コンポーネント指向の Java アプリケーション開発技法

Component-based Java Application Development Method

羽 田 昭 裕

要 約 日本ユニシスで開発したコンポーネント指向開発技法に基づく Java アプリケーション開発方法を紹介する．コンポーネント指向の基本は，インタフェースと実装の分離である．この開発技法は，以下の特性を持つ．1）インタフェースとユースケースを対応させることにより，開発プロセスは見通しよく，開発環境の支援を行ないやすい．2）情報隠蔽とカプセル化の単位を区別することで，データを保全しやすく，効率的で柔軟なコンポーネント・モデルを提供している．3）アーキテクチャを絞り込むことで，適用可能な手順を具体的に示している．以上の特性により，Java 言語の能力を活用したアプリケーション開発が実現できる．

Abstract This article provides an overview of for the component-based Java application development method which developed by Nihon Unisys Ltd. The component-based development is targeted forward the separation of interfaces and implementation.

This development method is characterized by the followings: 1) Corresponding interfaces to use-cases facilitates traceacbility of development phases and support activities by development tools, 2) Applying the information hiding and encapsulation separately to different objects provides effective and flexible component model that holds data integrity, and 3) Selecting architecture among the predefined set of architectures presents concrete development procedures. Thus, the application development environment can be derived through capabilities of Java programming language.

1. はじめに

オブジェクト指向開発はコンポーネント指向開発に移行することにより，現実的な開発技法となってきている．日本ユニシスでも，コンポーネント指向の開発技法を定めており，データ保全を不可欠な要件とするアプリケーション開発を主なターゲットとしている．エンタプライズ規模のアプリケーション開発は，データの保全だけでなく，分散・並行開発，プロジェクト管理，要員教育などの課題に応える必要がある．この技法はインタフェースと実装の分離を軸に解決策を示しており，事前定義されたアーキテクチャ別に具体的な内容を用意している．本稿では，Java 言語を利用してサーバサイドのアプリケーションを開発する場合の開発技法を紹介する．

2 章で開発技法を定めるアプローチを紹介し，定めた開発技法の骨格を開発プロセス（3 章）とアーキテクチャに分けて述べる．アーキテクチャは，枠組みとしてのコンポーネント・モデル（4 章）と具体的なアーキテクチャ（5 章）について記述する．このアーキテクチャは Java 言語と親和性が高い．しかし，Java 言語固有の特性もあり随時紹介していく．なお，手順の詳細は割愛する．

2. 開発技法の概要

この開発技法の特徴は，インタフェースを開発の中心に据えていることである．具体的には，開発プロセスにおいてはインタフェース・レベルで追跡可能性（traceability）を保証し，開発環境ではインタフェースを開発単位に対応させ，実行環境においては通信するコンポーネントの組み合わせでアプリケーションを構成する．

この章では，インタフェース中心で開発する理由を述べる．そして，コンポーネントとインタフェース中心開発との関係およびコンポーネントの粒度についての考えを述べる．

2.1 オブジェクト指向開発とコンポーネント指向開発

開発をインタフェース中心でアプローチすることは，オブジェクト指向開発の発展に対する以下のような認識に基づいている．

表 1　コンポーネント指向開発への発展

	第 1 世代 オブジェクト指向開発	第 2 世代 オブジェクト指向開発	コンポーネント指向 開発
実在	構造化分析の OOP 的拡張	同左	同左
仕様	—	—	コンポーネント仕様 （インタフェース）
実装	OOP（クラス）	OOP（タイプ） デザイン・パターン	同左 コンポーネント コンテナ

文献[1]ではオブジェクトの捉え方を実在（essential），仕様（specification），実装（implementation）という 3 種に分類している．実在モデルは，問題領域の概念を表現し，実装上のクラスと対応させる必要はない．仕様モデルは，システム領域をインタフェースで捉える．実装モデルは，実行環境・実装言語に依存する．この分類にしたがって整理すると，次のようになる（表 1）．

構造化分析をオブジェクト指向プログラミングのアイデアで拡張した第 1 世代のオブジェクト指向開発方法論は，仕様レベルでのオブジェクト把握を欠いていたため，本来乖離している実在モデルと実装モデルとの差を埋める工夫を必要とした．

インタフェースと実装の分離を軸とする第 2 世代のオブジェクト指向開発は，コンポーネント指向開発へと発展した．インタフェースと実装の分離は情報隠蔽を実現し，カプセル化のベースとなる．実在レベルでのインタフェースの独立は，一つのオブジェクトが複数のインタフェースを持つことに相当する．例えば，ある人が複数の役割を持つことや，ある商品が複数のサービスを持つことを表わす．

第 2 世代のオブジェクト指向開発は，抽象クラスと具体クラスの分離や，多相性（polymorphism）を活用したデザイン・パターンとして発達した．実装モデルでは，C＋＋の仮想クラスやテンプレートを使用した多相性や動的束縛で実現していた．その後，CORBA や Java の普及により，イディオムであったインタフェースが言語・実行基盤として提供され，仕様レベルのオブジェクトが具体化した．

以上の情報隠蔽や多相性はオブジェクト指向がソフトウェア開発にメリットをもた

らす源泉であり，インタフェースと実装の分離によって実現される．さらに，インタフェース中心に対応した開発方法も提唱され，コンポーネント指向開発へと移行している．

このように，オブジェクト指向が解決すると期待された，変更の局所化や再利用といった課題は，コンポーネント指向開発により現実的になってきている．

2.2 代 用 性

インタフェースと実装の分離によって，同一のインタフェースに対して実装を取り替えられるようになる．この性質を代用性（substitutability）と呼ぶ．コンポーネントを組み立ててシステムを構築する場合，この代用性により，他チームが開発するモジュール，実行環境のサービス，既存のモジュールなどは代用となるコンポーネントを利用して開発を進められる．それゆえ，並行・分散開発が容易になり，結合テストまで運用時の実行環境を必要としない．さらに，一から開発する場合と同様のプロセスで，既存システムの再構築が行なえる．そして，どの段階でもプロトタイプ可能となる．

ところで再利用性は，代用性をもったコンポーネントが複数の文脈で使用されることである．従って，論理的には，インタフェースの再利用は実装の再利用に先行する．このように，インタフェース中心の開発は，コンポーネント指向開発の基礎となる．

2.3 技法として具体化するアプローチ

一般的な開発方法論ではなく，実際の開発にそのまま適用できる具体的な開発技法を提供するために，設計の空間を制限する，というアプローチをとる．オブジェクト指向でのパターンやフレームワーク，ソフトウェア工学での問題フレームの研究は，問題を絞り込むことで，設計の空間を制限できることを示唆している[3][7]．この技法は，実行環境やアプリケーション構造を具体的に定めている．そして，技法にしたがって開発・実行できるように，開発環境を用意している．

この開発技法は，コンポーネント指向であるが，モデルの表現形式として UML（Unified Modeling Language）を用いている．UML は，前節で述べた開発方法の発展に追随している標準であり，多くのツールで採用されているためである．また，インタフェースの内部表現として XML（eXtensible Markup Language）を採用している．XML がプログラムで扱いやすい文書構造を提供している標準だからである．UML の XML への標準的な交換方式（XMI：XML Metadata Interchange）を，開発環境のツール間連携に採用している．また，XML はテストデータの記述にも利用している．

2.4 コンポーネントの粒度

この技法では，コンポーネントを狭義のコンポーネントとコンポーネント・システムに分けて捉えている．コンポーネント・システムのインタフェースは，システムと環境のインタフェースとしている．したがって，その粒度は従来のサブシステム・レベルである．一方，狭義のコンポーネントのインタフェースは，システム内のモジュール間のインタフェースである．OOSE[5]で言うコントロール・オブジェクトやエンティティ・オブジェクトの粒度である．

以後の説明のため，この技法の用語法を簡単に示す．より詳細な定義は 4 章で示す．

まず，狭義のコンポーネントをコンポーネントと呼ぶ．また，インタフェースは，コンポーネント・システムのそれを表わすものとし，コンポーネントのインタフェースはタイプと呼ぶ[*1]．クラス・インスタンスという用語は，コンポーネント内部を実装する言語要素としてのみ使用する．

3. 開発プロセス

この章では，技法の開発プロセスとそれを支援する開発環境について述べる．特に，ユースケース主導開発とインタフェース中心開発を対応させることにより，開発の見通しをよくし，開発環境を構築しやすくしていることを説明する．

3.1 開発プロセスの概要

開発プロセスは，UML に基づく一般的な開発方法と同様，ユースケース主導(use-case driven)・アーキテクチャ中心（architecture-centric）・段階的（incremental）開発となっている．段階的開発を貫くのは，リスクの観点である．

開発プロセスは，ユースケース・モデルとそこで使われる用語が引き出せる程度に要件が固まっているという条件で開始する．この条件まで要件を引き出すプロセスは，開発プロセスとは別に定める．UML においてもプロセスの具体化は，組織・文化・問題領域に依存するものとされている[6]．ビジネス・モデリングなど要件獲得プロセスは，実世界を対象にユーザと共に，実在モデルを構築する．このプロセスは，業界，業種，企業規模などに依存した類型を持つ．これはシステム領域で仕様モデルから出発する開発チームのもつ類型とは別である．このため，プロセスを要件獲得と開発に分割する．

前述のように，限定したアーキテクチャから出発する．このことで，技術的なリスクを減少させ，アーキテクチャ中心の開発プロセス（以下，プロセス）を実行しやすくしている．アーキテクチャは，システム構造・アプリケーション構造・実行環境を含む．また，プロセスを開発業務体系の枠組みに適合させることでプロジェクト管理と連携し，各種のリスクに歯止めをかけている．

インタフェース中心の開発は，ユースケース主導のプロセスを具体化している（図1）．ユースケース・モデルは，ユーザがシステムを使うやり方を意味するが，より一

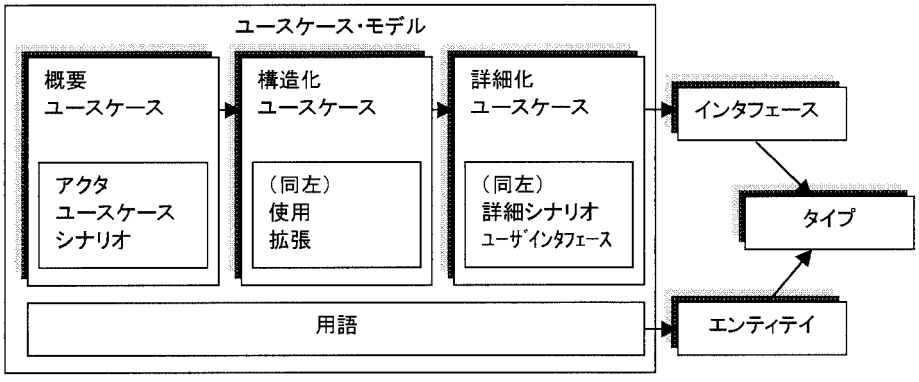

図 1　ユースケース主導のコンポーネント指向開発

般的には環境とシステムの相互作用を表現する．ユースケース・モデルは，環境を代表するアクター（ユーザや他システム）と，システムの機能を表すユースケースを要素とする．まず，ユースケースを構造化し，詳細化し，インタフェースに対応させる．ユースケースの構造化とは使用・拡張関連を用いてグループ化することであり，詳細化とはユースケースに対応するシナリオ，ユーザ・インタフェースを明記することである．使用関連で共通部分を括り出し，拡張関連で例外ケースを記述する．定められたアーキテクチャを加味して，ユースケースをインタフェースに翻訳する手順は，技法に含まれている．インタフェースとエンティティから，技法に従い，コンポーネントのタイプを定める．なお，エンティティの導出は，ユースケースで使われる用語とアクターを利用する．このようにアーキテクチャに埋め込む形でモデルを変換していく．

そして，インタフェース中心のプロセスは，システムの追跡可能性と要求充足性（sufficiency）の検証に貢献する（図 2）．追跡可能性は成果物（とその要素）間の関係づけができることであり，要求充足性はシステムが要求を実現することを検証できることである．まず，ユースケース・モデル―インタフェース―タイプは，仕様モデルレベルで追跡可能である．この階層と粒度は，テストの階層・単位や，配置（deploy）する単位と対応している．そこで，ユースケースに基づいた検証を行なうことにより，要求充足性を確認する．一方，アーキテクチャの実現可能性（feasibility）はプロトタイピングにより検証する．前述したように，インタフェースはどの段階でもプロトタイプ可能である．

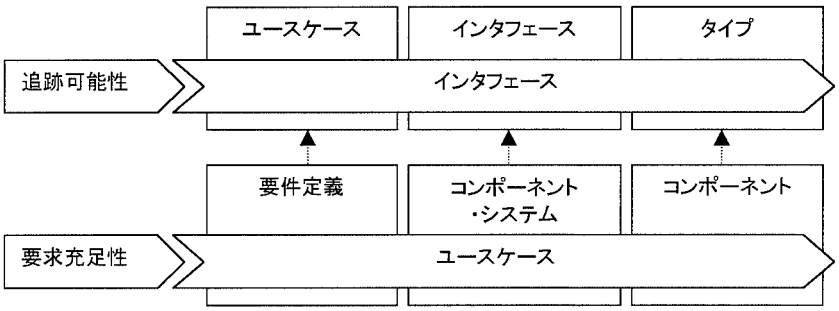

図 2　インタフェース中心の開発プロセス

3.2 開 発 環 境

開発環境の役割は，構築作業の支援と成果物の管理である．

第一の役割である構築作業支援の一つの柱は，コンポーネントやコンポーネント・システムを，システム制御の詳細を記述することなく，設計・実装できるようにすることである．これは，通信や制御の詳細を隠蔽するコンテナを用意することで実現する．そのため，システム制御用のコンポーネント・ライブラリやアプリケーションごとに生成するプログラムを用意する．後者のプログラムは，コンポーネント・システムのプロパティ定義を元に開発環境が生成する．

もう一つの柱は検証を容易にすることである．開発環境はインタフェース定義から

テストプログラムを自動生成する．このことにより，システム制御用のソフトウェア（Web サーバ，RDBMS など）の導入を待たずに，単体テストを行なうことができる．ちなみに，テストの入出力データを XML で表現することにより，テストケース・結果の比較を容易にしている．

第二の役割である成果物の管理の軸は構成管理である．インタフェースと実装が分離されているため，開発単位をインタフェースの単位とすることにより，管理対象が明確になる．開発チーム間ではインタフェースに関する情報のみを交換・共有し，実装に関する情報の利用は開発チーム内に限定する．変更の影響範囲も，実装の変更であれば，自開発チーム内に限定され，インタフェースの変更は追跡可能となる．

4. コンポーネント・モデルの構成要素

この章では，コンポーネント・モデルを明らかにし，アーキテクチャの導入とする．特に，コンポーネントの特性の一つである情報隠蔽とカプセル化について，データの保全性との関係を中心に述べる．このコンポーネント・モデルと Java 言語の関係についても触れる．

4.1 データの保全

この技法が対象としているシステム開発は，データの保全を要件としている．ここで，データの保全性（integrity）とは，システムの整合性が永久に保存されるようなトランザクション処理が行なわれることである．一般に，ビジネス・アプリケーションではエンティティ間に同時に成り立つ静的な関係がある．この静的な制約は，エンティティに相当するオブジェクトの中に納めることはできないので，外的に制約を宣言し，システム制御機構により保証することが必要である．それゆえ，一時的な（transient）データはオブジェクトごとにカプセル化できるが，永続（persistence）データは整合性を保証できるデータベース管理システムに格納する必要がある．

4.2 情報隠蔽とカプセル化

前節で述べたことを，情報隠蔽とカプセル化の観点で整理すると次のようになる．ここでは，構造と振る舞いをグループ化することをカプセル化と呼び，内部データ構造など実装を公開しないことを情報隠蔽と呼ぶ．一時的なデータはクラスでカプセル化し，コンポーネントで情報隠蔽する．永続的なデータはコンポーネントでカプセル化し，コンポーネント・システム内で情報隠蔽する．コンポーネント・システムをオブジェクトと捉えた場合，永続データが属性であり，その状態変更をもたらす操作がユースケースとみなせる．

4.3 コンポーネント・モデル

以上の方針で定めたコンポーネント・モデルを UML に対応させて定義する．UML のクラスは，C++や Java などのオブジェクト指向言語の要素であるクラスに対応していると思われる．この粒度のオブジェクトを最小粒度（fine-grained）のオブジェクトと呼び，コンポーネント・システムを大粒度（large-grained）のオブジェクトと呼ぶ．

コンポーネント・システム：UML のサブシステム（Subsystem）に相当する．仕様レベルで捉えたオブジェクトに相当する．永続データを情報隠蔽する単位であ

る．この方向性は，コンポーネント指向開発方法とも符合する．例えば，代表的なコンポーネント指向開発技法である Catalysis[2]は，ドメイン・モデル，コンポーネント仕様，内部モデルの 3 層に分けており，それぞれ実在・仕様・実装に対応する．Catalysis のコンポーネントとタイプは，本技法のコンポーネント・システムとコンポーネントにほぼ対応する．

代用性を実現するため，コンポーネント・システムはコンテナを必要とする．コンテナはコンポーネント・システム間の通信と，システム制御のサービスを抽象化する．また，アクターに関するデータをはじめとするコンテキスト情報を持つ．

コンポーネント：UML のパッケージ（Package）およびコンポーネント（Component）に対応する．一時的データを情報隠蔽する単位である．永続データは，コンポーネント単位でカプセル化する．

クラス：UML の実装クラス（Implementation Class）に対応する．クラスは，最小粒度のオブジェクトに対応し，コンポーネント内の実装要素である．クラス構造は，コンポーネントの範囲で記述する．クラスにおいて，手続きと一時的なデータをカプセル化する．

データモデル：UML のエンティティ・クラス（Entity Class）のみで構成されたクラス図（Class Diagram）に対応する．データ間の依存関係のみを記述する．コンポーネントとデータモデルのエンティティが同じ粒度であると考えている．アクターを除くエンティティは，コンポーネント・システム内にカプセル化する．データモデルは，関係データモデルを採用する．

4.4 コンポーネント・モデルと Java 言語

このコンポーネント・モデルから Java 言語を捉えると以下のようになる．Java 言語の最大の特長は，実装から独立したインタフェース[*2]を言語レベルでサポートしていることである．具体的には，extends 句と implements 句によりクラスやインタフェースの階層（拡張関係）とインタフェースとクラスの関係づけ（実現関係）を区別している．このことで，コンポーネントの特長である情報隠蔽と多相性が実現しやすい言語となっている．

情報隠蔽は，インタフェースとパッケージによって実現される．まず，インタフェースは，ポインタを廃止し参照によりオブジェクトへアクセスさせることと，操作のみ公開することで実装を隠蔽する．また，デフォルトのアクセス制御を用いることで，クラス内の変数をパッケージ外から参照不可能にできる．カプセル化の単位をクラス，情報隠蔽の単位をパッケージとすることで，安全で効率的なプログラムが可能になる．

また，多相性は一つのクラスが複数のインタフェースを持ち，一つのインタフェースを複数のクラスで使用することで自然に表現できる．インタフェースの階層と実装の階層が分離し，かつ木構造となるので，柔軟でシンプルなプログラム構造を実現できる．

5. アーキテクチャ

この章では，アーキテクチャをシステム構造，アプリケーション構造，実行環境に分けて述べる．ここで紹介するアーキテクチャは，Web を利用したトランザクショ

ン処理を対象にしている．このアーキテクチャについては，報告[8]に詳しい紹介がある．

5.1 システム構造

分散オブジェクトとしてコンポーネント・システムを扱うこと，コンポーネント・システムをシステム制御から分離することが基本構造である．プレゼンテーション・ロジックは，コンポーネント・システムから分離する．開発環境・実行環境で以上の構造を実現する．これはサーバ・アプリケーションの基本構造であり，サーバ・アプリケーションとクライアント・アプリケーションの間はインターネットを基盤とし，プロトコルは HTTP を基本とする（図 3）．他のコンポーネント・システムは環境の一部として捉える．ユースケース・モデル上，そのようなコンポーネント・システムはアクターとして扱う．

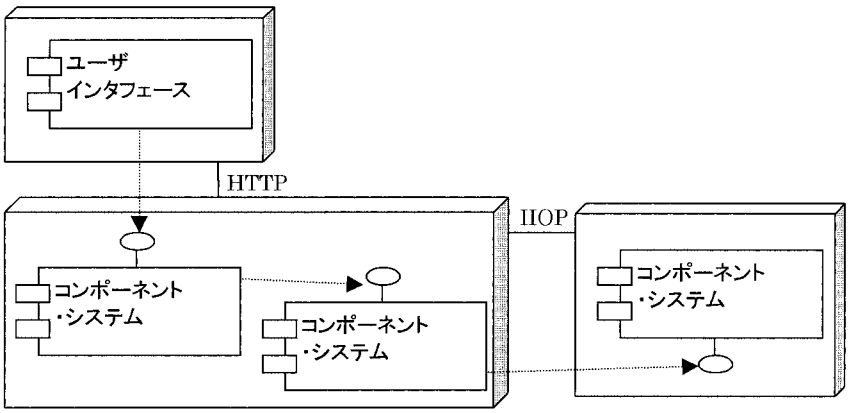

図 3　システム構造（配置図）

5.2 アプリケーション構造

アプリケーション構造とは，コンポーネント・システム内の構造である．アプリケーション構造は，ステレオタイプ間の関係として表現する．ステレオタイプは，タイプの類型である．ステレオタイプは基本的に OOSE[5]と同様である．エンティティとユースケースに対応するステレオタイプを軸とし，データベースなど制御システムとの関係で幾つかのステレオタイプを定義している．

5.3 実　行　環　境

この技法では，大粒度のオブジェクトはどのように実装してもよい．そのため，プログラミング言語透過性と異機種分散環境対応が必要である．そこで，分散オブジェクト環境として CORBA を採用している．また，トランザクション処理や負荷分散は CORBA ベースの OTM（Object transaction manager）を利用する．

コンポーネントは，Java 言語のパッケージに対応させている．従って，インタフェースとタイプの指標は，それぞれ OMG IDL と Java interface で表現する．

6. お　わ　り　に

以上見てきたように，インタフェースと実装の分離を軸にした開発技法は，そのア

ーキテクチャに支えられている．アプリケーション構造は、ユースケースとインタフェースを対応させた大粒度のオブジェクトを想定し，システム構造はアプリケーションの独立を目指している．

大粒度でオブジェクトを捉えることの有効性は，Java で企業レベル・アプリケーションを開発した経験の報告[4]にも見られる．また，Enterprise Java Beans やアプリケーション・サーバのサービスなど，新しいアーキテクチャが実装されてきている．しかし，これらもビジネス・アプリケーションを独立させるには不十分である．コンポーネント指向で開発するための技法は今も進化中であり，開発プロセス，アーキテクチャともに改善すべき事柄がある．事例からの経験をフィードバックしつつ，技法の改善を継続していく．

＊1　属性を含まないタイプをインタフェースと呼ぶ UML[6]の用語法とは異なる．また，インタフェースは操作の指標（signature）と作用/副作用を含むものとし，指標のみを指す場合は注記する．操作の指標とは，操作名および引数・返値・例外の型と引数名である．

＊2　この節では Java 言語の interface をインタフェースと呼ぶ．

参考文献

［1］ S. Cook and J Danniels, Designing Object Systems : Object-Oriented Modeling with Syntropy, Prentice Hall, 1994.

［2］ D D 'Sourza and A C. Wills, Catalysis : Objects, Framework and Components with UML, Addison-Wesley, 1998.

［3］ 羽田, オブジェクト指向によるモデリング, 技報 60 号, 1999.

［4］ G C. Dennis and J R. Rubin, Mission-Critical Java Project Management, 1999.

［5］ I. Jacobson et al, Object-Oriented Software Engineering : A Use-Case-Driven Approach, Addison-Wesley, 1993.

［6］ OMG Unified Modeling Language Specification, Object Management Group, 1998.

［7］ F. Buschmann et al, A System of Patterns, Wiley, 1996.

［8］ 溝上, Java/CORBA アプリケーション開発環境の実現, 技報 63 号, 1999.

執筆者紹介

羽　田　昭　裕（Akihiro Hada）

1984 年一橋大学卒業．同年日本ユニシス(株)入社．意思決定支援ソフトウェアの開発・適用に従事．その後，業務システムとその基盤の要求分析・開発に従事．現在，生産技術部情報技術室に所属．情報処理学会会員．