コンポーネントベース・フレームワーク開発手法における コンポーネントの抽出・設計方法論

吉田 和樹[†] 本位田 真一[††]

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の中で，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案する．方法論は抽出・設計に対する3つの限定— 1. コンポーネントは，ルートを1つ持つ有向非循環グラフ（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，2. コンポーネントが提供する処理の中で，ホットスポットになりうる部分を特定して，これを別クラスに分離して実装する，3. データの違いにともなう処理の違いは，ホットスポットとして分離される別クラス，あるいは，データの実装クラスに吸収させることにして，コンポーネントの内部に直接実装しないようにする—に基づいて，4つのフェーズと1つのチェックポイントから成る．そして，この方法論に従ってコンポーネントを抽出・設計した例として，帳票編集処理と業務トランザクション処理のコンポーネントセットについて説明し，それらを実システムでのアプリケーションフレームワーク開発に適用した実績をもとに，その有効性を評価する．

A Component Extraction/Design Methodology in a Component-based Framework Development Method

Kazuki Yoshida[†] and Shin'ichi Honiden[††]

In a method for composing an application framework from a set of components, especially a methodology to extract and design such components from application domain is proposed as a main subject in this paper. This methodology consists of four phases and one checkpoint on the basis of the following three restrictions on the extraction and design; 1. Components are connected in the structure of directed acyclic graph with one root and messages are sent on the arcs from source to destination successively. 2. In the process provided by each component, hot spots are specified and implemented separately in other classes. 3. A variation of process caused by a variation of data are absorbed in the above classes for hot spot or data classes, and never implemented directly in components. According to this methodology, two sets of components are extracted and designed from data processing on the report editing and business transaction processing, then these are applied to the application framework development in the real system. From these results, the effectiveness of this application is evaluated.

1. はじめに

アプリケーションフレームワークがあらかじめ用意されているコンポーネントの組合せで構築できるようになれば，フレームワークを一から開発する場合にともなう設計・実装作業の繰返しから開発者を解放し，また，実装作業そのものを効率化することもできるので，これまでの課題であったアプリケーションフレームワーク開発の生産性向上を期待することができるようになる．本論文では，このような手法をCBF（Component-Based Framework development method）と呼ぶことにして，その中でも，特にアプリケーションフレームワークを構築するために利用されるコンポーネントセットの抽出・設計方法論について提案する．そして，この方法論に従って抽出・設計されたコンポーネントセットの適用が，アプリケーションフレームワーク開発に有効であることを，実システムでの実績をもとに示す．

アプリケーションフレームワークを構築するためのコンポーネントセットの抽出・設計方法論では，そのコンポーネントを使って構築されるアプリケーションフ

† 株式会社東芝 SI 技術開発センター
System Integration Technology Center, Toshiba Corporation
†† 国立情報学研究所
National Institute of Informatics

レームワークが，再利用基盤としてより広い範囲のアプリケーションを扱えるものになっていなければならない．したがって，アプリケーション間での汎用性を確保することが重要になり，そのためには，コンポーネントがアプリケーションフレームワークのホットスポット[1)] を的確に実現できるようになっていることが求められる．したがって，方法論でもその点に焦点が当てられるようになる．もしそれがアプリケーションを開発するためのコンポーネントの場合には，抽出が個別のアプリケーションに密着したものになり，設計もデータや処理がより明確化された形で行われることになるため，上記の点が特に求められることはない．

本論文の以後の章は，次のように構成されている．

まず，2 章では，フレームワークの観点から，コンポーネントセットの抽出・設計に求められる必要条件を明らかにする．次に，3 章では，対象とするドメインと，コンポーネントセットを抽出・設計するうえでの技術的な課題を明確にして，これに答える形で方法論を説明し，その適用可能性についても考察する．4 章では，この方法論を適用して抽出・設計したコンポーネントセットを実システム開発に適用した際の評価結果について説明する．最後に，5 章で関連研究について述べ，6 章で本論文のまとめを行う．また付録では，この方法論の適用例として，業務トランザクション処理のコンポーネントセットの抽出・設計について説明する．

2. 抽出・設計に求められる必要条件

本論文では，コンポーネントを次のように定義する．

[定義]

コンポーネントとは，再利用部品の一実装形態を指し，**図 1** のコンポーネント x に示したように，1 つの代表的なクラスと，そこでの処理を実装するために使われる複数の補助的なクラスの組合せで構成され，外部に対しては，この代表的なクラスが提供する機能の実行やカスタマイズ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☆．通常，ビジュアルな開発支援環境とともに提供されて，そのうえでアイコンとして表現され，利用者は，このアイコンに対するマウス操作などにより，コンポーネントどうしの組合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．

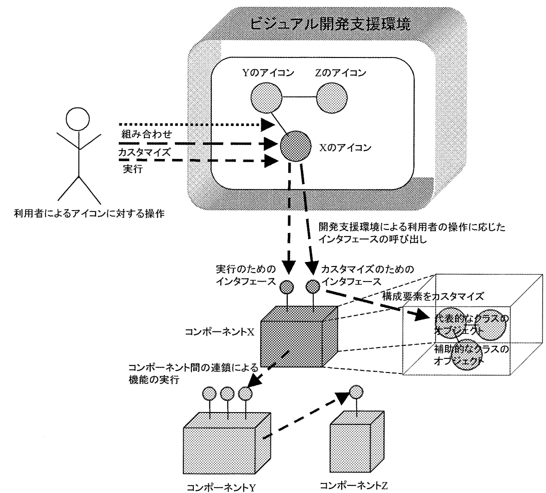

図 1　コンポーネントの概念図
Fig. 1　Conceptual model of components.

このような利用者の操作に応じて，開発支援環境が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，あるいは，呼び出すためのコード生成を行うので，利用者の実装作業は軽減される．

ところで，コンポーネントの組合せでアプリケーションフレームワークを構築する場合，アプリケーションフレームワーク内に存在するホットスポットをコンポーネントの側で問題なく扱えるようにする仕組みを考えなければならない．一般に，プログラム内でホットスポットになりうる部分としては，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（構造的側面）

したがって，コンポーネントの抽出と設計は，これらの点に柔軟性を与える形で行われなければならない．

まず，処理のシーケンスについては，コンポーネントを基本単位として，コンポーネントの組合せ方を変えることにより，処理のシーケンスを変えていく方針で扱う．ただし，その場合に，コンポーネントの組合せの中で，1 つのコンポーネントにおいて関連性のない複数の処理の流れが交差するようにシーケンスが組まれていると，その中の 1 つの処理の流れを変える目的で，そのコンポーネントを別のコンポーネントに入れ替えた際に，他方の処理の流れにも影響を及ぼすという副作用が起こりかねない．そこで，**図 2** に示すように，コンポーネントの組合せの中に，あるコンポーネントを起点にしてつねに一定方向の処理の流れが定まるものとして，この問題を未然に防ぐことにする．

☆ ここでい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．

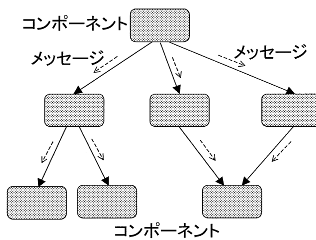

図 2　コンポーネントの組合せと処理の流れ
Fig. 2　Structure of components assembling and flow of processes.

ここで，図にも示されているように，複数の処理の流れに共通化できる部分が存在する場合には，その共通部分の処理を実装したコンポーネントに，複数のコンポーネントから関係をつなぐことは許されてもよい．

以上より，コンポーネントの抽出と設計に次のような必要条件を課す．

［必要条件 1］

コンポーネントは，ルートを 1 つ持つ有向非循環グラフ（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．

また，処理の内容については，ホットスポット部分の処理をコンポーネント単位に分割して，上述のようなコンポーネントの組合せであらかじめ処理のシーケンスを定めた後に，各コンポーネントにその分割したホットスポット部分の処理をプラグインすることで扱えるようにする．そのためには，コンポーネントが提供する処理が互いに他のコンポーネントの処理と結合することがないように抽出されており，そのうえで，提供する処理の中でホットスポットになりうる部分が特定されていなければならない．また，それを委譲の考え方に基づいて別クラスに分離して実装できるようになっていなければならない．

したがって，コンポーネントの抽出と設計にさらに次のような必要条件を課す．

［必要条件 2］

コンポーネントが提供する処理の中で，ホットスポットになりうる部分を特定して，これを別クラスに分離して実装できるようにする（このクラスをカスタマイズクラスと呼ぶことにする）．

これにより，フレームワーク開発者がコンポーネントの組合せをアプリケーションフレームワークとして提供し，アプリケーション開発者が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ように，工程上のフェーズや開発者の種類によって，対象とする部分を切り分けて扱うことが可能になる．このことは，さらに試験工程においても，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．

処理の対象となるデータの違いについては，それを構成の違いとして扱うのか，型の違いとして扱うのかを，コンポーネントの設計段階で選択しなければならない．構成の違いとして扱うとは，共通のクラスに，データの項目名と値を組みにして，これを任意の数だけ追加できるようにする方法で，データが受動的に扱われる場合に選択する．そして，構成の違いにともなう処理の違いは，必要条件 2 に従ってカスタマイズクラスで扱う．一方，型の違いとして扱うとは，データをその種類ごとにクラスを分けて実装する方法である．この場合には，各コンポーネントでデータの種類に依存した処理部分を，クラスとして実装されるデータの側に移すようにして，コンポーネントの側ではポリモルフィズムを利用してこれを呼び出すような実装を行う．これにより，型の違いにともなう処理の違いを扱えるようにできる．

したがって，これまでの必要条件に加えて次の必要条件を課す．

［必要条件 3］

データの違いにともなう処理の違いは，カスタマイズクラス，あるいは，データの実装クラスに吸収させることにして，コンポーネントの内部に直接実装しないようにする．

3. コンポーネントの抽出・設計のための方法論

本章では，アプリケーションフレームワークの構築に利用されるコンポーネントを，2 章で述べた 3 点の必要条件を満たすように抽出・設計するための方法論を中心に説明する．

3.1 対象とするドメイン

方法論の対象範囲は，これまでに CBF のためのコンポーネント開発に実績のある，業務アプリケーションドメインとする．一般に，業務のモデリングの世界では，モデルを構成する要素に，次のような種類が考えられる[14),17)]．

- 業務プロセス
  1 つの業務を構成する一連の処理の流れや状態遷移を表現する．
- 業務エンティティ
  業務を遂行する過程で作成，参照，操作されるデータ要素を表現する．

- 業務サービス☆
  業務プロセスとシステムの動作環境の間にあって，業務プロセスや業務エンティティによる処理を助けるものを表現する．
- 業務ルール
  上記3種類の構成要素が，つねに満たさなければならない業務的な条件や，一定条件下で実行される小処理を表現する．

本論文の方法論は，これらの中で，特に，業務プロセスと業務サービスの部分にフレームワークを開発するためのコンポーネントを抽出・設計するものである．また，これを適用することにより，業務ルールは，カスタマイズクラスとして実現される．

3.2　技術的な課題

方法論に対する技術的な課題としては，次のような点が存在する．

(P1)　コンポーネントの抽出方法について

［必要条件1］を満たすようにコンポーネントを抽出するために，業務機能をどのような観点から分析するのかを明確化しなければならない．

(P2)　コンポーネント化すべきか否かの判断基準について

コンポーネント化することでアプリケーションフレームワークを開発する作業の効率化を見込めるかどうかの判断基準を明確化しなければならない．

(P3)　コンポーネントの設計方法について

コンポーネントがすべての必要条件を満たすように，適切なデザインパターンを選択して，インタフェースの設計や共通クラスの構築などを行うための指針を明確化しなければならない．

(P4)　コンポーネントの状態や処理のコンテキスト，例外の扱いについて

コンポーネントの組合せにより機能を実現する際に，コンポーネントの状態や処理のコンテキスト，例外の設計上の扱いについて明確化しなければならない．

方法論は，(P1)〜(P4)への解をまとめた形で，コンポーネントセットの抽出・設計のための作業プロセスとして示されるが，これは一般的なウオーターフォール型のシステム開発プロセスにおいて，ソフトウェア基本/詳細設計フェーズでの作業の一部として位置づけられる．システムの構成やプログラムモジュールの配置およびそれにともなう設計などのシステム設計フェーズの観点は，ここでの主題からは外れるため，方法論から除外している．

3.3　方　法　論

ここで提案する方法論は，前節であげた技術的な課題(P1)〜(P4)のそれぞれに対して答える形で，4つのフェーズ（フェーズ1〜フェーズ4）と1つのチェックポイントから構成されている．

方法論の実施にあたっては，システム開発の要求定義フェーズでまとめられた，あるアプリケーションについての業務機能，GUI，データベースに関する仕様書が入力となる．業務機能仕様書には，ユースケース図やシステム構成要素間の処理の流れを表現したシーケンス図が含まれていることを想定する．アプリケーションフレームワークを構築するためのコンポーネントセットの開発には，本来は，対象とするドメインに属する複数の多様なアプリケーションの仕様書が必要になる．しかし，実際の作業においては，仮に，複数の仕様書が存在しても，その中の1つでまずコンポーネントの抽出・設計を行い，それを基にして別の仕様書で適用可能性を確認し，必要ならば洗練化を行って，という形で作業が進められることが多い．したがって，方法論では，最初の入力は1つのアプリケーションの仕様書として，その代わりに，複数のアプリケーションを考慮に入れた汎用性の見直しを，後の段階で行うことを定めている（複数の仕様書について繰り返される洗練化のための作業を，プロセスの中に明確な形で取り込むことは，プロセスが煩雑になり説明が難しくなると考え，本論文では行っていない）．方法論に従った作業の結果は，クラス図，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を順番に実施した後は，フェーズ3とフェーズ4は，互いに独立の関係にあるため，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，まず概要を述べ，次に作業プロセスの詳細を説明する．

3.3.1　フェーズ1

(1)　概要

フェーズ1は，(P1)に対する解を提示する．

業務アプリケーションドメインでは，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い．そこで，コンポーネントの候補となる再利用の基本単位を，機能の入力→処理→出力を明確にして，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として抽出するか，あるいは，それらとは独立に処理の部分から抽出する．その作業プロ

☆ 文献14)や17)ではあげられていない．本論文で独自に追加したものである．

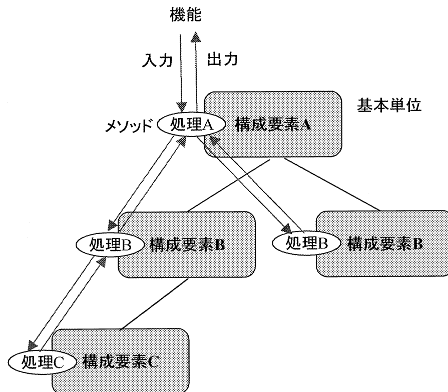

図 3 入出力データの構造を基本単位に
Fig. 3 Basic units extracted from a structure of input or output data.

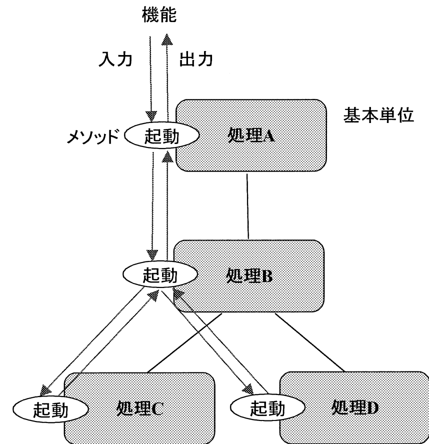

図 4 処理を基本単位に
Fig. 4 Basic unit extracted from process.

セスを，前者はステップ 1) で，後者はステップ 2) で説明する．

そして，これらの作業プロセスを通して，基本単位が［必要条件 1］を満たすように抽出できたかどうかを，ステップ 3) で確認する．

(2) 作業プロセスの詳細

ユースケース図に示された各ユースケースを，入力→処理→出力の観点でとらえ，次のステップを実行する．

ステップ 1)

① GUI 仕様書やデータベース仕様書を参考にして，入力データ，出力データの構造を，複数種類の基本単位から成る構成要素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り当てて，**図 3** に示すように，この構成要素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，シーケンス図を記述しながら確認する．

③ もし実現可能ならば，各クラスに割り当てられた処理を，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 ステップ 1) での作業において，入力データ，あるいは，出力データの構造に特定の基本単位が見い出せたとしても，入力データ，あるいは，出力データの構造そのものをホットスポットとして扱えるようにしておきたい場合には，各基本単位に割り当てられていた処理を抜き出して，基本単位に保持されていた処理対象のデータを，その処理に入力として与えることで同等の処理を実現するようにして，この処理を基本単位にする．そして，**図 4** に示すように，この基本単位の組合せにより，処理の全体を構成することができないかどうかを確認する．

② もしそれが可能ならば，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．

ステップ 3)

ステップ 1)，あるいは，ステップ 2) の作業を通して抽出された基本単位が，［必要条件 1］の次の 2 点を満たしているかどうかを確認する．

① 基本単位は，ルートを 1 つ持つ DAG 状の構成で組み合わされる．

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．

もし上記のいずれかを満たさなければ，仕様書が示しているアプリケーション，および，そのアプリケーションが属するドメインは，本方法論の適用範囲外であるとして，ここで作業を終了する．

3.3.2 チェックポイント

(1) 概要

チェックポイントは，(P2) に対する解を提示する．

コンポーネントの組合せによるアプリケーションフレームワークの開発は，そのアプリケーションドメインにコンポーネントの組合せ方に関するバリエーションが存在していなければ効果が薄い．したがって，フェーズ 1 で機能から基本単位が抽出された後にその確認を行う．

(2)　作業プロセスの詳細

アプリケーションが属するドメインの観点から（同じドメインに属する他のアプリケーションの仕様書などを参考にして），基本単位間の関係をより汎用的になるように見直して，必要ならばクラス図に修正を加える．そのうえで，次の基準で判断を行う．

[判断基準]

基本単位のクラス図内に，可変多重度，再帰，サブクラス化の要素が存在する☆．

上であげた3点は，基本単位の組合せ方にバリエーションを生むもとになるもので，もしこの基準が満たされなければ，そのアプリケーションドメインでは基本単位が固定的な組合せでしか利用されないことになるので，それらをコンポーネント化しておく必要はない．もし基準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（これらのクラスを以後の説明ではコンポーネントと呼ぶ）．ただし，コンポーネント数の増大を防ぐために，安易なサブクラス化は避けるべきである．複数のコンポーネント間の差異が，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吸収できるならば，そのようにする．

3.3.3　フェーズ2

(1)　概要

フェーズ2は，(P3) に対する解を，機能の実行のためのインタフェースに関して提示する．

アプリケーションフレームワークが［必要条件1］に基づいたコンポーネントの組合せにより機能を実現するために，コンポーネントのインタフェース設計をステップ2)，3) で行う．その際，各コンポーネントでの処理について考える中で，［必要条件2］に基づくカスタマイズクラスの抽出や，［必要条件3］に基づくデータの違いにともなう処理の違いへの対処を行う．

また，コンポーネントの汎用性を高めるために，コンポーネントに持たせる属性の種類を変更可能にする設計をステップ1) で行う．

さらに，データを生成する処理をコンポーネント化するための設計をステップ4) で行う．

(2)　作業プロセスの詳細

本フェーズを含む以後のフェーズでは，個々のコンポーネント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，そのコンポーネントが利用される状況によりバリエーションが存在するならば，それをホットスポットとして特定し，Dynamic Properties パターン[5] を適用することにより吸収する（**図5** 参照）．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．

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り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り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．

(1)　自身が呼び出されたときに処理を加える☆☆．

(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☆☆☆．

(3)　(1) と (2) の両方．

また，コンポーネントに割り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，その処理内容に関してアプリケーションドメイン内でバリエーションが存在するならば，それをホットスポットとして特定し，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する．そして，そ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．

これらのインタフェースを定義する際に，引数として渡すデータにアプリケーションドメイン内でバリエーションが存在するならば，それをデータの構成の違いとして扱うのか，型の違いとして扱うのかを選択する．

以上の作業結果から，コンポーネントやカスタマイズクラス，さらには，データのクラスを通した全体の処理の流れをシーケンス図に記述する．

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合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，その処理内容に関

☆ Composite パターン[3] はその典型的な例である．

☆☆ そこで加えられる処理を前処理と呼ぶ．これは構造化分析/設計における遠心型モジュールに相当する．

☆☆☆ そこで加えられる処理を後処理と呼ぶ．これは構造化分析/設計における求心型モジュールに相当する．

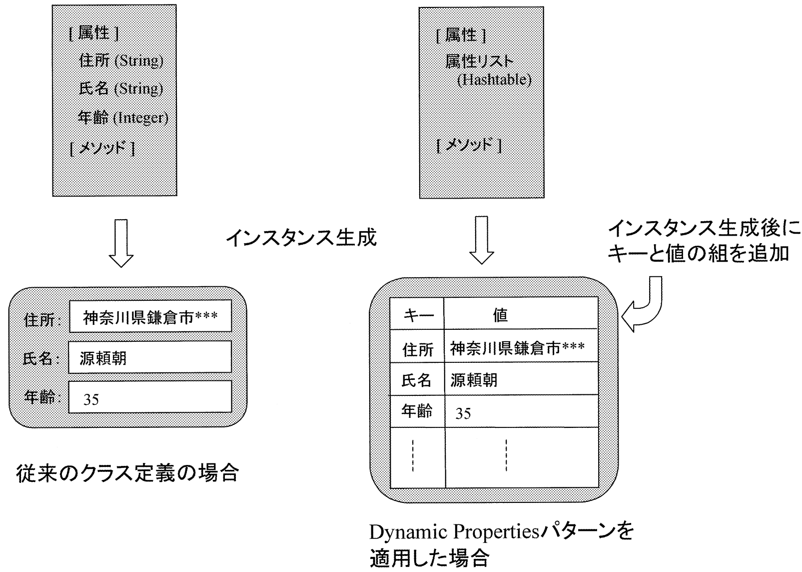

図 5 Dynamic Properties パターンの適用イメージ
Fig. 5 How is Dynamic Properties pattern applied ?

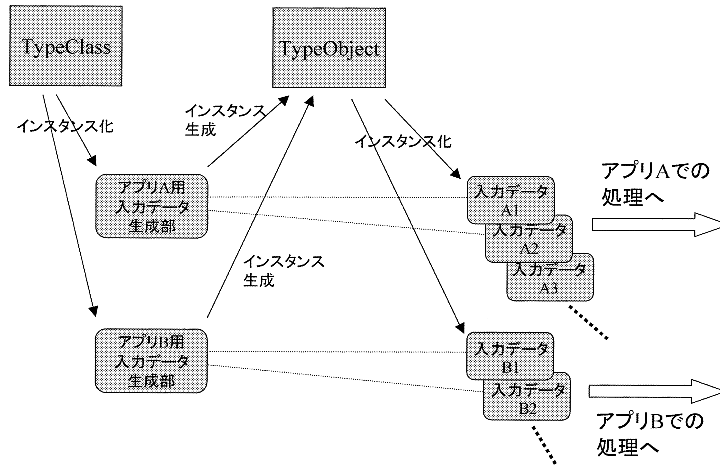

図 6 Type Object パターンの適用イメージ
Fig. 6 How is Type Object pattern applied ?

してアプリケーションドメイン内でバリエーションが存在するならば，それをホットスポットとして特定し，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する．そして，そ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．

これらのインタフェースを定義する際に，引数として渡すデータにアプリケーションドメイン内でバリエーションが存在するならば，それをデータの構成の違いとして扱うのか，型の違いとして扱うのかを選択する．

以上の作業結果から，コンポーネントやカスタマイズクラス，さらには，データのクラスを通した全体の処理の流れをシーケンス図に記述する．

ステップ 4)

特に，データの構成にアプリケーションドメイン内でバリエーションが存在するならば，データを生成する部分に Type Object パターン[4] を適用して，TypeClass にデータ（TypeObject インスタンス）の生成機能を持たせることで，吸収する（**図 6** 参照）．そして，TypeClass をコンポーネントの 1 つに加え，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，

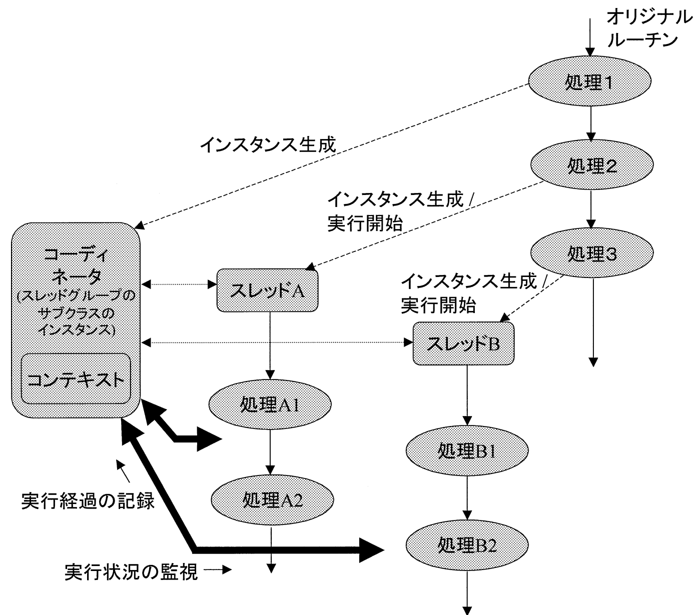

図 7　マルチスレッド実行時のコーディネータ
Fig. 7　A coordinator for multi threads execution.

それをクラス図とインタフェース定義書に記述する．

ステップ 5)

あるユースケースを基に行ってきたコンポーネントの抽出・設計作業の結果を，別のユースケースにあてはめる場合には，カスタマイズクラスでの処理の内容やデータの内容を変えることで，同じコンポーネントセットのままで機能を実現できないか，あるいは，別の基本単位をコンポーネントとして加えることで実現できないかを確認する．実現できそうな場合には，それまでの設計結果に対して整合性がとれるように，フェーズ 2 のステップ 1) に戻って詳細に見直しをかけ，必要ならば設計を修正する．実現できない場合には，そのユースケースについて，再びフェーズ 1 から作業を行う．

3.3.4　フェーズ 3

(1)　概要

フェーズ 3 は，(P4) に対する解を提示する．

コンポーネントの状態や，処理のコンテキスト，例外について，その扱い方の方針をステップ 1)，2)，3) でそれぞれ別々に説明する．

(2)　作業プロセスの詳細

業務機能仕様書のユースケース図やシーケンス図を基にして，状態やコンテキスト，例外を明確化する．

ステップ 1)

状態については，その状況に応じて，State パターン[3] を適用して，状態による処理の違いを，別途定義した状態クラスに委譲する形で扱う．その際，状態遷移の同期化による処理のボトルネック/デッドロックを考慮する．

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．

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．

さらに，Java の場合には，

②　スレッド☆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．

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☆☆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（**図 7** 参照）．

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を区別できるようにする．

(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

☆ java.lang.Thread クラス．

☆☆ java.lang.ThreadGroup クラス．

(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

(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．また，正常ケースとして意図的に例外を発生させる場合も考えて，先述の分類のさらに上位に，正常/異常の分類を表すクラスをそれぞれ作成して，これを先述のクラスに継承させるようにする．

以上の各ステップでの作業結果を，クラス図，シーケンス図，インタフェース定義書に記述する．

3.3.5　フェーズ 4

(1)　概要

フェーズ 4 は，(P3) に対する解を，機能のカスタマイズのためのインタフェースに関して提示する．

関係や属性の設定，取得などのためのインタフェースを，関係や属性の種類を識別子を使って区別しながら，すべてを統一的に扱えるようにする設計をステップ 1) で行う．これにより，関係や属性の種類ごとに accessor 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ことができる．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ともなうことにもなるので，実装にあたってはトレードオフの選択が必要である．

また，そのほかに，インタフェース追加や機能追加を動的に行えるようにするための設計を，ステップ 2) と 3) で各コンポーネントに対して行う．

(2)　作業プロセスの詳細

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．それを基に，関係や属性の設定，取得のためのインタフェースを統一的に設計する．

ステップ 2)

［必要条件 1］で規定された処理の連鎖により実現する機能を新たに追加できるようにするために，Visitor パターン[3] を適用する．

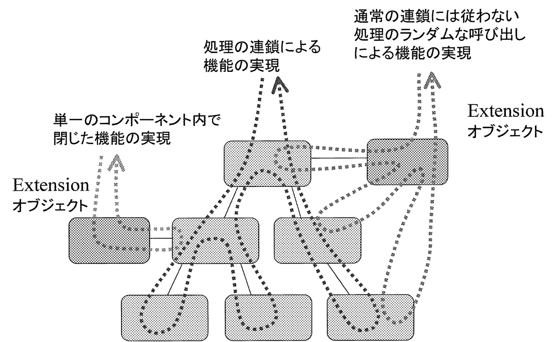

図 8　Extension Object パターンの適用イメージ
Fig. 8　How is Extension Object pattern applied ?

ステップ 3)

各コンポーネント内で閉じた機能や，［必要条件 1］で規定された連鎖には従わない，ランダムな処理の呼び出しにより実現する機能を，後からコンポーネントに追加できるようにするために，各コンポーネントには Extension Object パターン[4] を適用する（**図 8** 参照）．

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．

ここでの作業は，典型的な関係や属性について統一的なインタフェースを実装したクラス（**図 9** 参照）を，コンポーネントの共通クラスとして用意しておくことで，省略することも可能である．この共通クラスでは，たとえば，典型的な関係として，ツリー状の組合せにおける親への参照，子への参照，カスタマイズクラスのインスタンスへの参照を，また，典型的な属性として，入力項目，出力項目，内部属性を保持できるようにしておく．各メソッドでは，これらの関係や属性をどのようなデータ構造で保持しておくのかをあらかじめ決めて，処理を実装しておく．

3.3.6　方法論のまとめ

方法論に従ってコンポーネントセットを抽出・設計することにより，アプリケーションフレームワークのアーキテクチャは，［必要条件 1］に示したように，ルートを 1 つ持つ DAG 状の構成になり，その各ノードがコンポーネントになる．コンポーネントは，［定義］で述べたように，1 つの代表的なクラスと複数の補助的なクラスの組合せで構成される．代表的なクラスとは，機能の実行，あるいは，カスタマイズのためのインタフェースを，内部で補助的なクラスを利用して実装しているクラスであり，Facade に該当する．フェーズ 1 では，この代表的なクラスを抽出するための作業を行

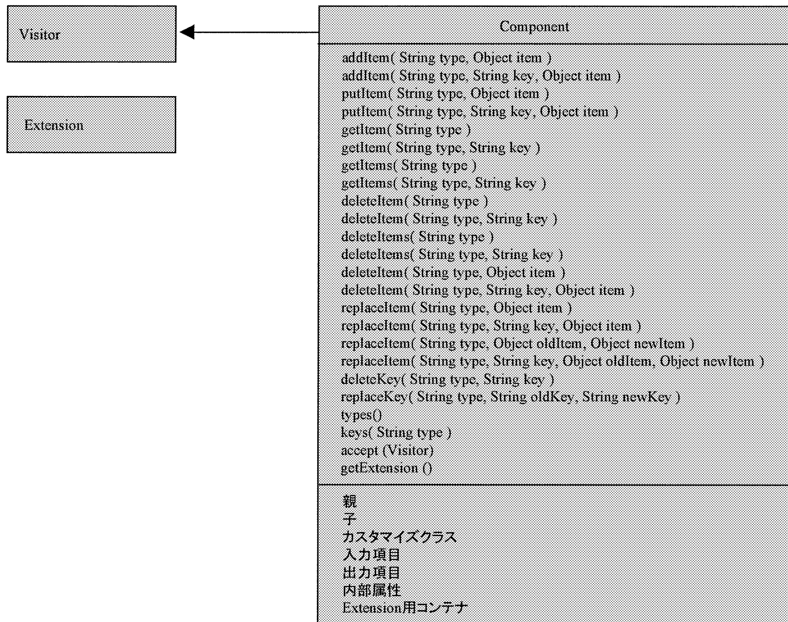

図 9 コンポーネントの共通クラス
Fig. 9 A common superclass for components.

表 1 作業内容の要約
Table 1 Summary of work items.

代表的なクラス	入力データ、出力データを構成する基本単位として抽出(1-1) 処理を構成する基本単位として抽出(1-2) 構成とメッセージの連鎖の確認(1-3) 組み合わせのバリエーションの確認(チェックポイント) インタフェース定義(2-2, 2-3, 4-1) データの生成処理部分を抽出(2-4) 別ユースケースへの適用(2-5)	
補助的なクラス	機能の実行	機能のカスタマイズ
	Dynamic Propertiesパターン適用(2-1) カスタマイズクラス抽出(2-2, 2-3) Stateパターン適用(3-1) スレッドクラス導入(3-2) スレッドグループクラス導入(3-2) 例外クラス定義(3-3)	Visitorパターン適用(4-2) Extensionパターン適用(4-3)

い，基本単位と読んでいるものが，その候補になる．そして，このクラスをコンポーネントの Facade として実装すべきか否かという判断を，チェックポイントで行う．一方，補助的なクラスについては，フェーズ 2，3，4 で扱う，各種のデザインパターンにより導入されるクラスや，処理を実装する際に状況に応じて選択されるクラス，などがこれに該当する．各フェーズでの作業内容の要約を，代表的なクラスと補助的なクラスに分けて，**表 1** にまとめる．

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，コンポーネントセットを抽出・設計した事例の説明を付録に載せる．

3.4 適用可能性

本章で示した方法論の適用可能性について，2 章であげた必要条件の観点から考察する．

フェーズ 1 では，[必要条件 1] を満たすようなコンポーネントの抽出を，入出力データの構造，あるいは，処理から行うが，データの階層的な構成は必然的にツリー構造になり，また，処理についても，モジュール間の呼び出し関係はツリー構造として表現することができる．したがって，コンポーネントの抽出を上記 2 通りで考える限り，DAG 状の構成ということはそれほど厳しい条件にはならないと考える．しかし，一般的なオブジェクト指向分析では，いわゆる現実世界の "もの" をオブジェクトの候補として抽出することもあり，それらの "もの" オブジェクト間で，メッセージがネットワーク状にやりとりされて連携が行われるような対象では，DAG 状の構成にまとめることは厳しい条件になると考えられる．また，仮に処理から DAG 状の構成が抽出できたとしても，各ノードにおける処理の内容が再利用単位として適切なものなのか，あるいは，データから DAG 状の構成が抽出できたとしても，上位ノードから下位ノードへのメッセージの連鎖を考えるだけで，求められる機能を実現できるのか，というところは条件としての厳しさにつながる可能性がある．

[必要条件 2] については，ホットスポットをドメイン分析を通していかに特定するかという問題に深く

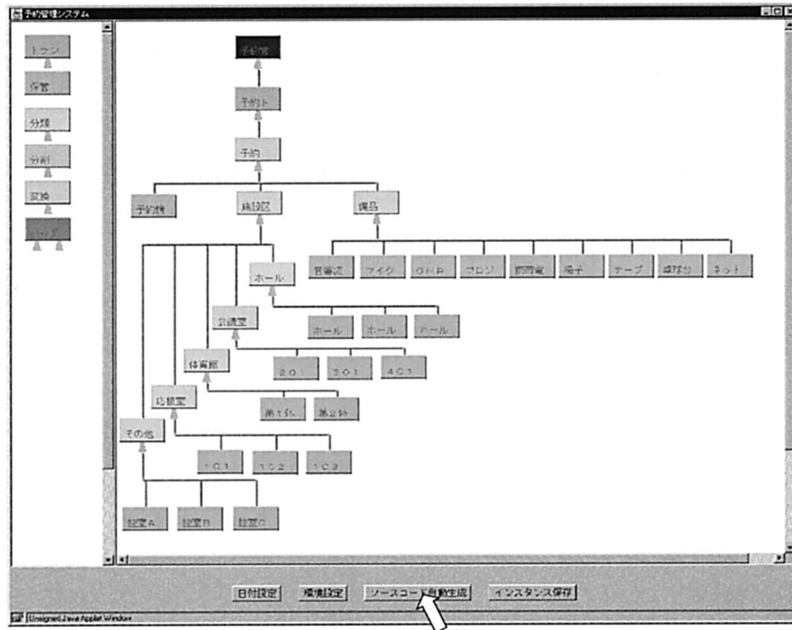

図 10 開発支援環境の基本画面
Fig. 10 Visual programming tool for components.

関係している．本論文では，この問題を直接は扱っていないが，これは条件自体の厳しさというよりも，開発者の分析・設計スキルに依存する問題であると考える．たとえば,［必要条件 3］は,［必要条件 2］の一例であり，データの違いによる処理の違いという典型的なホットスポットを，データのクラス，あるいは，カスタマイズクラスへの委譲により扱うことを定めたものであるが，これは設計上のテクニックで十分に実現可能なことであり，条件の厳しさにつながるものではない．

以上の考察から，3 つの必要条件の中では,［必要条件 1］が抽出・設計に対して最も厳しい条件を課すことになる．しかし，業務アプリケーションドメイン，特に，業務データの処理を行うような情報システムでは，この条件は大きな障害にはならないと考える．その反面，たとえば，機器間の協調動作を制御するようなリアルタイム制御システムや，処理ルーチンのジャンプや戻りが利用者からの要求や環境に応じて頻繁に発生するようなグループウェア類，さらには，通信系のシステムなどには，適用は難しくなると考えられる．

3.5 開発支援環境

方法論に従うことで,［必要条件 1］や［必要条件 2］を満たし，かつ，カスタマイズのためのインタフェースが統一されたコンポーネントが抽出・設計されるので，コンポーネントの組合せやカスタマイズクラスのプラグインなどをビジュアルに行えるようにするための開発支援環境を汎用的に用意することが可能になる．

図 10 は，Web ブラウザ上で動作する開発支援環境の構築例である．画面左端はコンポーネントパレットと呼ばれ，利用可能なコンポーネントの一覧がそれぞれアイコンで表現されて並べられている．画面右側はキャンバスと呼ばれ，その上でコンポーネントの組合せを行う．

開発者は，これを利用して，次の作業を行う．

① コンポーネントパレットからコンポーネントを選択して，キャンバス上に配置する．
② コンポーネント間を結線する．
③ コンポーネントのプロパティ設定を行う．
④ カスタマイズクラスを汎用プログラミング環境で実装する．
⑤ カスタマイズクラスをプラグインする．
⑥ フレームワークの構成を表すソースコードを自動生成する☆．

このように，開発者がやらなければならない作業は限られており，また，これらは④を除けば，開発支援環境の GUI 上でマウス操作などにより簡単に実施できるようになっている．

☆ 開発支援環境では，コンポーネントやカスタマイズクラスを組み合わせた結果を，動作確認のために直接実行する機能も提供している．

表 2　適用例の一覧
Table 2　Application frameworks assembled from components.

	受注管理システム	会計システム
帳票編集処理	事前リスト ダイレクトメール 配送伝票 郵パック 短冊のし	会計仕訳表(6種類) 出納確認票 修正原票
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

コンポーネントの基本的な狙いを具現化するこのような開発支援環境の提供は，生産性の向上や習熟の早さを実現する助けになると考える．

4. 評　　価

この章では，本論文で提案した方法論を通して抽出・設計されたコンポーネントセットの適用が，アプリケーションフレームワークの開発に有効であることを，実績をもとに示す．

これまでに，方法論を通して，帳票編集処理のためのコンポーネントセットと業務トランザクション処理のためのコンポーネントセットが抽出・設計されたが（付録参照），それらを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システムの 2 つの実システム開発にそれぞれ適用して，帳票編集処理で 13 個，業務トランザクション処理で 10 個，合計 23 個のアプリケーションフレームワークを作成することができた（**表 2** 参照）．

たとえば，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セットを適用した際には[10]，ファンクションポイント（IFPUG 法)[11] で，アプリケーションの全機能量 474.37 のうち，374.05 の部分をカバーすることができた（カバー率：78.9%）．この適用/非適用の区別は，その機能を実現する処理をトランザクションとして扱う必要があるか否かにより生じている．

また，これらの機能を実装しているフレームワーク部分において，コンポーネントを使って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，

表 3　生産性指標の比較
Table 3　A comparison of productivity index.

	LOC数	作業時間 (単位:h)	生産性指標 (LOC数/開発時間)	比率
コンポーネント不使用	12,382	984	12.58	1
コンポーネント使用	12,035	433	27.79	2.2

表 4　ラウンドトリップ型開発における生産性指標の比較
Table 4　A comparison of productivity index in a round trip-style development.

	LOC数	作業時間 (単位:h)	生産性指標 (LOC数/開発時間)	比率
ラウンド1	3,651	55	66.38	1
ラウンド2	9,697	61	158.97	2.4

前者が 27.79，後者が 12.58 となり☆，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いえ，コンポーネントを使った方で約 2.2 倍の生産性向上を示す値を得ている（**表 3** 参照）．ただし，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，開発支援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り，また，開発時間とは，コンポーネントの組合せに要した時間とカスタマイズクラスのコーディングに要した時間の合計である．すなわち，ここでは，コンポーネントセットを使う/使わないのそれぞれの場合について，アプリケーションフレームワークの開発とそのアプリケーションフレームワークを用いたアプリケーションの開発を合わせた形で評価が行われ，比較が示されていることになる．

さらに，このシステム開発において，コンポーネントの組合せによる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，生産性が約 2.4 倍向上している結果が出ている（**表 4** 参照）．ただし，作業時間としては，純粋にラウンドトリップで作業を行ったコンポーネントの組合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（本来，生産性は，習得やテストの時間も含めて評価されるべ

☆ プログラミング言語はともに Java.

きであるが，ここでは単に，ラウンドトリップにおける1回目と2回目の相対的な評価を行うためだけに，コード行数/作業時間を計算している．したがって，その値は，習得からテストに至るまでの一連の作業時間がすべて計上されている前述のコンポーネント使用時と不使用時の生産性指標の値と比較できるものではない）．この1回目と2回目を比較すると，1回目では，コンポーネントの組合せにより1種類のアプリケーションフレームワークを作成したのに対し，2回目では，この1種類に対する修正のほかに，新たに6種類のアプリケーションフレームワークを作成している．しかも，それを1種類のときとほぼ同程度の時間で作成していることを考えると，これは，利用者のコンポーネント利用に対する習熟を表していると考えらる．

コンポーネントセットを実システム開発で広い範囲に適用することができた理由として，まず，機能を入力→処理→出力の観点からとらえてコンポーネントの抽出を行うことで，何を汎用化すべきかが見極めやすくなった点があげられる．方法論は，そこでの分析結果に基づき，コンポーネントを，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として抽出する場合と，それらとは独立に処理の部分から抽出する場合の両方をサポートしている．また，フレームワーク内でホットスポットになりうる3点（処理のシーケンス，処理の内容，処理の対象となるデータ）に対して，方法論では3つの必要条件を満たす形で柔軟性を与える設計を行っていることもあげられる．さらに，コンポーネント内に属性や関係の種類の追加を動的に行えるようにして，その設定，取得などのためのインタフェースを統一的に設計している点も，コンポーネントの汎用性を高めるのに役立っていると考える．

また，コンポーネントを使用することで生産性向上を示す結果が得られたことは，次のような理由によると考えられる．

すなわち，コンポーネントを使用した場合のソースコードの中で，自動生成されるソースコード（5,315行）はコンポーネントの組合せ方を記述したもので，これは開発支援環境上でアイコンとして表現されたコンポーネントを，マウス操作によりビジュアルに組み合わせる作業の結果として生成されるものである．また，カスタマイズクラス（6,720行）は，あらかじめインタフェースが規定されており，この中で何を記述しなければならないかは明確に定義されている．ここで定義したカスタマイズクラスの数は24個で，テーブル内のデータ項目が列挙された形でのコーディングが多いため，1つのクラスあたりのコード行数はかさばる傾向にあるが，数としては多くはなく，また，インタフェースが規定された抽象クラスを継承して実装すればよいだけなので，利用者自身が継承階層を考えてクラス間の関係を整理していく必要もない．これに対して，コンポーネントを使用しない場合には，66個のクラスが定義され，この中には，大きく3つの継承階層が存在して，それぞれで5階層，5階層，3階層のクラス階層が形成されている．これは，フレームワーク設計の観点から見た場合には一般的なことだが，実際には，この階層を考える際に大きな労力が必要とされる．このように，コンポーネントを使用した場合に，利用者にとって比較的に容易に設計・実装作業に取り組めるようになっていることが，116時間という短い時間でコンポーネントを使用しない場合とほぼ同等のコード量を設計・実装できた大きな理由になっていると考える（コンポーネントを使用しない場合に設計・実装作業に費やした時間は637時間）．

そのほかに，試験に要した時間も，コンポーネントを使用しない場合が236時間であるのに対して，コンポーネントを使用した場合が63.5時間とかなり少なく済んでいる．これは，コンポーネントを使用した場合には，コンポーネントが提供する処理部分についてはすでに品質が保証されており，あらためて試験を行う必要はなく，コンポーネントとカスタマイズクラスを組み合わせて，それが全体として要求された処理を提供していることだけを確認すればよいためである．

また，設計ドキュメントを作成する時間についても，両者には差が存在する．コンポーネントを使用しない場合には，新規に定義したクラスに関して，クラス図やメッセージシーケンス図を作成しなければならず，これが複雑になる傾向にあるのに対して，コンポーネントを使用した場合には，開発支援環境上でビジュアルに表現されたコンポーネントの組合せ図そのものが設計仕様書として利用でき，また，カスタマイズクラスについてはJavaDocなどを利用することで，ドキュメント自体を作成するのに要する作業時間が無視できるほどになるためである．

その反面，コンポーネントを使用する場合は，その使い方を習得する時間が必要になり，これは作業時間全体の中でかなりの割合を占めることになる．今回の例では，253.5時間がこのために使われ，全体に占める割合は58.5%に及んでいる．しかし，これは初期コストにあたるもので，次回の開発からこの部分は省略できるようになるので，生産性のさらなる向上が期待できる．

ところで，一般にコンポーネントベースの開発における生産性評価では，コンポーネントを利用する際のメリットと開発する際のオーバヘッドが見合うかどうかを，コンポーネントを利用して開発された複数のアプリケーションフレームワーク，さらには，アプリケーションを対象としてトータルに議論する必要がある．しかし，CBF については，現段階ではまだ十分に適用実績が蓄積できていないなどの理由で，開発する側/利用する側双方からの評価が行えない状況にある．ただし，CBF のコンポーネントは，個別のアプリケーションのためだけに開発されるものと違い，アプリケーションフレームワークの開発を行うためのものであることから，汎用性が高く，より多くのアプリケーションに適用される可能性が高まる．したがって，コストの回収も比較的に短期間で可能になると考えられる．今後，ソフトウェア業界が，フレームワークやコンポーネントを開発する企業と利用する企業に二極分化していくことになれば，開発する側と利用する側でそれぞれのビジネスモデルも異なるものとなり，それにともなって生産性に関する評価も別の方法で分けて議論する必要が出てくるであろう．

コンポーネントを使用するためには，開発担当者がコンポーネントの使い方を習得する必要がある．前述のとおり，今回の事例では，特にこれが作業時間全体の中で大きな割合を占めている．もし，実際に使い始めた後に習熟の進みが早ければ，生産性の向上により，この習得のための初期コストを補ったうえで，開発コストの削減が早期に実現できるようになるため，習熟に関する上記の結果には大きな意義があると考える．

5. 関連研究

一般に，コンポーネントベースの開発方法論には，次の 3 点が示されている必要があると考える．

- コンポーネントの抽出・設計のための作業プロセスが示されていること．
- プロセス内の各作業に対してノウハウがまとめられていること．
- コンポーネント化すべきか否かの判断基準が明確化されていること．

これらの観点から，既存の開発方法論を比較する．まず，コンポーネントベースの開発方法論として広く知られているものに Catalysis [16] がある．そこでは，方法論が全部で 48 個のパターンにより構成されているのが特徴である（プロセス全体：4 個，ビジネスモデルの構築：13 個，コンポーネントの仕様化：13 個，コンポーネントの実装：18 個）．このスタイルは，作業ノウハウを一定のフォーマットで整理して記述できるという利点がある反面，作業プロセスが不明確で，ノウハウの寄せ集め的な印象を利用者に与えてしまうのが欠点である．これに対して，作業プロセスに焦点を当ててまとめられた方法論に Rational Unified Process（RUP）[15] がある．RUP も，コンポーネントベース開発への対応が謳われており，ロードマップにはコンポーネントの開発に適合させた反復ワークフローが示されている．しかし，逆に各作業に対するノウハウが欠如している．Kobra [18] は，specification から realization への詳細化を再帰的に行うことにより，1 つのシステムをコンポーネントの階層的な組合せにより実現するための方法論である．作業プロセスにコンポーネントの階層の概念が導入された点が特徴だが，個々の realization における作業内容は，従来のオブジェクト指向分析・設計のプロセスから変わるものではなく，コンポーネントとして実現する可能性や妥当性には疑問が残る．

Catalysis のように作業プロセスが不明確であれば，ソフトウェア開発の工業化を志向する産業界においては，工程管理や品質管理などの不備から，それ自体が方法論として認められるものにはならない．また，RUP のようにノウハウが欠如していると，成果物の品質が担当者の経験やスキルに依存する形になり，これも工業化にとっての阻害要因になる．さらに，Kobra のようにソフトウェアの構成要素すべてをコンポーネントとして扱うという立場では，本来は不必要な箇所に，コンポーネント化のための過剰な開発投資を招くことにもなりかねず，プロジェクトの管理者にとっては，受け入れ難い．これに対して，本論文で提案した方法論では，作業プロセスとノウハウ，コンポーネント化の判断基準をそれぞれ明確にしているため，上記の 3 つの方法論と比べても，実用的であるといえる．このようなコンポーネントベースの開発方法論の出現や，それに対する関心の高まりを受けて，UML でも，バージョン 1.4 からコンポーネントやフレームワーク，パターンといった概念のための表記が新たに追加されている．

フレームワーク設計とパターンの関係を扱ったものとして，Pree が提起した “メタパターン”[1] がある．メタパターンは，デザインパターン [3] にあげられているようなオブジェクト指向設計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，ホットスポットへのパターンの適用方法を示している．また，文献 8) では，パターンの適用によるフレームワークの設計方法が具体的な例を

使って説明され，さらに，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（これについては Catalysis でもより体系的にまとめられている）．コンポーネント設計は，フレームワーク設計の延長線上にあるものとしてとらえることができるが，本論文で提案した方法論に基づくコンポーネントでは，設計上でオープンな問題になっているのは，ホットスポットの中でも特に処理の内容に関係する部分だけなので，適用のために考慮しなければならないパターンも限定することができる．

文献 7) では，ホットスポット/フローズンスポット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．本論文で提案した方法論では，ホットスポットの特定について，特に具体的な手段を示していない．したがって，文献 7) の研究成果との融合により，この点を補強していくことが考えられる．また，業務アプリケーションドメインからのコンポーネントの抽出を視野に入れて，業務を定義するために使われる概念を体系的に整理して，業務のモデリングを支援する方法論も出てきている[14),17)]．今後，このような開発の上流工程における方法論と，本論文で提案した中流，下流工程を対象とする方法論の融合も考えられる．

ところで，コンポーネントを利用した開発では，たとえば，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あわせて新たに作り変える必要がある．文献 9) では，このモデルを一群のプロセスパターンとして表現している．CBF では，特に，コンポーネントの組合せ方に対する試行錯誤を減らすための方法が必要になるが，その作業についてまとめる際にも，上記のアプローチは参考にできる．

6. ま と め

本論文では，CBF を支えているコンポーネントの抽出・設計方法論を明らかにした．そこでは，次にあげる 4 つの技術的な課題，

- コンポーネントの抽出方法，
- コンポーネント化すべきか否かの判断基準，
- コンポーネントの設計方法，
- 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，

を解決して，コンポーネント開発者に対する指針を提示することができた．さらに方法論を通して抽出・設計されたコンポーネントを，実システムでのアプリケーションフレームワーク開発に適用して，評価を行った．その結果は，コンポーネントが広い範囲に適用でき，また，その適用により生産性が向上すること，さらに，利用者の習熟も早いことを示している．今後も適用と調査を繰り返し行いながら，方法論の改良を進めていく．

謝辞 本研究の機会を与えてくださった（株）東芝 SI 技術開発センターの堤本明史所長，調重俊技監，小林恵部長に感謝する．

参 考 文 献

1) Pree, W.: *Design Patterns for Object-Oriented Software Development*, Addison-Wesley (1994). 佐藤啓太，金澤典子（訳）：デザインパターンプログラミング，トッパン (1996).
2) Johnson, R.，中村宏明，中山裕子，吉田和樹：パターンとフレームワーク，共立出版 (1999).
3) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (1994). 本位田真一，吉田和樹（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. and Buschmann, F. (Eds.): *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) 名取万里，加賀谷聡，本位田真一：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, *Comm. ACM*, Vol.42, No.10, pp.38–45 (1999). 安藤　進（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. and Vilbig, A.: A Componentware Development Methodology based on Process Patterns, *Pattern Language of Programs 1998* (*PLO98*), Monticello, Illinois (1998).
10) 吉田和樹，田中誠一郎ほか：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) Jones, C.（著），鶴保征城，富野　壽（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. and Jacobson, A.: *The Object Advantage, Business Process Reengineering with Object Technology*, ACM Press (1995). 本位田真一（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) Herzum, P. and Sims, O.: *Business Component Factory*, John Wiley & Sons, Inc. (2000).
15) Kruchten, P.: *The Rational Unified Process: An Introduction*, Addison-Wesley (2000). 藤井拓（監訳）：ラショナル統一プロセス入門，ピアソン (1999).
16) D'Souza, D. and Wills, A.C.: *Objects, Components, and Frameworks with UML*, Addison-Wesley (1998).
17) Penker, M. and Eriksson, H.E.: *Business modeling with UML: Business Patterns at Work*, John Wiley & Sons, Inc. (2000).
18) Atkinson, C.: Component-based product line engineering with UML, *Enterprise Distributed Object Computing Conference 2000 Tutorial Proceeding* (2000).
19) Ehrig, H. and Mahr, B.: *Fundamentals of Algebraic Specification 1, Equations and Initial Semantics*, Springer-Verlag (1985).

付　録

3.3 節で示した方法論に従って，業務トランザクション処理のドメインから，コンポーネントセットを抽出・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，次に方法論のフェーズ/ステップから特にコンポーネントセットの抽出と，処理の連鎖の設計を行う部分を取り上げて，作業の詳細を説明する．

A.1 業務トランザクション処理

A.1.1 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セットは，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（たとえば，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．

日	付	摘　要	元丁	借　方	貸　方
1	10	（現金）	100	500,000	
		（資本金）	300		500,000
1	16	（コンピュータ）	150	400,000	
		（プリンタ）	151	70,000	
		（現金）	100		470,000
1	18	（広告費）	505	20,000	
		（買掛金）	200		20,000
1	22	（売掛金）	103	120,000	
		（サービス収入）	400		120,000

取引　エントリ　借方　貸方

図 11 取引処理の入力データの基本単位
Fig. 11 Basic units in and input data of accounting transaction.

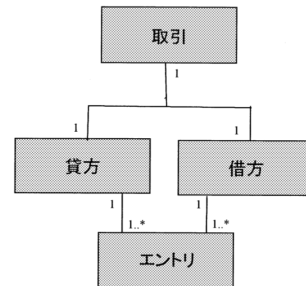

図 12 基本単位のクラス図（1）
Fig. 12 A class diagram of basic units (1).

A.1.2 方法論に基づく作業の説明

(1) フェーズ 1

ステップ 1)

① 取引処理の入力データの中には，階層状に組み合わされる 4 種類の基本単位，すなわち，「取引」，「貸方」，「借方」，「エントリ」が存在している（**図 11** 参照）．

② この基本単位をクラスとしてクラス図を作成すると**図 12** のようになる．

取引処理は，各クラスにそれに応じた処理を次のように割り当て，それらを連鎖させることで実現できる．すなわち，「取引」には，1 つのトランザクションとして，内部に保持している「貸方」と「借方」のオブジェクトそれぞれに勘定への記入を要求する処理を，「貸方」や「借方」には，内部に保持している複数の「エントリ」の 1 つ 1 つについて勘定への記入を要求する処理を，「エントリ」には，該当する勘定を選択する処理や，内部のデータを勘定へ記入する処理や，金額以外のデータの変換を行う処理をそれぞれ割り当てる（**表 5** 参照）．

そして，これらを**図 13** のように連鎖させることで，

表 5 取引処理の基本単位への割当て
Table 5 Allocation of accounting transaction processes to each of basic units.

基本単位	機能
取引	貸方と借方への記入要求
貸方	個々のエントリへの記入要求
借方	個々のエントリへの記入要求
エントリ	データの変換 勘定の選択 勘定への記入

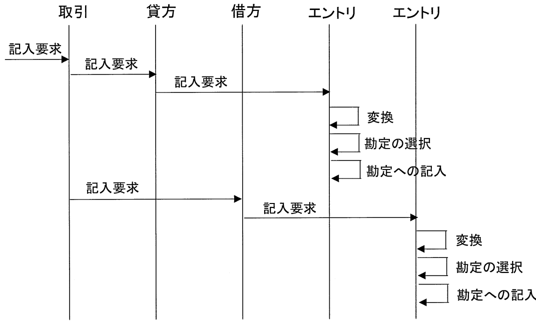

図 13 取引処理のシーケンス図
Fig. 13 A message sequence diagram for accounting transaction.

取引処理を実現する．

ステップ 2)

① 一方，ステップ 1) の基本単位に割り当てられた処理を，処理だけで独立させて，それらを代わりに基本単位として抽出してみる．その場合，処理の基本単位としては，次の 5 種類が存在する．

- 入力として与えられる取引データを貸方データと借方データに分割する処理
- 入力として与えられる貸方データ，あるいは，借方データを，構成要素である 1 つ 1 つのエントリデータに分解する処理
- 入力として与えられるエントリデータの内容を変換する処理
- 入力として与えられるエントリデータを，その内容に基づき分類する処理
- 入力として与えられるエントリデータを，特定の勘定へ記入する処理

以後，それぞれを「分割」，「分解」，「変換」，「分類」，「記入」と呼ぶことにする．そして，これらの基本単位を**図 14** のように組み合わせて，取引処理を構成する．

② これらの基本単位の組合せの順序は，おおむね「分割」→「分解」→「変換」→「分類」→「記入」という固定的なものになる．したがって，クラス図を作

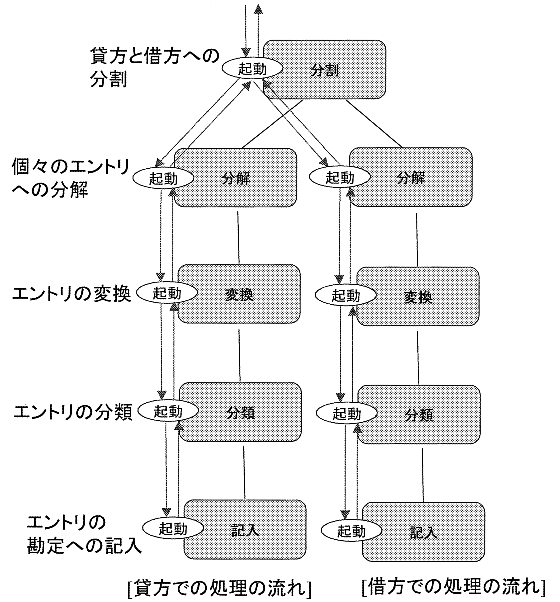

図 14 処理の基本単位による取引処理の構成
Fig. 14 Accounting transaction composed by basic units of process.

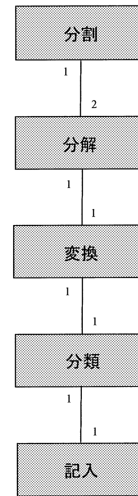

図 15 基本単位のクラス図 (2)
Fig. 15 A class diagram of basic units (2).

成すると**図 15** のようになる．

③ ステップ 1)，あるいは，ステップ 2) で抽出された基本単位は，いずれも次の条件を満たしている．

(1) 基本単位は，ルートを 1 つ持つ DAG 状の構成で組み合わされる．

(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．

したがって，会計業務の取引処理は，本方法論の適

用範囲内にあると考える．

(2) チェックポイント

ここでは，例として，ステップ 2) の結果に対してチェックポイントでの汎用化の作業，および，[判断基準] を適用してみる．

基本単位の中には，入力データに対する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，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」，「分解」，「変換」，「分類」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．したがって，次のようになる．

- 入力として与えられるデータを分割して，複数の子ノードに振り分ける処理
- 入力として与えられる集合データを，1つ1つの構成要素に分解して，非集合データと結合して子ノードに渡す処理
- 入力として与えられるデータの内容を変換して子ノードに渡す処理
- 入力として与えられるデータに基づき，複数の子ノードの中から次の処理を行うものを選択して，それにデータを渡す処理

それに加えて，適用性をさらに高めるために，前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合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，勘定に対して「記入」の処理を行っていたが，勘定に限らず，データベースをターゲットとして汎用化する方向でとらえた場合，「記入」に相当する処理には，SQL 文の作成やデータベースに対するコネクションを通してそれを実行するといった定型的な処理が必要になるため，この部分を，「保管」という名前に変え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合せが可能になるように，Composite パターンを適用してクラス図にまとめると，**図 16** のようになる．Composite パターンの適用により，基本単位として，「分割」，「分解」，「変換」，「分類」，「保管」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．

これにより，構成されるオブジェクト構造には十分な

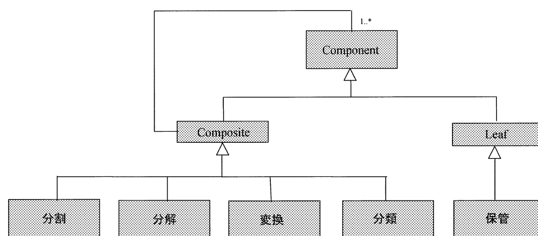

図 16　基本単位のクラス図 (3)
Fig. 16　A class diagram of basic units (3).

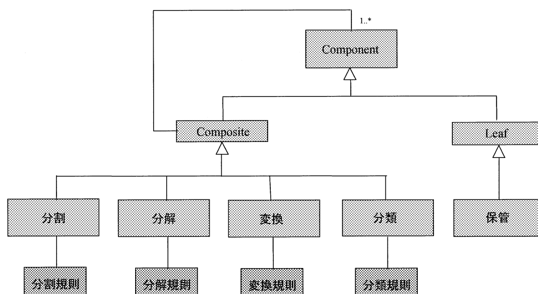

図 17　カスタマイズクラスの追加
Fig. 17　A class diagram including customize classes.

バリエーションが考えられるので，ここで抽出した各基本単位をコンポーネント化する意義があると判断する．

一方，ステップ 1) の抽出結果では，「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，上述のとおり，ステップ 2) で基本単位を抽出した方が，コンポーネントの汎用性も考えやすくなり，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 Composite パターンに基づいて次のように統一的に定める．

void post (Hashtable aTransaction)

引数の aTransaction は，データベースに登録するデータの構成の違いを，Hashtable を使って柔軟に実現できるようにしている．Hashtable の値は，文字列や数値のような基本型のオブジェクトのほかに，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ホットスポットを特定する．たとえば，次のように考えられる．

「分割」 aTransaction 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

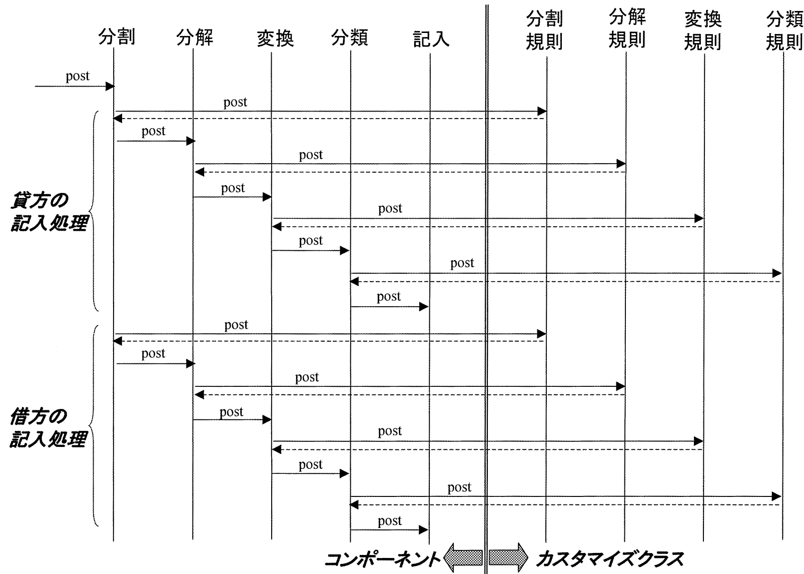

図 18 取引処理のシーケンス図
Fig. 18 A message sequence for accounting transaction.

る部分

「分解」 aTransaction 内部の集合データを，1 つ 1 つの構成要素に分解して，非集合データと結合する部分

「変換」 aTransaction 内部のデータをどのように変換するのかを決める部分

「分類」 aTransaction 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

これにより，アプリケーションごとに異なるデータの構成に特化させる形で，上記のホットスポット部分をカスタマイズクラスとして実装できるようにする☆．これらのカスタマイズクラスを，それぞれ「分割規則」，「分解規則」，「変換規則」，「分類規則」と呼ぶことにする．そして，そこでのインタフェースを次のように定義する．

分割規則：Hashtable post (Hashtable aTransaction)
分解規則：Vector post (Hashtable aTransaction)
変換規則：Hashtable post (Hashtable aTransaction)
分類規則：String post (Hashtable aTransaction)

それぞれの引数は，いずれも，「分割」，「分解」，「変換」，「分類」の post メソッド呼び出し時に引数として渡された Hashtable オブジェクトである．それを基にして，分割規則では，子ノードの ID とその子ノードに渡す分割後の Hashtable オブジェクトの組を，分解規則では，分解後の Hashtable オブジェクトの集合を，変換規則では，変換後の Hashtable オブジェクトを，分類規則では，選択した子ノードの ID をそれぞれ返すようにしている．

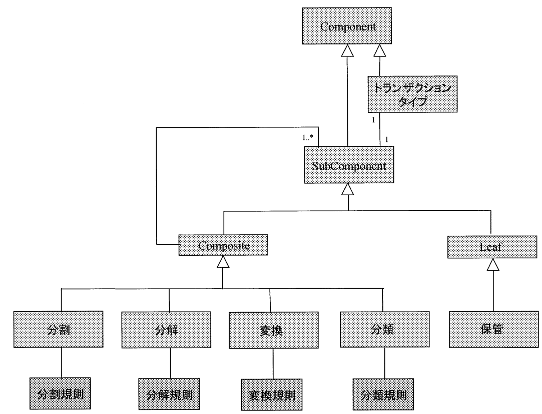

図 19 「トランザクションタイプ」の追加
Fig. 19 A class diagram including TransactionType.

ここで新たに追加されたクラスを含めてクラス図を記述し直すと**図 17** のようになる．

また，会計業務の取引処理は，これらのクラス間を流れる**図 18** のようなメッセージのシーケンスで実現できる．

☆ DAG 状の構成の中で，親から渡されたデータをカスタマイズクラスでの処理結果に従って子に渡す処理は，すべてのコンポーネントにおいて，フローズンスポットとしてコンポーネント内に実装される部分である．

ステップ 4)

コンポーネント間を受け渡しされる Hashtable 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違いをそこに吸収させる．この「トランザクションタイプ」は下位に任意のコンポーネントをつなぐことができ，生成した Hashtable オブジェクトを，下位のコンポーネントの post メソッドを呼び出す際に引数として与える．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，下位につながる処理全体を 1 つのトランザクションとして宣言する意味を持たせることにする．そして，これをコンポーネントの 1 つに加える．これにより，クラス図は**図 19**のようになる．

(4)　フェーズ 3

ステップ 1)

「分割」，「分解」，「変換」，「分類」は，入力された Hashtable 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の一部として呼び出された post メソッドの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の処理は待ち状態にする必要がある．この排他の仕組みは，「保管」に対して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，別途用意されたコネクションプールなどからコネクションを取得して，それを post メソッドの引数である Hashtable オブジェクトを介して「保管」に提供するなどして，「保管」が自身でコネクションを管理することがないようにすれば，「保管」から状態をなくすことも可能であり，ロックのためのパターンを使う必要もなくなる．ここでは，この後者の案を採用することにする．コネクションは，「トランザクションタイプ」で Hashtable オブジェクトを生成する際に，コネクションプールから取得され設定される．

ステップ 2)

枝別れした複数の処理全体を，1 つのトランザクションとしてコミット/ロールバックすべきか否かの判断は，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て制御が「トランザクションタイプ」に戻ってきた際に，コネクション内部で SQLException など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．一方，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する．そして，このスレッドグループに，コーディネータの役割を担わせて，発生した例外なども各スレッドからそこに記録させるようにして，複数のスレッドに分かれて進行している処理全体に対する最終的なコミット/ロールバックの判断と処理を，このスレッドグループに行わせるようにする．ここでは前者のシングルスレッドで実行する案を採用することにする．

(平成 13 年 1 月 1 日受付)
(平成 13 年 8 月 1 日採録)

吉田　和樹（正会員）

1987 年東京工業大学工学部経営工学科卒業．1989 年同大学大学院総合理工学研究科システム科学専攻修士課程修了．同年（株）東芝入社．1995 年〜1997 年米国イリノイ大学アーバナ・シャンペン校客員研究員．現在，SI 技術開発センター所属．主として，オブジェクト指向によるソフトウェア再利用/部品化技術の研究に従事．日本ソフトウェア科学会会員．

本位田真一（正会員）

1976 年早稲田大学理工学部電気工学科卒業．1978 年同大学大学院理工学研究科電気工学専攻修士課程修了．（株）東芝を経て，2000 年より国立情報学研究所教授．2001 年より東京大学大学院情報理工学系研究科教授を併任．工学博士（早稲田大学）．主として，エージェント技術，オブジェクト指向技術の研究に従事．1986 年度情報処理学会論文賞受賞．日本ソフトウェア科学会，IEEE，ACM 各会員．