モデル/ビュー分離アーキテクチャBeaMの機構とその評価

鷲崎弘宜† 白銀純子† 深澤良彰†

GUIアプリケーションの開発時に，最も基本的な汎用コンポーネントを組み合せ，さらにアプリケーションロジックを付加することで，特定のアプリケーションドメインに特化された，クラスレベルでの再利用を目的とするGUIアプリケーション部品を開発，利用することがある．従来の手法では，GUIアプリケーション部品内部でのアプリケーションロジックとGUI部分との依存性の排除は不完全なものであり，結果として部品全体の再利用性および拡張性の欠如をもたらす．本稿ではGUIアプリケーション部品の開発において，アプリケーションロジック部とGUI部が相互に依存しないアーキテクチャBeaMを提案し，Java言語による実装を行った結果についての評価を行う．

The Mechanism of the Model-View Separation Architecture BeaM and Its Evaluation

Hironori Washizaki,† Junko Shirogane† and Yoshiaki Fukazawa†

GUI application components, which are developed by combining the most atomic GUI components and appending the required application logic, have been widely utilized. Their target is domain-specific and instance-level software reuse. In conventional methods, the separation between its application logic part and GUI part is incomplete. As the results, enough reusability and extensibility of the GUI application components can not be obtained. In this paper, we propose a Model-View separation architecture, "BeaM", which realizes the fully independence between application logic part and GUI part. Also the evaluation results are analyzed based on the implementation in Java language.

1. はじめに

近年，オブジェクト指向技術に基づいたソフトウェア部品の組合せ開発を中心とするコンポーネント技術が，ソフトウェア開発期間の短縮と開発コストの削減を目的として広く注目されてきている[1)]．この普及にともない，インスタンスレベルでの再利用を目的として，コンポーネント対応RAD（Rapid Application Development）ツールやOS等にコンポーネントライブラリが提供されるようになってきている．

このようなライブラリに含まれるパネル，ボタン等の汎用コンポーネントをアプリケーションの最小構成要素として組み合せ，アプリケーションロジックを付加することで，新たなコンポーネントを開発し利用することがある．このようにして開発された複合コンポーネントは，クラスレベルでの再利用が行われ，目的とするアプリケーションドメインに依存することからアプリケーション部品と呼ばれる．

汎用コンポーネントには，可視的な（visible）なものと非可視的（invisible）なものがある．本稿では，可視的なものを汎用GUI（Graphical User Interface）コンポーネントと呼び，また，汎用GUIコンポーネントを内包し，実行時にGUIを提供するアプリケーション部品をGUIアプリケーション部品と呼ぶ．

GUIアプリケーション部品を開発・利用する目的は，サブシステムとして利用する方法と，再利用を目的とするサービスパッケージングに大別される．

サブシステムとして位置付けられたグループ化複合オブジェクト群は，ビジネスコンポーネントと呼ばれることもある[2)]．ビジネスコンポーネントとしての意味の強いGUIアプリケーション部品は，アプリケーションの開発過程において開発・利用されるものであるため，必ずしも再利用性を重視したものではないが，将来にわたっての保守を考えるとき柔軟な拡張性を備えることが重要である．特にGUIの構造設計は頻繁に変更されることが多く[3)]，GUIアプリケーション部品の内部アーキテクチャは，内部の汎用GUIコンポー

† 早稲田大学理工学部
School of Science and Engineering, Waseda University

ネントの修正・変更と，内部のアプリケーションロジック中で対象汎用GUIコンポーネントを直接参照している箇所（GUI参照部）のそれにともなう修正に，柔軟に対応できる必要がある．

GUIをともなうソフトウェアサービスのパッケージ化では，再利用性が最重要視され，内部のアプリケーションロジック部やGUI部の修正，変更に柔軟に対処できなければならない．

GUIアプリケーション部品においては，GUI部とアプリケーションロジック部間の依存性を低減することが，再利用性・拡張性の向上につながる．このための手法として，アプリケーションに特化されたモデル部とGUIであるビュー部を分離するModel-View Separation（モデル/ビュー分離）パターン[4]，およびそれを実現したModel-View-Controller（MVC）アーキテクチャ[3]が使われてきた．

しかし，これまでに提案・実現されてきているモデル/ビュー分離では，アプリケーションロジック部内にGUI参照部が潜在するため，結果としてアプリケーションロジック部とGUI部との分離が不完全なものとなる．

本稿では，GUIアプリケーション部品内部において，アプリケーションロジック部からGUI参照部を完全に分離することで，アプリケーションロジック部とGUI部との間で，互いを認知することなく通信を行うアーキテクチャBeaMを提案し，Java言語による実装を行った結果について，開発方法の説明とともに評価を行う．

2. GUIアプリケーション部品とモデル/ビュー分離

GUIアプリケーション部品を開発する際に，内部においてGUI部とアプリケーションロジック部間の依存性を低減する重要性が広く認識され，モデル/ビュー分離の仕組みの内部構造への適用は一般に行われている．そのアーキテクチャとして，RADツールやOSが実現するものがその入手のやさしさにより一般的に普及している．

2.1 従来のGUIアプリケーション部品

GUIを提供可能なRADツールやOSは，標準的な仕組みとして，不完全なモデル/ビュー分離を実現可能なオブジェクト指向アプリケーションフレームワークを提供することが多い．例としてMacApp[5]や，MFCにおけるDocument-Viewアーキテクチャ[6]，Java2SEにおけるAWT/JFC[7]等があげられる．それらの標準で提供されるアプリケーションフレームワークにより

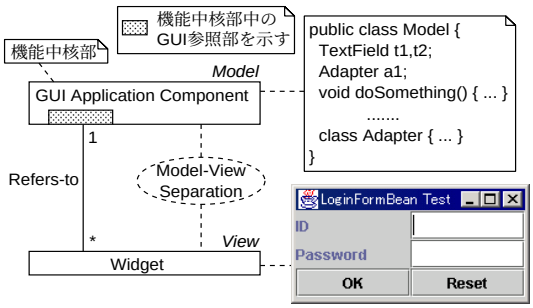

図1 従来のGUIアプリケーション部品
Fig. 1 Conventional GUI application component.

実現するGUIアプリケーション部品は，汎用GUIコンポーネント群より構成されるGUI部と，目的とするアプリケーションの問題領域に特化したアプリケーションロジック部よりなる．UML[8]を用いて図式化したものを**図1**に示す．

2.1.1 部品構成

(1) GUI部

RADツールもしくはOS付属のコンポーネントライブラリによって提供され，稼働対象とするコンポーネントシステムに対応した汎用GUIコンポーネント（Widget）群より構成される．GUI部はロジックを持たず，GUI出力についての設定調整のみ可能である．

(2) アプリケーションロジック部

GUIアプリケーション部品の構成要素のうち，GUI部以外をアプリケーションロジック部と呼ぶ．標準のアプリケーションフレームワークは，GUIその他からのイベント刺激を全体の動作制御の中心とするイベント駆動型であるため，アプリケーションの問題領域にかかわる処理とデータを保持する機能中核部に加え，その結果の正しいGUI部への出力とGUI部からのユーザ入力イベント通知の受取り・対応について責任を持つGUI参照部から構成される．

- 機能中核部

目的とするGUIアプリケーション部品が，組み込まれるアプリケーションにおいて責任を持つべき問題対象のデータと，データに対する操作手続きを持つ．

- GUI参照部

GUI部を構成する各汎用GUIコンポーネントへの参照を保持し，アプリケーションロジック部からGUI部への方向についての通信，各汎用GUIコンポーネントの直接操作を行う部分である．

2.1.2 通信方式

GUI部とアプリケーションロジック部間の通信方式は，標準のアプリケーションフレームワークでは，次のように特徴付けられる．

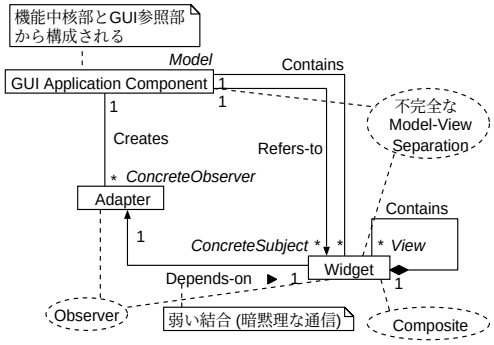

図 2 従来の GUI アプリケーション部品内部クラス図
Fig. 2 Class diagram of conventional GUI application component's inside.

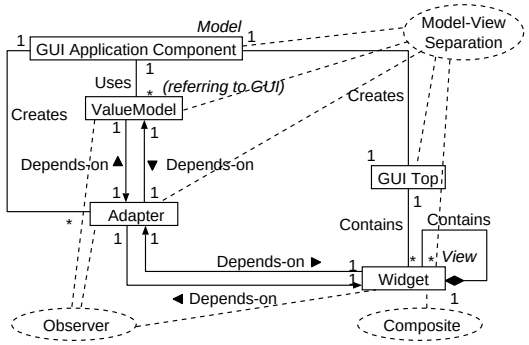

図 3 完全なモデル/ビュー分離の内部クラス図
Fig. 3 Class diagram of complete Model-View separated GUI application component.

(1) GUI 部 → アプリケーションロジック部
GUI アプリケーション部品内部において，アプリケーションロジック部は汎用 GUI コンポーネントからのイベント通知を正しく受け取り反応する必要がある．この処理は，その方向についての依存性を低減するために，Observer パターン[9]を利用した各汎用 GUI コンポーネントから登録されたアプリケーションロジック部への暗黙理な通信か，もしくは Template Method パターン[9]を利用したアプリケーションロジック部中のフックメソッドへのコールバックで対処する．例として Java2SE の委譲イベントモデルにおける，GUI 部からアプリケーションロジック部方向への通信方式を**図 2** に示す．
アダプタを介在させた Observer パターンの適用により暗黙理な通知が行われるため，各汎用 GUI コンポーネントとアプリケーションロジック部の依存性は低減され，この方向についての GUI 部とアプリケーションロジック部との独立性は保たれる（図 2：[Adapter] Depends-on [View]）．
(2) アプリケーションロジック部 →GUI 部
GUI アプリケーション部品は問題対象とするデータを保持し，その変化や何らかのアプリケーションロジックの動作状況にともない，自身の GUI を更新しユーザに表示する必要がある．標準的なモデル/ビュー分離において，この処理はアプリケーションロジック部の内部から直接，各汎用 GUI コンポーネントを参照し操作を行う．（図 2：[Model] Refers-to [View]）．

このアプリケーションロジック部から GUI 部への片方向の強い依存性が，標準アプリケーションフレームワークによるモデル/ビュー分離が不完全なことの原因であり，以下の問題を引き起こす．

- 保守・拡張性の低下

各汎用 GUI コンポーネントへのアプリケーションロジック部からの直接参照により，仕様変更にともなう GUI の一部またはすべての修正が必要なとき，アダプタだけでなくアプリケーションロジック部内の GUI 参照部をも修正する必要が生じる．GUI 参照部は，本来のアプリケーションロジック内に潜在するため，修正の必要な箇所が判別しにくく，結果として拡張についてのコストの増大と保守性の低下をもたらす．

- 分離同時開発性の欠如

開発者は，GUI アプリケーション部品の設計・開発過程において，GUI 部とアプリケーションロジック部を切り分けて作業を進めることができない．一般に，GUI の設計者とアプリケーションロジックの設計者は異なるため，独立して開発を進められることが望ましいが，内部の依存形態によりそのような分離同時開発は不可能である．

2.2 完全なモデル/ビュー分離

アプリケーションロジック部から GUI 参照部を分離独立させることで，内部におけるアプリケーションロジック部と GUI 部間の相互依存性をより排除する試みがなされている．例として，Smalltalk における Pluggable-MVC[10]，および Java についての追加的な UI（User-Interface）フレームワーク Jaguar[11]等があげられる．

これらは共通してモデルを，機能中核部と，GUI 参照部としてのモデル用アダプタ群（ValueModel）に分離している．内部におけるアプリケーションロジック部と GUI 部の間の通信に，両方向についての完全な Observer パターンを適用することで依存性を排除し，完全なモデル/ビュー分離を実現している（**図 3**）．

各汎用 GUI コンポーネントが表現するデータ出力と，アプリケーションロジック部内のデータを，アダプタにより 1 対 1 対応させることで，標準的なモデル/ビュー分離におけるアプリケーションロジック部から GUI

部方向への依存性を排除する（図3：[ValueModel] Depends-on [Adapter] Depends-on [Widget]，および [Widget] Depends-on [Adapter] Depends-on [ValueModel]）．

しかしながら，このようなObserverの合成による完全なモデル/ビュー分離の実現は，データと汎用GUIコンポーネントのプロパティとの1対1の対応付けに起因して，両者間を結び付けるアダプタの数が肥大化し，結果として両者間のアダプタおよびメッセージフローの複雑化をもたらす．

これは仕様変更時において修正の必要な箇所が，GUIアプリケーション部品内にアダプタ群として散在することを意味し，将来における両者の依存性変化にきわめて弱い構造を持つ[12]．特に新しいユーザインタフェース（ビュー）の追加やデータモデルの大幅な変更について，修正・追加箇所を特定し対応することは大幅なコストの増大につながる．

3. BeaMアーキテクチャの提案

任意のコンポーネントシステムに沿ったGUIアプリケーション部品内部において，アプリケーションロジック部とGUI部の分離を解決し，両者の完全な独立性と，仕様変更時における修正箇所の局所化，および内部における柔軟な処理記述を同時に実現するアーキテクチャBeaM（Bean for Model-View Separation）を提案する．BeaMアーキテクチャを設計・実装するにあたり，汎用GUIコンポーネントの扱いとイベント駆動型の制御を可能とするコンポーネントシステムを基盤とし，そのコンポーネントシステムを形成するコンポーネントフレームワーク上への追加的な実装形態をとることとした．汎用GUIコンポーネントの扱いとイベント駆動型の制御を可能とするコンポーネントシステムとして，たとえばActiveX/COMやJavaBeans等がある[13]．

3.1 アーキテクチャモデルの設計指針

BeaMアーキテクチャモデルでは，名前通知，中央管理，および外部転送の実現を基本的な設計指針とした．

- 名前通知

アプリケーションロジック部とGUI部間の通信には，両方向について，Jaguar等で用いられている完全なモデル/ビュー分離方式と同様に，アダプタを介したObserverパターンを用いる．Observerパターンには，更新通知の際に，更新にかかわる詳細な情報のすべてを通知するpushモデルと，更新されたことのみの最低限な情報を通知して，詳細な情報の取得は通知の受取り側に行わせるpullモデルがある[9]．本アーキテクチャでは，アプリケーションロジック部とGUI部間の通信には，依存性の排除とそれにともなう再利用性に優れるpullモデルを採用し，両者の更新通知時の最低限な情報として単純な名前を用いて，名前を受け取った際の情報取得動作を設定することで，適切な通信を行う．

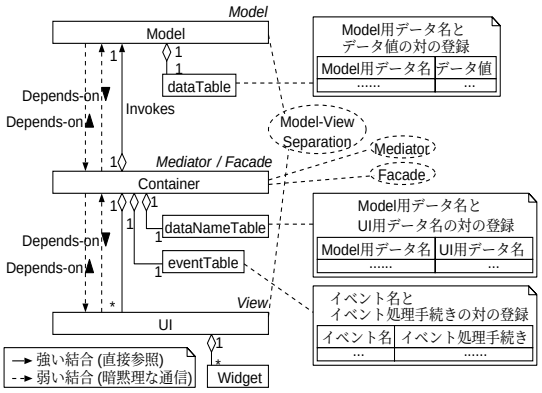

図4 BeaMアーキテクチャモデルのクラス図
Fig. 4 Class diagram of BeaM architecture.

- 中央管理

前述の名前通知の導入により，アプリケーションロジック部とGUI部間の依存性は排除されるが，アダプタの数が肥大化するという問題を依然として持つ．そこで，Mediatorパターン[9]の導入により，両部間の通信を1つのMediatorを介して実現し，全体の内部構造の複雑化を防ぐ．名前通知の実現に必要なアダプタ群を，1つのMediatorとしてまとめた．Mediatorは，片方から通知された名前を受け取り，もう一方に解釈可能な名前に変換するための名前変換表を内部に持つ．

- 外部転送

GUIアプリケーション部品は，対象とするコンポーネントシステムにおいて，1つのコンポーネントとして振る舞う必要がある．この外部への振舞いは，Facadeパターン[9]の適用により，アプリケーションロジック部とGUI部への本来外部から許可されるべきアクセス，および，アプリケーションロジック部からの本来外部へ通知すべきイベント通知を，中央管理におけるMediatorをFacadeとして外部へ転送することで実現する．

3.2 基本構造

設計指針に基づき，BeaMアーキテクチャは内部において，以下のModel部，UI部，Container部の3要素より構成される（**図4**）．

(1) Model部

GUIアプリケーション部品が保持すべきデータの実際

の値と，Container との通信に用いる Model 用データ名を対として登録するデータ表（dataTable）を持ち，アプリケーションロジックとしてデータに対する操作手続きを持つ．Model 用データ名はデータと 1 対 1 に対応し，Model 部内で一意である必要がある．従来の開発手法におけるアプリケーションロジック部から GUI 参照部を除いた機能中核部にあたる．

(2) UI 部

GUI アプリケーション部品としての処理の結果を GUI の更新としてユーザに伝え，また逆にユーザからの入力を GUI を経由して受け付け，GUI アプリケーション部品としての内部データに反映する役割を持つ．UI 部は，対象とするコンポーネントシステムに従って提供される汎用 GUI コンポーネントを内部に持つ．Container 部との通信は UI 用データ名とイベント名を介して行われる．UI 用データ名は，UI 部内の各汎用 GUI コンポーネントの GUI 出力用データと 1 対 1 に対応し，UI 部内で一意である必要がある．イベント名は，各汎用 GUI コンポーネントへのユーザ操作に 1 対 1 に対応し，UI 部内で一意である必要がある．従来の開発手法における GUI 部にあたる．

(3) Container 部

1 つの Model 部と 1 つ以上の UI 部を生成し内包する．Model 部と UI 部間で適切に通信を行うための仲介者（Mediator）として，内部にデータ名変換表（dataNameTable）とイベント表（eventTable）を持つ．データ名変換表には，Model 用データ名と UI 用データ名が対として登録され，データ名の通知時に，もう一方で解釈可能なデータ名への変換と通知を行う．イベント表には，イベント名とイベント処理手続きが対として登録され，イベント名の通知時に，対応するイベント処理手続きが実行される．イベント処理手続きは，イベントが起きた際に，Model 部に対して行うべき操作手続きと，イベントにともなう入力データに対応する UI 用データ名をカプセル化したものである．操作手続きは，Command パターン[9]の適用によって，対象とするコンポーネントフレームワークで実現可能な Command オブジェクトとして実装される．データ名変換表およびイベント表に対する登録は固定的ではなく，動的に登録の追加・削除が可能である．従来の開発手法におけるアダプタ群およびモデル用アダプタ群にあたる．

3.3 通信方式

Model 部と UI 部間の通信は，両者の独立性を保つためにすべて Container 部を介して行われる．また，外部との通信もすべて Container 部を介して行われる．

(1) データ共有機構（Model 部 ↔Container 部 ↔UI 部）

Model 部内のデータのうち，UI 部と共有すべきデータを設計時に定め，両方向についての Observer パターンの pull モデルに従うデータ名の通知により，Model 部中のデータと UI 部による GUI 出力の完全な一致，データの自動的な共有メカニズムを提供する．データ名を渡された側が自ら自動的に実際の値を取得する仕組みであり，データ共有についての両方向の依存性を極力排除している（図 4：[Model] Depends-on [Container] Depends-on [UI]，および [UI] Depends-on [Container] Depends-on [Model]）．

Model 部内の共有データは初期化時に，Model 部内のデータ表に Model 用データ名と対として登録される．値の変更時には，Model 用データ名が Container 部に通知され，Container 部内のデータ名変換表を用いて自動的に UI 用データ名に変換されて，すべての UI 部へ通知される．UI 部は通知された UI 用データ名を用いて，Container 部を経由し Model 部から値を得て（Observer パターンの pull モデル），GUI 出力を自動更新する．

逆に UI 部内の汎用 GUI コンポーネントに値が入力されると，対応する UI 用データ名が Container 部により Model 用データ名に変換されて Model 部に通知される．Model 部は通知された Model 用データ名を用いて，Container 部を経由し UI 部内の入力データを取得し自身のデータを自動更新する．

(2) イベント処理機構（Model 部 ←Container 部 ↔UI 部）

UI 部におけるユーザ入力にともなうイベントは，すべて GUI アプリケーション部品内部で処理される．このイベントは，UI 部より Container 部へ対応するイベント名として通知される．通知後，イベント名をキーとして，Container 部のイベント表に登録済みの対応するイベント処理手続きが取り出され，最初にイベント処理手続きより得られる UI 用データ名を用いた UI 部内の入力データの通知・自動更新が行われる（(1) データ共有機構参照）．続いて，イベント処理手続きより得られる操作手続きを用いて，Model 部に対する操作が Template Method パターンの適用により自動実行される．

Container 部は操作手続きを用いた Model 部への直接操作を行うが，この設定は Command パターンの，サブクラス化ではなくパラメータ化による適用により，イベント名と Model 部の対象操作を指定する方法（および後述の 4.2 節 (3) Integrator）を採用し，Con-

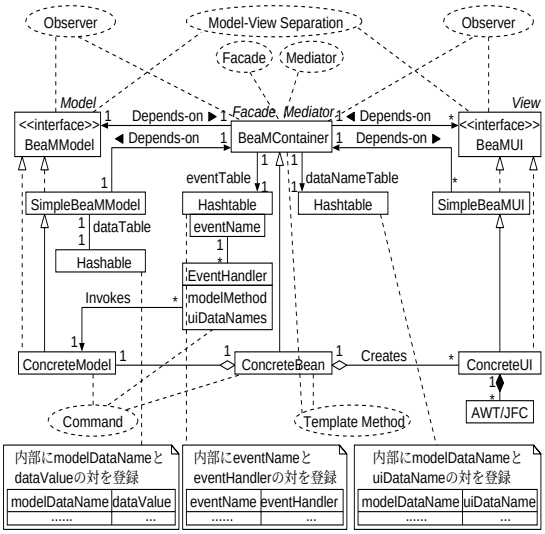

図 5 BeaM アーキテクチャの実装クラス図
Fig. 5 Class diagram of BeaM component.

tainer 部が Model 部の特定なインタフェースに依存することを防ぐ（図 4：[Container] Invokes [Model]）．UI 部と Container 部間の両方向の通信については Observer パターンの pull モデルの適用により依存性が極力排除されている（図 4：[UI] Depends-on [Container]，および [Container] Depends-on [UI]）．

(3) 外部転送機構

Model 部・UI 部の本来 GUI アプリケーション部品としての機能を，Container 部を Facade として外部に対して公開し，外部からのアクセスを転送することで，Container 部は外部に対して GUI アプリケーション部品として振る舞う．

3.4 JavaBeans への実装

BeaM アーキテクチャモデルの実装を，Java 言語におけるコンポーネントシステム JavaBeans[14] 上に追加的に行った．構成するクラス図を**図 5** に示す．

(1) BeaMModel

BeaM アーキテクチャモデルの Model 部に対応し，BeaMModel インタフェースを実装した Java オブジェクトである．BeaMContainer オブジェクトへの参照を持ち，内部にハッシュテーブルを用いたデータ表の実装を持つ．

(2) BeaMUI

BeaM アーキテクチャモデルの UI 部に対応し，BeaMUI インタフェースを実装し java.awt.Container クラスを継承した Java オブジェクトである．内部に汎用 GUI コンポーネントとしての AWT/JFC コンポーネントを持ち，GUI アプリケーション部品としての GUI を構成する．

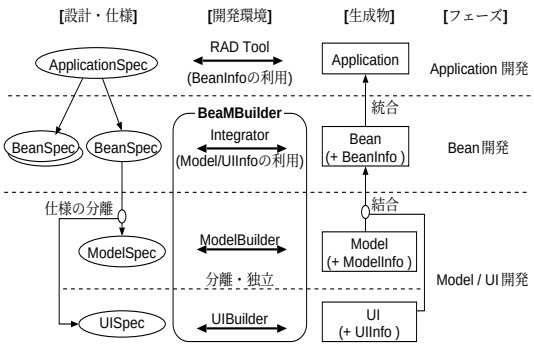

図 6 BeaMBuilder によるアプリケーション開発モデル
Fig. 6 GUI application component development model using BeaMBuilder.

(3) BeaMContainer

BeaM アーキテクチャモデルの Container 部に対応し，BeaMContainer クラスを継承した Java オブジェクトである．1 つの BeaMModel オブジェクトと，1 つ以上の BeaMUI オブジェクトを内包する．内部に，ハッシュテーブルを用いたデータ名変換表およびイベント表の実装を持つ．イベント名に対応付けられるイベント処理手続きは，EventHandler オブジェクトが対応する．EventHandler オブジェクトは，Model に対する処理手続きとしての BeaMModel オブジェクトの Method オブジェクトと，UI 用データ名を持つ．外部に対する JavaBeans コンポーネントとしての振舞いは，BeaMModel オブジェクトおよび BeaMUI オブジェクトのプロパティへの外部からのアクセスと，外部へのイベント通知を，BeaMContainer オブジェクトを経由して転送することで実現する．

4. BeaM 開発支援環境

4.1 設計手法と開発モデル

BeaM アーキテクチャはその目的をモデルとビューの完全な分離・同時開発に置くため，開発支援ツールを用いた実装段階の前に，Bean の要求仕様定義をモデルとビューの 2 つの仕様に分離する必要がある．この仕様の分離は，VMT（ビジュアル・モデル化技法）分析フェーズに基づくユーザ・ビュープロトタイピングとユースケースモデリング[15]や，ユースケース写像 UI 設計手法に基づくモデルとビューの要求仕様定義の分離[16]等により達成され，その後開発支援ツールを用い Model 部・UI 部をそれぞれ独立に設計・実装する．この開発モデルを**図 6** に示す．

4.2 統合開発環境 BeaMBuilder

コンポーネントは本来 WYSIWYG を実現する

RAD ツールによりビジュアルに使われることを目的とするため[14]，BeaM アーキテクチャに沿った BeaM コンポーネントを，ビジュアルに開発可能なツールが必要である．本稿で提案する BeaMBuilder は BeaM コンポーネントを開発するためのツールであり，ModelBuilder（BeaMModel 用），UIBuilder（BeaMUI 用），Integrator（BeaMContainer 用）から構成される．

（1） ModelBuilder

コーディングを中心として BeaMModel オブジェクトを開発する．データ定義時に，共有データとして Model 用データ名とともに設定することで，データ共有に必要なコードを自動生成し，ModelInfo オブジェクト内に共有データの設定情報を登録する．

（2） UIBuilder

BeaMUI オブジェクトのビジュアルな GUI 設計を可能とし，各属性値の設定時に，UI 用データ名を用いて，将来 BeaMContainer オブジェクトで BeaMModel オブジェクト内のデータと対応付け可能なように宣言することで，データ共有に必要なコードを自動生成する．イベントの定義は，汎用 GUI コンポーネントから受け取りたいイベントについて，イベント名を設定することで行う．共有データおよびイベントについての情報は，UIInfo オブジェクト内に自動的に登録される．

（3） Integrator

ModelInfo・UIInfo 両オブジェクトより共有データについての情報を入手し，Model・UI 両オブジェクト間でのデータ名の結び付けをビジュアルに行う．また，BeaMUI オブジェクトからの各イベント名に対する BeaMModel オブジェクトの処理メソッドを選択・決定する．設定後に BeaMContainer クラスのコードを自動生成し，BeanInfo オブジェクト内に JavaBeans コンポーネントとして外部に示すべき情報（プロパティ・外部へのイベント）を登録する．

5. 評　　価

BeaMBuilder による開発手法を，システムへのログイン時に使用される LoginFormBean の開発を通して示し，その従来手法に対する修正・拡張についての優位性を述べる．従来の開発ツールとして JBuilder2.0[17] を比較対象とする．

LoginFormBean は，AuthenticatorBean[18] の機能を簡略化したもので，ID とパスワードの入力を行う（**図 7**）．またプロパティとして id および password を持ち，各プロパティを取得・設定するためのアクセスメソッド（getId・setId，getPassword・setPassword）が外部に公開される．

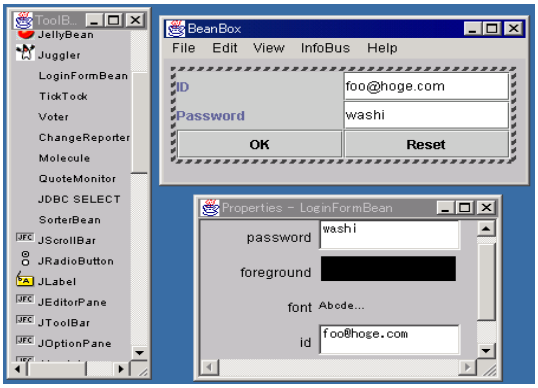

図 7 BeanBox 上での LoginFormBean
Fig. 7 LoginFormBean running on BeanBox.

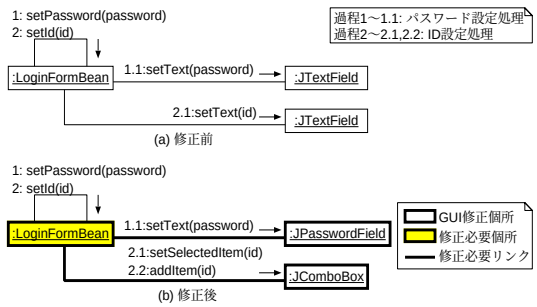

図 8 従来手法：既存 GUI 定義修正時のコラボレーション図
Fig. 8 Traditional style: code modifications for GUI customization.

5.1 定性的評価

（1） 部分単位での詳細なカスタマイズ

パスワードを入力する GUI 部品の JTextField から JPasswordField への変更，および，ID を入力する GUI 部品の JTextField から JComboBox への変更作業を行う．

従来の手法では，記述されたアプリケーションロジック部内の両 JTextField についての操作記述（GUI 上の値の取得，更新操作）をすべて変更する必要がある．特に，**図 8** に示すように，パスワード設定処理（過程 1）および ID 設定処理（過程 2）について，プロパティ id および password のアクセスメソッド内（setId，setPassword）から各 JTextField へ操作（setText）が行われているため，修正作業にともない，各アクセスメソッドを修正する必要がある．コンポーネントのプロパティは，アプリケーション部品にとって重要なデータであり，GUI 設計の変更時にそのアクセスメソッドの修正が本来なされるべきでない．これによりコンポーネントの著しい保守性の低下を招き，誤りの生じる可能性も高い．

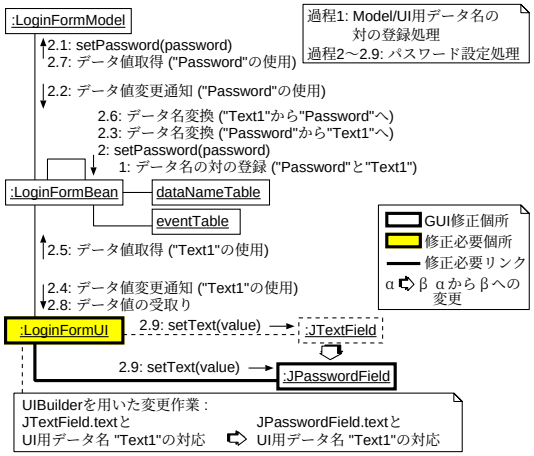

図 9 BeaMBuilder：既存 GUI 定義修正時のコラボレーション図
Fig. 9 BeaMBuilder: visual modifications for GUI customization.

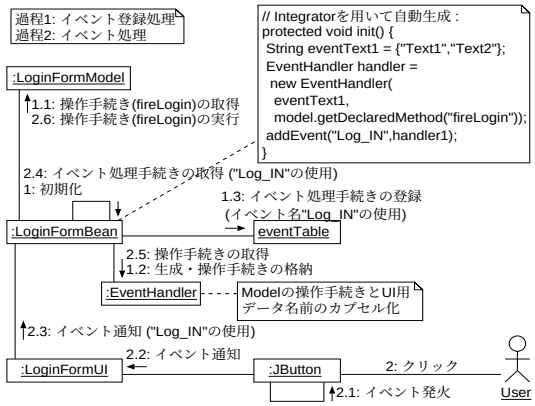

図 10 BeaMBuilder：イベント処理コラボレーション図
Fig. 10 BeaMBuilder: collaboration diagram for event handling.

これに対して BeaMBuilder を用いた開発においては，**図 9** に示すように，パスワード設定処理（過程 2）について LoginFormModel から直接 JTextField を操作することはなく，Model/UI 用データ名を介して更新処理が行われるため，変更作業は，UIBuilder を用いて変更前と同じ UI 用データ名を設定するだけでよく，BeaMModel オブジェクトについていっさいの修正を必要としない．ID 設定処理についても同様である．UIBuilder によるデータ名の設定は，各 GUI コンポーネントのプロパティについてビジュアルに編集可能であるため，GUI アプリケーション部品の GUI 変更は容易であり，正確なコード生成が保証される．また，従来手法では，ロジックの修正に関して，開発者が必ず直接手作業で修正する必要があるのに対して，BeaMBuilder を用いれば，ユーザ入力にともなうイベントへの対応処理を変更する作業に関して，EventHandler オブジェクトを用いたビジュアルな設定によりイベント処理コードを自動生成するため，プロパティの追加等の本質的なアプリケーションロジックの変更でない限り手作業による修正作業を必要としない．**図 10** は，BeaMBuilder によって開発された LoginFormBean のイベント登録処理（過程 1）およびイベント処理（過程 2）を示す．図 10 から，Integrator によるビジュアルな事前の設定により以下が分かる．

- LoginFormBean について両処理に必要なすべてのプログラムコードは自動生成される．
- 実行時のイベント処理において，LoginFormUI は LoginFormModel を参照することなく，ボタン（JButton）からのイベントを LoginFormBean を経由して通知し，適切に LoginFormModel の操作手続きが起動される．

このように BeaMBuilder を用いれば，ロジック・GUI の一方についての変更・修正の際に，BeaMContainer オブジェクトにおける結合定義の修正が必要になる可能性はあるが，結合定義は BeaMModel・BeaMUI 両オブジェクトより独立しており，他方のオブジェクトに影響が出ることはなく，仕様変更にともなう修正箇所の局所化に成功しており，正確なコードの自動生成により信頼性も高い．

(2) データの GUI 出力一貫性の実現

5.1 節 (1) で見たように，LoginFormBean においてプロパティpassword は更新時に，更新のきっかけがアプリケーションロジックか GUI を用いたユーザ入力かにかかわらず，その GUI 出力を適切に更新する必要がある．これはアプリケーション部品開発者が，部品内部のデータと GUI 出力との対応付けを注意深く設計することで回避される．しかし BeaMBuilder を用いれば，共有データの変化は BeaMContainer オブジェクトを経由して BeaMUI オブジェクトに自動通知されるため，BeaMUI オブジェクトによる GUI 表示は BeaMModel オブジェクト内の実際の値と同一であることが保証される．よって開発者は，BeaMModel・BeaMUI 両オブジェクトでのデータの整合性を注意する必要がなく，アプリケーション部品のロジックと GUI を完全に分離した設計を促進できる．

(3) 同一ロジックに対する複数ビューの提供

従来の手法では，アプリケーション部品内で同一処理ロジックに対して複数の GUI 出力を提供するには，アプリケーション部品の GUI 出力にかかわる処理記述を直接修正するか，もしくは，GUI 出力処理を切り離し

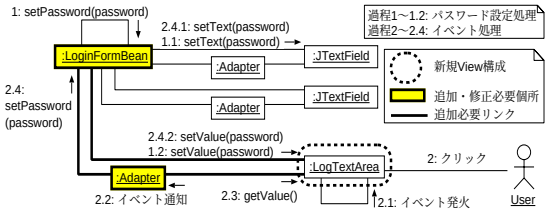

図 11 従来手法：新規 GUI 出力追加時のコラボレーション図
Fig. 11 Traditional style: collaboration diagram for GUI addition.

て，別の複数の GUI 出力を行うコンポーネントをデータの変更に対するイベントリスナとして登録するしか実現方法がなかった．たとえば LoginFormBean では，新たな GUI 出力としてログ出力機能を持ったコンポーネント LogTextArea を追加するとき，**図 11** に示すように，パスワード設定処理（過程 1）の結果を正しく表示するために，プロパティ password の値の更新時に，新しい LogTextArea への操作記述（setValue）を LoginFormBean 中（setPassword）に追加する必要がある．また，LogTextArea からパスワード入力を受け付ける設定（過程 2）を行うと，その入力を受け取った際に，他のビューとしての汎用 GUI コンポーネントも同じ値を出力するようにする必要がある．このように従来手法において，新しいビューを追加する際に，修正・追加に必要な箇所を見つけ出し適切に作業を行うことは容易ではない．

これに対して，BeaMBuilder を用いた BeaM コンポーネントであれば，BeaMModel オブジェクトが BeaMUI オブジェクトの存在を知ることなく，BeaMContainer オブジェクトを介して自動的にデータの更新通知が行われるため，BeaMUI オブジェクトの容易な複数登録が可能となる．

たとえば LoginFormBean に対して，前述の従来手法と同様の GUI 出力の追加を行うには，コンポーネント LogTextArea を持った新たな BeaMUI オブジェクト LogTextAreaUI を開発し LoginFormBean へ追加する．その際に，パスワード設定処理（過程 1）と新しい LogTextArea からの入力受付け（過程 2）への対応は，LoginFormBean の共有データおよびイベント結合定義を，Integrator によりビジュアルに修正・追加することで容易に可能となる（**図 12**）．図 12 に示すように，パスワード設定処理および入力イベント処理への対応は，データ名変換表への Model 用データ名と UI 用データ名の対の追加登録，および，イベント表へのイベント名と UI 用データ名を持ったイベント処理手続きの対の追加登録によってなされる．

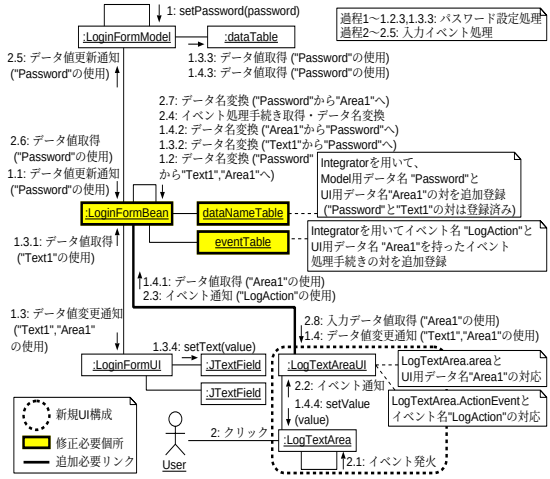

図 12 BeaMBuilder:新規 GUI 出力追加時のコラボレーション図
Fig. 12 BeaMBuilder: collaboration diagram for GUI addition.

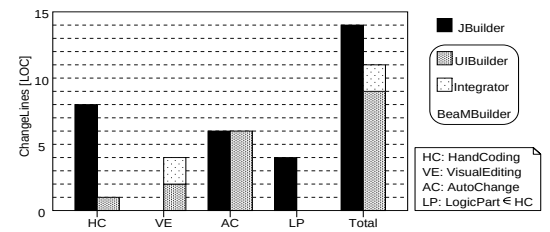

図 13 GUI 変更にともなうコード修正量（単位：ライン数）
Fig. 13 Number of modified code lines for GUI customization [lines].

このように BeaMBuilder を用いれば，BeaMModel・UI 両オブジェクトはお互いを直接認識しないため，BeaM コンポーネント内の BeaMModel オブジェクトに対し，複数の BeaMUI オブジェクトを BeaMContainer オブジェクト経由で依存させることができ，同一データに対する複数の表示方法の同時提供を容易に実現する．また，BeaMContainer オブジェクトについてデータ名およびイベントに関する登録は実行時に追加・削除可能であるため，BeaMUI オブジェクトの BeaMContainer オブジェクトに対する依存関係は実行時に変更可能であり，データの GUI 表示方法を実行時に変更可能である．

5.2 定量的評価

(1) 部分単位での詳細なカスタマイズ

定性的評価 (1) で述べた，GUI 変更についてのコード修正量を，JBuilder2.0 と BeaMBuilder をそれぞれ用いた場合で比較したグラフを**図 13** に示す．BeaMBuilder を用いることで，手作業によるコーディング作業（HandCoding）が大きく低減し，さらに，修正

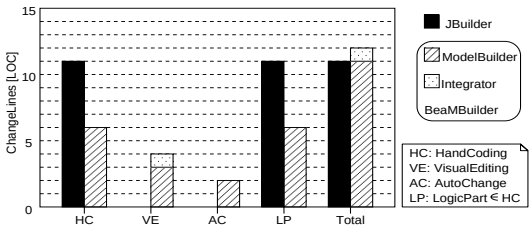

図 14 ロジック仕様変更にともなうコード修正量（単位：ライン数）
Fig. 14 Number of modification code lines for changing model specification [lines].

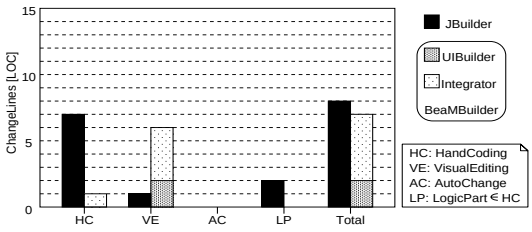

図 15 GUI 追加にともなうコード修正量（単位：ライン数）
Fig. 15 Number of modification code lines for GUI addition [lines].

に際してロジックにかかわる部分（LogicPart）に手を加える必要がないことが分かる．

一方，ユーザによる入力操作にともなうイベント通知時に，ID もしくはパスワードのどちらかが未入力である場合は入力済の方の値を仮に用いるというロジックにかかわる部分の仕様変更について，修正の必要なコード量を**図 14** に示す．コード修正量の総計こそ従来手法が少ないものの，BeaMBuilder ではビジュアルな設定を使用することから，手作業による実際のコーディング作業（HandCoding）が低減することが分かる．ロジック部の修正は，誤りのないように最も神経を使う作業であり，ModelBuilder および Integrator はその作業の一部をビジュアルな設定による自動生成が補うことで，エラーの可能性を低減する．

(2) 同一ロジックに対する複数ビューの提供

図 15 は，5.1 節 (3) で述べた GUI 追加についてのコード修正（追加）量を，JBuilder2.0 と BeaMBuilder をそれぞれ用いた場合で比較したものである．BeaMBuilder を用いれば JBuilder2.0 におけるほとんどの手作業によるコーディング作業（HandCoding）をビジュアルな編集により低減し，さらに，修正に際してロジックにかかわる部分（LogicPart）に手を加える必要がないことが分かる．図 15 における Integrator で必要な手作業のコード修正量は，GUI の追加にともなう BeaM コンポーネント全体としてのレイアウト変更に必要なものであり，将来 Integrator にレイアウト変更機能を追加することで，修正作業の完全なビジュアル化が可能となる．

表 1 繰返しテストにおけるメソッド呼出し数（単位：[回数]，[%]）
Table 1 Number of method calls [times], [%].

Loop[回]	直接呼出し回数 [回] 従来手法	BeaM	総回数中の割合 [%] 従来手法	BeaM
	SingleView			
100	3830	20660	0.064	0.260
1000	38030	205160	0.068	0.275
10000	380030	2050160	0.064	0.260
	DualView			
100	5054	28128	0.051	0.294
1000	50054	279228	0.050	0.295
10000	500054	2790228	0.048	0.280

oldstyle プロファイル使用 (-Xrunhprof:cpu=old)

表 2 繰返しテストの総実行時間（単位：[msec]）
Table 2 Execution time for LoginFormBean [msec].

Loop[回]	SingleView 従来手法	BeaM	DualView 従来手法	BeaM
100	17125	18828	18697	18787
1000	169804	176374	179588	183494
10000	1809102	1937265	2015458	2024081

Sun Java2SEv1.2.2, WindowsNT4.0 Server, 500 MHz/PentiumIII, Memory256 M

(3) 実行効率の検証

BeaM アーキテクチャにおける，その再利用性向上の代償としての実行効率の低下が，従来手法に比べてどの程度かを検証するため，LoginFormBean についての値入力と LoginEvent の発火，入力値リセット，という一連の動作の自動繰返しテストを，従来手法と BeaM アーキテクチャで開発した場合の両手法について行い，メソッドの呼び出し回数，および実行時間を計測した．LoginFormBean として，変更をいっさい加えずに単一のビューを持つもの（SingleView）と，5.1 節 (3) における LogTextArea および LogTextAreaUI をビューとして追加したもの（DualView）の両方を扱った．

表 1 は，自動繰返しテスト時に LoginFormBean について直接起動されたメソッド呼出しの数（直接呼出し回数）と，その数の JavaBeans フレームワークを含む全メソッド呼出し数における割合（総回数中の割合）を示す．**表 2** は，同じ自動繰返しテストにかかる総時間を示す．

表 1・表 2 より，全メソッド呼出し数における直接メソッド呼出し数の割合の，従来手法に対する BeaM での増加は，平均して SingleView で 0.199 ポイント，DualView で 0.24 ポイントであり，総実行時間についての増加の割合は，SingleView で 3.9%~9.9%，Du-

alViewで0.4%〜2.0%である．
直接の起動メソッド呼出し数は，BeaMコンポーネント内の通信が名前通知を基本としたpull型の通知の仕組みを採用し，実行時に頻繁な更新通知が行われるため，従来手法に対してBeaMでは約4倍増加している．総実行時間の増加の割合は，メソッド呼出し数の全体に対する割合の増加分に比較して大きく，実行効率の低下を招いているが，これは，BeaMコンポーネント内のすべての通信が用いるデータ名・イベント名の実装において，Stringオブジェクトを逐次生成して用いているため，このオーバヘッドが現れた結果である．
しかしながら，BeaMアーキテクチャは既存のGUI構築フレームワーク上に追加的に実装することを主目的とするため，実装上の実用的な評価は利用するGUIフレームワーク等の環境をすべて含んだ結果が重要である．したがって，表2において実行効率の低下は0.4%〜9.9%の範囲であり，GUIアプリケーション部品を用いたアプリケーションの使用は，主にユーザとの対話的な操作を中心とすることから実用上問題ないといえる．
なお，SingleViewと比較して，DualViewでは実行効率の低下が小さいが，これはDualViewでは従来手法においても，BeaMと同様に，同一のデータ値を複数のビューに通知する仕組みを必要とするため，BeaMと大きな差が出なかったことに起因する．

6. 関連研究

(1) PAC（Presentation-Abstraction-Control）
PAC[19]は，一般的な拡張MVCアーキテクチャとは異なり，Presentation-Control-Abstractionの対を基本エージェントとして階層的に持たせることで，仕様変更に対する柔軟性を持たせている．BeaMアーキテクチャと同様に，GUIを担当するPresentationとアプリケーションドメインのデータを担当するAbstractionは互いを知らず，内部における依存性の排除がなされている．
しかしながら，PACにおけるAbstractionとPresentation間の通信は，Observerパターンによる更新通知に限られていないため，Controlが著しく拡張性を欠く実装を開発者に許してしまう[3]．
これに対してBeaMアーキテクチャは，Model/UI用データ名の対応付けが正しく設定されている限り，BeaMUIオブジェクトのGUI表示の一致が保証されるため，ビューの追加を容易に行うことが可能であり保守性に優れる．
またPACはJaguar等の拡張MVCアーキテクチャ同様に，Controlの数と上下階層を伝わるメッセージフローが肥大化する問題をかかえる．
(2) MVP（Model-View-Presenter）
MVP[20]は，構成要素をModel，View，Presenterとし，従来のモデル/ビュー分離に対してGUI表示特有の情報を管理するPresenterを追加することで，より高度なGUI，特に複合GUI等を単一もしくは複数のModelに対して協調動作させる仕組みである．
完全なモデル/ビュー分離におけるアダプタやBeaMにおけるBeaMContainerオブジェクトとは異なり，PresenterはModelのGUI表現に限定した役割を持つため，MVPにおけるViewとModelのView→Model方向の依存性の低減は不完全である．アダプタを挟むことでこの依存性は解消できるが，アダプタ・メッセージフローの複雑化と仕様変更にともなう修正箇所の散在化を引き起こす．

7. おわりに

GUIアプリケーション部品の開発手法として，従来の内部におけるアプリケーションロジック部とGUI部間の依存形態に起因する拡張性・保守性の問題を克服する新たなアーキテクチャBeaMを提唱し，対応開発ツールBeaMBuilderとともに，従来手法および各種関連研究に対するその有用性を示した．
BeaMアーキテクチャはGUIアプリケーション部品全般について，共通に内部でのモデル/ビュー分離を目的とするものであり，その仕組みはJavaBeansに限定されるものではない．BeaMアーキテクチャは，GUIを主体としイベント駆動型の制御を可能とするコンポーネントシステムに適用可能であり，今後他のコンポーネントシステムへの移植，および有用性の提示を進めていく予定である．

参考文献

1) Hopkins, J.: Component Primer, *Comm. ACM*, Vol.43, No.10 (2000).
2) D' Souza, D. and Wills, A.: *Objects, Components, and Frameworks with UML*, Addison-Wesley (1998).
3) Buschmann, F., Meunier, R., Rohnert, H., Stal, M. and Sommerlad, P.（著），金澤典子，水野貴之，桜井麻里，関富登志，千葉寛之（訳）：ソフトウェアアーキテクチャ，トッパン (1999).
4) Larman, C.（著），今野 睦，依田智夫（監訳），依田光江（訳）：実践UML，ピアソンエデュケーション (1998).
5) Schmucker, K.: *Object-Oriented Programming*

for Macintosh, Hayden Book Company (1986).
6) Shepherd, G. and Wingo, S.（著），玉井 浩（訳）：MFC インターナル，アジソンウェスレイ (1997).
7) Fowler, A.: A Swing Architecture over View: The Inside Story on JFC Component Design, *The Swing Connection* (Aug. 1998).
8) OMG ADTF: *OMG Unified Modeling Language Guide Specification* (1999).
9) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.（著），本位田真一，吉田和樹（監訳）：デザインパターン，ソフトバンク (1995).
10) Krasner, G. and Pope, S.: A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, *Journal of Object-Oriented Programming*, Vol.1, No.3 (1988).
11) Lovejoy, A. and Mervine, F.: Jaguar: A UI Framework for Java, *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (1999).
12) Vlissides, J.: The Trouble with Observer, C++ Report (Sep. 1996).
13) Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley (1997).
14) Hamilton, G.: *JavaBeans 1.01 Specification*, Sun Microsystems (1997).
15) Tkach, D., Fang, W. and So, A.（著），矢部啓司，澤谷由里子，酒井之子（監訳）：ビジュアル・プログラミングによるオブジェクト技術（VMT），トッパン (1997).
16) 満田成紀，沢田篤史，鯵坂恒夫：操作情報を中心としたユーザインタフェース設計法，電子情報通信学会ソフトウェアサイエンス研究会報告，SS99-45 (1999).
17) Inprise Corporation: Borland JBuilder. http://www.borland.com/jbuilder/
18) Sun Microsystems: HotJavaBeans. http://java.sun.com/products/hotjava/bean/
19) Hussey, A. and Carrington, D.: Comparing two user-interface architectures: MVC and PAC, *International Workshop on Formal Aspects of the Human Computer Interface* (1996).
20) Potel, M.: *Model-View-Presenter*, Developer-Works (Jan. 2000).

(平成 12 年 3 月 31 日受付)
(平成 13 年 9 月 12 日採録)

鷲崎 弘宜

昭和 51 年生．平成 13 年早稲田大学大学院理工学研究科修士課程修了．現在，同大学院理工学研究科博士課程に在学中．コンポーネント指向ソフトウェア開発の研究に従事．電子情報通信学会，ソフトウェア科学会，JapanPLoP 各会員．

白銀 純子（正会員）

昭和 49 年生．平成 11 年早稲田大学大学院理工学研究科修士課程修了．現在，同大学院理工学研究科博士課程に在学中．GUI ソフトウェア開発支援の研究に従事．

深澤 良彰（正会員）

昭和 51 年早稲田大学理工学部電気工学科卒業．昭和 58 年同大学大学院博士課程中退．同年相模工業大学工学部情報工学科専任講師．昭和 62 年早稲田大学理工学部助教授．平成 4 年同教授．工学博士．ソフトウェア工学，コンピュータアーキテクチャ等の研究に従事．ソフトウェア科学会，IEEE，ACM 各会員．