修 士 論 文

自律オブジェクト群の制御を目的とするデザインパターンの拡張

指導教官　落水浩一郎 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

鈴木大輔

1998 年 2 月 13 日

要 旨

ソフトウェアアーキテクチャを利用したソフトウエア開発においては、仕様化設計段階と実装段階にギャップがある。本論文では、そのギャップを埋めるためのデザインパターンの拡張を行った。

本研究では、ソフトウェアのごく一般的な解法を示したデザインパターンを、ドメイン実装言語を特定し、拡張を行うことで、アーキテクチャとして利用可能にした。具体的には、ドメインとして分散共同開発の支援、実装言語として Java 特化した形にデザインパターンを拡張・変更する。このようにドメインと実装言語を特定することにより、デザインパターンの持つ問題点を解消出来ることを示し、その有効性について検討する。

目 次

図 目 次

表 目 次

第 1 章

はじめに

1.1 背景

近年、ネットワーク等を介した分散共同開発が行なわれるようになってきている。しかし、分散共同開発においては、その地理的分散が原因でさまざまなトラブルが生ずる。

例えば、トラブルの 1 つとして、共同作業においては作業者間の合意事項や中間成果物の内容の認識に関するズレが発生し、このようなズレは作業者の地理的分散によってさらに拡大し、その収束が遅延してしまう。このような状況は開発状態の不安定さをもたらし、中間成果物の内容や作業のコンテキストとして存在する合意事項に多くの矛盾や不確実さを発生させる。

そこで、何らかの支援が必要になると考えられる。これに対して、落水研究室で開発中の「自在」[1, 2, 26] 環境では、これらの問題の起こりやすい情報群をリポジトリとして定義し、その中に含まれる矛盾や不確実さといった状態を管理・制御することで、この調整支援を行う試みが進められている。

「自在」では、状態情報を含んだ上記リポジトリを CSCSD モデルとして定義している。このモデルは、「一貫性と不確実性の管理」を実現するために必要な機能の階層を示した概念モデルになっている。(図 1.1)

このような分散共同開発支援環境では、

- 機能の変更、追加が容易である
- 分散・ネットワークを意識させない

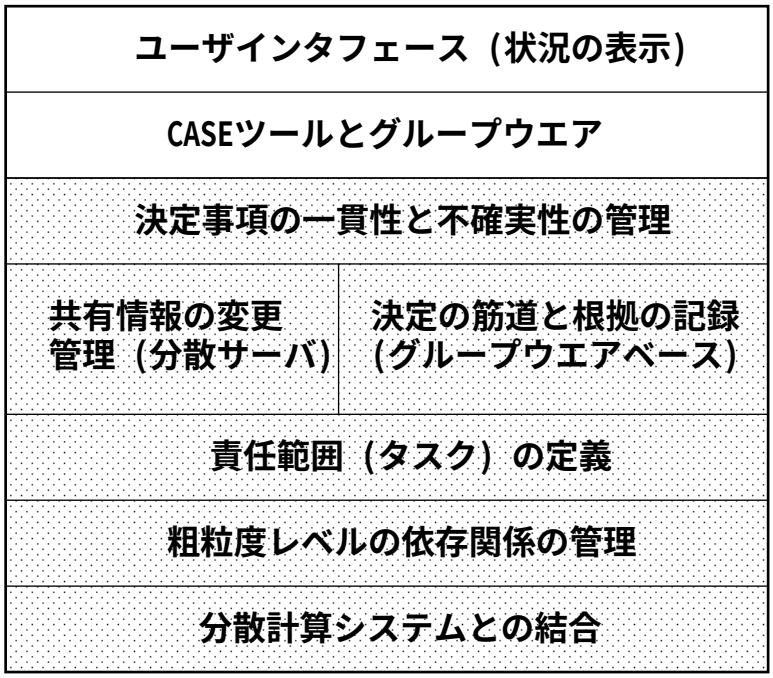

図 1.1: CSCSD モデル

- 環境に適応しやすい

のような要件を満たす支援システムのアーキテクチャを考える必要がある。「自在」では、この支援システムを自律オブジェクトとそのインタラクションとして実現する予定である。

本研究では、この「自在」を実現する上で必要な機能や要件を満足しつつ、容易にシステムを実現できるようなフレームワークやツールキットを構築するにあたっての問題点を整理し、解決方法の提案をする。

フレームワークやキットを利用して開発を行う場合、それを柔軟で再利用可能なものにするために、このアーキテクチャとして Gamma によるデザインパターン [3] を利用する「パターン指向」と呼ばれる開発手法がとられることが多い。

そこで、本研究では、システム構築にデザインパターンを利用する方針を取り、これを「自在」環境構築に適用する場合の問題点とその変更点に関して考察する。

1.2 研究の目的

本研究の目的は、前節で述べた支援システムのフレームワークとキットを構築するための設計指針を提案し、これに基づいたフレームワーク開発を行うことである。

柔軟なフレームワークやキットを容易かつ高品位に構築するためには、その設計と実装においてしっかりとしたアーキテクチャが必要である。このアーキテクチャをどのように表現するか、なにを基礎とするかを考えることは非常に重要である。本研究ではこの基礎となるものとしてデザインパターンを利用する。しかし、デザインパターンではシステム全体のアーキテクチャとしては不十分であることが予想される。

本研究では特に、自律オブジェクト群とその動作環境というドメインを対象に、このデザインパターンの変更・拡張点を示し、その拡張パターンの有用性を確認する。

1.3 本論文の構成

2 章では、Gamma のデザインパターンとアプリケーションフレームワーク、ドメインモデル、アーキテクチャとの関係を考察し、「自在」環境構築における重要性や関係などを述べる。特に、ソフトウエア開発におけるデザインパターンの役割と問題点について考察する。

3 章では、本研究が対象とする自律オブジェクトについて、既存の研究の動向と一般的な定義を述べる。

4 章では、「自在」環境構築にあたっての問題点を明らかにし、具体的な構築手法について議論する。また、デザインパターンを「自在」へ適用するにあたっての考え方や、パターンの特殊化について考察する。

5 章では、「自在」に特化したパターンの具体例を提示する。

6 章では、5 章で提示したパターンの有効性と利用方法について議論し、最後に 7 章で本論文のまとめと今後の課題を述べる。

第 2 章

デザインパターンとドメインモデル、アーキテクチャスタイル

本章では、前章で述べた自在フレームワークとキットを開発する上で重要となる、デザインパターン・ドメインモデル・アーキテクチャとの関係を文献 [11, 12] に従ってまとめる。

2.1 相互関係

William M. Tepfenhart らによると [15]、ドメインモデル、アーキテクチャスタイル、デザインパターン、フレームワーク、キットの相互関係は、図 2.1のように表現される。

一般に、アプリケーション開発では、ドメインモデルの分析を経て、アーキテクチャスタイルを決定する。その後、ドメインに非依存な部分をまとめたフレームワークを作成し、そのホットスポット (特定のアプリケーションやドメインに特有な部分) をキットで埋めるという工程をたどる。

しかし、アーキテクチャスタイルの決定はドメイン非依存とされているが、実際にはドメインモデルに応じて、選択が行われる。また、フレームワークはドメイン非依存とされているが、これもドメインモデルが重要になる。

実際にフレームワークやキットを作成する場合の工程では、アーキテクチャスタイルとドメインモデルを元にデザインパターン等のパターンを利用してフレームワークやキットを作成することになる。

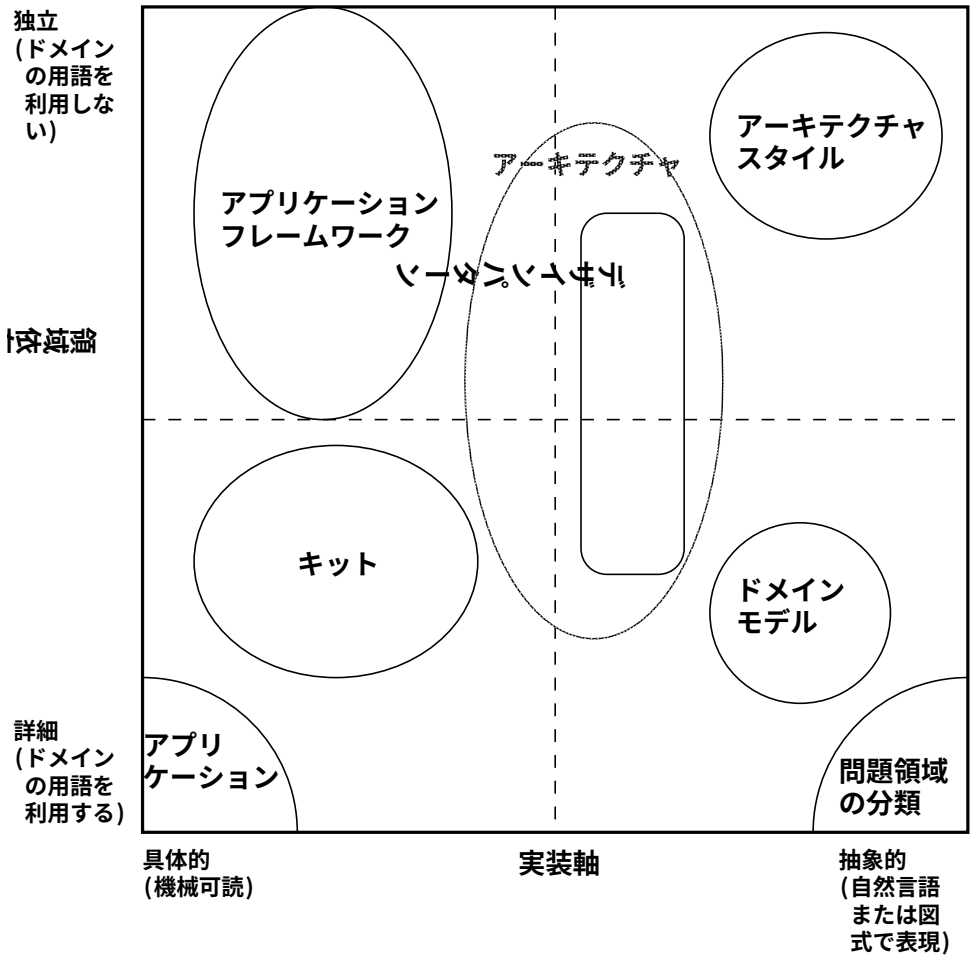

図 2.1: 諸定義 (Tepfenhart)

2.2 ドメインモデル

ドメインモデルとは、ある特定のドメインを的確に表現したオブジェクトモデルのことを示し、ドメイン分析 [21][22] によって作られる。一般的に実装に関しては一切言及をしていない。本研究においては、自在におけるオブジェクトモデルがドメインモデルに該当する。

2.3 アーキテクチャスタイル

アーキテクチャスタイル [13] とは、ドメインに非依存なシステムの構造についてのパターンを示し、システム全体の大きな枠組の様式を表現する。M. Shaw と D. Garlan による分類学的な成果 [13] が存在する。

以下に Shaw と Garlan により分類された具体的なアーキテクチャスタイルの例を示す。

パイプとフィルタ (Pipe and Filters)

UNIX の shell スクリプトに代表されるバッチ処理型システムの様式。

データ抽象とオブジェクト指向構成

(Data Abstraction and Object-Oriented Organization)

抽象データ (データとそれに対する操作を隠蔽したもの) やオブジェクトの集まりとしてシステムを表現する様式。

イベント駆動、暗黙起動 (Event-Based, Implicit Invocation)

GUI プログラミングに代表される、入力等のイベントがシステムの構成要素を暗黙的に起動するようなシステムの様式。

階層化システム (Layered System)

オペレーティングシステムやネットワークシステムの様に、システムを「使う-使われる」の関係で階層化する様式。各層はすぐ上下の層としかやりとりができない。

リポジトリ (Repositories)

データベースが中心となる形式の応用を規定する様式。プログラムはデータベース問い合わせ言語で書かれる。

インタプリタ (Interpreters)

構文と意味が定義された言語を処理する様式。

プロセス制御 (Process Control)

熱機関などの系を制御する工程の一部をソフトウエアで実現する場合の様式。

このようにアーキテクチャスタイルはごく大雑把な分類であり、ドメインにも実装にも依存しない様式を示したものである。この様式を決めることで、全体的な枠組が決まると言える。

自在では、リポジトリを定義し、それを監視制御する形になっているため、ここで言うところのリポジトリスタイルをとることになる。しかし、リポジトリと言ってもその内部の表現方法はさまざまで、具体的な実装には全く結び付かない。

2.4 デザインパターン

デザインパターン [3] は、設計における一般的な解法をパターン言語によって整理している。その目的の 1 つは変更に強く再利用性の高い設計の方法を提示することがある。また、設計における共通の用語を提供しようという目的もある。

デザインパターンでは、「実装の継承を元にした静的クラス階層は再利用性を向上しない」といった考え方に基づいて、委譲と合成を利用した設計を勧めている。

Gamma らの定義したパターンは、生成パターン (Creational Patterns)、構造パターン (Structural Patterns)、振舞パターン (Behavioral Patterns) の 3 種類に大別されている。それぞれ、インスタンス生成に関わる問題、複数のオブジェクト間の構造に関連する問題、データ構造と独立したメソッドの定義に関する問題の一般的な解法を示している。Gamma らのデザインパターンでは、5 個の生成パターン、7 個の構造パターン、および 11 個の振舞パターンが提唱されている。

以下に、それぞれのパターンについて簡単に説明する。

2.4.1 生成パターン

Abstract Factory 互いに関連し合うオブジェクト群を、その具象クラスを明確にせずに生成するためのインタフェースを提供

Builder Abstract Factory における生成アルゴリズムのカプセル化

Factory Method 作り方の違うインスタンスの生成

Protptype インスタンス複製による新たなインスタンスの生成

Singleton インスタンスを 1 つに限定する

2.4.2 構造パターン

Adapter インタフェースが期待するものと違う場合のラッピング

Bridge 仕様と実装を分離する

Composite 合成オブジェクトを作る

Decorator 部品に装飾を施す

Facade サブシステムのインタフェースを統一する

Flyweight 細かいオブジェクトを共有により効率的に利用する

Proxy 代理機能の実現

2.4.3 振舞パターン

Chain of Responsibility 要求を処理するオブジェクトを分散してリストをたどって要求を渡していく

Command 要求をオブジェクトにカプセル化する

Interpreter 言語の中間形を木構造でもち、木の接点に実行方法を記述する

Iterator 集合データの内部表現を気にせず、要素に順にアクセスする方法を提供

Mediator オブジェクト間の相互関係をカプセル化する

Memento カプセル化を破壊せずにオブジェクトの内部状態を外面化

Observer MVC を規定する

State オブジェクトの状態に伴う振る舞いカプセル化

Strategy アルゴリズムの実装を分離

Template Method アルゴリズムの骨組みを提供する

Visitor データ構造とその操作を分離

このように、狭い範囲で、粒度の微小な一般的な構造を示したものであることから、デザインパターンはマイクロアーキテクチャと呼ばれる。

しかし、デザインパターンでは、ごく一般的な設計に関する解法を示しているだけで、当然、実装に関しては示されていない。また、粒度が非常に細か過ぎるため、プログラミングの助けにはなるが、システムとしてどのように組合せ、どのように使うのかが分からない [19]。ドメインや言語にも一切依存していないため、それらに特有の解法と言ったものも考慮されていない。

2.5 アーキテクチャ

アーキテクチャとは、アーキテクチャスタイルよりも具体化したもので、ドメインに一部依存している。Garlan によるとアーキテクチャの定義は以下のとおりである。[14]

> 「プログラムまたはシステムの構成要素の構造、構成要素間の関係、およびそれら (構成要素よその間の関係) の設計と時間経過による進化を支配する原則と指針。」

デザインパターンはマイクロアーキテクチャと呼ばれるため、デザインパターンと何かがアーキテクチャであるという考え方もできる。全体の大きな枠組を示すものがアーキテクチャスタイルで、その内部の構造を規定するものがデザインパターンとすれば、これらの組合せがアーキテクチャと言える。

しかし、どのように組み合わせるかという問題や、デザインパターンをそのまま利用できるのかといった問題があり、実際にこれをドメイン非依存のアーキテクチャとして一般化するのは難しい。

2.6 フレームワーク

フレームワークとは、ある特定の種類のソフトウエアを対象にした、再利用可能な設計プロダクトを構成する協調動作するクラスの集合を言う。アプリケーションの実装のために必要となるドメインに非依存の枠組 (実装) を提供する [15]。フレームワークを構成している抽象クラスがドメイン固有のサブクラスを生成することで、フレームワークを特定のアプリケーションにカスタマイズすることが可能である。フレームワークはアプリケーションのアーキテクチャを現実化する [3]。フレームワークには、システム基盤フレームワーク、ミドルウェア統合フレームワーク、アプリケーションフレームワークがあり、一般にフレームワークの理解には多大な労力を要する [18]。

2.7 キット

キットとは、特定のドメインで利用される論理的に関連しあっているオブジェクトのことを言う。ドメインに非常に強く依存しており、完全に実装されているものである [15]。キットは有益な機能を提供するが、アプリケーション設計までは決めてしまうことが無いようなクラスの集合であると言われる [3]。

2.8 本研究との関連

本研究では、アーキテクチャスタイルとドメインモデルからデザインパターンを利用してフレームワークとキットを開発することが目的であるため、開発の進み方は、図 2.2 のようになる。本研究の本論であるデザインパターンの拡張は図 2.2における**拡張部分**となる。

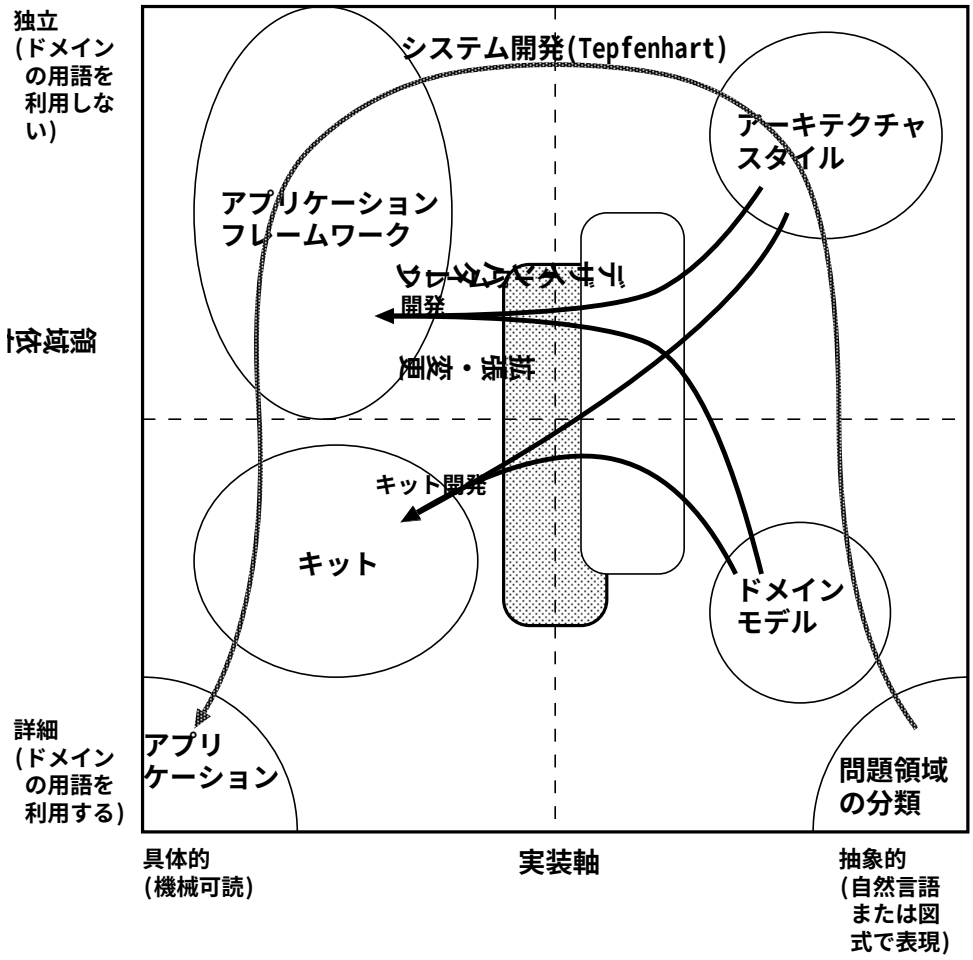
独立
(ドメイン
の用語を
利用しな
い)
システム開発(Tepfenhart)
アーキテクチャ
スタイル
アプリケーション
フレームワーク
キット開発
詳細
(ドメイン
の用語を
利用する)
キット開発
キット
ドメイン
モデル
アプリ
ケーション
問題領域
の分類
具体的
(機械可読)
実装軸
抽象的
(自然言語
または図
式で表現)

図 2.2: 本研究の役割

第 3 章

自律オブジェクト

この章では、本論文が対象とする**自律**オブジェクトについて述べる。自律オブジェクトに関する関連研究と本論文における自律性に関する議論を行う。

3.1 自律性の定義

自律性に関しては、明確な定義をしたものは見当たらない。しかし、M. Jackson は以下のような領域に分類することで、これを定義をしている。[4, 5]

- **能動的 (Active)**
  外部からの刺激無しに自発的は動作を行うことができる。
  - **自律的 (Autonomous)**
    外部から制御不可能
  - **指示可能 (Biddable)**
    外部からの指示によって制御可能 (ルールで決められている範囲内ならば自発的な動作が出来る)
  - **プログラム可能 (Programmable)**
    あらかじめプログラムしておくことにより能動的に動作可能
- **反応的 (Reactive)**
  外部世界からの入力したがった反応動作のみ可能

- **不活性 (Inert)**
  自動的な動作は一切できない。

しかし、ここで述べられている分類の定義は非常に曖昧であり、オブジェクトに関する自律性を述べるには、不十分である。

3.2 関連研究における自律オブジェクト

自律オブジェクトを扱う研究は他にもいくつか存在する。たとえば、AutO (A Distributed System of Autonomous Objects) [6, 7] では、本研究と同様に自律オブジェクトを「プログラム可能な能動性を持たせたオブジェクト」としており、これらの集合で分散システムを表現するものである。

また、MESSENGERS (Distributed Computing using Autonomous Objects) [8] も、自律オブジェクト同士のメッセージ通信による分散システムの構成法について述べており、ある種のプログラムのインタプリタを内蔵したオブジェクトを自律オブジェクトと呼んでいる。これは、M. Jackson の言うところのプログラム可能領域である。

DEGAS [9] はルールベースの能動的なオブジェクトを自律オブジェクトと呼んでおり、この能動性を持たせたオブジェクトによるデータベースの構成法を提案している。これもやはり、M. Jackson の言うところのプログラム可能領域である。

P. Bichler らの Active Object-Oriented Database [10] では能動的なオブジェクトと振舞いを図式表現している。ここでは自律という概念はでてこないが、他の自律オブジェクトの研究と同様な能動性を持ったオブジェクトを対象としている。この研究では、一般的な受動的 (passive) なオブジェクト指向データベースに対して、イベント–条件–動作のルールによる能動的なオブジェクトを扱うデータベースが扱われている。

3.3 本研究における自律オブジェクト

このように、多くの研究では、Jackson の言う**プログラム可能**なオブジェクトに対して、**自律 (Autonomous) 的**オブジェクトや**能動 (Active) 的**オブジェクトと呼んでいる。

本研究における自律オブジェクトは、 M. Jackson の言うところの自律ではなく、プログラム可能なオブジェクトというレベルの能動性を持たせたオブジェクトを示している。

しかしながら、本研究の対象とする領域では、オブジェクトが存在する空間そのものが利用者から見ると閉じた別の空間であり、複数人で利用するため刻一刻と状態が変わっている。このため、利用者を外部世界と考えると、オブジェクト空間は利用者の関知しないところで動作をする場合があることから、一種の自律的動作とも言える。

第 4 章

分散自律オブジェクト実行環境

本章では、本論文で対象とするドメインである分散自律オブジェクト群とその実行環境に関する要求事項を、その一例である「自在」構築を考えて洗い出し、デザインパターンをどのように適合させるかを議論する。まず、自在フレームワークとツールキット双方について要求事項を洗い出し、デザインパターンの適用の可能性を探る。その後、簡単なプログラム (自在に必要となるであろう機能の一部を持つもの) を試作することで、デザインパターンの適用可能性を確かめ、同時にデザインパターンのみでは不十分と推測される事項を検討する。これによって得られた経験や知見をもとにデザインパターンを適合修正した**自在パターン**を示す。

4.1 自在の要求事項

「自在」の様な環境特有の必要となる機能がある。まず、これらの必要機能をフレームワークとツールキットという 2 つの視点から分類する。そして、それぞれに対して適合する可能性のあるデザインパターンを推測する。しかし、自在特有の機能を実現するには Gamma のデザインパターンそのままでは不十分である。そのため、本章でデザインパターンに対する変更案も考察する。

4.1.1 自在フレームワーク (自律オブジェクト実行環境)

本研究の対象とする自在フレームワークは、自律オブジェクト群が動く実行環境 (サポート環境) とする。具体的には自律オブジェクト群の動作をサポートするための汎用的なオ

ブジェクトの集合となる。このため、主にオブジェクト間の通信や相互作用、自律オブジェクトの生成などが必要となる。以下に、このフレームワークに必要であろう機能と、関係するデザインパターンを示す。

- 自律オブジェクト間通信を支援
  - 目的

    自律オブジェクト間のメッセージ (事象など) 通信をサポートする。
  - 対応パターン

    Iterator, Mediator など
  - パターンの適合性

    デザインパターンでは Iterator はオブジェクトに含まれているすべての要素をたどるパターンになっている。しかし、自在では、オブジェクトの集合の中から必要なオブジェクトだけを対象にたどることが必要となる。そのために必要なものだけを取り出す仕掛けに関して変更を加えなくてはいけない。また、自在では常に要素の追加削除が起こることが考えられるが、特にこれに対処する方法を考える必要がある。
- 状態遷移の連鎖
  - 目的

    自律オブジェクト間の状態の連鎖反応をサポートする。
  - 対応パターン

    Observer、Iterator など
  - パターンの適合性

    自在では Observer だけでなく Observable をどうするかが重要となる。実際には、自在における各自律オブジェクトは observer でもあり observable でもある。しかし、デザインパターンにおける Observer パターンが対象としているのは「１対多」の依存関係で、Observer と Observable を別のものを想定している。そこで、自在では「多対多」となるように変更しなくてはならない。相互多重に参照することは避けたいので、Mediator パターンの考え方を取り

入れ、依存関係をオブジェクトにカプセル化する。また、subject と observer は両方を兼ねているため、片方だけにまとめる。

- ロールバック支援
  - 目的

    いつでも状態をロールバック出来るようにする。
  - 対応パターン

    Memento, Command など
  - パターンの適合性

    オブジェクトの内部状態を完全に隠したまま、ある時点での状態のスナップショットを作成し、いつでもその時点に戻れるようにしなければならない。Memento では内部状態を外面化することで状態を外部で保存するが、オブジェクトそのものを保存し、ロールバックのときには差し替えてしまうことで、外部に状態を漏らすことなく、ロールバックを可能にできる。
- 自律オブジェクトの生成、複製
  - 目的

    自律オブジェクトの生成、削除、複製を行う。
  - 対応パターン

    Factory Method, Composite, Template Method, Prototype など
  - パターンの適合性

    Prototype パターンでは、生成したいものがいくつかのオブジェクトの合成である場合に、それぞれのオブジェクトのプロトタイプを用意して後で合成することが可能である。しかし、バラバラに生成させて合成するよりも、組合せを指定することで合成されたオブジェクトが得られるようにすると良いのではないだろうか?

    また、分散環境を前提としているため、生成する場所を考える必要が合ったり、場合によっては生成の依頼を委譲することもありうる。このため、Factory Method などで、どのクラスが実際に生成するのかをカプセル化するようにしないといけない。

- 構造の再構成
  - 目的

    自律オブジェクトの構造 (配置) が動的に変更できるようにする。
  - 対応パターン

    Abstruct Factory, Prototype, Mediator など
  - パターンの適合性

    次に述べるネットワーク透過性の実現とも関係するが、動的な構成の生成を考える必要がある。
- ネットワーク透過性
  - 目的

    物理的な配置を気にしなくても良くする。
  - 対応パターン

    直接は無いが、Mediator などが参考になる。
  - パターンの適合性

    分散環境ではこのような場合が特に多いと考えられるため、新規のパターンが必要になる。
- 担当者を探す
  - 目的

    必要な機能を提供するオブジェクトを動的に探し、利用する。
  - 対応パターン

    Chain of responsibility, Iterator など
  - パターンの適合性

    複数の同じ機能をサポートするオブジェクトの中から、ネットワーク的に一番近いオブジェクトを捜し出して利用する必要がある。Chain of Responsibility では最初に見付かったオブジェクトに要求を依頼するが、分散環境で最適なものを探せるような仕組みを与えられるようにしなければならない。構造と照らし合わせて、探すようにしなければならない。

この中でも特に「自在」においては、状態遷移図をどのように定義するか、ロールバックをどのように処理するか、状態遷移の依存関係をどのように扱うのかといったことがらが重要であると考えられる。

また、分散をどう扱うかと言う点も考える必要がある。分散に伴うコピーの処理や見かけ上の構造と実際の構造の違いをカプセル化することも必要である。

4.1.2 自在ツールキット (自律オブジェクトキット)

本研究では、自在ツールキットは、具体的な自律オブジェクトを作成するための部品群とする。このため、オブジェクトに自律性を持たせるための機構や、オブジェクトのテンプレート、オブジェクト間通信、等と行った機能が必要となる。以下に、自在ツールキットに必要であろう機能と、関係するデザインパターンを示す。

- 状態遷移図を持つ
- 状態に応じてアクションを起こす
  - 目的

    オブジェクトに状態遷移図を持たせ、状態と状態遷移に付随した振舞いをカプセル化する。
  - 対応パターン

    State, Singleton
  - パターンの適合性

    デザインパターンでは、あるオブジェクトの状態に依存した振舞のカプセル化を行うパターンになっている。しかし、「自在」では、単独の状態だけではなく、その状態遷移によって他のオブジェクトの状態にも影響を与えなくてはならない。そのため、State パターンでは不十分となる。具体的には、ある状態に依存した振舞だけではなく、ある状態遷移に依存した振舞も記述する必要がある。つまり、状態と状態遷移の 2 つについてそれぞれ考えなくてはいけない。
- 状態を通知する
  - 目的

オブジェクトが状態遷移した場合などに、自らその状態を通知できるようにする。

- 対応パターン

  Observer, Iterator など

- パターンの適合性

  変更点はフレームワークの場合と同様

- オブジェクトの機能の動的な追加

  - 目的

    オブジェクトに動的に機能を追加削除し、仮想的に 1 つのオブジェクトとして扱えるようにする。

  - 対応パターン

    Composite

  - パターンの適合性

    Composite パターンに関しては、合成したものとしていないものを同じように扱えるようになると言う点は有効となる。しかし、Composite パターンでは、静的な合成、つまり合成するオブジェクトが静的に実装に書かれてしまうため、自在で必要とする動的な合成には変更が必要である。Java 言語の Refrection API を利用することで、パターンを拡張する。クラス内に直接合成する対象のオブジェクトの参照を書かずに、場合に応じて必要なオブジェクトを動的に結合するように変更する。

    また、自在では合成するオブジェクトには主従の関係があるため、この点に配慮した形態が必要である。

- 状態のロールバックが可能

  - 目的

    要求に応じて、オブジェクトの状態をロールバック可能にする。

  - 対応パターン

    Memento, Command など

– パターンの適合性

内部状態の履歴やアクションや遷移の履歴を内部に持つ必要があるかどうかは考える必要がある。Memento パターンでは内部状態を外面化しておき、別のオブジェクトでそれを保存しているが、オブジェクトが分散されている環境では、外部に保存等の管理を委ねるのではなく、オブジェクト自身が自分の状態を保存できるようにする必要がある。

4.2　機能とパターンの対応

ここでは、自律オブジェクト環境の必要機能とパターンの関係について整理し、デザインパターンの拡張について考察する。この各機能の一部については、その機能を的確に表現できる小規模なプログラムを用意し、実際のパターンの適用可能性などを確認した。以降では、この小規模プログラムをエッセンスプログラムと呼ぶ。

表 4.1はこの対応をまとめたもので、各カラムは左から、

- 機能：自律オブジェクト環境に必要な機能
- デザインパターン：適用できる可能性のあるデザインパターン
- 変更・拡張点：自律オブジェクト環境において、デザインパターンに変更や拡張が必要な点
- 例題：機能の一部を表現するエッセンスプログラムの例
- 適用例：自在に適用した場合の例

を表す。

例題の具体的な内容は次節で述べる。

表 4.1: 機能とパターンの関係

機能	デザインパターン	変更・拡張点	例題	適用例
オブジェクト間の相互監視	**observer**		ダブルカウンタ	中間成果物間の状態の連鎖
	1 対多 見るものと見られるものが固定	多対多 見るものと見られるものが同一 参照関係をカプセル化		
探索	**iterator**		探求的探索	必要な中間成果物を参照する
	どのように探索するかをカプセル化 探索対象：要素全て、変化しない	どれを探索するかもカプセル化 探索対象の増減に対応		
オブジェクトの合成	**composite**		合成 Object	動的機能追加
	静的参照	動的参照 スタブは使わない オブジェクトイントロスペクション		
状態遷移図	**state**		ダブルカウンタ	中間成果物の自律化
	状態に関する振舞いをカプセル化	状態遷移に関してもカプセル化		

前ページより　　表 4.1:機能とパターンの関係

機能	デザインパターン	変更・拡張点	例題	適用例
スナップショット	**Memento**			ロールバック
	オブジェクトの内部状態を外面化	内部状態を知らずに、オブジェクトそのものと過去のログを保存		中間成果物に問題が起こった場合、問題のあった個所までロールバックする。
オブジェクト間通信	**Mediator**		ダブルカウンタ	各オブジェクト間の通信
	オブジェクト間の相互作用をカプセル化	主に他のオブジェクトと組み合わせる		
オブジェクト生成	**Factory Method**			自律オブジェクトの生成削除
	どのクラスを生成するかをカプセル化	どのクラスで生成させるかもカプセル化		
複製管理	**Prototype**			共有空間と私有空間の関係
	複製作成によりインスタンスを生成	複製と原本の関係を保持する		

前ページより　　表 4.1:機能とパターンの関係

機能	デザインパターン	変更・拡張点	例題	適用例
ネットワーク透過性	**Pullout**		分散オブジェクト参照	多ホストのデータにアクセス
	Mediator を参考	参照するオブジェクトの位置をカプセル化、論理的配置と物理的配置の差異を吸収	分散したホスト上のオブジェクトを論理配置に基づき参照する	各自のワークスペースマネージャが散在するオブジェクトに透過的にアクセス

4.3　エッセンスプログラム試作

前節で示した必要であろう機能とデザインパターンを考慮して、「自在」に必要となると考えられる機能を持ったエッセンス的なプログラムをいくつか試作した。これを試作することにより、デザインパターンの不十分さの検証と必要な機能の実現方法を考察する。

まず、前節の要件をもとに具体的な要求事項を定義し、デザインパターンを考慮しつつ Java 言語によって実装を行った。この際に後に必要となるデザインパターンの変更点、不足点等を明らかにしておいた。

以下にその具体的なプロトタイプの例をいくつか挙げる。

4.3.1　ダブルカウンタ

目的

オブジェクト間の相互作用の動作とその実現方法、Observer パターンの関係を再確認する。同時に、デザインパターンでは不足する部分を確認する。

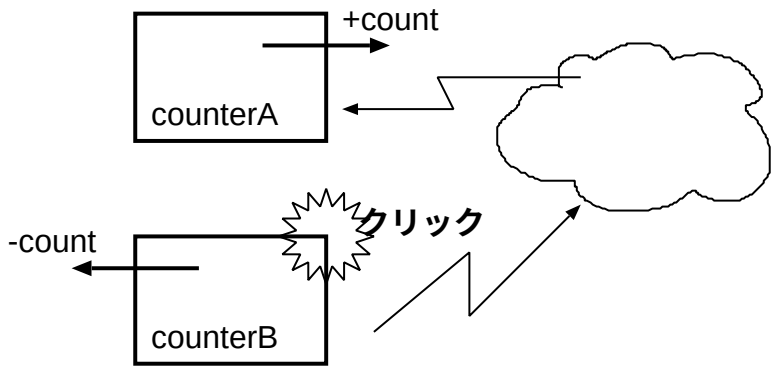

図 4.1: ダブルカウンタ

要求事項

- 図 4.1のように、2 つのカウンタが同時に動いている。
- それぞれ +カウンタと -カウンタとなる。

- 一方をクリックするとカウント方向が逆転する。他方はそれを受けて同様に逆転する。ただし、自分の動きを相手に明示的に教えることはしない。
- どちらか一方のカウンタの値が初期値と等しくなったら、カウント動作を止める。他方はそれを受けて止まる。
- (**制限**) 相手の動きを常に監視する、あるいは自分の動きを直接伝えることはしない。カウンタの数が増えた場合でも対応できるようにしておく。カウンタ自身は自分の状態の変化を誰かに伝えなければいけない事だけ知っているが、誰に何を伝えるかは知らない。

結果とまとめ

2 つのカウンタをそれぞれ observable として作り、その間の関係を知るオブジェクトを observer として作成することになった。しかし、実際には observer となったオブジェクトは受け取ったメッセージを該当するカウンタに受渡しているだけで、本質的に observe しているわけではなく、リピータという役割になっている。本質的に observer となっているのはやはりカウンタ自身であり、カウンタは observer/observable 両方の役割を果たしていると考えられる。

よって、当初の予測通り observer/observable を統合して、新たに repeater を導入するようにパターンを変更することで、本研究に適したパターンにすることができる。

4.3.2 関係要素取得

目的

あるオブジェクトに関係するオブジェクトを全て取り出すという機能と、Iterator パターンとの関係を再確認する。また、同時に Iterator パターンでは不足する部分、変更が必要な部分を確認する。

要求事項

- 図 4.2の様に多数のオブジェクトがいくつかにグループ化されている。
- 各オブジェクトはそれぞれ値を持っている。

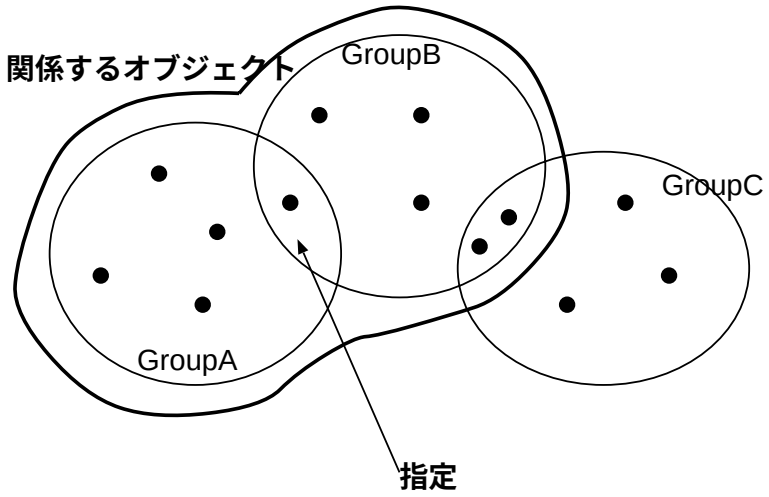

図 4.2: 関係要素取得

- あるオブジェクトを指定すると、それに関係する (グループの) オブジェクトを全てたどり、それぞれの持つ値を合計する。
- ただし、各オブジェクトは互いに参照はしない。自分がどのグループにあるかは知らない。
- 対象となるオブジェクト群はランダムに増減している。

結果とまとめ

たどる対象が常に変化するという状況を考え、iterator はその変化を知るための仕組みが必要となる。つまり、iterator と observer を組合せて、たどる対象の動的な選択をするオブジェクトを利用することで、対応できると考えられる。

つまり、仮想的なオブジェクトの集合を表すオブジェクトを用意することで、iterator 自身は、要素の変化に関して関知する必要が無くなる。

4.3.3 合成オブジェクト

目的

オブジェクト同士の動的な合成に関する手法とその可能性を確認し、Composite パターンとの関係を再確認する。また、同時に Composite パターンでは不十分な部分や拡張が必要な部分などを、Java での動的な合成の実装手法と絡ませて検証する。

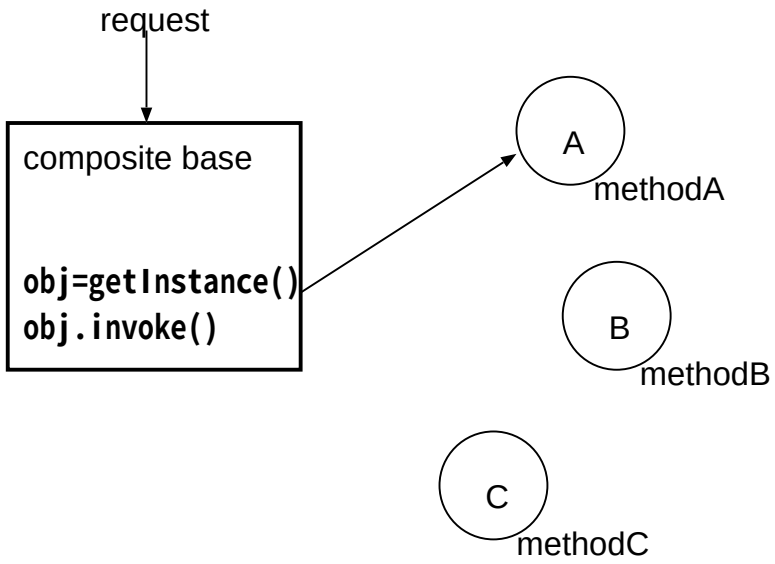

図 4.3: 合成オブジェクト

要求事項

- ある機能をを実現するいくつかのオブジェクト A を用意する。これはある決められたメソッドを持ち、それぞれユニークな名前が付けられている。
- 上のオブジェクト A は常に生成消滅の可能性がある。
- 上で用意されたオブジェクト A の一部を参照するオブジェクト B があり、指定されたオブジェクト A を合成し、あたかもオブジェクト B がその機能を持っているかのように振舞う。

結果とまとめ

言語特有な実現方法があることを確認できた。つまり、言語無依存のデザインパターンでは対応できない場合があることが確認できた。

オブジェクト同士の結合を弱めるためのパターンはデザインパターンには数多くあるが、オブジェクト合成のためには静的に結合してしまっていた。しかし、Java 言語を特有の機能を利用することで、これを動的な結合にすることができた。これをパターンとするために、メタレベルインタフェースを使う。

4.3.4 状態遷移図のカプセル化

目的

状態遷移図の定義方法とカプセル化について、State パターンとの関係を確認する。同時に、デザインパターンにおける State パターンでは不十分な点、変更が必要な点を実装を通して検証する。

要求事項

- 1 つ目のモデルを拡張する。
- 1 つ目のモデルの、各オブジェクトの状態とそれに伴う処理、状態遷移とそれに伴う処理の双方をカプセル化する。
- このとき、オブジェクト自身が状態遷移に関して深く関与しなくても良いようにする。

結果とまとめ

状態とその振舞いをカプセル化すると同時に、状態遷移中の振舞いも別にカプセル化する必要性が確認できた。具体的には、ある状態から別の状態に移る場合にほかのオブジェクトへの影響を考慮しなくてはいけないために、ある程度時間がかかってしまうことがある。これは、分散環境ではより顕著になると予想されるため、より複雑な処理が行われる。よって、この部分に関して状態のクラスから分離するべきだと考えられた。

4.3.5 分散オブジェクトの参照

目的

分散オブジェクトの参照方法とそのカプセル化について、今回新規に定義したPulloutパターンの利用可能性や有効性を確認する。同時に、デザインパターンにおける既存のパターンでは不十分な点、変更が必要な点を実装を通して検証する。

要求事項

- 分散して存在するデータを内蔵したオブジェクトが存在する。
- それらの間の論理的な配置を持つオブジェクトと物理的な配置を持つオブジェクトがある。
- Pullouterはこの２つの間の関係を保持していて、必要に応じた処理によって、分散したオブジェクトの参照を提供する。
- クライアントは物理的配置は一切知る必要は無い。

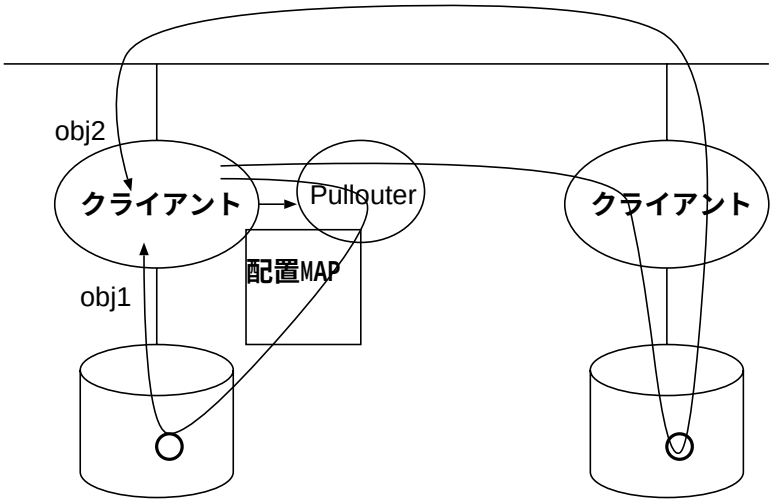

図 4.4: 分散オブジェクトの参照

結果とまとめ

分散したデータへ透過的にアクセスするためのインタフェースを提供する必要性が確認できた。データ自身は複数のデータベース等で管理されていても、利用者からの視点では論理的なデータ構造が見えることが重要である。これを透過的に扱うために、オブジェクト参照の取得をカプセル化すると同時に、ORB へのインタフェースも提供する。

これは、複数のデータベースを１つのデータベースとして扱う場合のパターンとして使うことが出来ると考えられ、同様に分散を隠すために有効であろうと思われた。

第 5 章

自律オブジェクトに対するパターン、キット、フレームワーク

本章では、自律オブジェクト群が動作する環境の構築に必要なパターン、キットフレームワークについて議論する。本論文で取り上げている「自在」はその例題である。

ここでは、まず前章の例題設計とその結果を元に、自律オブジェクトに対するパターンをまとめる。次に、このパターンを利用した自律オブジェクトキットと自律オブジェクトフレームワークを設計する際の方針について議論する。

5.1 自在パターンカタログ

自在の様な分散開発環境というドメインから見た視点と、実装時に Java 言語を利用するという視点の、両方に特化させた形でデザインパターンを利用する。ここでは、このときにデザインパターンに対して必要となる変更点と、変更したパターンについて議論する。

5.1.1 State パターン

目的

状態に依存する振舞をカプセル化するためのパターン。

適用場面

- オブジェクトに動的に状態遷移図を持たせたい場合。
- ある状態に依存した振舞をカプセル化する場合。
- 状態遷移に伴う振舞が動的に変わる場合。

デザインパターンからの変更点

デザインパターンでは、あるオブジェクトの状態に依存した振舞のカプセル化を行うパターンになっている。しかし、「自在」では、単独の状態だけではなく、その状態遷移によって他のオブジェクトの状態にも影響を与えなくてはならない。そのため、State パターンでは不足となる。具体的には、ある状態に依存した振舞だけではなく、ある状態遷移に依存した振舞も記述する必要がある。つまり、状態と状態遷移の 2 つについてそれぞれ考えなくてはいけない。

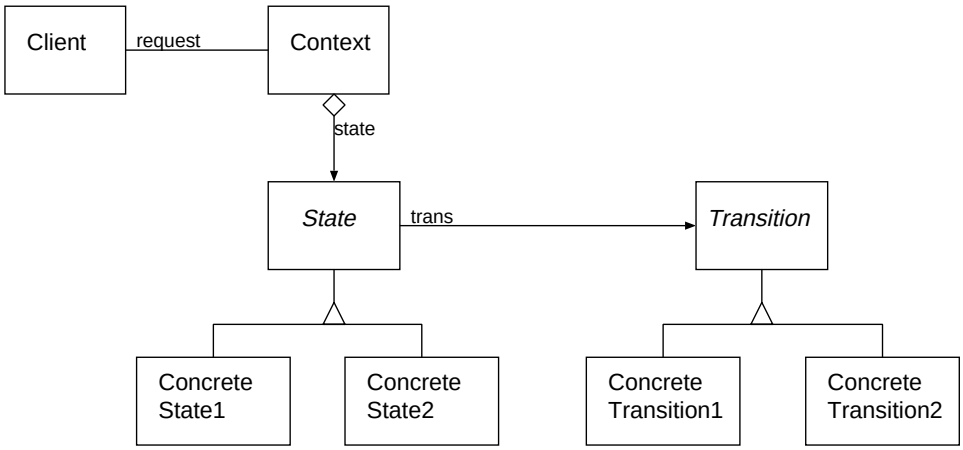

図 5.1: State パターン

5.1.2 Iterator パターン

目的

物の集合を示すオブジェクトの内部表現を隠したまま、その要素を順にたどるためのパターン。

適用場面

- 必要とするオブジェクトを順にだどって取り出す場合。
- 対象となるオブジェクトが常に変更される場合。

デザインパターンからの変更点

デザインパターンではオブジェクトに含まれているすべての要素をたどるパターンになっている。しかし、自在では、オブジェクトの集合の中から必要なオブジェクトだけを対象にたどることが必要となる。そのために必要なものだけを取り出す仕掛けに関して変更を加えなくてはいけない。また、自在では常に要素の追加削除が起こることが考えられるが、特にこれに対処する方法を考える必要がある。具体的には、対象となるオブジェクトの集合を仮想的に表現するオブジェクトを配置し、これに要素の選択や増減への対応をカプセル化することで、iterator 自身には影響が及ばないようにする。

5.1.3 Singleton パターン

目的

あるクラスのインスタンスがただ 1 つであることを保証する。

適用場面

- 各自律オブジェクトの状態を示すオブジェクトを 1 つに限定する。
- State パターンと組み合わせて使う。

デザインパターンからの変更点

特に変更の必要性はないが、Java 言語を使う場合には Singleton パターンを汎用的にするために、Class クラスのインスタンス生成に Singleton を使うようにする。具体的には、Class.forName を使ってクラス名からそのクラスのインスタンスを生成する機能を拡張し、同時にすべての生成されるインスタンスの管理をする。よって、必ずこのメソッドを使ってインスタンス生成をすることで、すべてのインスタンスの管理が行えるようにな

る。このようにすることで、それぞれのオブジェクトが Singleton を意識することなく利用でき、コードの分散も防ぐことができる。

たとえば、State パターンなどで、状態を管理するオブジェクトの生成にこれを使うことで、1 つのオブジェクトに 1 つの状態ということを保証できる。

5.1.4 Observer パターン

目的

1 対多の依存関係を定義して、あるオブジェクトが変化したときにそれに依存するすべてのオブジェクトの状態を自動的に更新する。

適用場面

- 自律オブジェクトの状態遷移を連鎖させる。
- 1 つのオブジェクトの状態が変わるとき、他のオブジェクトも同時に変化させる場合。しかも対象となるオブジェクトが固定でない場合。自律オブジェクト群の状態を監視する。

デザインパターンからの変更点

単純な Observer と Observable の関係ではなく、Observable 自身が Observer でもありうる状況をどうするかが重要となる。自在においては各自律オブジェクトは互いに参照しあっている。つまり、相互に複雑な参照関係が存在している。しかし、デザインパターンにおける Observer パターンが対象としているのは「１対多」の依存関係であり、Observer と Observable は別のものであることを想定している。よって、自在ではこの関係が「多対多」となるように変更しなくてはならない。相互多重に参照することは避けたいので、Mediator パターンの考え方を取り入れ、依存関係をオブジェクトにカプセル化する。また、Observable と Observer は両方を兼ねているため、片方だけにまとめる。この結果、状態の通知はあるオブジェクトを介して伝えられることになる。このオブジェクトを Repeater と呼ぶことにする。図 5.3では Repeater を介して ObserverObservable クラスの各インスタンスへメッセージが振り分けられることになる。

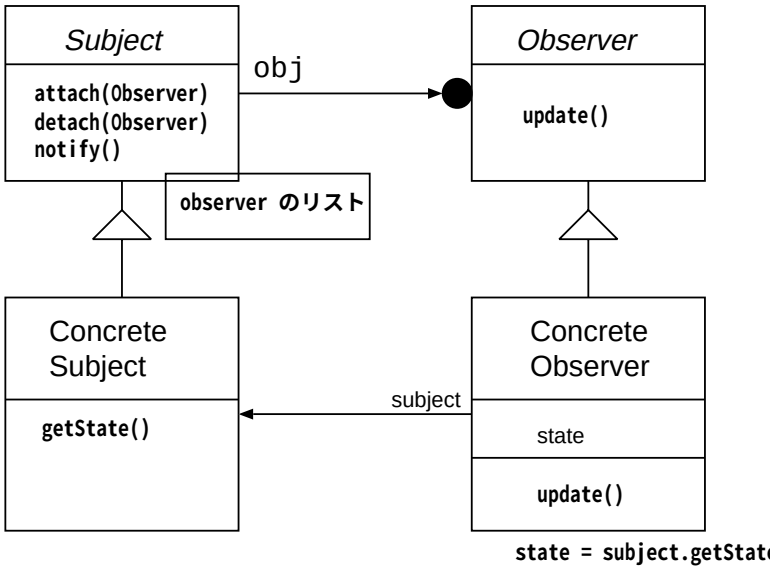

図 5.2: 元の Observer パターン

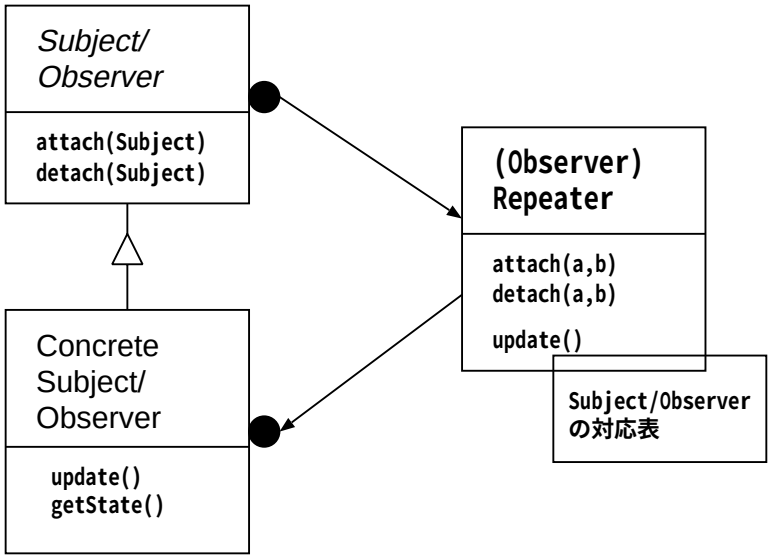

図 5.3: 変更後の Observer パターン (未完成)

5.1.5 Composite パターン

目的

合成オブジェクトをつくる。

適用場面

- あるオブジェクトに別のオブジェクトを合成し、合成する前と同様に扱えるようにしたい場合。
- 合成・分離を動的に行いたい場合。

デザインパターンからの変更点

合成したものとしていないものを同じように扱えるようになると言う点は有効となる。しかし、Composite パターンでは、静的な合成、つまり合成するオブジェクトが静的に実装に書かれてしまうため、自在で必要とする動的な合成には変更が必要である。Java 言語の Refrection API を利用することで、パターンを拡張する。クラス内に直接合成する対象のオブジェクトの参照を書かずに、場合に応じて必要なオブジェクトを動的に結合するように変更する。

また、自在では合成するオブジェクトには主従の関係があるため、この点に配慮した形態が必要である。

5.1.6 Mediator パターン

目的

オブジェクト群の相互作用をカプセル化するオブジェクトを定義する。

適用場面

- オブジェクト同士の関係をオブジェクトとは独立にしたい場合。
- オブジェクト間の関係を動的に変更したい場合。
- 個々のオブジェクトが相互参照をしないようしたい場合。

デザインパターンからの変更点

オブジェクトが他のオブジェクトを参照する場合に、必ず Mediator オブジェクトを中継することで、オブジェクト間の関係をカプセル化する。このための Mediator を通してオブジェクトにアクセスするための interface を定義しておき、すべての参照を Mediator を使うようにする。

しかし、Mediator を単独で考えることは非常に少ない。実際には他のパターンに対して Mediator を考えて、拡張するために使うことになる。

5.1.7 Prototype パターン

目的

オブジェクトを生成する場合に、元になるインスタンスを指定しそれの複製を作ることで、生成されるオブジェクトを明確にする。

適用場面

- 生成する自律オブジェクトを実行時に動的に追加・削除を行いたい場合。
- あるオブジェクトの複製を生成したい場合。

デザインパターンからの変更点

Prototype パターンでは、生成したいものがいくつかのオブジェクトの合成である場合、それぞれのオブジェクトのプロトタイプを用意して後で合成することになる。また、生成するクラスは固定となってしまう。そこで、合成されたオブジェクトを直接得たり、複数のクラスから選択して生成したりするための抽象クラスを用意する。本研究では Java 言語を対象とするため interface を使う。

5.1.8 Memento パターン

目的

カプセル化の規則を破らずにオブジェクトの内部状態を外部に見せる。

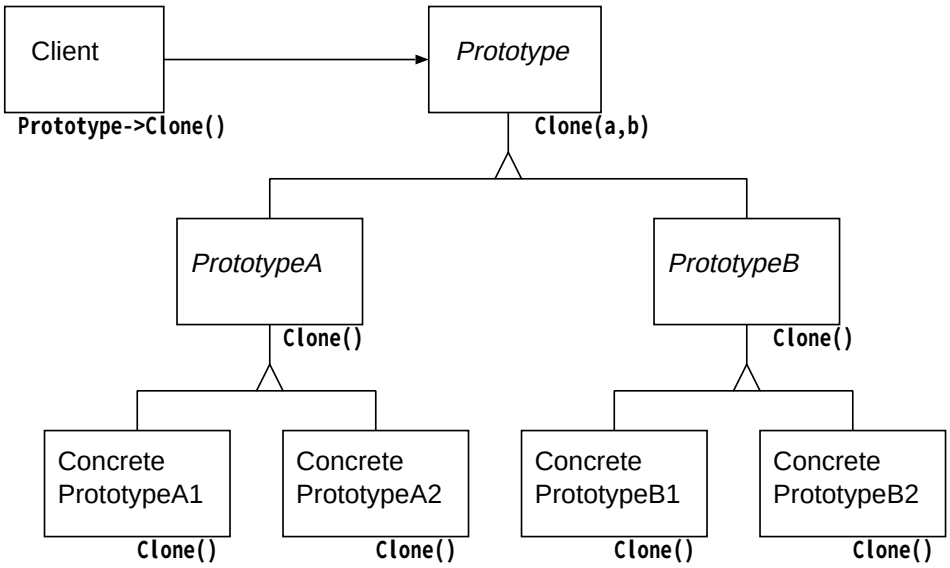

図 5.4: 変更した Prototype パターン

適用場面

- ロールバックのために、必要な時点でオブジェクトの内部状態のスナップショットをとりたい場合

デザインパターンからの変更点

デザインパターンでは、オブジェクトの内部状態を外面化することで、オブジェクトを後で元の状態に戻すことが出来るようになっている。しかし、内部状態を知ることは必ずしも必要ではなく、出来れば見えない方が良いと考えられる。このため、オブジェクトの状態遷移のログと現在のオブジェクトそのものを残すことで、元に戻すことが可能にする。具体的には、過去のアクションと遷移のログを提供する様にして、現在のオブジェクトはそのままスナップショットを取るようにする。

5.1.9 Command パターン

目的

状態遷移とそれに伴うアクションのログをとる。

適用場面

- ロールバックのためにオブジェクトが起こしたアクションのログを取れるようにする。

5.1.10 Abstract Factory パターン

目的

互いに関連し合っているオブジェクトをどのように生成するかを考えずに実装したい場合に利用する。

適用場面

- 様々な種類の自律オブジェクトの中から 1 つ選択して生成する場合。
- システムとしてどの自律オブジェクトも同様に扱いたい場合。

デザインパターンからの変更点

デザインパターンにおける Abstruct Factory では、生成する Product ごとにメソッドを用意している。このため生成できる Product が限定されてしまっており、Product の追加や変更に問題が起こる。そこで、Java 言語の持つ Reflection API を利用して、生成メソッドのパラメータによって動的なクラス指定ができるようにする。

5.1.11 Factory Method パターン

目的

オブジェクトを生成するためのインタフェースを定義し、どのクラスのオブジェクトを生成するのかをサブクラスで決める。

適用場面

- どの種類の自律オブジェクトを生成すれば良いのかあらかじめ分からない場合。
- どの種類の自律オブジェクトを利用しているのかを意識させたくない場合。

デザインパターンからの変更点

特に変更の必要はないが、Java 言語の場合は interface を利用することや、Abstruct Factory の場合と同様にクラス指定を動的にすることが挙げられる。

5.1.12 Command パターン

目的

オブジェクトに対する要求をカプセル化することで、要求の内容のログを取ったり、バッファリングしたり、パラメータとして渡せるようにしたりする。

適用場面

- オブジェクトに対する要求のリストを使って、状態のロールバックを実現したい場合。
- 要求をバッファリングして、後でまとめて処理したい場合。

デザインパターンからの変更点

Java ではオブジェクトイントロスペクションの機能を利用して、起動したいメソッドをパラメータとして渡すことが出来る。そこで、この機能を拡張する形で、メソッド起動をバッファリングしたり、ログを取ったりする機能を Command オブジェクトにカプセル化する。これを Method クラスを拡張するように実装する。

5.1.13 Pullout パターン (新規)

目的

必要なオブジェクトを取り出すための操作をカプセル化する。

適用場面

- オブジェクトの存在する位置を意識しないで参照したい場合。
- オブジェクトを参照するときに行う操作をカプセル化したい場合。

これは分散データにアクセスする場合の 3 層構造アーキテクチャを示すパターンとも言える。具体的なデータにアクセスする部分とその抽象インタフェースを持つ部分とに分ける。

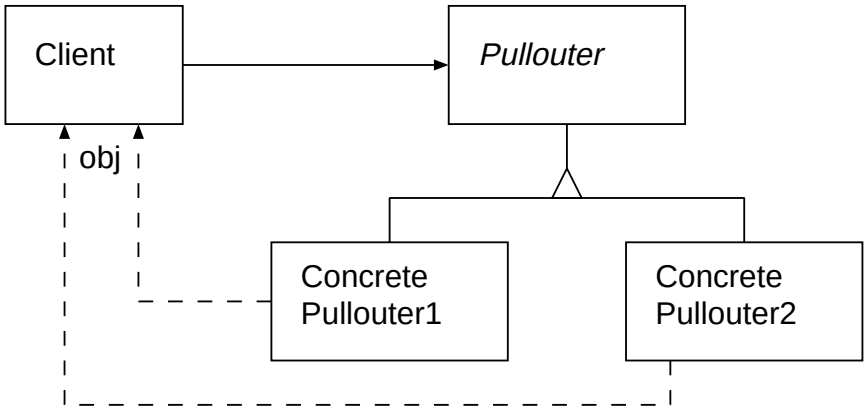

図 5.5: Pullout パターン

第 6 章

議論

この章では、デザインパターンの変更・拡張に関する方針と、このパターンのフレームワークへの適用方針について議論する。

6.1 パターンの設計方針

本論文では、アプリケーションフレームワークとキットの開発を目的としたドメインと言語に特化したパターンを、自在を例に述べてきた。ここで挙げてきた各パターンは、エッセンスプログラム等の作成を経ることで、一定の有効性は確認されているが、これらのパターンの汎用的な有用性については未確認である。

また、ドメインに特化した形でパターンを変形させてしまうことで、デザインパターンの本来持っていた再利用性の促進などの特長が失われてしまっていないかという検証は必要である。

このようにいくつかの問題はあるが、変更・拡張を行う上での方針を明確にすることで、これらの問題点を解消できると考えられる。

まず、対象とする言語として Java を選択した。このため、Java 特有のパターンの適用方針となった。具体的には、多重継承ができないため、複数の機能を持つ場合には interface を利用した。また、抽象クラスを使う場面では極力 interface を利用するようにした。また、クラスやメソッドの参照が固定的にならないように Java の Reflection API を利用するようにした。このようなことを気を付けながらパターンを変更した。

次に、分散を隠すためにカプセル化を随所に利用した。分散していることをクライアン

トから隠すことで、クライアントは普通のパターンを利用することができるようにした。例えば、データへのアクセスやメソッドの起動、オブジェクト (インスタンス) への参照といった部分に必要となるパターンに関しては、これに対応できるような変更を施した。

また、いくつかの自律オブジェクトを実現するための変更も必要となった。たとえば、自律性を持たせるために、 MVC モデルで言うところのの View が、前面に出ず、内部に View に該当するものが必要となった。

このように、パターン単位では多数のパターンに影響が及んだが、実際には、いくつかの主要な概念を元にパターンを変更、あるいは拡張したことになる。現在の段階では、この整理ができていないため、やや繁雑になってしまっているが、今後、パターンの整備が進むにつれ、より系統的なパターンのカタログとなると考えられる。

ここで、最初の問題点であるデザインパターンの変更による影響を考えてみる。前述したように、一定の方針を持って系統的にパターン設計をすることで、デザインパターンの修正・変更によるデザインパターンの利点が失われる可能性はすくないと考えられる。これに関する検証は今後の課題としておく。

6.2 フレームワークの設計方針

前章で示したパターンを、実際にフレームワークへどのように適用し、どのように設計していくかを議論する必要がある。本研究が対象とするフレームワークは、一般的な「穴の空いたメインプログラム」といった形式ではなく、自律オブジェクトが動作するために必要な機能をもった実行環境を想定している。

4章で示したように、フレームワークが必要とする機能群を、それぞれオブジェクトとして、本論文で述べたパターンを使って設計する。具体的には各機能をプロセス単位に分け、サーバとして実装する。図 6.1に自在システムの概略図を示す。

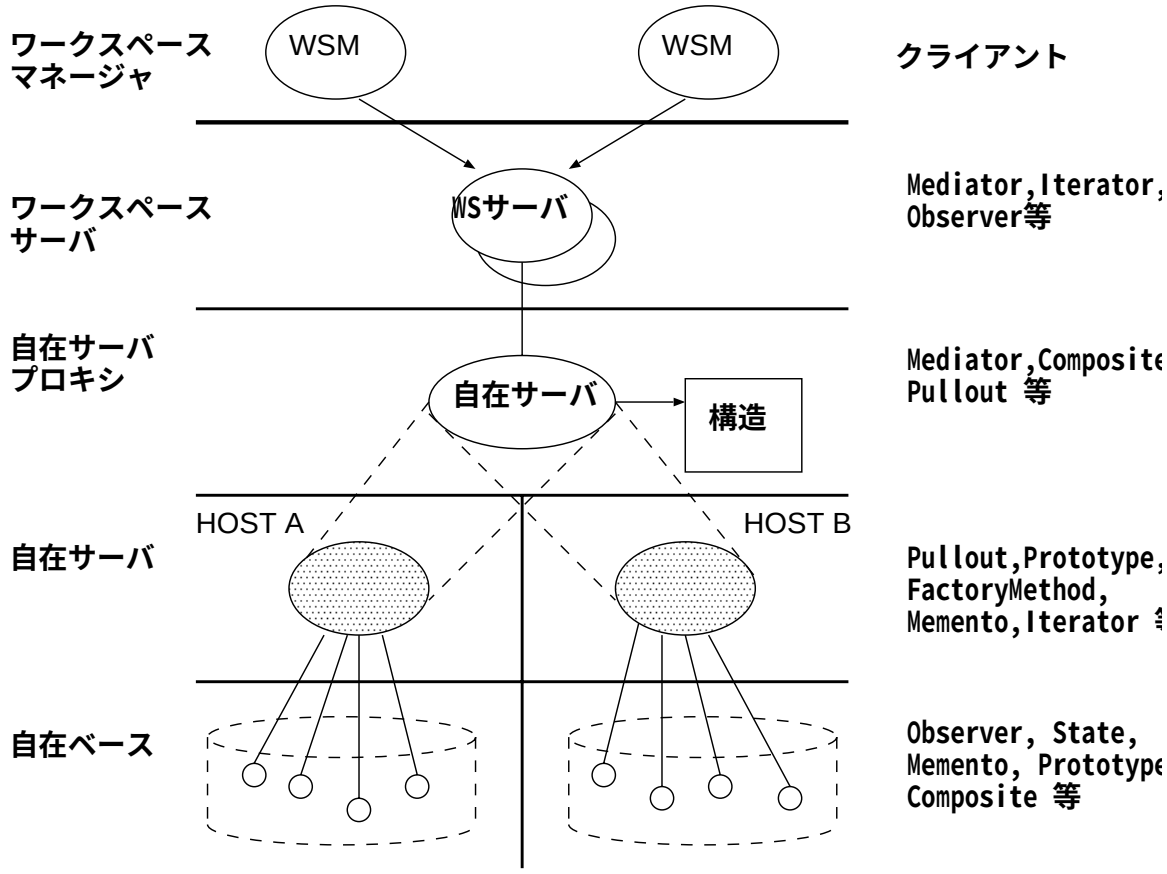

図 6.1: 自在システム概略

この図を下から順に説明すると、

自在ベース 自律オブジェクト群が存在するデータベース。データへのアクセスは自在サー

バを介して行われる。

自在サーバ 実際に自律オブジェクトをアクセスする。ホスト毎にそれぞれデータの管理を行う。自律オブジェクト間のメッセージ通信もサポートする。

自在サーバプロキシ 複数の自在サーバを透過的に扱えるようにするインタフェースを提供する。またサーバ間の間のメッセージ通信をサポートする。

ワークスペースサーバ ワークスペースマネージャ等のクライアントが自在サーバへアクセスする際のインタフェースを提供する。また、ワークスペースマネージャ間のメッセージ通信をサポートする。

ワークスペース 自在システムにおけるクライアント群。分散ワークスペースマネージャや栞がこれにあたる。

となる。主に、自在ベース、自在サーバにおいて自律オブジェクトの動作環境が提供され、これに対するクライアントからのインタフェースをその上の層で提供される。

それぞれの層の右側に示されているものが、本論文で示したパターンが適用部分である。多くのパターンは複数の場所で利用されると考えており、とくに、層間に跨った形での利用が大部分であると思われる。各層の詳細な設計は今後の課題である。

第 7 章

おわりに

7.1 まとめ

本論文では、最初に分散開発の支援環境である「自在」の開発方針について論じ、フレームワークやキットとアーキテクチャやパターンの重要性と問題点について述べた。次に、「自在」の要求事項を洗い出しデザインパターンの適合性について論じ、エッセンスプログラムの試作を通じてそれを検証した。その上で、「自在」に適合するように変更・拡張を施したデザインパターンの例を提示し、これを利用したフレームワークやキットの開発手法を提案した。

デザインパターンに対して、自在のような「分散共同開発支援環境」というドメインや言語に特化した変更や拡張を施すことにより、デザインパターンの持つ

- 粒度が細かすぎるためシステムとしてどう組み合わせて利用すれば良いか分かりにくい
- 設計のごく一般的な解法であるため、具体的な適用方法が分からない
- 設計のパターンであるため、実装が示せない
- 実装言語特有の解法に則していない

といった、問題点が以下のように解決できる可能性があることを示した。

- ドメインに特化したため、パターン間の組合せに関するパターンを示すことができた。

- パターンを統合して新たなパターンとすることで、粒度の問題を解決できた。
- 具体的な適用場面との対応を明らかにできた。
- 実装言語を Java に特定することで、パターンに対する実装を示すことができる可能性を与え、実装に則したパターンに変更することもできた。

さらに、「自在」フレームワーク／キット開発の基礎となるパターンを示すことで、これらフレームワークとキットの開発と理解が促進される可能性を与えた。

7.2 今後の課題

現在、プロトタイプ実装を踏まえて、自在パターンのさらなる整備に取り組んでいる。また、不十分なパターン、変更が必要なパターン、新たなパターンの設計を進めている。

また、本論文で提示したパターンを利用したフレームワークとツールキットのプロトタイプを実際に開発することで、開発方針とパターンの有効性・問題点などを検証する必要がある。

また、現在はパターンの拡張方針の決定がヒューリスティックであるが、これをシステマティックに行う方法に関する考察が必要である。

また、Java のもつメタレベルオブジェクトの機能を積極的に使うように考えて来たが、Class オブジェクトや Method オブジェクトを多用することの弊害についても今後考察する必要がある。

また、具体的な実装に関して、パターンカタログへの追加について考察する必要もある。

謝辞

最後に、本研究を行うにあたり、終始御指導頂きました北陸先端科学技術大学院大学情報科学研究科の落水浩一郎教授に心より感謝申し上げます。

また、論文審査あたって適切な御助言、御意見を頂きました北陸先端科学技術大学院大学情報科学研究科の篠田陽一助教授、中島達夫助教授に深く感謝申し上げます。

また、情報処理学会ワークショップにおいて御助言を頂きました南山大学情報管理学科の野呂昌満助教授に感謝申し上げます。

そして、本研究に関して多くの有意義な助言を頂きました海谷治彦助手、博士課程の藤枝和弘氏、堀雅和氏、村越広享氏、ならびに落水研究室の皆様に感謝申し上げます。

参考文献

[1] 落水浩一郎：「分散開発に適したソフトウェアプロセスモデルの構築に向けて」, FOSE'97, pp.43-50, 1997.

[2] K.Ochimizu, C.Kadowaki, M.Hori: "Design of an Information Repository to Support Cooperative Works over a Computer Network", IPSJ International Symposium on Next-Generation of Information Technologies, September, pp.79-86, 1997.

[3] E. Gamma 他著、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", Addison-Wesley 1995.

[4] M. Jackson : "Software Requirements and Specifications - a lexicon of practice, principles and prejudices.", Wokingham, Addison-Wesley, 1995.

[5] Birgitte Krogha, Lars Mathiassenb : "Coordination Problems and Method Features", IRIS 20 conference Proceedings, 1997.

[6] A. Kemper and P. C. Lockemann and G. Moerkotte and H. D. Walter : "Autonomous Objects: A Natural Model for Complex Applications", Journal of Intelligent Information Systems (JIIS), Vol.3, Nr.2, pages 133-155, 1994.

[7] N. Krivokapic, S. Grieβer, M. Islinger, M. Keidl, S. Prols, S. Seltzsam, and A. Kemper "AutO - A Distributed System of Autonomous Objects"

[8] Lubomir F. Bic Department of Information and Computer Science : "MESSENGERS: Distributed Computing using Autonomous Objects", ICS-TR-95-19, May 1995.

[9] Johan F. P. van den Akker, Arno P. J. M. Siebes : "DEGAS: A temporal active data model based on object autonomy", Report CS-R9608, ISSN 0169-118X, 1996.

[10] Peter Bichler, Michael Schrefl : "Active Object-Oriented Database Design Using Active Object/Behavior Diagram", IEEE Software, 1994.

[11] 落水浩一郎 :「ドメインモデル、アプリケーションフレームワークの構築におけるソフトウェアアーキテクチャの役割」, IPSJ Winter Workshop in Ena, pp.1-4, 1998.

[12] 野呂昌満: 「ソフトウエアアーキテクチャと新工法」, IPSJ Winter Workshop in Ena, pp.5-8, 1998.

[13] M. Shaw, D. Garlan : "Software Architecture", Prentice Hall 1996.

[14] D. Garlan, D.E. Perry : "Introduction to the Special Issue on Software Architecture", IEEE Trans. Soft. Eng., Vol.21, No.4, pp.269-274, April 1995.

[15] W.M Tepfenhart, J.J Cusick : "A Unified Object Topology", IEEE Software Jan./Feb., pp.31-35, 1997.

[16] Robert T. Monroe, Andrew Kompanek, Ralph Melton, and David Garlan : "Architectural Styles, Design Patterns, and Objects", IEEE Software Jan./Feb., pp.43-52, 1997.

[17] Norman L. Kerth and Ward Cunningham : "Using Patterns to Improve Our Architectural Vision", IEEE Software Jan./Feb., pp.53-60, 1997.

[18] M.E. Fayad and D.C. Shhmit : "Object-Oriented Application Frameworks", CACM Vol.40 No.10, pp.32-38, 1997.

[19] Alexander L. Wolf : "Software Archtecture", ACM SIGSOFT Software Engoneering Notes vol.22 no.1, pp.42-56, 1997.

[20] Doug Lea : "Concurrent Programing in Java – Design Principles and Patterns", Addison-Wesley 1997.

[21] Prieto-Diaz R. , Arango G. : “Domain Analysis and Software Modeling”, IEEE, 1991.

[22] 伊藤 潔 他 編: 「ドメイン分析・モデリング」, 共立出版, 1996.

[23] 戸松豊和 : 「Java プログラムデザイン」, SOFTBANK 1997.

[24] Peter Coad and Mark Mayfield 著, 今野 睦 監訳 : 「Java オブジェクト設計」, プレンティスホール 1997.

[25] Patrick Naughton 著, 豊田 光智衣、細井 一雄 訳 : 「Java プログラミング徹底解説」, 日経 BP 社, 1997.

[26] K.Ochimizu “Towards a Software Process Model for Distributed Cooperative Software Development”, Proc. of the 2nd ISFST, October, pp.55-62, 1997.