ソフトウェア工学 115－13
（1997．8．1）

アーキテクチャパターンに基づく設計手法の考察

岸 知二

NECマイコンソフト開発環境研究所

適切なソフトウェアアーキテクチャを構築することは重要であるが困難な作業である。本稿ではパターンを用いた設計手法について考察する。本手法では、パターンを用いて分析モデル中の構造を整理するとともに、実装時に考慮しなければならない各種パラメータをそのパターンに対応づけることによって、要求される非機能的要件を定義する方法を採る。さらにパターン毎の実装方式を検討しそれらを総合することにより、全体のアーキテクチャを構築する。

On Design Method based on Architectural Pattern

Tomoji Kishi

*Microcomputer Software Development Laboratory
NEC Corporation

It is important and difficult to construct proper software architecture. In this paper, we propose design method based on architectural pattern. In the method, we extract important structure in analysis model as pattern. Then we attach some parameter to the pattern to express nonfunctional requirements to the system. Based on these parameterized pattern, we decide the implementation of the patterns and construct overall software architecture..

1 はじめに

適切なソフトウェアアーキテクチャ（以下アーキテクチャ）に基づいてソフトウェアを開発することは、ソフトウェアの性能、信頼性、拡張性などにとって重要である。しかしながら既存の設計手法は、アーキテクチャの構築方法やアーキテクチャの持つ特性の捉え方に関して必ずしも十分な指針を与えてない。

我々はオブジェクト指向でソフトウェアを開発する際に、どのようにしてアーキテクチャを構築し、それに基づいて設計を行うかを検討してきた[7]。オブジェクト指向の開発においては、対象世界を捉える論理的なモデルを作成することが重要な作業となるが、アーキテクチャを構築するためには、そうした論理的なモデルには明示的に現れない性能や信頼性といった非機能的な要件を考慮しなければならない。またこうした上位からの要件だけでなく、ソフトウェアを実装する環境においてどのような機構を利用できるかという下位からの制約もあわせて考慮しなければならない。こうした点がアーキテクチャの構築を困難にしているといえる。

我々は信頼性等のいくつかの重要な非機能的要件が与えられた中規模のソフトウェアをオブジェクト指向で開発する中で、アーキテクチャパターンに基づいてアーキテクチャを構築する手法を適用している。本手法では、アドホックにアーキテクチャを決定するのではなく、アーキテクチャに求められる要件をアーキテクチャパターン毎に整理し、それらを統合しながら全体のアーキテクチャを構築するという考え方をとっている。本稿では本手法を紹介するとともに、その利点や課題について考察する。

2章では本稿におけるアーキテクチャとパターンという用語の意味を明確にする。3章では本稿の想定するオブジェクト指向開発の概要と、その中におけるソフトウェアアーキテクチャ構築の位置づけ、さらにその際の課題について述べる。4章では我々が試行中のアーキテクチャパターンを用いた設計手法を紹介する。5章では本手法に関連した議論を行う。

2 アーキテクチャとパターン

本章では、本稿においてアーキテクチャおよびパターンという用語がどのような意味で用いられているかを述べる。

2.1 アーキテクチャ

アーキテクチャとは、ソフトウェアの構造、すなわちソフトウェアの構成要素とそれらの間の関係を指す。関係には構成要素間の静的な関係だけでなく、構成要素の協調関係など動的な関係も含まれる[7]。

この定義は広範な内容を含むため、本稿ではアーキテクチャが何の構造を表すものであるかという観点から、論理アーキテクチャと実装アーキテクチャとの２つに分類する。論理アーキテクチャとはソフトウェアの対象世界を記述するときに用いられる構造を指し、実現のしくみに拘束されない本来あるべきソフトウェアの論理構造を示す際に用いられるものである。これはオブジェクト指向開発での分析モデルに対応するものである。一方実装アーキテクチャとはそうしたソフトウェアを実現するときに用いられる構造を記述するものであり、オブジェクト指向開発での設計モデルに対応するものである。

分析モデルや設計モデルはアーキテクチャに基づいて構築される、あるいはそれらのモデル中に潜在する構造を明示化したものがアーキテクチャである。定義上は分析モデルや設計モデルそのものをアーキテクチャと呼ぶことも可能だが、同一であればアーキテクチャを区別して認識する現実的な意味はあまりない。実用面から考えるなら、なんらかのアーキテクチャを認識することによって特定の分析モデルや設計モデルを構築したり、読み解いたり、利用したりする際にメリットがあるかどうかがポイントとなる。

本稿で取り上げるアーキテクチャは実装アーキテクチャであり、以下特に断らない限りアーキテクチャとは実装アーキテクチャを指すものとする。

2.2 パターン

パターンとは繰り返し現れる、あるいは繰り返し用いられる構造を示す[2]。従ってアーキテクチャパターンとは繰り返し現れるアーキテクチャの構造を意味する。Buschmann はパターンをアーキテクチャパターン、デザインパターン、イディオムに分類し、アーキテクチャパターンという用語を基本的なシステムレベルのアーキテクチャを意味するものとして定義しているが[2]、本稿では特にそうした限定的な意味では用いず、上述したアーキテクチャに対するパターンをすべてアーキテクチャパターンと呼ぶ。

なお本稿ではパターンの記述にはデザインパターン[3]的な記述を応用する。すなわちその構造の中にどのような役割を持ったコンポーネントが存在するかをクラス図で示し、それらがどのような協調をしながら一連のふるまいをするかをインタラクション図で示す。しかしながら対象とするアーキテクチャの種類によって、アーキテクチャパターンの利用目的は異なるため、デザインパターン的なカタログ記述をそのまま利用することはしない。

3 アーキテクチャ構築と課題

本章では、我々がどのようなオブジェクト指向開発を想定し、その中においてアーキテクチャ構築がどのように位置づけられているのか、どのような構築上の課題があるのかを述べる。

3.1 分析モデルと設計モデル

オブジェクト指向で開発する際には、分析モデルと設計モデルの２種類のモデルを作成する。

分析モデルは、対象世界の論理構造を記述したものであり、実装上の制約を考慮せずに作成される。分析モデルは主としてソフトウェアの構築、修正、拡張、あるいは再利用が容易となることをねらって作られる。

一方設計モデルは、分析モデルで表されたあるべきソフトウェアの構造を、現実のソフトウェアとして実現するための構造を記述するものである。ソフトウェアを実現するためには、実装環境で提供される機構（言語、ＯＳ、ライブラリ等）を利用する必要があるため、設計モデルは、対応する分析モデルが要求する要件を満たさなければならないだけでなく、実装環境から様々な制約を受けることになる。

分析モデルをそのまま設計モデルとして利用できることがひとつの理想形かもしれないが、多くの場合それは困難である。その理由は、設計モデルにおいては、性能や信頼性といった分析モデルでは明示的に示されない要件を考慮する必要があるからである。規模の小さなソフトウェアや、特段の工夫がなくても求められる性能や信頼性を達成できるような環境で実装する際には、分析モデルと設計モデルの違いは大きくならないかもしれない。しかしながら規模の大きなソフトウェアや、何らかの工夫がなければ性能や信頼性が確保できない環境においては、分析モデルと設計モデルとはかなり異なったものとなる。

分析モデルと設計モデルとが異なってくる他の理由として、分析モデルを構築する際の仮定が、そのまま実装環境ではなりたたないことも指摘できる。例えば分析モデルを作成する際にはすべてのオブジェクトが自律的にふるまいながら動作をしているという仮定をおくことも多いが、Ｃ＋＋のクラスとしてオブジェクトを実装すると、そのままの仮定で設計モデルを作ることは困難となる。

3.2 アーキテクチャの位置づけ

前節で指摘したように、設計モデルの作成は単に分析モデルに示された要件だけから演繹的に行えるものではない。実装環境で利用できる機構を考慮して、それらの要件を受け止めることのできるしくみを考え、そのしくみに基づいて設計モデルを構築しなければならない。さらにそうしたしくみを考える際には、性能や信頼性などの非機能的要件を達成できるかどうかを考慮しなければならない。性能や信頼性は

実装に用いる個々の機構だけによって決定されるものではなく、それらの機構をどう有機的に関連づけて用いるかによって決定される。従って設計モデルを作成する際には、それらの非機能的要件を受け止めることのできる機構の構造、すなわちアーキテクチャへの考慮が必要となる。

すなわち設計においては、分析モデルに示される機能的な要件だけでなく、そこには明示的に示されない非機能的な要件を明確にすることと、実装環境で利用できる機構を関連付けてそれらの要件を満たすことのできるアーキテクチャを構築することが必要となる。設計モデルはこのアーキテクチャに基づいて構築されることになる。なおひとつの分析モデルに対応する設計モデルが複数存在し得るように、ひとつの分析モデルは様々なアーキテクチャ上で実現することができる。

3.3 アーキテクチャ構築の課題

分析モデルや設計モデルの作成がそうであるように、アーキテクチャの構築を機械的な手順で行うことはできない。しかしながらそこになんらかの構築の指針が与えられることが望まれる。次章ではアーキテクチャ構築とそれに基づく設計モデルの作成の手法について述べるが、その際に留意すべき課題を以下に整理する。

- 構造の重要性：アーキテクチャの重要な特性が構造によって決定される以上、アーキテクチャの構築においても構造をより明示的に捉えることが必要となる。分析モデル上で示された機能や情報をアーキテクチャ上のコンポーネントにマッピングするというような方法だけでは、そうした構造の持つ特性を捉えることが困難である。
- 非機能的要件の取り扱い：アーキテクチャ構築にとって重要な性能や信頼性といった非機能的要件は、通常分析モデル上には明示的に示されない。従ってそうした非機能的要件どう取り扱うべきであるか、指針を整理する必要がある。
- 設計モデル構築の手順化：設計モデルはアーキテクチャに依存するため、開発過程のどのタイミングでそれを作成するかを明確にすることが重要である。特に動的モデルをステート図等を用いて一般的に定義する作業はコストがかかるため、作業効率の面からも手順の明確化が必要となる。

4 パターンを用いたアーキテクチャ設計

我々はアーキテクチャ構築においてパターンを活用した手法を検討し、実際の開発への適用を進めている。本章では、本手法について述べる。

4.1 ねらい

アーキテクチャの特性が構造によって決定されるとするならば、アーキテクチャがどのようなコンポーネントを含むかということだけでなく、どのような構造を含むかが重要な問題となる。そこでアーキテクチャに潜在的に含まれる構造[1]をパターンとして捉え、パターンをアーキテクチャ検討の基本単位として利用することを考える。本手法の基本的な発想は、アーキテクチャに内在するパターンを捉え、そのパターン個々の特性を明示化するとともに、求められる要件がどのパターンによって実現されるかを検討することによって、アーキテクチャ構築を行おうというものである。

なお我々はパターンの記述にデザインパターンの記法、すなわちクラス図により役割と関係の記述を行い、インタラクション図によりふるまいの例示を行う方法をとっている。この方法は分析モデルや上位設計を行っている段階では極めて有用である。その理由のひとつは、そうした段階では動的側面に関しては重要な局面にお

[1] アーキテクチャとして明示的に示される構造だけでなく、そのアーキテクチャ上にどのような設計モデルが構築されることを想定しているかが重要となるため、そうした設計モデルの構造を含めるために「潜在的に含まれる」という表現を用いている。

けるふるまいの制約が断片的に与えられているだけであることが多く、状態モデル等で一般化することが困難であることが多いからである。インタラクション図は例示であるため、与えられた断片的な制約を記述することに適している。もちろん例示だけではモデルとしては不十分であるが、動的モデルがアーキテクチャに強く依存することを考えると、そうした動的側面の厳密な一般化はアーキテクチャが決定された後に設計モデル上で行うことが有利なことが多い。

4.2 パターンを利用した設計

本節ではパターンを利用したアーキテクチャ構築の考え方を述べる。

まず分析モデルの中に内在するパターンを抽出する。ここで抽出したいパターンは、主として動的なふるまいを捉えたパターンであり、その非機能的特性が設計において重要となるものが対象となる。分析モデル中から抽出されたこれらのパターンを論理パターンと呼ぶ。論理パターンはパターンカタログに載っているような著名なパターンである必要はなく、開発しているソフトウェア中だけに繰り返し使われている特有のパターンであって構わない。パターンが抽出されたら、そのパターンに当てはまる構造が分析モデル中のどこに存在するかを明らかにする。パターンの抽出とパターンの対応付けは順次的に行われるのではなく並行して行われる。この一連の作業によって、分析モデル上に存在する構造、特にふるまいにかかわる構造をパターンに照らして説明できるようにする。

次に分析モデルから抽出された主要な論理パターンを実現するための実装構造を考え、それをパターン化する。このパターンを実装パターンと呼ぶ。

例えば、分析モデル中でデータ値を監視するという構造が複数存在するとき、それらを「監視パターン」という論理パターンで説明することができる。「監視パターン」は監視側、被監視側、被通知側といった役割を含み、特定の状況を監視し通知するための一連のインタラクションをパターン化したものとなる。一方データ値を監視する方法としてポーリングという実装方式を利用するなら、それを「ポーリングパターン」という実装パターンで捉えることができる。また「監視パターン」は監視結果を通知することも必要なため、その部分は例えば「送信・受信パターン」という実装パターンで実現することができる。すなわち「監視パターン」全体はこれらの実装パターンの組み合わせで実現されることになる（図 1参照）。

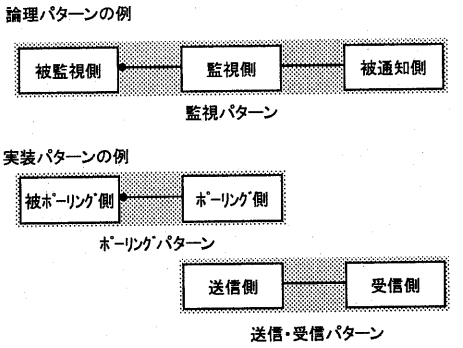

図 1　論理パターンと実装パターン

この「ポーリングパターン」や「送信・受信パターン」を特定の実装環境で実現することを考えると、そのパターンが持つ具体的な特性を認識することができるようになる。例えば実装するハードウェアや OS などが決まれば、どの程度の周期でポーリングすることが可能か等の特性を議論することができる。こうした実装パターンの持つ特性が「監視パターン」の要件を満たすかどうかによって、非機能的要件が実現可能かどうかを判断する。

分析モデル中には複数の論理パターンが存在するため、それを受け止めるためには複数の実装パターンが必要となる。したがってそのソフトウェアを実現するためのアーキテクチャは、それらの実装パターンをすべて含む必要がある。もちろん論理パターン群を見つけ、それらに対応する実装パターン群を定義したからといって、それらを受け止めるアーキテクチャが機械

的に求められるわけではない。しかしながらアーキテクチャに求められる要件が具体的に与えられることによって構築作業ははるかに容易になる。また特定のアーキテクチャとそれが内在する実装アーキテクチャが明確にされれば、そのアーキテクチャの妥当性をより適確に検討できるようになる。

4.3 パターンとパラメータ

前節で議論したように、本設計手法では論理パターンに対して非機能的要件を対応づける必要がある。別の言い方をすると、論理パターンに対してはその実装を考えるときに重要となるいくつかの実装上の要件が存在する。例えば「監視パターン」では、監視の周期、監視対象の数、監視内容などが重要な要件となる。ここで重要とは、これらの要件によって実現の方法が変化したり、性能や信頼性に影響を及ぼしたりするという意味である。本稿ではこうしたパターンにとっての重要な要件をパターンのパラメータと呼ぶ。

同一の論理パターンとして表現される分析モデル中の構造も、それぞれのパラメータの内容によって、異なる実装パターンによって実現されることがあり得る。逆に異なる論理パターンを同一の物理パターンで受け止めることもあり得る。一般に論理パターンは複数の実装パターンに展開することができ、どのような実装パターンに展開できるかを判断するための情報がパラメータとして表現されていると考えられる。

4.4 設計フロー

以上説明した考え方に基づいた典型的な設計フローを以下に示す。このフローは逐次的に進められる必要はなく、繰り返しを含みながら完成度が上がっていくものである。

(Step1)分析モデルの構築：ＯＯＳＥ[5]等を用いて分析モデルを導く。本手法はＯＯＳＥに依存するものではないが、動的なふるまいを捉えた論理パターンを抽出するという観点から、責任主導のモデルが得られていることが望ましい。データ主導のモデルの場合、そうしたパターンを捉えることが困難である。

(Step2)論理パターンの抽出：分析モデルから論理パターンを抽出する。抽出されたパターンには名称をつけ、クラス図によりそのパターンに参加するクラスの役割と関係を定義し、典型的なふるまいをインタラクション図で記述する。前述したようにインタラクション図はこのパターンに当てはまる具体的な構造が満たすべき制約あるいは必要条件を表していると考えられる。次に抽出された論理パターンに対して、どのようなパラメータが必要かを定義する。必要なパラメータは論理パターン毎に異なるが、典型的なパラメータの例としては、インタラクションの回数や頻度、扱われるデータの量、そのパターンに当てはまるインタラクションの総数（トランザクションの総数）などがある。図 2に抽出されたパターンの記述例を示す。

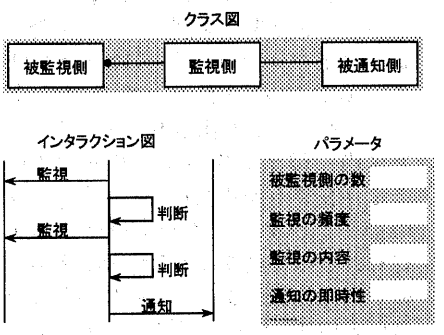

図 2　パターンの記述

(Step3)論理パターンとの対応付け：分析モデル中の構造を論理パターンと対応づける。対応付けはふるまい毎に独立して記述する方が分かりやすい。さらに対応付けらえた構造毎にパラメータの内容を決定する。定量的な値が決まらない場合には

定性的であっても構わない[2]。(Step2) と (Step3) の作業は並行して進行するので、ある程度両方の作業が進行しないとパターンは安定しない。そういう意味でパラメータの定義等はあまり早期から行うことは得策ではない。図 3にパターンとの対応付けとパラメータ記述の例を示す。

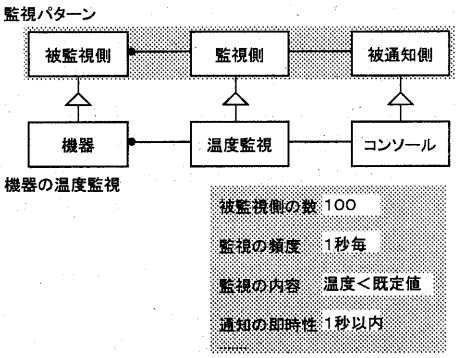

図 3　パターンとの対応付け

(Step4)アーキテクチャの検討：抽出された論理パターン毎に実装方式を検討する。その方式の中から実装パターンを抽出する。もっとも単純には論理パターンに対してそれを実装するための実装パターンを定義する。同一の論理パターンであってもパラメータの内容や実装環境によって用いる実装パターンが異なることもある。

次にこうして発見された実装パターン群を実現することが可能なアーキテクチャを構築する[3]。複数の実装パターンをひとつのアーキテクチャにまとめる際には、同一化できる実装パターンがあるかどうかを判断する必要がある。同一化には2種類あり、パターンは同一だが実体が異なる場合と、実体も同じ場合とがある。例えばソケットを利用した実装パターンがあった場合、ソケットの利用方法は同一だが実行時にはそれぞれ別々のソケットを利用する場合と、同一のソケットを共用する場合とが考えられる。この両者は設計上異なってくるので留意が必要である。

このようにしてアーキテクチャが構築されると、その中に含まれる実装パターンの特性と、対応する論理パターンに要求される要件（パラメータの内容）を比較することによってそのアーキテクチャの妥当性の検討が可能となる。

(Step5)設計モデルの構築：アーキテクチャが決定されたら、論理パターンと実装パターンとの対応を考慮しながら設計モデルを構築する。この段階では動的モデルを一般化して厳密に定義する。

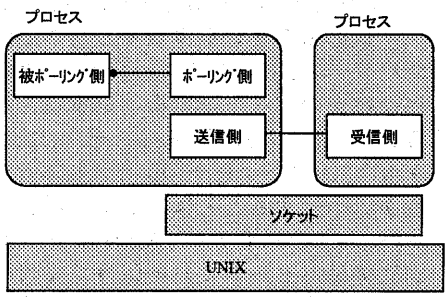

図 4　アーキテクチャとの対応付け

5　議論

従来の手法においてもシステムを構成するサブシステム、モジュール、ブロックなどを認識し、あるいはそれらの典型的なアーキテクチャを用いて、具体的な設計を行うことが指摘されている。しかしながらアーキテクチャの特性そのものをより明確に捉え、その構築や妥当性確認を行うという視点は十分でない。

そうした中でアーキテクチャに対する視点を掘り下げた手法もいくつかある。例えば Hatley らはアーキテクチャ上のコンポーネントに機能や情報を配置したり、その結果導かれるコンポーネント間のチャネルの妥当性を検討する方法を提案して

[2] 実現を検討するにあたって内容を決定する必要がないのであれば、そのパラメータは不要である。

[3] アーキテクチャの構築には実装パターンだけでなく、実装環境、ソフトウェアの物理的な配置、修正、拡張、再利用の容易性等様々な要因を検討しなければならない。

いる[4]。また Buhr らはユースケースマップを用いてシステムのマクロなふるまいをモデル化する方法を提案している[1]。

本手法はアーキテクチャに内在する個々のパターン要素に注目して、そのパターンがアーキテクチャを特性づけている基本単位だとして捉え、分析モデルと設計モデルの橋渡しに利用している点などに特徴がある。

パターンに対して定義するパラメータは、単に非機能的要件を記述するという役割だけでなく、そのパターンから派生するいくつかの構造のバリエーションを想定し、そのバリエーション毎に変化する点、気をつけなければならない点などを明示的に提示したものと考えられる。別の言い方をするならば、各パターンがどういう構造に対して適用すべきであるかの枠組みを提示するものであるとも考えられる。

一連のふるまいを表現するために複数のパターンが使われることも多い。その際には複数のパターンの役割を重ね合わせるなどの工夫が必要である。そうした状況を含め、パターンをどのように組み合わせるかを検討することはひとつの課題である[8]。また本手法では論理パターンを個別に実装パターンに対応付けて妥当性を確認したが、複数のパターンのふるまいに何らかの関連や相互の影響が起こることも考えられ、そうした状況に対する考慮も必要であろう。

6 おわりに

本稿ではパターンを用いたアーキテクチャ構築手法について議論した。現在、実際のソフトウェア開発に本手法を適用しながらその妥当性やより適切な運用について検討を深めている段階である。今後その結果等を踏まえ、より有用性の高い設計手法へとリファインしたい。

参考文献

[1] Buhr, R.J., et.al.: *Use CASE Maps for Object-Oriented Systems*, Prentice-Hall, (1996).

[2] Buschmann, F., et.al.: *Pattern-Oriented Software Architecture - A System of Patterns*, Wiley, (1996).

[3] Gamma, E., et.al.: *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, (1995).

[4] Hatley, D.J., et.al., *Strategies for Real-Time System Specification*, Dorset House Publishing, (1988). 邦訳: リアルタイム・システムの構造化分析, 日経BP.

[5] Jacobson, I., et.al: *Object-Oriented Software Engineering - A Use Case Driven Approach*, Addison-Wesley, (1992).

[6] 岸,他: オブジェクト指向設計における制御アーキテクチャの定義, ソフトウェア光沢研究会, Vol.94, No.55. (1994).

[7] 岸,他: ソフトウェアアーキテクチャモデルに基づく設計手法について, FOSE'96, (1996).

[8] 野田,他: パターンを用いたアーキテクチャ設計, 情報処理学会, OO'97, (1997).

[9] Rumbaugh, et.al.: *Object-Oriented Modeling and Design*, Prentice-Hall, (1991).

[10] Shaw, M., et.al.: *Software Architecture, Perspectives on an Emerging Discipline*, Prentice-Hall, (1996).