムがアクセスするような場合でも、矛盾なくデータを管理することができるようになった。
しかし、データというものは、それ単独で存在するものではなく、他のデータと関連を持ちながら存在している。データAを更新すれば、それに対応してデータBも更新しなければならない、という関係がいくつもあるのである。例えば、薬の濃度が変われば摂取量も対応して変わる、という具合である。また、階層構造のデータの場合、その構造を表現するために "car" と "cdr" のようなデータを持っているが、これは、勝手に更新されてよいものではなく、階層を表現するように、ある制約のもとで更新されるべきものである。
従来、このようなデータを矛盾なく更新するのは、個々の目的システムの役割であった。したがって、一つのデータに対して多くのプログラムがアクセスする場合、それぞれに同じような処理が書かれていたことになる。一つの目的システムに誤りがあって、階層構造を正しく表現しないデータを作ってしまうと、その影響はすべての目的システムに及ぶ。
しかし、こういった処理は本来、目的システム側でなく、データの側で行うべきものである。こうすることによって、データの整合性を保持するための手続きは1箇所にカプセル化され、仮に更新処理の中身が変更になったとしても、個々の目的システムとは独立に保守することができる。
オブジェクト指向データベースについて、実用上の便宜からは、継承機能などいろいろと要件が挙げられているようである。しかし本質的に必要なのは、目的システムと独立に保守できるメソッドが管理できること、というのがここで採用している定義である。
このように定義したうえで、ここでは、「オブジェクトベース」という言葉を用いる。これは「オブジェクト指向データベース」の短縮形という以上の意味をこめている。
図3のように、従来、オブジェクト指向データベースは、階層型、関係型などと並ぶ、新しい型のデータベースである、という理解のしかたが一般的であった。しかし、上述のように、目的システムと独立なメソッドを管理する機能を本質ととらえると、データベースと対立する(または包含する)概念として、「オブジェクト(=データ+メソッド)ベース」というものをとらえることができる。そして、階層型、関係型などのデータベースがあるのと同様に、XX型、YY型のオブジェクトベースが考えられる。但しXX、YYにどんな言葉が入るのかはまだわからない。
定義により、メソッドは、目的システムとあらかじめリンクされて動くものであってはならない。メソッド変更時に再リンクが必要となり、目的システムと独立な保守が不可能になるからである。このことを強調するために、DynMethod という言葉でこれを表現する。一方、これを満たさないものを BindMethod と呼ぶ。

Unix の場合、ダイナミックリンク機能が充分でないため、通常のオブジェクトファイルの形式で実行させることは難しい。そこで、独自に定めた形式の実行形式 DynMethod ファイルに変換しておき、ソフトウェア的に CPU の動作を模擬する方式を採っている。

4．2．論理的な属性の定義

「テキスト」クラスを代表とするいくつかの入出力部品では、その内容が、オブジェクトベース中のあるインスタンスと直接対応している。「テキスト」以外にも、例えば「スケール」は、数値型のテキストと同様に扱うことができる。属性の定義はそのままオブジェクトベース中のクラスの定義に反映される。

オブジェクトベースでは、データの性質をメソッドの形で規定するというのが建前である。例えば「更新」メソッドは、新しい値を受け取って、それが一定の条件を満たしているかどうか確認し、実際の更新処理を行う。このように、メソッドの記述という形でオブジェクトの性質を記述するのである。

しかし、データ管理の観点から考えると、制約式、生成式のようなものは、メソッドとは独立した場所に記述できることが望ましい。そこでこれらをクラス定義の特別な属性として記述できるようにする。式評価の契機、制約式を満たしていないときに起動するメソッドを付加できる。

オブジェクトベースを共用する環境では、これらの属性は目的システムごとに決めることはできない。原則として、従来のデータベース管理者に対応する、オブジェクトベース管理者が規定し、メタベースで管理する。

但し、特定の目的システムで、より厳しい制約条件を課することは許されてよい。

4．3．画面作成部

オブジェクトベースの内容を反映するように画面を作る部分である。オブジェクトベースの内容が変わるような操作を施したメソッドは、必ず画面作成部の「描画」メソッドを起動することにして、整合を保とうとする。

画面作成部の中身は、更に２つの部分からなる。抽象描画部は、入出力部品の表示形式に依存しない処理を行う。例えば「階層構造」オブジェクトの場合、親子関係という形で論理的に把握されていたものを、平面上にどう配置するかという形で把握し直すのである。具体表示部は、表示形式情報に基づいて、画面に描画できるデータにする。例えば「階層構造」を、どちらを根にして表示するかに従って反転し、節の大きさも考慮して、具体的な座標値を得ていく。

表示形式情報は目的システムごとに異なるので、どのようにして具体表示部に取り込むかが問題となる。

一つは、目的システムの初期化の際、表示形式情報をロードしておき、解釈しながら実行する方法である。目的システムの試行ツールはこれを採用している。

もう一つは、表示形式情報を取り込んだ形のプログラムを生成してしまう方法である。上記の解釈プログラムに、表示形式情報をパラメーターとして与え、あらかじめできるところは先に部分計算してしまうことに相当する。最終的な目的システムはこの方法を採る。

さて、理論的には上記のように構成してみたが、実用的なシステムでは、意図的にオブジェクトベース通りでない表示をすることがある。階層構造のうちのある部分構造を抽出表示するとか、表構造である条件を満たす行だけを選択表示するとかいう場合である。

かといって、無原則な表示を許すことは整合性を崩す危険があるので、画面作成部に、部分描画のためのメソッドを設けることにした。階層構造では根となる節の番号と深さを指定する部分表示、表構造では行の満たすべき条件式を指定する部分表示の入口点である。そして、全部の表示を行うメソッドを含め、スクリプトから呼び出す形とし、原則を崩している。これはまた、再描画のタイミングをスクリプトで指定できることにもなる。

この結果、常にオブジェクトベースの全データを知る必要がないことをも勘案し、画面作成部がオブジェクトの参照メソッドによって必要なデータを得て画面を作るというように、制御の順序を反転して実現している。

5．まとめ

オブジェクトベースを中心としたシステム構成を採ることは、いわゆる電算部門の役割をも変えていくであろう。複数の業務部門に関係する情報資源を整理し、オブジェクトベースを管理する責任を負うことになる。逆に、個々の目的システムの開発は、それを利用する業務部門に任される。

目的システムの開発者は計算機に関する知識を持たないということで、スクリプトを意図通りに作成できるよう、さまざまな工夫が必要となるであろう。また、業務部門間の情報交換ということも考慮する必要がある。試作の結果を踏まえ、更に使いやすいものにしていきたい。

参考文献

[1] Myers : User-Interface Tools: Introduction and Survey
(IEEE Soft. Jan. 1989 pp.15-23)
[2] Young, Taylor, Troup : Software Environment Architectures and User Interface Facilities (IEEE trans. soft. vol.14 No.6, June, 1988 pp.697-708)