(7) ソフトウェア設計の変遷と再利用

奈良先端科学技術大学院大学・情報科学研究科／情報科学センター
飯田元

はじめに

本稿では，ソフトウェアを取り巻く状況の変化に対応して変遷を重ねてきたソフトウェア設計手法と，ソフトウェア設計における再利用技術について，商用のアプリケーションソフトウェア開発等で広く用いられてきた設計手法を中心に解説する．

1. ソフトウェア設計とは

ソフトウェアとは，広い意味においては，コンピュータシステム(ハードウェア)を動作させるために必要なプログラムコード，及び，ドキュメント(文書)やデータなどを指す．また,狭い意味ではプログラムコード，つまり，ハードウェアを動作させるために必要な命令とデータ定義を記述したものを指す．ソフトウェアと対立する概念としてハードウェアという言葉も用いられるが，この場合のハードウェアとは，機械部品や，コンピュータのメモリや CPU(中央演算装置)などのシリコンチップなど，容易に改変ができないシステムの物理的な実体を指す．近年ではソフトとハードの境目自体があいまいになってきており，たとえば，VLSIやCPUの多くは一種のプログラミング言語を用いて設計されている．また，プログラムを書き換えるように何度でも構成を書き換えて動作を変更することの出来るチップもある．

ソフトウェアがソフトウェアと呼ばれる所以は，その柔軟さにある．ハードウェアは容易に変更ができなくても，その中で動作するソフトウェアを変更することで，さまざまにシステムの機能を変更したり追加したりすることが可能である．現在のいわゆるフォンノイマン型と呼ばれる電子計算機は，プログラムをメモリ(記憶装置)に格納してから実行する形式をとっているため、メモリに格納するプログラム変更することで，さまざまな用途に応じた仕事をさせることができる．

1.1. ソフトウェアの特質

ソフトウェアは他の工業製品と比べて，開発や製造の管理が極めて難しいといわれ続けているが，その理由は，ソフトウェアの持つ以下のような特質にある:

1) **無形である**: ソフトウェアは本質的にはビットパターンによってあらわされたロジックの集合体であるが，ハードウェアや格納メディアと独立には存在できないため，プロダクトとしての境界が不明瞭で，扱いが難しい．さらに，無形であるがゆえに品質を表現することは非常に困難であり，不具合の特定も容易ではない．
2) **改変が容易である**: ソフトウェアの内容を単に「改変」することはきわめて容易であるが，正しく「変更」(あるいは「改訂」)することは容易ではない．ユーザが，変更が容易であることと混同しているために，開発の終盤になってから仕様の変更を要求して，プロジェクトを混乱に陥れることは珍しくない．また，開発者側も，リリース後の修正が容易であると思いがちで，完成度の低い製品をとりあえず納入するといったことを行いかねない．
3) **製品分野として未成熟である**: 1)とあわせて，商品としての自立がしばしば困難である．ハードウェアに従属するため，しばしばハードウェアの変更に応じて修正を余儀なくされる．また，ソフトウェアの生産と消費のためのビジネスモデルやカルチャーが十分に形成されておらず，機能や品質，対価などにおける開発側・顧客側双方の合意形成が困難である．

1.2. ソフトウェア設計の特徴

ソフトウェア開発の見通しの悪さを排除しようと，さまざまな工学的アプローチの適用が試みられてきた．まず，系統的開発を行うための基本的手段として，開発作業を複数の工程に分割して進捗や品質の管理が行われている．企画から納入・保守までの一連の流れは「ソフトウェア・ライフサイクル」としてモデル化されている．ソフトウェア・ライフサイクルの参照モデルはいわゆる SLCP (Software Life-Cycle Process) モデルとしてISOによる標準化も行われている(ISO12207)が，大まかには，図1のような流れをたどる．

このような単一方向に一度だけ実行される手順の流れはウォーターフォール型と呼ばれているが，現実にはやり直しのための反復が行われる．また，リスク回避のために作業目標を限定して反復的に作業を行う方式の工程管理へ主流となりつつある．

日本機械学会〔No.04-97〕設計工学シンポジウム「関係性のデザイン：つくることから育てることへ」講演論文集〔04.12.14,15,東京〕

図1. ソフトウェア・ライフサイクルの概要

ソフトウェア分野では古くから建築分野の手法を手本としてきた部分が多く,「アーキテクチャ」など多くの用語や概念が建築分野から輸入されている. しかし, 現状では, いまだにピラミッド建設のようなデスマーチ形式[1]での開発プロジェクトも珍しくなく, 開発プロセスの評価・改善の取組みも大きな課題である.

ソフトウェアにおける設計とは, 狭い意味では, 外部設計(モジュールへの分割とインタフェースの定義)と内部設計(データ構造やモジュール内ロジックの定義)の2工程を指すが, 一つ前の要求分析工程(顧客からの要求獲得と分析, システム仕様の定義)も含めて議論されることが多い.

なお, 最近のソフトウェアの多くは, グラフィカルユーザーインタフェース(GUI)を備えており, ウィンドウの色やフォント, ボタンの配置やメニューの構造といった, Look and Feel に関わる部分での設計の重要性が増大しているが, GUI 設計については本稿では扱わない.

ソフトウェア設計は 1.1 で挙げたプロダクト自体の特性に由来して, 以下のような特徴を持つ:

1) **工程の定義や境目が必ずしも明確ではない**: しばしば, 設計途中に不明な要求事項の存在が明らかになったり, 設計と称してほとんどプログラミングに等しい作業に立ち入っていたりという事例に事欠かない. また, 小さな組織では工程や設計手法が整備されていないことも多い.
2) **設計内容の妥当性や正当性を検証する手段に乏しい**: ソフトウェアの仕様書や設計書は一般の顧客に馴染みのない記法が多く用いられているために, 顧客が内容を理解できず, 自分の要求が満たされているかの判断や, 仕様と設計との整合性の検証が困難である. また建築物のように, 強度などの重要な品質を満たすための標準規格が整備されておらず, 稼動してみるまでパフォーマンスや使いやすさなどの品質が要求を充足するか予測できないことが多い.
3) **様々な設計手法が存在する**: 実現手法の自由度が高いため, デファクトスタンダードとなっている手法が存在する一方で我流の方法論も数多く存在する. また, 設計情報を書き記す方法も統一されていない.

2. ソフトウェア設計の変遷

本節では, ハードウェアの進歩や社会的要求の変化に応じてフトウェアシステムが辿ってきた変化の道のりに沿ってソフトウェア設計手法の変遷をたどることで, ソフトウェア設計における課題を俯瞰する.

2.1. 黎明期～アルゴリズムに基づく数式処理プログラムの開発

そもそも計算機は弾道計算や暗号解析, 気象予測などを目的として開発された. プログラムはいくつかのパラメータを入力値として読み込み, それに基づいて方程式の解を求めるなどの処理を行う. 従って, プログラム P は入力値の集合 I と出力値の集合 O との間の数学的あるいは論理的対応関係を表すものとして抽象化される. プログラムの入力とは, 計算を開始するに当たっての初期パラメータであり, 出力とはPの実装によって定義される関数値 $f(i) \mid i \in I$ の計算結果である. 関数 f の定義に従った出力値を十分に実用的な精度と時間・空間コストのもとに計算する手順(=アルゴリズム)を考案・実装することが, プログラム設計の大部分を占めていた.

FORTRAN(1957)はこのような用途に設計されたプログラミング言語であり(その名前も Formula Translation に由来している), いくつかの改訂を経て現在でも科学技術計算用途に用いられている.

数学的に定義された要求に従った計算を, 十分な精度を保証して限られたメモリ空間と計算時間の範囲内で完了させることが, この時代におけるプログラム設計の課題であったと言えるが, 本稿で特に取り上げるべき系統的なプログラミング法・設計法は存在しなかったといってよいであろう.

2.2. 構造化の始まり～構造化プログラミングによるバッチ処理プログラムの開発

1950 年代前半にはすでに事務処理用計算機が IBM によって開発され, 以来, 計算機は軍事目的や科学技術計算などごく限られた用途だけではなく, 事務データの処理などより広い用途に対して用いられるようになる. 事務処理計算においては, 従来のような複雑な計算ではなく, 蓄積された大量なデータに対して一定の処理を繰り返し施すことが計算機利用の主

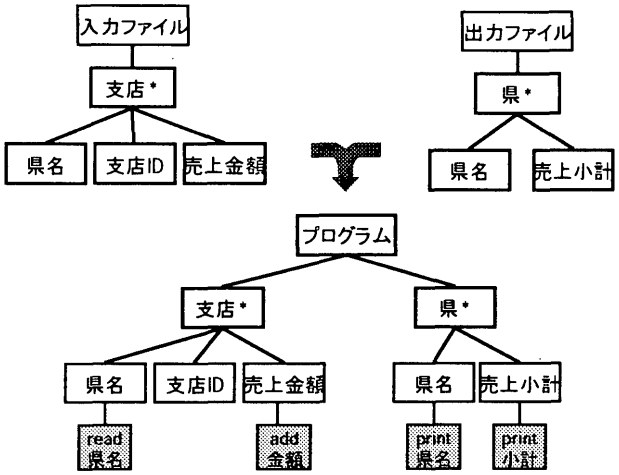

図2. ジャクソン法：入出力ファイルの構造木の合成によるプログラム構造木の導出

たる目的となる. このように入力となるデータを蓄積して一度に計算を行う方法はバッチ処理と呼ばれる.

バッチ処理型プログラムの入出力データは, ばらばらの数値ではなく, 多数のデータの連なりとして扱われるが, このようなデータの連続体を格納したものを「ファイル」と呼ぶようになる.

バッチ処理型のプログラム P は, 何らかの繰り返しや順序関係を含んだ構造を持つ入力ファイル F_I を, 同様に何らかの繰り返しや順序関係を含んだ構造を持つ出力ファイル F_O へと変換する順序処理として抽象化される. 典型的なプログラムの動作は入力ファイル中のレコードを読み込み, 処理を施し, 結果を書き出すという手順を繰り返すものである.

具体的な手順は, 入出力ファイルのデータ構造に大きく依存するため, 入出力ファイルの構造に着目して系統的にプログラムの構造を求める方法が編み出された. 代表的なものとしては,ワーニエ法やジャクソン法(JSP:Jackson Structured Programming)などがある.

ここでは例としてジャクソン法を簡単に紹介する. ジャクソン法は入出力ファイル構造を図2に示すようなツリー型のダイアグラムによってわかりやすく表現し, それをもとにプログラムの基本構造を求める手法である. プログラムの構造もファイル構造同様にツリー型ダイアグラムとして表現される. ここで用いられている図法はジャクソン木と呼ばれ, ソフトウェア設計情報を表現するための図式言語としては(フローチャートの系譜を除けば)最も古いもののひとつである.

2.3. 構造化の発展～構造化分析・設計による大規模オンライン処理システムの開発

2.3.1. システムの大規模化と分割統治による対応

ソフトウェアの規模が次第に拡大し, 一つのソフトウェアに様々な機能が含まれるようになると, 見かけ上ひとつである入力ファイルにも, 実際には異なる機能のための様々な種類のデータが混在して含まれるようになる. このような単純な大規模化に対しては, データ入出力のための主プログラムと各機能を実現するための複数の副プログラムを設け, 主プログラムにおいてデータ入力し, それを各副プログラムに振り分けを行い, それらからの出力を再統合する単純な分割統治が可能である.

2.3.2. オンラインシステムによる入出力分散とデータフロー分析

計算機の低価格化・小型化が進んだ結果, 様々な場所に端末を配置してそれらの入力を対話的に処理する「オンラインシステム」が登場する. たとえば, 日本における初のオンラインシステムとしては, 全国に配置されたオンライン端末を通じてリアルタイムに座席の予約を行うことのできる国鉄の座席予約システム MARS101 が 1963 年に早くも納入・稼動している. このようなシステム形態は一般にオンライン・トランザクション処理と呼ばれている(以降ではオンライン処理と略記).

オンライン処理型プログラムとバッチ処理型プログラムの最大の相違点は, リアルタイム性よりも, むしろ, 入出力処理の物理的位置や時間順序が一箇所に特定されないと言う点である. このため, システムに対してどのような入力データと出力データが存在し, それらの間のデータ変換処理として何が必要かを明確にする「システム分析」という作業を行った上で, 各処理を階層的モジュールとして設計を行うという手法がとられるようになった.

このような手法を一般に構造化分析・設計法, あるいは構造化手法と呼ぶ. システム分析とシステム設計とを別工程として明示的に区別するようになったこともひとつの特徴である. 構造化手法にはいくつものバリエーションがあるが, 代表的なものとしては DeMarco による「構造化分析法」[7]や Mayers の「複合設計法」[8]などがある.

構造化分析においては DFD (Data Flow Diagram)と呼ばれる図を用いて, システム内に存在する基本的なデータ変換処理(プロセス)とそれらの間でやり取りされるデータの流れ(データフロー)が記

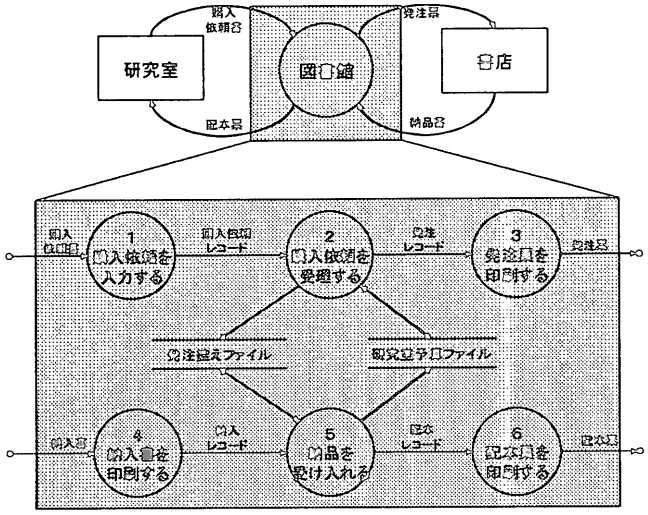

図3. DFDを用いた構造化分析の例

述される. 図3は大学の研究室が図書館を通じて書籍購入を行う業務の分析例である. 一番上の DFD はコンテキストダイアグラムと呼ばれ, システム全体を現すプロセスと, そこへの入出力を表している.「図書館」プロセスは複雑な処理を内包しているので, さらに細かい DFD へと階層的に分解される(図 3 下側). 各データフローはバッチ処理型プログラムの入出力ファイルと同様の考え方でその構造を分析し, データ辞書と呼ばれる形式で整理・記述される. 分割が繰り返された結果として得られる最下層のプロセスは, 入力データから出力データへのアトミックな変換処理とみなされ, その処理内容は擬似プログラミング言語などの手段を用いて「ミニ仕様書」としてドキュメント化される.

設計段階では, このようにして得られた分析モデルをもとに階層的なモジュール構成を求める. 各モジュールにはひとつ以上のプロセスが含まれる. モジュール間の独立性を高めるために, どのモジュールにどのプロセスを含めればよいかを決定することが設計における重要な課題となる. モジュール分割法にはデータ入力部と変換部, 出力部の 3 層に分割する STS 分割法や, トランザクションごとにプロセスをグループ化し, 主モジュールにディスパッチャを置く TR 法などがある.

2.4. 構造化の限界～オブジェクト指向による分散処理システムの開発

2.4.1. 構造化手法の限界

構造化手法によるシステム設計は, 一定の成功を収め, 70 年代から今日に至るまで, 多くのソフトウェア開発に適用され, 大規模で安定した構造のシステムを多人数で分担して構築するうえで, 自然でかつ有効な手法であることが示された. 計算機システムが高価で特別な設備であった時代には, システムの構成は一度決定されると長期間にわたり変更されることなく運用されてきたためである.

しかし, ハードウェアの高速化や低価格化, インターネットの普及などにつれ, ソフトウェア需要の増加と短命化が進み, 徐々に, 構造化手法の弱点が明らかになりはじめた. オンラインシステムが汎用の PC とインターネットを用いて構築されるようになると, システムの構成が容易に変更可能となり, ソフトウェアもそれに対応することが求められるようになったからである.

柔軟性の低さと再利用性の低さは, 構造化手法の弱点として従来から指摘されていた. たとえば, 構造化手法による設計では, 共有された大域変数やその構造定義を仲介としたモジュール間の依存関係(データ結合, スタンプ結合などと呼ばれる)が多数存在する. このため, データの構造を変更する場合にインタフェースの整合性やデータ操作の正当性が破壊されないように注意を払う必要がある. つまり, モジュール同士が互いに依存する度合いが高いため, 特定部分の変更に対する波及を分析し修正するコストも高くなる.

同様に, 外部に共通変数の存在を仮定するようなモジュールは, 見かけ上モジュール化されていても, 単体での再利用が困難である. また, 機能や内部アルゴリズムはほぼ同一にもかかわらず, 扱うデータ型が異なるために再利用できず, 良く似たモジュールが多数作成されることも多い.

これらの問題は、モジュール間の呼び出し関係とその間のインタフェース(モジュール間でのデータ受け渡し方法やデータ構造)を外部設計段階で決定・固定するという、構造化手法の本質的な仮定に起因している. また, システムの分散化がさらに進んだことにより, 分析段階においてデータフローを的確に洗い出すことが非常に困難となってきたことも大きな要因である.

2.4.2. オブジェクト指向の登場

オブジェクト指向の概念はシミュレーション用言語 Simul67(1967)にその源流があるといわれているが, オブジェクト指向言語 Smalltalk80 とその実行環境により広く一般に知られることとなった.

オブジェクト指向では, まず, データや手続き, 関数といった従来の手続き型プログラムにおける構成要素に対して, 関連の深いもの同士を「抽象データ型」としてひとつにカプセル化したものをオブジェクトと呼び, プログラムの基本構成要素として用いる. プログラ

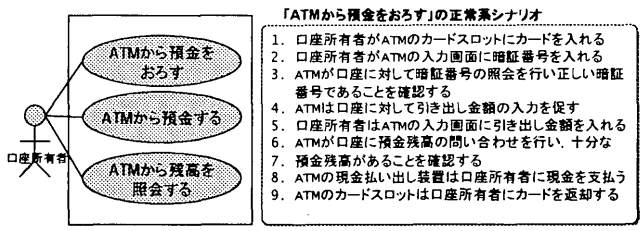

図 4-a UML のユースケース図

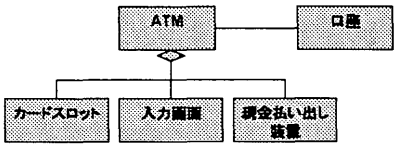

図 4-b UML のクラス図

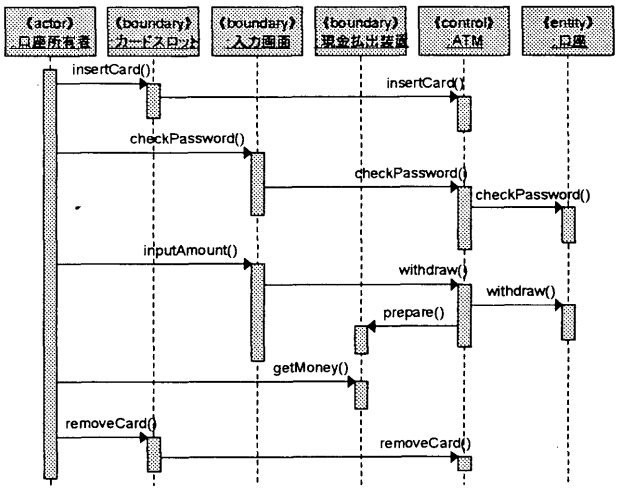

図 4-c UML のシーケンス図

ムの構造は，構造化アプローチにおいては何重にも静的に階層化されていたものから，オブジェクトの平盤な集合へと変化し，オブジェクト間のインタフェースはオブジェクトの持つ関数への対等な呼び出し関係で実現されるようになった．

さらに，

扱うデータ型のみが異なる複数の関数には同じ関数名をつけられるようにしたり，類似するオブジェクトはクラスとして抽象化し，さらに，あるクラスを部分的に変更して新しいクラスを定義する仕組みを用意したりすることで，概念の階層的整理や再利用を可能としている．

2.4.3. オブジェクト指向を用いた分析・設計手法

初期のオブジェクト指向の研究では，これらの概念をプログラミング言語という枠組みにいかに結実させるかについて注力がなされていたが，システム分析や設計への応用が次第に進み，さまざまなオブジェクト指向開発法が提案された．

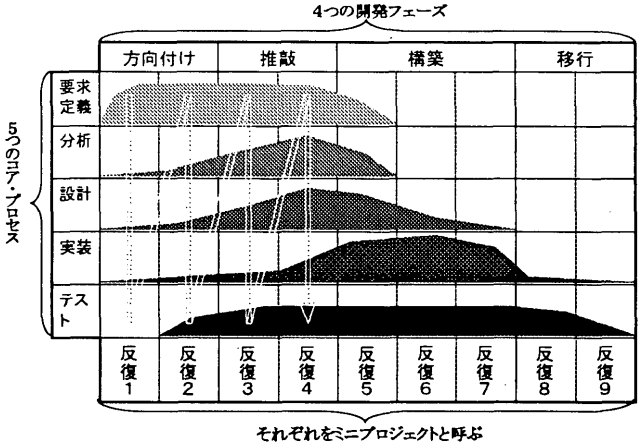

図5. Rational Unified Process における反復プロセス

オブジェクト指向開発法とはオブジェクト指向言語の利用を意味するのではなく，オブジェクト指向を概念整理のためのガイドラインとして活用することを意味している．このことによってたとえば，クラスの階層構造や関連づけを簡潔な概念整理に役立てることが出来る．

2.4.4. 統一プロセスとモデリング言語 UML

これまでに提案されたオブジェクト指向開発法のなかでも，Booch による Booch 法，Rambaugh による OMT(Object Modeling Technique)法，Yacobson による OOSE（Object Oriented Software Engineering）法は，それぞれ，実装，モデリング，要求分析といった部分において優れたものであった．後に，三者が Rational 社に集結したことで，これらを統一した方法論の開発が進められた．結局，方法論のうち，モデルの記述言語の部分のみが切り離されて UML(Unified Modeling Language)として OMG (Object Management Group：オブジェクト指向関連技術の標準化・推進団体）による標準となった．UML では，問題に対する複数の視点を提供し，それぞれの視点に応じたシステム記述を行うためのダイアグラムを 9 つ(UML1.5 の場合．2004 年にリリースされた UML2.0 ではさらにダイアグラムが追加されている）定義している．複数視点の提供は OMT において静的ビュー，動的ビュー，機能的ビューの 3 視点が定義されていたものが元となっているが，UML2.0 においては，14 種類の図が定義され，構造と振る舞いの2つの大視点に分類されている．図4は UML によるダイアグラムの一部である．

一方，UML から切り離された方法論自体は RUP(Rational Unified Process)として Rational 社（現在は IBM の一部門）からリリースされるに至った．名称に方法論（Method）ではなくプロセスという言葉

が用いられていることが示すとおり，RUP では，オブジェクト指向的考え方の適用方法だけでなく，UML を用いた様々なドキュメントの作成手順や，反復を含んだ工程の定義など，詳細なガイドラインが定義されている．図5は RUP における反復的プロセスを概念的に表したものである．この図が示すように，分析からテストまでの各工程に対する重心を4つのフェーズ(方向付け，推敲，構築，移行)を通じて上流から下流へと変化させつつ実行する．

RUP を含め，多くのオブジェクト指向設計法では，システム全体のモジュール構造をクラス関連図などから導くための具体的手法は示していない(若干の指針は示されている)．その代わりに，システム構成の概要を示すモデルであるアーキテクチャを積極的に活用することを勧めている．アーキテクチャについては4節でさらに解説する．

2.5. オブジェクト指向を超えて

オブジェクト指向はソフトウェア設計法の主流となりつつあるが，それまでの方法論が完全に駆逐されたわけではない．構造化アプローチに基づく開発を今日でも継続している組織はまだまだ多く，また，ハードウェア組み込みソフトウェアにおいては職人芸的な開発がいまだに続けられている分野も数多く残っており，近年ようやく組み込み分野に特化した設計開発法の研究と普及が進みつつある．

一方，オブジェクト指向設計自体の問題点や限界についても様々な指摘がなされている．たとえば，以下のような問題点が指摘されている：

- クラスによる概念や実装の再利用は従来手法に比べれば再利用性は格段に向上しているが，それでも，利用のための前提が完全になくなったわけではない(たとえばあるクラスの利用には別のクラスが必須であることがある)．
- オブジェクト指向は分散システムの開発に向いてはいるが，それで分散システムの設計でのすべての課題が解決されるわけではない(分散オブジェクト技術を用いるとしても)．
- 平板なオブジェクト同士の集合内のメッセージパッシングのみで構成されたシステムでは，制御のフローが直感的にわかりにくく，解析も容易ではない．このため，不具合箇所の特定が困難になる(これはオブジェクト指向の登場時から指摘されていることではある)．
- すべての機能がオブジェクト単位で切り分け可能とは限らないため，複数のオブジェクト間にまたがって構成される機能の追加や修正に際しては，複数モジュール(=オブジェクト)に対する変更が要求される(これは構造化アプローチ以来持ち越されている問題でもある)．

これらの問題点を包括的に解決できるような「画期的に新しい」設計手法は，実用レベルでは登場していないが，個別の問題に関してはオブジェクト指向を踏まえたうえで様々な手法が提案されている．以下ではそのいくつかを紹介する．

2.5.1. コンポーネント指向開発

ソフトウェアを機能単位で部品に分解し、得られたソフトウェア部品を組み合わせることによって開発を進める方法をコンポーネント指向ソフトウェア開発と呼ぶ．同開発方法は、クラスライブラリやフレームワークに代表されるオブジェクト指向開発技術が効果的な再利用を実現できなかった反省に基づき、それらの研究を継承・発展させる形で登場してきた．コンポーネントという用語はソースコードからコンパイル済みのバイナリ形式でそのままシステムに動的に組み込んで利用可能なソフトウェア部品を指している．コンポーネントを利用するためには，コンポーネントが動作できるための基盤機構を予め採用する必要があるため，特定のベンダに依存するという問題があり，一般に広く普及してはいない．コンポーネントについては 4 節でも紹介する．

2.5.2. サービス指向アーキテクチャ(SOA: Service-Oriented Architecture)

今日の分散化されたシステムでは，必ずしもすべての機能がひとつの組織内で実現される必要がなくなってきており，処理のアウトソーシング化が推進されつつある．サービス指向アーキテクチャ(SOA)は，細かいオブジェクト個々ではなく，それらの組み合わせの結果として外部に提供される「サービス」に着目する考え方である．

オブジェクト指向では，データの定義と操作をカプセル化して隠蔽することでモジュール間インタフェースに一定の柔軟性を持たせ，モジュール間の独立性を高めることに成功した．しかし，どのオブジェクトがどのような操作を持つか，つまり，どのような機能が外部に対して提供されるかは，利用する側で予め十分に理解しておく必要がある．

SOA では，オブジェクトの実装方法を(言語を含めて)完全に隠蔽し，インタフェース(＝提供するサービスの定義情報)をふくめたリクエストのやり取りを行うため，あらかじめ利用するサービスの内容やサービス提供者の位置を予め決定する必要がなく，動的で柔軟

なシステム構築が可能である.

典型的なSOAシステムとしてWebサービス型のシステムがある. Web サービスでは, WWW 用のネットワークプロトコルHTTPやデータ記述規格XML等を組み合わせた方法で異なるサイト間でのサービスの提供・利用を実現している. SOAを実現するための基盤技術の整備が現在進められているが, SOA 型システムを効率的に設計・開発する手法としては SOA のためのアーキテクチャ・フレームワークの整備が推進されている.

2.5.3. モデル駆動型アプローチ(MDA : Model Driven Architecture)

UMLを制定しているOMGにおいて提唱されている手法である. UML によるモデル記述をプラットフォーム(CPU や OS、ミドルウェア、言語など)に非依存のレベルとプラットフォーム依存のモデルの 2 段階に分け, 対象プラットフォーム上で動作するプログラムをモデルからの段階的な変換により直接得ようとするものである.

2.5.4. アスペクト指向(AOP: Aspect-Oriented Programming)

アスペクトとは, 各オブジェクトが本質的に持つような機能とは別に, たとえばログファイルの出力などの副次的な機能を指す. このような機能は必然的に複数のオブジェクトにまたがって存在するが, 設計段階においては, アスペクト毎に切り離して定義され, 最終的に各クラスのコードに分散して埋め込まれる. 現状ではアスペクト指向言語処理系の研究が進められており, 設計手法としては整理されていない.

2.6. 設計法の変遷についてのまとめ

本節では, 数学者によるアルゴリズム設計や職人芸的プログラミングから出発したソフトウェアの開発法が, システム自体の進化や社会状況の変遷にともなって, いかに変化を遂げてきたかについて概観した. 設計法の成り立ちの多くは, 熟練したエンジニアによって実践されてきた, その時々のシステム開発の方法を体系化したものであり, 様々な基本原理を含んではいるものの, 基本的には経験に基づいて構築されたものである. これらの方法論は, まったくのゼロから独立に考案されたものではなく, それ以前の方法論を通じて見出された知見や原理を整理しなおすことによって得られたものと捉えることが妥当であろう.

3. ソフトウェア設計における再利用

ソフトウェアの再利用は, ソフトウェア工学における最重要課題のひとつである. ソフトウェアの再利用を積極的に行うことには, 開発期間の短縮と, コスト削減や信頼性の高い部品を再利用することによるソフトウェア信頼性の向上というメリットが存在する.

構造化アプローチのもとでも, ソフトウェア部品化の試みは数多くなされてきたが、それらはソースコードレベルでの部品の再利用であり、実際に使用する際にはインタフェースの細かな調整のための書き換えを要したり、特定の組織内での利用や特定のシステム構造を前提としたりするものであった. そのため, 汎用システムライブラリの活用など, 一部の例外を除けば, 必ずしも効果的な再利用は行われていなかった.

オブジェクト指向の登場以降は, モジュールの独立性の向上により, ソフトウェア再利用にも一定の進歩が見られた. また, モジュール独立性の向上は, モジュール単位での再利用のみならず, モジュール間の組み合わせかたの再利用にも進歩を与えた.

本節では, 再利用を部品レベルと構造レベル(部品の組み合わせ方), 概念レベルの 3 つの再レベルに分類し, それぞれにおける再利用手段を紹介する.

3.1. 部品レベルでの再利用

3.1.1. クラス定義

クラス定義はもともとオブジェクト指向に用意されている強力な再利用のための手段である. 分析・設計段階においては, 概念モデルやインタフェース設計の再利用手段として機能し, プログラミングレベルにおいても, そのままコードの再利用手段として機能する. サブクラスや同名のメソッドの多重定義, スーパークラスのメソッドの上書き定義などがクラスの再利用を用意にする仕組みとして有効に機能している.

3.1.2. コンポーネント

コンポーネントとは, ソフトウェアの一部を機能毎に分解して保存したもので, ある特定のインタフェース規約に従った, 自己完結型のソフトウェア部品である[14]. コンポーネントは通常バイナリ形式で保存されており, 共通インタフェースにより動的に組込んで処理を行うことができる. この共通インタフェースの仕様は一般にコンポーネントモデルと呼ばれており, 基本的に, コンポーネントモデルに準拠したプラットフォームにコンポーネントを組込む形で再利用が行われる[15]. コンポーネントは単独での動的な生成が可能で, 一定の内部隠蔽性を保ちながら, 提供する機能を再

表 1. 一般的なパターンランゲージ

名前(Name)	短くて印象的な名前
意図(Intent)	パターンの概要. 可能なら図を用いる
前提 (Initial Context)	パターンを適用可能な状況
課題(Problem)	パターンが解決すべき課題の説明
解(Solution)	解の具体的内容を詳細に記述する
結果 (Resulting Context)	解を適用した結果得られる状況を説明する
関連パターン (Related Patterns)	関連するパターンや, 内包する・されるパターンを列挙する
例(KnownUses/ Examples)	どのような状況で適用されるかなどの例を示す

利用できる.

広く用いられているコンポーネントモデル/アーテキテクチャの例として, ActiveX/COM/DCOM[16], JavaBeans[17], EJB[18], CORBA/CCM[19]などが存在するが, これらは基本的にオブジェクト指向に基づいて設計されており, コンポーネントはある機能を実現する為の関連オブジェクトをまとめてカプセル化した物としてモデル化されている. したがって, オブジェクト指向アーキテクチャを持つシステムでの利用は容易であるが, それ以外のシステムでは, 必ずしも容易に利用できない構造となっている.

3.2. 構造レベルでの再利用

構造レベルでの再利用手法には, フレームワークに基づくものとパターンに基づくものとがある.

3.2.1. フレームワークに基づいた再利用

ここでのフレームワークとは, ある一定のアーキテクチャに基づいて予めある程度部品が組み込まれた半完成品に, コンポーネントを組み込んだり, 新たにソースコードを追加したりしてシステムを完成させることの出来る仕組みを指す. 例えば, GUI フレームワーク(Java の AWT/SWING や OpenStep など)や, 抽象クラスから具象クラスを穴埋め形式で生成するためのクラスライブラリ, MFC (Microsoft Foundation Class)や JFC(Java Foundation Class)のような, カスタマイズ可能なアプリケーションの骨組や, SOA (Service Oriented Architecture)に基づく Web サービス構築のためのフレームワークなど, 様々な粒度のものがある.

3.2.2. パターンに基づいた再利用

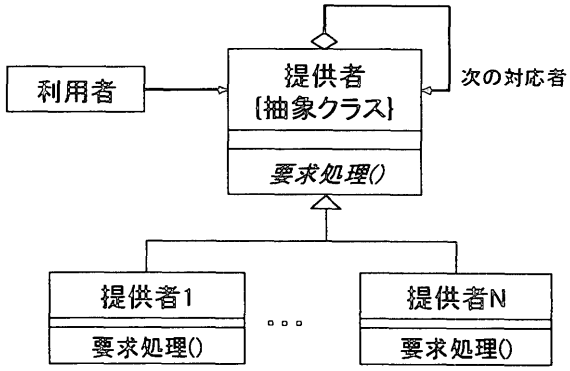

図6.デザイン・パターン Chain of Responsibility

ソフトウェア工学におけるパターンの再利用の概念は構造化手法の時代から存在する. たとえば[16]では, COBOL プログラムによる制御構造をパターン毎に整理してテンプレートとして再利用した実例が紹介されている. しかし, 今日では「パターン」という言葉は, 多くの場合, Gamma らがデザイン・パターンの概念を整理するために, Alexander が建築分野において提唱するパターンランゲージの概念[17]を取り入れたものを指すことが多い.

デザイン・パターンをきっかけとして, アーキテクチャなど他の粒度におけるソフトウェア構造はもとより, 組織構成や開発プロセスなど様々な知識の再利用の手段としてパターンランゲージの活用が進んでいる[5]. ここではデザイン・パターンとアーキテクチャ・パターンを紹介する.

デザイン・パターン

デザイン・パターンとは, オブジェクト指向設計における再利用可能なクラスの集合と組み合わせ方法について, パターンランゲージに基づいた形式でまとめたものである. Gamma によるデザイン・パターンの定義は以下のとおりである[4]:

- 問題とそれに対する解を集めたもので適切に分類され, カタログ化されていることが望ましい. 単なる方針や戦略ではなく明確な形を持った解でなければならない. 自明の解はパターンとは呼べない.
- システムの構造を具体的に示すような, 物事の関連を示すものがパターンである. 単一のモジュールによって解決されるようなものはパターンとは呼べない.
- ある一定の枠組み(パターンランゲージ)に基づいて記述される. パターン記述言語はできるだけ明確な意味定義を持ち曖昧さが少ないことが望ましい.

表2. アーキテクチャ・パターンの例

分類	パターン名	説明
システムの構造化	レイヤ	サブシステムを特定の抽象レベルでグループ化する
	パイプ&フィルタ	データストリームを扱うシステムの構造
	ブラックボード	共有データを用いて独立したプログラム間での協調動作を実現する
分散システム	ブローカー	クライアントにサーバを動的に紹介する仕組み
対話型システム	MVC	GUIとモデルと制御の分離
	PAC	アプリケーションの意味概念の分離
適合化システム	マイクロカーネル	システムの中核サービスを拡張機能やアプリケーションと分離独立させる
	リフレクション	システムの構造や振る舞いを動的に変更する仕組み

パターンランゲージはいくつかの記述要素から構成されている. 表 1 は一般的なパターンランゲージの例である. また, 図 6 にデザイン・パターン"Chain of Responsibility":受け手を明確にせずに, 複数あるオブジェクトの一つに対して要求を発行する(責任のたらいまわし)の概要を UML で記述した例を示す.

アーキテクチャ・パターン

システム分析の結果得られる問題に対する論理的構造とシステム設計の成果物である物理的構造との間にはいまだに大きな隔たりがある. オブジェクト指向設計においても, オブジェクトをシステム内に物理的に配置して運用するためには, サブシステムへの分割などの外部的構造を決定する必要があるが, RUP を含め, オブジェクト指向設計法の多くは, システム構造の定め方に対する明確な方針は定めておらず, システムアーキテクチャを決定した上で, アーキテクチャ上の各部分に必要となるオブジェクトを配置するという考え方をとっている.

個々のアーキテクチャは, 特定のプラットフォームに依存するものであってもかまわないが, そこにはプラットフォームに依存しない数々のパターンが発見されており, これをアーキテクチャ・パターンと呼ぶ. 表2は文献[18]で紹介されているアーキテクチャ・パターンの一覧である. たとえば, 今日の一般的な Web アプリケーションの多くはユーザ端末と業務処理用サーバとデータベースサーバの 3 層から構成されているが, これは, レイヤードアーキテクチャ・パターンに基づいている.

3.3. 概念の再利用

3.3.1. 概念モデル

概念モデルとは, システムが対象とする世界での約束事をモデルとして記したもので, 要求分析段階において作成される.

たとえば, 金融システムの開発であれば, 金融業界における重要なデータやアクタ(その世界で重要な役割を果たすオブジェクト)にどのようなものがあるかを, クラス図などを用いて記述する. このようにして記述された概念モデルは, 他の金融システムの開発時などにそのまま再利用可能な場合もあれば, ある組織固有のモデルとなっていて再利用困難な場合もある. そのままの再利用が困難な場合においても, 類似の概念モデル間で共通して出現する一定のパターンを発見することが出来れば, それを再利用することが可能となる. 概念モデル再利用のためのパターンとしては Fowler のアナリシスパターン[3]が有名である.

3.3.2. メタファー

メタファーとは, あるコンテクストにおける模式的表現によって, 実際の表現を置き換える手法のことで, たとえば, ストレージ内の不要なデータを家事におけるゴミにたとえるようなものである. メタファーはオリジナルの表現に対する, ある視点からの特性(たとえば上記の例であれば「始末しなければならない不要なもの」という性質)を強調して表現することができる [9]

無形の産物であるソフトウェアの開発においては, メタファーはイメージ共有のための強力なツールとしてしばしば無意識のうちに用いられる. また, メタファー自体は概念モデルのように概念を目に見える形で明確に整理されたものではなく, 人間の直感に頼る手段であるため, 概念モデルの記述が理解できないようなユーザとのコミュニケーションには有効である. しかし, 内容を過度に単純化してしまったり, 誤解を生んだりする危険性もあるため, メタファーを利用するには, 当事者間での世界観の一致が非常に重要な前提となる.

おわりに

限られた紙数で設計手法や設計知識の再利用技術について網羅することは不可能であり, 本稿においても, 話の流れに不要な技術や話題については数多く割愛したことを改めてお断りしておく.

ソフトウェアの設計においては, オブジェクト指向技術の成熟により, ようやく,「部品とその構成」というメタファーが有効に機能する基盤が整ったところであると

いえる. この基盤の上で, パターンをはじめとする様々な「構成のための技法」が, 研究レベルではなく実践レベルにおいて活発に議論されている.

今後, オブジェクト指向の次の「画期的新手法」が登場するとしても, これらの議論の成果やそこで明らかになった問題点がそこに反映されているとみるべきであろう.

文献

[1] Edward Yourdon (松原他訳):「デスマーチ - なぜソフトウエア・プロジェクトは混乱するのか」, シイエム・シイ,2002.

[2] 経済産業省「2004 年版 組込みソフトウェア産業実態調査報告書」 http://www.meti.go.jp/policy/it_policy/technology/softjittaityousa-gaiyou.pdf

[3] Martin Fowler (堀内他訳):「アナリシスパターン—再利用可能なオブジェクトモデル」, ピアソン, 2002.

[4] Gamma 他 (本位田, 吉田訳)「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」ソフトバンクパブリッシング, 1995.

[5] 井上, 松本, 飯田,「ソフトウェアプロセス」共立出版, 2000.

[6] カーネギーメロン大学ソフトウェアエンジニアリング研究所, (日本 SPI コンソーシアム訳)「CMMI®モデル-公式日本語版」,2004. http://www.sei.cmu.edu/cmmi/translations/japanese/models/index.html

[7] Tom DeMarco :「構造化分析とシステム仕様—目指すシステムを明確にするモデル化技法」

[8] Grenford Myers (國友他訳) :「ソフトウェアの複合/構造化設計」

[9] Heinz Zuellghover: "Object-Oriented Construction Handbook", Elsevier, 2004

[10] Peter Herzum, and Oliver Sims (長瀬他訳), 「ビジネスコンポーネントファクトリ」,翔泳社, 2001.

[11] 青山幹雄," コンポーネント指向ソフトウェア開発へのいざない," FUJITSU, vol50-2, pp.62-71, Mar.1999.

[12] D. Rogerson, D., "Inside COM," Microsoft Press, 1997.

[13] Sun Microsystems, "JavaBeans Documentations," http://java.sun.com/products/javabeans/

[14] Sun Microsystems, "EJB (Enterprise JavaBeans) Documentations," http://java.sun.com/products/ejb/

[15] Jon Siegel, "CORBA Fundamental and Programming," Jon Siegel, ed., John Wiley & Sons, Inc., New York, 1996.

[16] 大野治他," 多次元部品化方式によるソフトウェア開発の自動化—自動生成系の開発とその評価—," 信学論(D-I), Vol.J84-D-I, No.9, pp.1372-1386, Sep.2001.

[17] C.Alexander(平田訳):「時を越えた建設の道」, 鹿島出版会, 1993.

[18] Frank Buschmann 他(金澤他訳) :「ソフトウェアアーキテクチャー : ソフトウェア開発のためのパターン体系」近代科学社, 2000.