ソフトウェア工学 128－4
（2000．7．28）

コンポーネントベース・フレームワーク技術における
コンポーネントの抽出/設計方法論

吉田和樹† 本位田真一‡
†(株) 東芝 SI 技術開発センター
‡ 国立情報学研究所

コンポーネントを使ってアプリケーションフレームワークを構築する手法を提案し、その中で特に課題として挙げられる、アプリケーションフレームワークの構成要素となるコンポーネントをアプリケーションドメインから抽出し、設計するための方法論を提示する。そして、その方法論に従ってコンポーネントを設計した例として、業務トランザクション処理ドメインについて説明し、それらのコンポーネントの適用例をもとに、手法の有効性を評価する。

A component extraction/design methodology
in a component-based framework technology

Kazuki Yoshida† Shinichi Honiden‡
† System Integration Technology Center, Toshiba Corporation
‡ National Institute of Informatics

A method for building an application framework from a set of components is proposed, and especially a methodology to extract and design such components from application domain is explained as a main subject in this paper. Then this methodology is applied to the business transaction processing domain and from these results, i.e. extracted components and application frameworks based on them, the effectiveness of this method is evaluated.

1. はじめに

近年のアプリケーションシステム開発は、高機能なアプリケーションを短納期で開発することを余儀なくされ、そのために、多くの企業でアプリケーションフレームワークの必要性に対する認識が高まってきている。しかし、アプリケーションフレームワークの開発には、ドメインに対する深い知識や洞察力が前提となり、設計/実装の繰り返しによる洗練化の作業に多大な労力が必要とされるため、そのことが、アプリケーションフレームワークの開発を阻む要因になっていた。

本論文では、アプリケーションフレームワークを予め用意されたコンポーネントの組み合わせで実装できるようにする手法を提案する。コンポーネントとは、機能のカスタマイズや実行のためのインタフェースだけが公開されたブラックボックス部品である[1]。通常、ビジュアルな開発環境とともに提供され、利用者は、この環境上で GUI を通して、コンポーネントどうしの組み合わせや各コンポーネントのカスタマイズ、さらに、その結果をすぐに実行することが可能になっている。その際、このような利用者の操作に応じて、公開されているコンポーネントのインタフェースを実際に呼び出すのは開発環境が行うので、利用者のプログラミング作業は軽減されることになる。

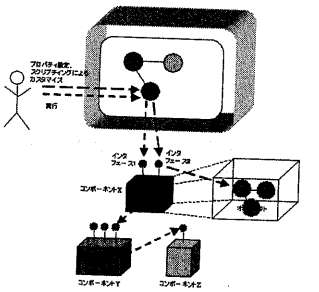

図 1. コンポーネントの概念図

[1] ここで言うコンポーネントとは、ActiveX や JavaBeans のような特定の基盤に限定したものではない。

したがって、洗練化の作業に要する労力を生産性の向上によりカバーして、アプリケーションフレームワークの開発にとって障害になっている問題点を克服することが、ここで提案する手法のねらいである。

その際、考えなければならない技術的な課題としては、次のような点が存在する。

(P0) アプリケーションフレームワークの構成方法
　コンポーネントとの関係の明確化
(P1) コンポーネントの抽出方法
　分析における着眼点
　コンポーネント化すべきか否かの判断基準
(P2) コンポーネントの設計方法
　適切なパターンの選択
　インタフェースの設計や共通クラスの構築
　処理における状態やコンテキスト、例外の扱い
(P3) 方法論の整備
　(P1)と(P2)への解になる、コンポーネントの抽出・設計のための作業プロセスの明確化

本論文では、これらの課題に対する解を提示して、その上で、この手法が前述の問題点を克服するのに有効であることを実例を通して評価するところまでを行う。

2. アプリケーションフレームワーク開発手法

2.1 アプリケーションフレームワークの構成

2.1.1 コンポーネント

Pree は、著書[1]において、アプリケーションフレームワーク内に存在する"hot spot"と"frozen spot"という概念を明らかにしたが、プログラム内で hot spot になり得る部分としては、次のようなものがある。

- 処理のシーケンス (動的側面)
- 処理の内容 (機能的側面)
- 処理の対象となるデータの構成/型 (構造的側面)

コンポーネントの組み合わせでアプリケーションフレームワークを構成する場合、コンポーネントの側でこれらの hot spot を問題なく扱えるようにする仕組みを考える必要がある。

そのために、まずコンポーネントに対して次のような前提を置くことにする。

[前提]
コンポーネントは、ルートを 1 つ持つ DAG 状の構成で組み合わされ、これらの間でメッセージはアークの向きに従って連鎖的に流れるものとする。

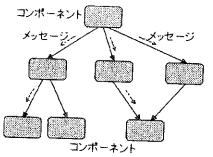

図 2. コンポーネントの構成

これにより、hot spot の1つである処理のシーケンスについて、コンポーネントの組み合わせ方を変えることで、そのバリエーションを実現できるようにする。

また、コンポーネントが提供する処理の中で、hot spot に相当する内容が存在する場合には、その部分を委譲の考え方に基づいて、別クラスに分けて実装する。このクラスを以後カスタマイズクラスと呼ぶことにする。

図 3. hot spot 部分の分離

カスタマイズクラスは、hot spot である処理内容を実行するためのインタフェースを定義した抽象クラスとして提供して、利用者が、その抽象クラスを継承するクラスを作成して、その中でこのインタフェースに必要な処理を実装して、これをコンポーネントにプラグインして使うことにする。

また、処理の対象となるデータの構成にアプリケーションプログラム間でバリエーションが存在する場合には、後節でも述べるように、データを生成する処理をコンポーネント化して抽出し、その中でこのバリエーションをカバーできるようにする。

2.1.2 アプリケーションフレームワークの形式的定義

本節では、前節の内容をもとに、アプリケーションフレームワークを形式的に定義する。

各コンポーネントは下位につなげられるコンポーネントの種類に関して何らかの制約を持っていることが考えられるが、これは BNF の形式で表現することができる。例えば、コンポーネント $Comp_1$が、下位にコンポーネントを複数つなげることができ(その種類は $Comp_2$、$Comp_3$ のどちらでもよい)、また、必ずカスタマイズクラスがプラグインされて使われる場合には(カスタマイズクラスのもとになる抽象クラスを $Cust_{1-1}$ で表わす)、次のような規則で表現することができる。

$tempComp_1 \rightarrow emptyComp_1 \times (tempComp_{2\vee3}*)$
$emptyComp_1 \rightarrow Comp_1(Cust_{1-1})$
$tempComp_{2\vee3}* \rightarrow tempComp_{2\vee3} + tempComp_{2\vee3}*$
$tempComp_{2\vee3} \rightarrow tempComp_2 \mid tempComp_3$

このような生成規則を持つ文法 G_{comp} から生成される言語の

集合を $L(G_{comp})$とすると、これはコンポーネントの組み合わせで実現される処理のシーケンスの集合を表していることにもなる。さらに、これと同様に、処理対象となるデータについても、その構成のための式をBNFで表現し、その文法 G_{input} から生成される言語の集合 $L(G_{input})$で、構成のバリエーションを表すことができる。

また、$Impl(Cust_{i-j})$により、抽象クラスとした与えられるカスタマイズクラス Custi-j を継承する実装クラスの集合を表すことにする。

これにより、アプリケーションフレームワーク(APF)は次のように形式的に定義することができる。

APF = <FS, HS>

すなわち、frozen spot(FS)と hot spot(HS)の 2 つ組からなり、それぞれは次のように定義される。

$FS = \{ Comp_1, \ldots, Comp_N \}$

$HS = L(G_{comp}) \cup L(G_{input}) \cup \{ Impl(Cust_{i-j}) \mid 1 \le i \le N,\ 0 \le j \le M_i \}$

2.2 開発環境

ここまでで述べたコンポーネントの組み合わせやカスタマイズクラスのプラグインは、コンポーネントと共に提供されるビジュアルな開発環境を使って容易に実現できるようになっている[2]。

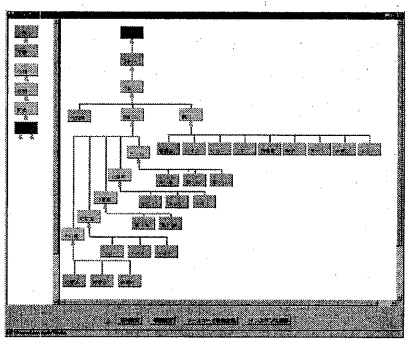

図 4. 開発環境の基本画面

このようにしてビジュアルに作成されたコンポーネント/カスタマイズクラスの組み合わせとしてのアプリケーションプログラムは、開発環境上で直接実行することができる。また、開発環境は、組み合わせの構造を再現するためのソースコードを自動生成できるようにもなっているので、それをコンパイルして実行することもできる[3]。

2.3 コンポーネントの抽出・設計方法論

ここで提案する方法論は、次のような4つのフェーズと1つのチェックポイントから構成されている。

フェーズ1

コンポーネント候補の抽出

チェックポイント

コンポーネント化すべきアプリケーションドメインか否かの判断

フェーズ2

コンポーネントの実行機能のインタフェース設計

フェーズ3

実行機能を考える上で不可欠な、コンポーネントの状態や処理のコンテキスト、例外の扱いの設計

フェーズ4

コンポーネントの共通機能のインタフェース設計

本方法論に基づく作業は、オブジェクト指向分析/設計の一般的な作業プロセスに対して付加的に行われるもので、コンポーネントに特化して、その抽出/設計方法を提供するものになっている。

方法論の実施にあたっては、アプリケーションプログラムの要求定義書やユースケース図が入力となり、作業の結果はクラス図、メッセージシーケンス図、クラスのインタフェース定義書に反映されて出力されることになる。

フェーズ1 → チェックポイント → フェーズ2を順番に実行した後は、フェーズ3とフェーズ4は、互いに独立の関係にあるものなので、フェーズ2の結果を基に並行に進めることも可能である。

以下、各フェーズ/チェックポイントについて作業内容を説明する。

2.3.1 フェーズ1

アプリケーションドメインを分析して、クラス図を作成する際に、コンポーネント候補の抽出のために、アプリケーションプログラムの機能を、入力 → 処理 → 出力の観点で捉え、次のステップを実行する。

ステップ 1)

① 入力データ、出力データ、あるいは、アプリケーションドメイン内で抽出されるエンティティの構造を、複数種類

[2] この環境では、コンポーネントがアイコンとして表現されてパレット上に並べられており、開発者は、それらをマウスによるドラッグ&ドロップ操作でキャンバス上に配置し、そしてマウスを使ってキャンバス上のアイコン間に関係線を引くことで、コンポーネントの組み合わせを表現できるようになっている。また、アイコンをクリックするとメニューが開き、そこで各種の GUI が選択できるようになっており、そのGUI を通して、カスタマイズクラスをコンポーネントにプラグインできるようにもなっている。ただし、カスタマイズクラス自体の作成は、外部のプログラミング環境を使って行う。

[3] コンポーネント/カスタマイズクラスの実装、さらに、開発環境によるソースコードの自動生成等は、現状ではすべて Java を対象にして行っている。

の基本単位から成る階層に分割できないかどうかを確認する。

② もし分割可能ならば、基本単位をクラスとして、クラス図を作成する。そして、処理の部分も分割して各基本単位のクラスに割当てて、この階層に沿って上位から下位へメソッド呼び出しを連鎖させることで機能を実現できないかどうかを、メッセージシーケンス図を記述しながら確認する。

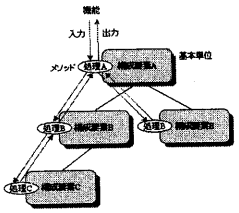

図 5. 構造を基本単位に分割

③ もし実現可能ならば、各クラスに割当てられた処理を、凝結度について一般的に適切なレベルでまとめるように、必要ならばさらに分割する。

ステップ 2)

① 処理の部分を、アプリケーションドメイン内で抽出されるエンティティからは独立した、複数種類の基本単位の組み合わせに分割できないかどうかを確認する。分割に際しては、構造化分析/設計の適用が有効である。

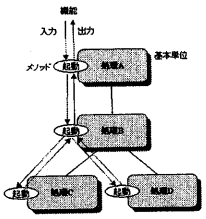

図 6. 処理を基本単位に分割

② もし分割可能ならば、基本単位をクラスとして、クラス図を作成する。その際、クラス間の結合度や処理の凝結度が一般的に適切なレベルになるように、必要ならば処理を再分割する。また、類似の処理については共通化を図る。

2.3.2 チェックポイント

コンポーネント化すべきアプリケーションドメインか否かを判断するためのチェックポイントとして、次の3点を挙げる。

(1) アプリケーションプログラムは、ルートを 1 つ持つ DAG 状の基本単位の組み合わせになる。

(2) 基本単位間で、アークの向きに従ってメッセージが連鎖的に流れる。

(3) 基本単位のクラス図内に、オブジェクト構造におけるバリエーションを生む、可変多重度、再帰、サブクラス化[4]の要素が入っている[5]。

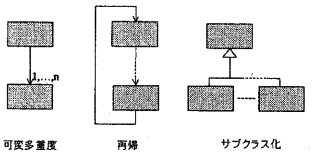

図 7. オブジェクト構造のバリエーションを生む要素

もしこれらの条件の中でどれか1つでも満たされなければ、そのアプリケーションドメインはコンポーネント化すべきでないと判断し、この方法論を終了する。もしすべてが満たされるならば、フェーズ1で基本単位として識別したクラスをコンポーネントにすべく、以後のフェーズに進む(これらのクラスを説明ではコンポーネントと呼ぶことにする)。

2.3.3 フェーズ 2

本フェーズを含む以後のフェーズでは、コンポーネントを対象にして、その設計について説明する。

ステップ 1)

もし、コンポーネントに持たせる属性について、アプリケーションプログラム間で差異が存在するならば、それを hot spot として特定し、Dynamic Properties パターン[5]を適用することにより吸収できるようにする。これにより、属性や関係の種類の追加が動的に行えるようになる。

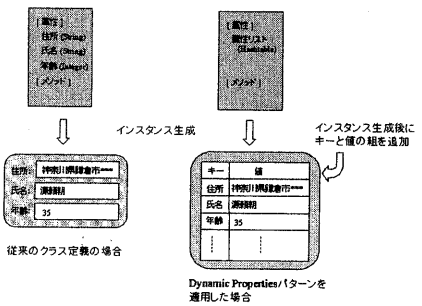

図 8. Dynamic Properties パターンの適用イメージ

ステップ 2)

フェーズ1のステップ 1)で述べた、処理を階層に沿って上位から下位へメソッド呼び出しを連鎖させることで実現す

[4] コンポーネント数の増大を防ぐために、1つのコンポーネントを基にしてそこにプラグインされるカスタマイズクラスで差異を吸収できるようにして、サブクラス化は極力避ける。

[5] Composite パターン[3]はその典型的な例である。

る場合に、この連鎖のためのメソッドと、各コンポーネントに割当てられる処理のためのメソッドを分離する。

そして、この連鎖のためのメソッドのインタフェースを各コンポーネントに定義して、これをクラス図とインタフェース定義書に記述する。その際、メソッド内でコンポーネントに割当てられた処理を呼び出すタイミングとして、次の区別を明確にすることで、インタフェースにも差異が現れるので留意する。

(1) 自身が呼び出された時に処理を加える[6]
(2) 自身が呼び出した先から返された値に対して処理を加える[7]
(3) (1)と(2)の両方

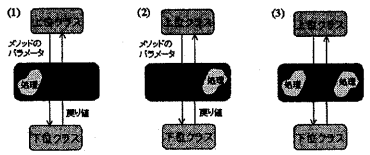

図 9. メソッド内で処理を加えるタイミング

また、コンポーネントに割当てられる処理のためのメソッドについても、インタフェースを定義して、クラス図、インタフェース定義書に記述する。もし、この処理にアプリケーションプログラム間で差異が存在するならば、それを hot spot として特定し、この部分はカスタマイズクラスに委譲することにより吸収できるようにする。そして、その際のカスタマイズクラス側のインタフェースも定義して、これらをクラス図、インタフェース定義書に記述する。

ステップ 3)

フェーズ 1 のステップ 2)で述べた、処理の部分を複数種類の基本単位の組み合わせに分割して実現する場合に、各コンポーネントで処理を実装するメソッドのインタフェースを定義して、これをクラス図とインタフェース定義書に記述する。もし、この処理にアプリケーションプログラム間で差異が存在するならば、それを hot spot として特定し、この部分はカスタマイズクラスに委譲することにより吸収できるようにする。そして、その際のカスタマイズクラス側のインタフェースも定義して、これらをクラス図、インタフェース定義書に記述する。

ステップ 4)

処理に与える入力データを生成する部分について、アプリケーションプログラム間で入力データの構成に差異がある場合には、Type Object パターン[4]を適用して、TypeClass に入力データ(TypeObject インスタンス)の生成機能を持たせることで、吸収できないかどうかを検討する。もし、それが可能ならば、TypeClass をコンポーネントの1つに加えることにして、また、入力データ生成のためのインタフェースも定義して、それをクラス図とインタフェース定義書に記述する。

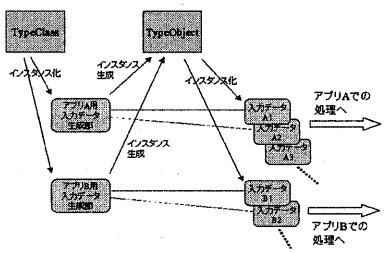

図 10. Type Object パターンの適用イメージ

ステップ 5)

アプリケーションの他の機能についても、フェーズ1のステップ 1)で述べた同じ階層上で、処理の連鎖により実現することができないかどうか、あるいは、フェーズ1のステップ 2)で述べた処理の基本単位の集合に、別の基本単位を加えたり、カスタマイズクラスの処理の内容や入力データの内容を変えることで、実現することができないかどうかを、それぞれ確認する。

ステップ 6)

ステップ 5)において実現できない、あるいは、実現できても複雑になるような場合には、インタフェースを明確に区分するために、Extension Object パターン[4]を適用して、Extension として処理を定義し、それをコンポーネントに持たせることにする。ただし、1つのコンポーネント内で閉じた機能については、そのコンポーネント自体に処理を定義する。

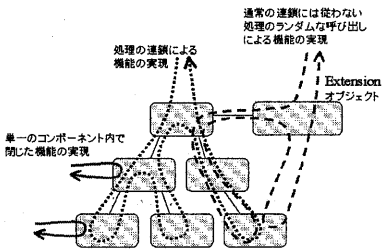

図 11. Extension Object パターンの適用イメージ

そして、インタフェースについてはクラス図とインタフェース定義書に反映させ、また、処理についてはメッセージシーケンス図を作成して明確化する。

2.3.4 フェーズ 3

[6] 構造化分析/設計における遠心型モジュールに相当する。
[7] 構造化分析/設計における求心型モジュールに相当する。

ユースケース図やメッセージシーケンス図をもとにして、コンポーネントの状態や処理のコンテキストを明確化する。

ステップ 1)

状態については、可能ならば State パターン[3]を適用して、状態による振舞いの違いを別途定義した状態クラスに委譲する形で扱うようにする[8]。その際、状態遷移の同期化によるボトルネック、デッドロックを考慮に入れておく。

ステップ 2)

コンテキストについても、それ自体を別クラスとして抽出するが、それをコンポーネント間で受け渡す方法については、次のような設計上の選択肢が存在する。

① コンテキストを、メソッドの引数で受け渡すようにする。

特に Java の場合には、

② スレッド[9]のサブクラスを作成して、それを使って処理を実行する形にして、そこにコンテキストを持たせるようにする。

③ 処理をマルチスレッドで実行し、コーディネータ的な存在が必要なときには、スレッドグループ[10]をサブクラス化して、これにコーディネータの役割を担わせ、コンテキストもそこに持たせるようにする。

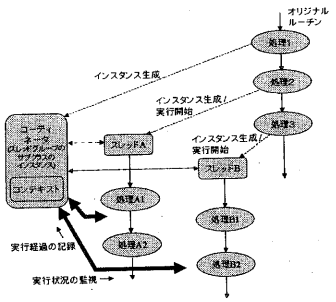

図 12. マルチスレッド実行時のコーディネータ

ステップ 3)

例外の扱いについては、例外の種類を次のように分類して、呼び出し側での対処方法の区別を明確にできるようにする。

(1) カスタマイズクラス内部で発生する例外

(2) コンポーネント内部で発生する例外

(2-1) コンポーネント自体の処理により例外が引き起こされる場合

(2-2) カスタマイズクラスでの処理の結果返された値により例外が引き起こされる場合

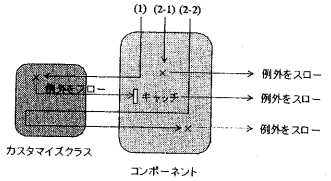

図 13. 例外の種類の分類

(1)、(2-1)、(2-2)のそれぞれについて個々に例外クラスを作成する。そして、(1)については、カスタマイズクラスの作成者により定義された、あるいは、処理系からスローされる任意の例外クラスを呼び出し元のコンポーネント側でキャッチして、それをここで作成した例外クラスでラップして上位に返すようにする。また、(2-1)、(2-2)については、ここで作成した例外クラスを継承させる形で、発生原因に応じて例外クラスを作成しておき、それを上位に返すようにする。特に、処理をマルチスレッドで実行している場合には、コーディネータとなるクラスを介して、上位に例外を返すようにする。

以上の各ステップでの作業結果を、クラス図、メッセージシーケンス図、インタフェース定義書に記述する。

2.3.5 フェーズ 4

コンポーネントの共通機能のインタフェースを設計する。

ステップ 1)

コンポーネントの関係や属性に識別子を定義する。

ステップ 2)

関係や属性の設定、取得、等のための統一的なインタフェースを定義する。

ステップ 3)

インタフェース追加を行えるようにするために、Extension Object パターンを適用する。

ステップ4)

コンポーネントを組み合わせたオブジェクト構造上での機能追加を行えるようにするために、Visitor パターン[3]を適用する。

以上の各ステップでの作業結果を、クラス図、インタフェース定義書に記述する。

関係や属性の設定、取得、等のためのインタフェースは、関係や属性の種類ごとに set***、get***といったメソッドを追加していくことによるインタフェースの増大を防ぐために[11]、

[8] 排他も状態の一種と考えられるが、そのときにはロックのためのパターン[6]を適用する。

[9] java.lang.Thread クラス

[10] java.lang.ThreadGroup クラス

[11] ただし、このような設計は、コンポーネントの汎用性を高めるが、性能的なオーバヘッドを伴うことにもなるので、実装にあたっては慎重な判断が必要である。

関係や属性の種類を識別子を使って区別しながら、その操作をすべて統一的に扱えるように定義する。これにより典型的な関係や属性について、このインタフェースを実装したコンポーネントの共通クラスを予め用意しておき、それを再利用することも可能になる。

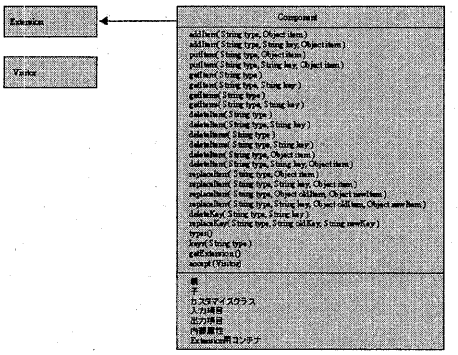

図 14. 共通機能のインタフェースを実装した共通クラス

3. 事例

本章では、2 章で示した方法論に従って、業務トランザクション処理のアプリケーションドメインから、コンポーネントを抽出/設計した事例について説明する。まずドメインの概要を述べ、次に方法論の各フェーズから主要なステップを取り上げて、作業の詳細を説明する。

3.1 ドメイン概要

業務トランザクション処理向けのコンポーネントは、初期段階では、会計業務における取引処理を対象にして抽出が行われた。

会計業務では、一般に、複式簿記に見られるように、取引を貸方と借方に分けて、それぞれをある勘定の貸方と他のある勘定の借方に記入するという仕訳処理を行う。その際、取引は貸方の金額の合計と借方の金額の合計が必ず一致するように分割されなければならないという原則がある。このことは、システムの観点から見ると、貸方と借方のデータの登録をデータベースに対する1つのトランザクション処理として扱わなければならないことを意味している。また、会計業務では、取引のデータとして金額以外のデータが渡されて(例えば、時間データなど)、それを仕訳の過程で、金額データに変換するような処理も、しばしば必要になる。

3.2 方法論に基づく作業の説明

(1) フェーズ1

ステップ 2)[12]

仕訳処理は、入力として渡される取引データを貸方と借方に分割する処理と、それぞれを分類して該当する勘定を見つけ出す処理と、実際に勘定への記入を行う処理と、金額以外のデータの変換を行う処理に分けて考えることができる。これらは、会計業務における処理の基本単位として考えることができるので、以後、それぞれを「分割」、「分類」、「記入」、「変換」と呼ぶことにする。

ただし、「記入」については、エンティティとして識別される「勘定」との関連が極めて強く、この処理だけで基本単位として独立に抽出する必要性がないため、「勘定」内のメソッドとして実装することにする。

これらの基本単位の組み合わせの順序は、概ね「変換」→「分割」→「分類」(→「勘定」)という固定的なものになる。

ところで、基本単位の中には、上述の処理とともにトランザクション処理を扱う仕組みを入れる必要があるので、これらの基本単位を、本来の会計業務だけでなく、データベースに対するトランザクション処理を行う他の業務にも適用可能になるように抽出しておくことを考える。その場合に、「分割」、「分類」、「変換」の各基本単位は、これまでのような取引やそこから派生した貸方、借方データを扱うものという役割付けから、データベースに登録する任意のデータを扱うものという役割付けに変わることになる。それに加えて、適用性をさらに高めるために、上述のような固定的な順番から、任意の順番で同一種類の基本単位を何度でも使用した組み合わせが可能になるようにしておく。

また、これまでは会計業務を対象にしていたため、「勘定」エンティティに「記入」の処理を任せていたが、勘定に限らず、データベースをターゲットとして明確に捉えた場合、「記入」に相当する処理には、データベースに対するコネクションの確保や SQL 文の作成と実行などの定型的な処理が必要となるため、この部分も、「保管」という名前であらためて基本単位として抽出しておく。

抽出された基本単位をクラスとして、これらを任意の順番で組み合わせ可能にするように、Composite パターンを適用する[13]。

---

[12] ステップ1)を通じて、「取引」、「貸方」、「借方」、「エントリ」といった形で基本単位を抽出することも可能である。しかし、この場合、チェックポイントにおいて、オブジェクト構造のバリエーションを生む要素として、「貸方」、「借方」と「エントリ」の間に可変多重度が入ることが確認されるだけなので、バリエーションの度合いは少ない。その反面、ステップ 2)で基本単位を抽出した方が、後に述べるように、コンポーネントの汎用性も考え易くなり、バリエーションを多く持たせられるようにもなる。したがって、ここではステップ 2)による抽出を採用し、説明上、ステップ 2)だけを取り上げることにした。

[13] Composite の子クラスの中で、「分割」と「分類」は、その処理の性格上、複数の子ノードを持つものになるが、「変換」だけは1つの子ノードしか持てないようにしている。

Composite パターンを適用したことにより、基本単位として、「分割」、「分類」、「変換」以外のものが新たに抽出された場合にも、容易にこのフレームワークに追加することが可能になる。

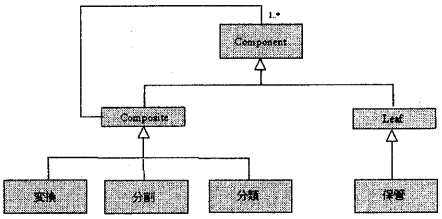

図 15. 業務トランザクション処理のクラス図 (1)

(2) チェックポイント

アプリケーションプログラムは、Composite パターンを適用した図 15 でも明らかにしてあるように、ルートを 1 つ持つツリー状のオブジェクト構造になる。この中を、ルートからリーフに向かってメッセージが連鎖的に流れることで処理が行われる。また、「変換」、「分割」、「分類」は任意の順番での組み合わせが可能で、構成されるオブジェクト構造にはバリエーションが考えられる。以上のことから、ここで抽出した「変換」、「分割」、「分類」、「保管」をそれぞれコンポーネントとする。

(3) フェーズ 2

ステップ 3)

コンポーネント間での処理の呼び出しのためのインタフェースを次のように統一的に定める。

void post(Transaction anInput)

ここで、Transaction オブジェクト anInput は、内部で Hashtable を使って、("名前", "聖徳太子")といったような key と value の組みで、データベースに登録するデータを保持する形になっている。データは、文字列や数値のような基本型の値の他に、ユーザ定義のオブジェクトであってもよい。

このインタフェースについて、各コンポーネント内で hot spot を特定する。簡単な特定の例として、次のような案が考えられる。

「分類」 Transaction オブジェクト内部のデータに基づき、複数の子ノードの中から次の処理を行うものを選択する部分

「分割」 Transaction オブジェクト内部のデータをどのように分割して、複数の子ノードに振り分けるのかを決める部分

「変換」 Transaction オブジェクト内部のデータをどのように変換するのかを決める部分

これにより、アプリケーションごとに異なる key と value の組みに特化して、分類、分割、変換の規則をカスタマイズクラスとして実装できるようになる。そして、そこでのインタフェースを次のように定義する。

分類規則 : String post(Transaction anInput)

分割規則 : Hashtable post(Transaction anInput)

変換規則 : Transaction post(Transaction anInput)

フックメソッドの引数は、いずれも、「分類」、「分割」、「変換」の post メソッド呼び出し時に引数として渡された Transaction オブジェクトである。それを基にして、分類規則では、選択した子ノードのIDを、また、分割規則では、子ノードのIDと分割後のTransactionオブジェクトの組みを、また、変換規則では、変換後の Transaction オブジェクトを、それぞれ上記の型で返すようにしている。

ステップ 4)

コンポーネント間を受け渡しされる Transaction オブジェクトを最初に生成する部分に、Type Object パターンを適用して「トランザクションタイプ」というクラスを抽出し、アプリケーション間でのデータの構成の差異をそこで吸収できるようにする。また、トランザクション処理を管理するプリミティブな機能も、この「トランザクションタイプ」の内部に持たせることにして、これをコンポーネントの一つに加えることにする。

ステップ 5)

本ベースコンポーネントの抽出プロセスは、会計における取引を勘定へ記入するという業務から出発したこともあり、データベースに対するトランザクション処理も、登録機能を中心に設計/実装を行ってきた。しかし、トランザクション処理を扱う業務処理全般に適用可能なフレームワークに仕上げるために、登録機能以外にも、主キーの値に基づく個々のレコード単位での検索、更新、削除機能を、post と同じ処理シーケンスを持つ別メソッドとして各コンポーネントに実装する。

以上の結果をまとめたものを、図 16 に載せる。

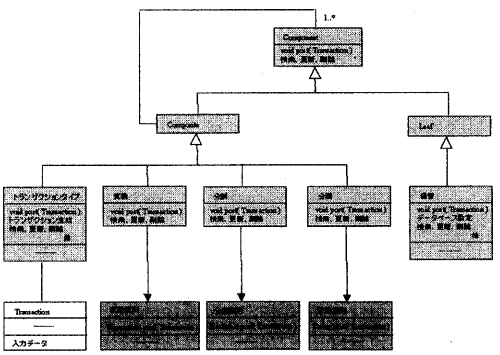

図 16. 業務トランザクション処理のクラス図 (2)

(4) フェーズ 3

ステップ 1)

「分割」、「分類」、「変換」は、入力された Transaction オブジェクトをカスタマイズクラスに渡して、そこでの処理の結

果を下位のコンポーネントに出力するだけなので、そこには状態の概念は存在しない。また、「トランザクションタイプ」についても同様で、やはり状態の概念は存在しない。

一方、「保管」については、そこでデータベースへのコネクションを管理させる場合には、あるトランザクション処理を実行している最中は、ACID 性を確保するために、別のトランザクション処理は待ち状態にする必要がある。この排他の仕組みはロックのためのパターンを使って実現することができる。ただし、個々のトランザクション処理が、別途確保されたコネクションプールからコネクションを取得して、それをTransaction オブジェクトを介して「保管」に提供するなどして、「保管」にコネクションを管理させないようにすれば、「保管」から状態を無くすことも可能である。

ステップ 2)

枝別れした複数の処理全体で、トランザクションをコミット/ロールバックすべきかどうかの判断は、シングルスレッドで実行する場合には、すべての枝での処理を逐次で終えた後に、コネクション内部で SQLException 等の例外が発生していないかどうかを確認して行うことができるが、枝ごとに別スレッドで処理を実行する場合には、各枝のスレッドを、同一トランザクションの処理を行っていることを識別できるようにするために、同一のスレッドグループに所属するものとして生成し、このスレッドグループに全体の処理の実行状況や実行経過を把握させて、最終的なコミット/ロールバックの判断と処理もそこに行わせるようにする。

(5) フェーズ 4

ステップ 1)

各コンポーネントの関係と属性を、予め識別子が決められている典型的な関係や属性のリストをもとにして整理すると次のようになる。

	トランザクションタイプ	変換	分割	分類	保管
親(PARENT)	——	(単一)	(単一)	(単一)	(単一)
子(CHILD)	(複数)	(単一)	(複数)	(複数)	——
カスタマイズクラス(CC)	——	変換規則	分割規則	分類規則	——
入力項目(IN)	——	——	——	——	——
出力項目(OUT)	入力データ項目(Transaction)	——	——	——	PreparedStatement用カラム名
内部属性(ATTR)	データベース関係情報 コネクションプール	——	——	——	テーブル名

表1. 関係と属性の分類（登録機能）

ここで、トランザクションタイプにおける"データベース関係情報"と"コネクションプール"のように、1つの識別子に対して複数の属性が存在する場合には、新たに識別子を定義することにして、それ以外では予め決められている識別子を使う。

ステップ 2)

図 16 の Component クラスの部分に図 14 で挙げた共通クラスを導入して、表1で整理した内容に基づいて、必要に応じて各コンポーネントで継承したメソッドの処理の上書きを行う。

4. 評価

本章では、3 章で示した適用事例の結果から、本論文で提唱するアプリケーションフレームワークの開発手法について、その有効性と課題を考察する。また、方法論を通して抽出されたコンポーネントが、アプリケーションフレームワークを作る上で有用であることを、これまでの適用実績をもとに示す。

4.1 開発手法の有効性と課題

この手法では、アプリケーションフレームワークをコンポーネントの組み合わせで実装できるようにするために、方法論の中で、そのようなコンポーネントの抽出・設計のための観点と手順を明らかにしている。

抽出の観点とは、部品の汎用性を高めるために、機能の入力→処理→出力を明確にして、処理の部分を適切な単位で分割するというものである。そして、分割された処理が、入力データ、または、出力データの構造内に現れるエンティティに関連付けられるものなのか、あるいは、独立のものなのかによって、以後の手順も一部は分けて考えるようにしている。特に事務処理の世界では、一般に、入力されたデータに何らかの操作を加えて出力するという処理が多く、4.2 節でまとめるこれまでのアプリケーションフレームワークの構築実績を見ても、この観点は有効であると考えられる。

一方、設計の観点としては、実行機能と共通機能の設計を区別して、特に共通機能においては、インタフェースを統一することで、予めそれらを実装したクラスを用意しておき、そこから派生させたクラスに実行機能を追加してコンポーネントを作成するという方法を可能にしている。また、実行機能については、コンポーネントどうしの組み合わせ、あるいは、コンポーネントとカスタマイズクラスの組み合わせで、オブジェクトコンポジションにより機能の実装を行っていく点が特徴である。

コンポーネントとカスタマイズクラスをそれぞれ別のクラスに分けたことにより、例えば、コンポーネントの組み合わせまでを行ったものをアプリケーションフレームワークとして提供し、それをアプリケーションプログラムに仕上げる際に、固有の処理をカスタマイズクラスに実装してプラグインするといった具合に、工程上のフェーズごとに対象とする部品を

分けて扱うこともできるようになる。このことは、試験工程において、試験範囲の明確な区分やリグレッション試験などの規模の縮小を図ることを可能にする。

ただし、カスタマイズクラスの基になる hot spot の特定については、本方法論内では特に具体的な手段を示していない。しかし、これについては、他の研究成果[7]との融合により、補強していくことは可能である。

ところで、実行機能の設計において、コンポーネントの状態や処理のコンテキストの概念は、独立した処理の集合体としてアプリケーションを作っていくというコンポーネントベースの一般的な考え方には合わないものなので、方法論のフェーズ2で特にそれらを取り上げることにした。しかし、そこでも十分に手順的なブレークダウンができているとは言えず、今後も引き続き調査と改良を行っていく必要がある。

4.2 コンポーネントの有用性

前章で説明した業務トランザクション処理のためのコンポーネント群と、その他にこの方法論に従って開発した帳票編集処理のためのコンポーネント群(付録参照)を、これまでに流通業向けの受注管理システム、および、ある社会インフラ系の企業における会計伝票発行システムの2つの実製品開発にそれぞれ適用して、業務トランザクション処理で 10 個、帳票編集処理で 13 個、合計 23 個のフレームワークを作成した。

	受注管理システム	会計伝票発行システム
業務トランザクション処理	ステータス管理データ管理 OCRスプールデータ管理 受注データ管理 前回受注データ管理 実績顧客データ管理 実績修正データ管理 ストップ管理データ管理	会計伝票データ管理 事例伝票データ管理 修正原票データ管理
帳票編集処理	事前リスト ダイレクトメール 配送伝票 納パック 短冊のし	会計仕訳表(6種類) 出荷確認票 修正原票

表 2. 適用例の一覧

例えば、流通業向けの受注管理システムに業務トランザクション処理のコンポーネント群を適用した際には[10]、アプリケーションの機能量のうち、ファンクションポイント(IFPUG 法)[11]にして 374.05 の部分をカバーすることができた。また、これらの機能を実装しているコード部分において、コンポーネントを使ったアプリケーションフレームワークをもとに作成したモジュールと、一から作成したモジュールとの間で、コード行数/作業時間(h)による生産性指標を比較すると[14]、前者が 27.79、後者が 12.58 となり[15]、両者の間にはもともと処理の複雑度に多少の差異があるとは言え、約 2.2 倍の生産性向上を示す値を得ている。また、このシステム開発において、コンポーネントを使ったアプリケーションフレームワークの作成はラウンドトリップ型で進められ、大きく 2 回に分けてその作成が行われているが、1回目と2回目でコード行数/作業時間(h)を比較すると、前者が 66.38、後者が 158.97 となり[16]、利用者の習熟により生産性が約 2.4 倍向上している結果が出ている。

	LOC数	開発時間(単位:h)	生産性指標(LOC数/開発時間)	
基本部品群不使用	12,382	984	12.58	2.2倍
基本部品群使用	12,035	433	27.79	
ラウンド1	3,631	55	66.38	2.4倍
ラウンド2	9,697	61	158.97	

表 3. 生産性指標の比較

以上より、本手法の考え方に基づくコンポーネント群が、アプリケーションフレームワークの作成に幅広く適用でき、そのアプリケーションフレームワークを使うことで開発の生産性が向上すること、また、アプリケーションフレームワークの洗練化作業においても、高い生産性を実現できることを示すことができた。

5. 関連研究

hot spot/frozen spot の概念を明らかにした Pree は、hot spot を満たすようにアプリケーションフレームワークを設計するにはどうすればよいのかを示すものとして、"メタパターン"を提起した[1]。メタパターンは、デザインパターン[3]に挙げられているようなオブジェクト指向設計のための典型的なパターンを、メタレベルで分類し記述することを可能にする包括的な枠組みである。

[7]では、この hot spot/frozen spot をドメインの中で特定するための手法を、データ中心アプローチ[12]とユースケース[13]の 2 つの観点を統合する形で提唱している。ここで特定された hot spot/frozen spot は、フレームワークパターンと呼ばれる形で表現され、以後の開発に容易に適用できる

[14] コンポーネントを使った場合のコード行数とは、カスタマイズクラスのソースコードの行数と、開発環境により自動生成されるソースコードの行数の合計である。

[15] プログラミング言語はともに Java

[16] 作業時間としては、コンポーネントの組み合わせとカスタマイズクラスの実装に費やした時間のみを対象として、部品の使い方の習得やテストに費やした時間は除外している。

ようにしている。

[8]では、フレームワークの設計と実装のための方法論が具体的な例を使って説明されている。この中では、特に、パターンの適用ということが重要な位置を占めているが、その他にも、フレームワークの拡張方法のバリエーションが簡潔にまとめられている点が有用である。

コンポーネントを使ったソフトウェアの開発では、例えば、コンポーネントの検索や評価などの作業が新たに加わるなど、従来のプロセスモデルをコンポーネントの導入に併せて新たに作り変える必要がある。[9]では、このモデルをプロセスパターンと呼ぶ一群のパターンを使って表現している。

コンポーネントの開発は産業界において活発に進められているが、その中でも、東芝が提供しているソリューションサービス C Solution[14]のアプリケーションフレームワークは、本論文の手法に基づいて開発が進められているものである。他にも、フレームワークに基づくコンポーネント体系の構築を行っているものとして、富士通の ComponentAA[15]などがある。

6. まとめ

本論文では、コンポーネントの組み合わせでアプリケーションフレームワークを開発する手法を提案した。そして、このようなコンポーネントを抽出・設計するための方法論を明らかにして、その有効性の評価を実製品開発への適用を通して行った。今後も適用と調査を繰り返し行いながら、手法と方法論の改良を進めていく。

付録

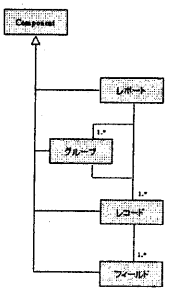

図 17. 帳票編集処理のクラス図

コンポーネント	機能
レポート	ページ制御(改ページ、ページ番号発番) 集計処理(ページ小計、総計) 出力制御(ページブレイク)
グループ	出力制御(同一内容出力制御)
レコード	フィールド分割
フィールド	データ編集・加工

表 4. 帳票編集処理のためのコンポーネント群

参考文献

[1] Pree,W.: Design Patterns for Object-Oriented Software Development, Addison-Wesley (1994). 佐藤, 金澤 (訳):デザインパターンプログラミング, トッパン (1996).

[2] Johnson, R., 中村, 中山, 吉田: パターンとフレームワーク, 共立出版 (1999).

[3] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley (1994). 本位田, 吉田 (監訳):オブジェクト指向における再利用のためのデザインパターン, ソフトバンク (1995).

[4] Martin, R., Riehle, D., Buschmann, F.(editor): Pattern Languages of Program Design 3, Addison-Wesley (1997)

[5] Fowler, M.: Dealing with Properties, http://www.awl.com/cseng/titles/0-201-89542-0/apsupp/properties.pdf, 1997.

[6] Lea, D.: Concurrent Programming in Java - Design Principles and Patterns, Addison-Wesley (1997).

[7] 名取, 加賀谷, 本位田: データ中心アプローチとユースケースに基づくオブジェクト指向フレームワーク構築手法、情報処理学会論文誌, Vol.38, No.3, pp.634-656 (1997).

[8] Larsen, G.: Designing Component-Based Frameworks Using Patterns in the UML, Communications of the ACM, Vol.42, No.10, pp.38-45 (1999). 安藤 (訳):UML パターンを使用したコンポーネントベースのフレームワーク設計, 情報処理, Vol.41, No.4, pp.419-424 (2000).

[9] Bergner, K., Rausch, A., Sihling, M., Vilbig, A.: A Component Development Methodology based on Process Patterns, (1998)

[10] 吉田, 田中, 他: コンポーネントベース・フレームワーク技術(CSolution APF) 実システムへの適用, 情報処理学会第 60 回全国大会, 講演論文集 (2000).

[11] Capers Jones (著), 鶴保, 富野 (監訳): ソフトウェア開発の定量化手法 第 2 版, 共立出版 (1998).

[12] 堀内: データ中心システム設計, オーム社 (1988).

[13] Jacobson, I., Ericsson, M., Jacobson, A.: The Object Advantage, Business Process Reengineering with Object Technology, ACM Press (1995). 本位田 (監訳): ビジネスオブジェクト, ユースケースによる企業変革, トッパン (1996).

[14] (株)東芝: CSolution (システム構築フレームワーク), http://www3.toshiba.co.jp/ccc/

[15] 富士通(株): ComponentAA シリーズ, http://www.fujitsu.co.jp/jp/soft/product/use/compo/index.html