ソフトウェア工学 117－2
（1997．11．21）

# 階層構造をもつデータの入力システムのための フレームワーク

三ツ井 欽一
日本アイビーエム 東京基礎研究所
E-Mail: mitsui@trl.ibm.co.jp

本論文では、複雑な階層構造を持つデータの入力システムの開発に利用できるアプリケーションフレームワークを開発した経験について述べる。このフレームワークを用いると基本的なデータ編集機能は自動的に提供される。また、GUIは手続き的なプログラミングをせずに柔軟に作成・変更することができる。このフレームワークの設計は、初期の具体的なアプリケーション設計の段階から、フレームワーク化、ブラックボックス化を経て、最終的にアプリケーションビルダーの試作へと進化した。この過程を理解することは、フレームワークをより容易に使えるようにする上で重要であると考えられる。

# Application Framework for Hierarchical Data Entry Systems

Kin'ichi Mitsui
Tokyo Research Laboratory, IBM Research

This paper describes our experience on the development of an application framework usable for complex hierarchical data entry systems. The application framework automatically provides basic editorial functions and facilitates creation and modification of the GUI configuration without procedural programming. The design of the framework has been evolving from the initial specific application, a generalized one, a black-box framework, and to the final one equipped with an application builder. We believe that understanding this evolution process is important to make frameworks easier to use.

## 1 はじめに

本論文では、複雑な階層構造を持つデータの入力システムの開発に利用できるアプリケーションフレームワークを開発した経験について述べる。フレームワークが規定している基本的なアーキテクチャの説明に加えて、特に、フレームワークの設計の変更から最終的にアプリケーションビルダーの試作までの進化の過程について議論する。

## 2 データ入力システム開発の問題点

我々が対象としたのは、診療オーダリングシステムである。図1は、処方箋を編集中の画面のスナップショットである。処方箋は動的に変化する数段の集約階層を持つ構造データである。

このようなデータ入力システムの開発における問題点を述べる。データ入力システムは、オーダーシステム等を含む典型的なビジネスアプリケーションのクラスである。それらは、大きく分けてデータベースアクセス、ビジネスロジック、プレゼンテーションのためのコンポーネントを持つ。この種のアプリケーションは、少なくとも表面的には、複雑なアルゴリズムを必要とするわけでも、構造が非常に複雑なわけでもない。[1]また、最近ではGUI作成やデータベースアクセスに関する強力な開発ツールが利用できるようになってきた。にもかかわらず、開発コストが非常に高くなる場合がある。理由は次のようなものが考えられる。

- データベース、ビジネスロジック、プレゼンテーションの間の対応付けには、多大なプログラミング労力を必要とする。
- オーダー等の構造を持ったデータの入力はデータの編集作業であり、結局エディタに相当する機能が要求される。エディタの開発は、従来それがフレームワークの良い適用例になっていることからもわかるように容易ではない。
- データ入力システムはGUIが非常に重要である。

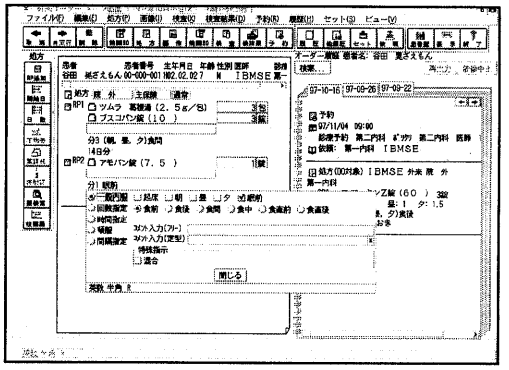

図1 診療オーダリングシステム

その設計は、画面の空間的な制約に依存し、利用者の種別、表示データの変更、機能の拡張等に対して微妙に調整する必要があり、稼動後にも変更要求が多い。

- 結局、GUIの変更、データモデルの変更、ビジネスルールの変更、データベースの変更が異なる状況で発生するが、それぞれの変更が他の部分に大きく影響してしまう場合は変更は容易ではない。

我々の第一の目的は、この種のアプリケーションの開発コストを下げることである。有望な技術としては、フレームワークおよびアプリケーションビルダーが考えられる。しかしながら、フレームワークの利用はオブジェクト技術の高いスキルを必要とする、目的に特化したアプリケーションビルダーを効率的に開発するのは難しい、といった問題がある。その基本となるフレームワークの設計は更に難しい。

## 3 データ入力フレームワーク

ここでは、我々が設計したデータ入力システムのためのフレームワークについて説明する。紙面の都合上、その設計全体を詳細に取り上げることはできないので、本節では特に注目すべき幾つかの設計上の判断について説明し、次節ではその設計と先に述べた問題点についてより一般的な議論をする。

---

[1] 我々が対象としたのは比較的な単純なクライアント・サーバー型のアプリケーションであり、以降の議論は、この種のシステムのフロントエンド側の開発に関わる。分散環境で動作するデータ入力システム全体を考えるとそれはより複雑なものになる。

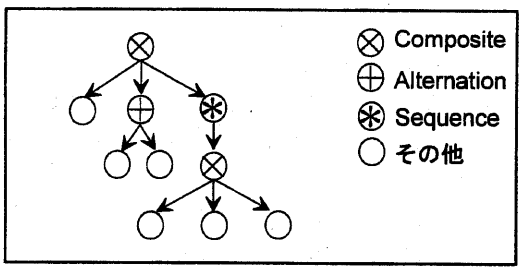


図2 モデルオブジェクト

### 3.1機能

本フレームワークが現在サポートしている機能をまとめると次のようになる。

- 追加、削除、コピー、取り消し、ドラッグ・ドロップなどの標準的な編集機能の実現支援
- GUIや印刷レイアウトの作成支援
- データの妥当性検査の扱い

### 3.2データ構造の定義と状態の管理

本フレームワークを最も特徴付けているのは、編集対象となるデータの構造を定義するための次のようなクラス群を用意していることである。

- Composite
- Alternation
- Sequence
- その他String, Symbol, Integer, Actionなどの構造を持たないデータ型

データ構造定義を図示した例が図2である。Compositeは、固定個の部分構造からなる構造型、Alternationは、部分構造の複数の選択肢、Sequenceは、任意個の要素の順序列を表すクラスである。これらのクラスのオブジェクトをモデルオブジェクトと呼ぶことにする。なお、モデルオブジェクトは名前付けされている。

このように、データ構造自体をモデルオブジェクトで表現することにより、フレームワーク本体から参照・操作できるようにする。例えば、後で説明するようなデータ構造とGUIとの対応情報からGUIを導出するといったことが可能になる。また、操作の対象となるデータの内部状態は全てこれらのクラスにより表現され、データの変更は、それらのクラスのメソッドを介して行うので、フレームワーク本体が状態変化の管理をすることができる。これにより、例えば取り消し等の機能を自動的に提供できる。

集約構造は木構造になるが、複数の集約構造は部分構造を共有してもよいものとする。したがって、一般にはデータ構造はサイクルのない有向グラフ構造になっている。

### 3.3オブザーバーパターン

オブザーバー[2]は、ある特定のオブジェクトの状態変化に反応して動作するオブジェクトである。オブザーバーを用いたプログラミングの利点は、観察されるオブジェクトの記述と観察するオブザーバーの挙動とを明確に分離できることである。アプリケーションにおけるオブザーバーの追加や削除は観察されるオブジェクトの記述の変更をすることなく行うことができる。本フレームワークの基本的なアーキテクチャは、既に説明したモデルオブジェクトと、それへの以降説明される様々な種類のオブザーバーから構成される。

### 3.4GUIコントロール

本フレームワークの強力な機能の一つは、GUIの自動生成である。これを実現するために、フレームワークは差し込み可能な(pluggable)GUIコントロールを提供する。GUIコントロールは、画面への表示とユーザー入力の最初の処理を行う。例えば、入力フィールド、ラジオボタン、ノートブック等が従来使われている。差し込み可能であるとは、そのコンポーネントが、あらかじめ想定されたある抽象インターフェースを持ち、そのインターフェースを会してフレームワークと相互に対話できることとする。フレームワークは、抽象インターフェースを知るのみであり、具体的なコンポーネントが何であるかを知る必要はない。これにより簡単な関連付けのみでフレームワークと差し込み可能なコンポーネントが協調動作する。

GUIコントロールは、モデルオブジェクトのオブザーバーであり、モデルのインターフェースの知識を前提にプログラムされている。ユーザー入力をモデルオブジェクトの状態変化に反映したり、逆にモデルオブジェクトの構造や値に変化があっ

た場合に反応し、表示を変更する。

本フレームワークでは、典型的なGUI設計でよく使われるGUIコントロールをあらかじめ数十個用意している。したがって、それらで十分な場合は、必要なものを選択するという形でGUIを実現することができる。

GUIは、一般にウィンドゥ（通常は画面上の矩形領域）の入れ子、折り畳み、ポップアップなどに相当する木構造で表現される。我々のフレームワークを用いて作成されるGUIは、モデルオブジェクトの有向グラフ表現を何らかの木構造に対応づけ、木構造の各点に具体的なウィンドゥの種別を指定したものになる。ウィンドゥの木構造は、モデルオブジェクトの構造が変化するのに対応して自動的に変化する。

GUIを定義するには、プログラムはまず次のような宣言を記述する。

*DefRootView("モデルオブジェクトの名前", "GUIコントロールの属性");*

これは、ある名前に相当するモデルオブジェクトが生成されたときには、トップレベルのウィンドゥに相当するGUIコントロールを対応させて生成することを宣言する。モデルオブジェクトの階層構造のある点から出発して有向辺をたどると木構造が得られる。このそれぞれの節に対して動的にGUIコントロールを生成するために、更に次のような形式の宣言をあらかじめ与えておく。

*DefView("モデルオブジェクトの名前", "GUIコントロール名", "GUIコントロールの属性");*

このフレームワークの利点は、GUIの設計がモデルオブジェクトとGUIコントロール種別の対応関係を宣言的に指定することでできることである。

印刷レイアウトの設計も、実行時には対話的である必要はないので動的な変化は必要ないが、同様の方法を用いることができる。

なお、色などGUIコントロールの属性を動的に変更したい場合は、次のいずれかの方法をとる。

- モデルの状態の変化をGUIコントロールの属性の変化として解釈する新しいGUIコントロールを元のクラスのサブクラスとして定義しフレームワークに登録する。
- モデルの文字列表現、表示／非表示、編集可／不可など、一般性の高い表示属性に関してはモデルオブジェクトのインタフェースを通して属性を変更できる。対応するGUIコントロールはこれらの属性の変化の通知を受ける。

前者は、自由度は高いがフレームワークの拡張になる。後者は、より容易な対応が可能である。

## 3.5ビューオブザーバー

GUIコントロール以外の差し込み可能なオブザーバーとして、ビューオブザーバー[2]が提供される。ビューは、元のモデルオブジェクト構造の一部を射影、並べ替え、併合、グループ化するためのみに用いる。その主な目的は、モデルオブジェクトの自然な構造とあるGUI設計でのウィンドゥの構造の間のギャップを容易に埋めることである。ビューオブザーバーは、別のオブザーバーから見た場合、モデルオブジェクトのように振る舞う。GUIコントロールと同様、想定される操作に対するビューオブザーバーがあらかじめフレームワークにより与えられるので、利用する場合は、それを選択してモデルオブジェクトと対応づければよい。

## 3.6ユーザー定義オブザーバー

差し込み可能なオブザーバーの他に、データの整合性管理や妥当性検査などのいわゆるビジネスロジックのうち状態の変更に反応して動作すべきものは、オブザーバーとしてプログラミングされる。整合性管理は、ある値が変化した場合で別の値が不整合になったときに適当な値へ変更する。妥当性検査は、値を検査し必要があればユーザーにフィードバックするなどエラー処理を行う。

ユーザー定義オブザーバーは、決められたインターフェースを持つクラスとして定義され、クラスは予めフレームワークに登録される。[3]これをモデルオブジェクトと関連付けるためにプログラム中で次のような宣言を行う。

---

[2] ビューは、関係データベースのビューと同様の目的をもつ。

[3] 我々はC++で実装したので、ファクトリ[2]を登録することになる。

*DefObserver("オブザーバークラス名", モデルオブジェクトの名前の集合);*

この宣言で指定された名前をもつモデルオブジェクトが生成されると自動的にオブザーバーオブジェクトも生成され関連付けられる。名前が集合になっているのは、一般に複数のモデルオブジェクトの状態を監視できるようにするためである。

### 3.7オブザーバー管理

オブザーバーパターンを使ったプログラミングの問題点の一つは、複数のオブザーバー間の関係を制御しにくい点である。オブザーバーの登録（または登録の削除）が他のオブザーバーの存在とは独立に行えることは、このプログラミング技法の利点である。そのために通常は、あるオブジェクトに複数のオブザーバーがある場合、変更の通知の順序は不定である。しかし、この順序をある程度制御したい場合もある。例えば、GUIの表示の変更は、最終的な状態を表示できればよいので、別の種類のオブザーバーの動作の後に通知を受けたいような場合である。制御の必要性の別の理由は、オブザーバー間にある実行時の間接的な呼び出し関係が問題になる場合である。あるオブザーバーが基本データの状態を更新した場合、別のオブザーバーを活性化するかもしれない。その結果、必要以上の回数オブザーバーが通知されたり、オブザーバー同士が相互に影響しあって無限に通知を繰り返すかもしれない。このようにオブザーバーどうしは、フレームワークによって動的に関連づけられるのであり、直接的にプログラムされるわけではないので、オブザーバーの追加や削除に対してプログラム全体の挙動が予期せぬ影響を受けてしまう可能性がある。

この問題に対処するために、本フレームワークでは次のような仕組みを入れている。

- まず、「インタラクション」を定義する。インタラクションは、マウスやキーボードからの入力イベントの発生時点から一連のイベント処理が終了して次のイベント待ち状態になるまでをさす。アプリケーションの実行は、有限時間で停止するインタラクションの繰り返しである。
- オブザーバーとモデルオブジェクトの関係の宣言中に優先度を指定する。
- あるオブジェクトの状態を変更する操作が実行されるとそのオブジェクトに登録されているオブザーバーは即時に通知されずに、待ち行列に積まれる。待ち行列は優先度付きである。優先度を付けることにより、オブザーバー間の順序を制御することが可能になる。
- 次に処理すべき操作がなくなると、待ち行列からもっとも優先度の高いオブザーバーを取り出し通知する。この操作を待ち行列が空になるまで繰り返す。待ち行列が空になって次に実行すべき操作がなくなったらインタラクションを終了する。
- 待ち行列から取り出されるオブザーバーの優先度は、あるインタラクションの中では単調に減少するようにする。もし、前に取り出されたオブザーバーの優先度よりも大きな優先度のオブザーバーが待ち行列に登録された場合は、これを無効とするか、例外処理を呼び出す。
- オブザーバーは、あるインタラクション中の活性化が高々１回であることを指定できるものとする。この場合、２回目の呼び出しの可能性がある場合は、無視または例外処理を呼び出す。

### 3.8編集機能

本フレームワークは、モデルオブジェクトへの状態変更操作に対する無制限回数のUndo/Redo機能を提供する。Undo/Redo操作により、過去のインタラクション間の境界の時点の状態のいづれかに遷移することができる。モデルオブジェクトの状態変化はフレームワークにより完全に管理されているので、この機能は自動的に提供される。

また、インタラクションの途中で、そのインタラクションの処理で起こったそれまでの状態変化を無効にすることができる。例えば、値の妥当性の検査において致命的なエラーがあった場合に、強制的にそのインタラクションの前の状態に戻るような場合である。状態が完全に管理されていないような環境で、このようにオブザーバーに拒否権を持たせることはプログラマーにとっては大きな負担である。

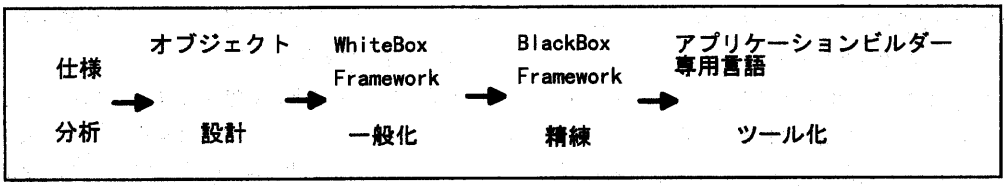


図3 フレームワーク設計の進化

その他、一般的な編集機能として、モデルオブジェクトの削除、コピー等が自動的に提供される。また、フォーカスの移動、デフォルトアクションの実行など、プログラミングが必要な一般的対話的操作の定義は、モデルオブジェクトに対してイベントハンドラーオブジェクトを関連付けることで行う。

## 4 フレームワーク設計の進化過程

我々は最終的には、フレームワークに加えてアプリケーションビルダーの試作まで行った。この最終段階にいたるまでのフレームワーク開発の中に図3に示すような進化の過程を見ることができる。本節では、最初に延べた二つの基本的な問題—フレームワークの設計の難しさと利用の難しさ—に関して、特にこの進化の過程に注目して議論する。

### 分析

初期の設計段階は、基本的な要求を明らかにすることから始めた。我々はオブジェクト指向方法論は用いなかったが、この段階で、対象とするアプリケーションのデータモデル、ビジネスルール、GUIのイメージの概要、そしてこのアプリケーションが本質的にエディタとしての機能を必要とすることが明らかになった。ただし、実際の要求仕様の全貌はこの時点では明らかではなく、仕様策定は開発と同時に進行した。

プロジェクトの目的はアプリケーションの再利用による開発コストの低減であったが、オブジェクト指向開発の高いスキルを前提とすることができないため、最終的にはできるだけプログラミングを少なくし、アプリケーションビルダーの実現をしたいという要求があった。

また、最初の時点では明確でなかったが、フレームワーク設計の観点から、アプリケーション内部が、*"安定度の異なるコンポーネント"*に分割できるかに注意が払われた。結果的には、次のようにまとめることができる。

- GUIの構成は、機能の追加やデータ内容の変化に応じた配置の最適化、ユーザー種別による違い、描画性能との兼合い等の要因で頻繁に変更が必要であり、非常に不安定である。
- データモデルの変更、特に追加は保守中にも定期的に起こりうる。また、同じアプリケーションでも顧客が変わると、データモデルも変更が必要である。
- データの整合や妥当性検査は、その顧客のポリシーであり、柔軟に変更できる必要がある。
- データベースは、既存のシステム（例えば病院では会計システム等）との連携が必要であるため、不安定要因の一つである。

これらの要求はほぼ独立に発生する。

- 一方、安定しているものとして、編集操作は従来のオフィスアプリケーションと同等のものがあればほぼ必要十分であり、また、データの検査やエラー処理の手順も比較的安定している。

### 設計

最初のプロトタイプでは、分析結果から自然に導かれるオブジェクトが導出された。対象としたアプリケーションは、処方箋だけでなく複数の種類のオーダーを処理する必要があるため、そのような共通部分を一般化しクラス階層が抽出された。また、このフレームワークの初期設計段階において、前節で説明したようなGUIを自動的に導出する、undo機能を入れるというようなアイデアが試された。この時点の設計の特徴を次のようにまとめることができる。

- 一般化に注意が払われ、継承を多用している。

特に多重継承も多く利用した。

- フレームワークの利用は、基本的にユーザーがサブクラスを定義するというものである。
- ビジネスドメインでの継承と、GUIの生成や編集機能の自動化といった、よりシステムよりの機能に関する継承とが混在している。ユーザーは両者を十分理解しないとサブクラスを設計できない。これは*"関心事の分離"*(separation of concerns)が十分できていないことを意味する。
- データの検査などの制御フローが十分整理されていない。このためサブクラスのプログラミングは容易ではない。

この時点の設計は、再利用や拡張に十分な自由度を持たせることに注意が払われた。実際に、アプリケーションの開発がこの段階から進められたが、十分なドキュメントが無いなど利用者には負担が大きく、オブジェクト技術のスキルという表現で問題点が指摘され始めた。

## ホワイトボックスからブラックボックスへ

文献でもフレームワーク利用の難しさがよく指摘されている[4]。一部の研究者は、比較的利用が容易なものとしてブラックボックスフレームワーク[1]という言葉を用いる。我々は、ブラックボックスフレームワークは、次のように特徴付けることができると考えている。

- 継承よりもオブジェクトの合成(composition)を積極的に用いる。
- フレームワークからインタフェースが知られている差し込み可能なコンポーネントがある程度そろっている。
- 可能ならば、フレームワークのパラメタは手続きではなくデータになっている。フレームワークは、このデータを解釈し挙動を決める。
- したがって、プログラムはより宣言的に表現されるようになる。

我々のフレームワークも、このような指針のもとに設計の変更を行った。行うべきことは基本的に、柔軟性をできるだけ損なわずに、手続き的な要素をできるだけフレームワークのユーザーから隠蔽することである。もちろん、完全にブラックボックス化できるとは限らないので、ホワイトボックスアプローチとのバランスが重要であり、また、ブラックボックス化されたコンポーネントが不十分である場合は、より高度なフレームワークの拡張という(ホワイトボックス的な)作業が必要なのは言うまでもない。

本質的に難しい問題は、ブラックボックス化をいかに行うかである。我々の設計例においても次のような、最初の分析や設計からボトムアップに自然に導かれるとは思われないような設計上の変更が観察できる。

- データモデルそのものをオブジェクトのグラフとして表現する。これによりフレームワークがデータ構造および状態の変更に関するメタ情報を利用することができる。これによりGUIコントロールやビューオブザーバーなどの利用がブラックボックス化される。
- データモデルと、関連するビジネスロジックを分離する。これによりデータモデルの定義自体もブラックボックス化する。これは一見情報隠蔽とは逆の立場であり、保守性を悪くする可能性が考えられるが、データモデルとロジックは安定性の差の観点から分離する根拠があり、分離により関連するモジュールが分散する問題よりも、分離により得られる利点のほうが大きいとの判断による。

このような設計上の判断を導くことを保証できる一般的な方法論を考えるのは困難に思われる。かなりの部分は設計者の創造性に依存し、設計上もっとも困難な問題である。しかしながら、結果的に成功した例としてデザインパターン[2]を利用することはできる。例えば、我々の設計の例では、安定度の違いを吸収したり、関心事の分離を実現するためにオブザーバーパターンが使えないかどうかを設計の中で積極的に探している。また、優先度によるオブザーバーの通知制御のように試行錯誤により行き着いた設計を新しいパターンとして記述しておけば、別の場面では、検討材料として利用することができるであろう。フレームワークを洗練する過程では、デザインパターンをトップダウン的に検討するのが有効である。

### アプリケーションビルダー

GUIのレイアウトの設計のみならず、アプリケーションを作成するのにビルダーツールが利用できれば便利である。アプリケーションビルダーは、対話的な直接操作を用いてコンポーネントの合成をサポートする。コンポーネントの検索や合成時のコンポーネント間の整合性の検査などが即時に行うことができ、フレームワークの学習コストをさげ、アプリケーション開発の生産性も向上することが期待できる。

アプリケーションビルダーは、各領域で有効な設計（フレームワーク）や再利用可能な部品があることが前提である。現状をみると、ウィンドウシステムにおける画面の設計をするリソースエディタや特定のアプリケーション開発環境の例を除いては、ビルダーが広く利用できているとは言い難い。一般に、領域に特化したアプリケーションビルダーを開発するのはコストがかかる。

我々の場合も、これまで述べたようなフレームワーク進化のプロセスは時間のかかるものであった。しかし、フレームワークがブラックボックス化できてくると、差し込み可能なコンポーネントの選択や宣言的記述は比較的容易にツールに反映できる。したがって、自然にツール化が期待できる。これに対して手続き的記述に対しては、ビルダーツールを作成したとしてもテキストエディタ以上の効果を出すのは容易ではない。

我々は実際に、本フレームワークに基づいてデータ入力システムのためのアプリケーションビルダーを試作した[3]。[4]このビルダーは、モデルオブジェクトの定義とGUIの定義を宣言的に行う部分を支援する。

以上は、フレームワーク進化の典型的な流れを示していると思われる。もちろんその中で次の段階に移るにあたっては、設計の再分析や新たな開発にかかるコストに注意し、効果と必要性を検討しなければならない。

今後必要な技術としては、フレームワークを利用しやすくするようにブラックボックス化するためのガイドラインやデザインパターンの整理、そしてアプリケーションビルダーをコストをかけないで作るための構築環境（あるいはメタビルダー）の開発が挙げられる。

## 5 まとめ

我々は、複雑な階層構造をもつデータの入力システムを開発するのに利用できるフレームワークを開発した。このフレームワークを用いると基本的な編集機能は自動的に提供される。また、GUIは手続き的なプログラミングをせずに柔軟に作成・変更することができる。このフレームワークの設計にあたっては、プレゼンテーション、ビジネスロジック、データベースに関わる各コンポーネントを独立に変更しやすいように分離し、更にそれらの統合が容易になるようにアーキテクチャが決められた。

フレームワークの設計は進化する。一般化と柔軟性を重視した初期の設計から必要以上の自由度を減らし、ブラックボックス化することにより、利用は容易になり、またそのフレームワークに特化したアプリケーションビルダーを低コストで開発できる可能性が高くなる。

## 参考文献


[1] Fayad, M.E. and Schmidt, D.C.: "Object-Oriented Application Frameworks," Comm. of ACM, Vol.40, No.10, 1997.

[2] Gamma, E. Helm, R., Johnson R. and Vlissides, J.: "Design Pattern: Elements of Reusable Software Architecture," Addison-Wesley, 1995.

[3] 広瀬紳一：階層構造を持つデータの入力システムのためのアプリケーション・ビルダーの試作, 情報処理学会 第55回全国大会, 1997.

[4] Johnson, R. and Foote, E.: "Designing Reusable Classes," J. of Object-Oriented Programming, Vol. 1, No. 2, 1988.


[4] ちなみに、我々のフレームワークはデータの編集を対象としたものであり、ビルダーを設計データの編集ととらえることにより、そのフレームワーク自身を試作に用いることができた。