修 士 論 文

パターンを利用したクラスブラウザの設計と実現

指導教官　篠田 陽一 助教授

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

萩原 豊隆

1997 年 2 月 14 日

要旨

本研究では、プログラマが協調して振る舞うクラス集合を理解することを支援するため、クラスブラウザの設計および実装を行った。このクラスブラウザは、オブジェクト指向パターン Party に基づいたブラウジングが可能である。オブジェクト指向パターン Party の連鎖をたどることによって、協調して振る舞うクラス群を表現できる。また対象言語とした Java のもつインターフェース機構に対処するため、オブジェクト指向パターン Party を拡張した。

目 次

第 1 章

はじめに

本章では、研究の背景と目的を述べる。

まず、ソフトウエア開発の現状と問題点を述べる。次にその問題点の原因を明らかにする。最後にそれらを踏まえ、本研究の目的を述べる。

1.1 ソフトウエア開発の現状と問題点

本節では、オブジェクト指向プログラミング技術を導入したソフトウエア開発現場の現状と問題点を述べる。

1.1.1 オブジェクト指向技術の普及

オブジェクト指向技術は、ソフトウエアのライフサイクルのさまざまな段階で積極的に利用されるようになった。そのなかでも実装時にオブジェクト指向プログラミング言語 (以下、OOPL) を導入する例は多い。分野にもよるが、C++[11] に代表される OOPL が、大規模ソフトウエア開発の主要実装言語として、採用されるようになった [1, 2, 3]。

OOPL によるコーディングの成果物として、クラス群が蓄積される。それらのクラス群を整理し、ユーザ自身のクラスライブラリとして使用することもできる。また、そのようなクラスライブラリの実例蓄積が、言語供給元によるクラスライブラリ整備も促進することになった。

その結果、OOPL とそれに対応したクラスライブラリの利用が、実用的なコスト内での実装に不可欠となってきた。この傾向は、クラスライブラリの内容が充実、整備されてい

る GUI をもつアプリケーション開発で大きい。特にある規模以上の商用アプリケーションでは、全体的な開発効率を考えると、OOPL と商用クラスライブラリの利用が欠かせない。

このように OOPL による実装では、クラスライブラリの充実および整備という条件が不可欠な要素となる。しかし、それら充実し整備されたクラスライブラリの量自体が、新規にそのクラスライブラリを活用し、コーディングを進めようとするプログラマの障害になってきている。

1.1.2 人員追加時のオーバーヘッド

商用クラスライブラリであれば、ドキュメントの存在と、その整合性について保証はされてはいる。しかしソフトウエア資産に、既存ソースコード、さらに開発中のソースコードまで含めると、必ずしもそれらは保証されない。

既存コードであれば、ドキュメントとの整合性に疑念がある場合も存在する。また開発中のコードであれば、ドキュメントが完備は期待できない。また、ソースコード自体も流動的であることが多い。

しかしこのような状態でも、既存コードを理解し、コーディングの開始をもとめられることがある。特にプロジェクトへの人員追加などでは、十分な時間がとれない場合が多く、プログラマは既存コードを早急に理解することを求められる。

1.1.3 初期学習コストの増大

既存プログラマのすべてがオブジェクト指向プログラミングに精通しているわけではない。例えば、そのプログラマが構造化プログラミング言語に精通しており、さらに C++ 等の構造化プログラミング言語の発展型オブジェクト指向言語を使用した場合でも、得られる利点は限られる。

オブジェクト指向の原理を理解し、クラスライブラリの設計思想および構造を理解しなければ、ライブラリのもつ能力を十分に使用できない。その場合のクラスライブラリ活用は、単なるサブルーチンとして使用にとどまる。クラスライブラリの設計を再利用するような使い方はできない。

OOPL のもつ能力を十分に活用しながら実装を進めるには、ポリフォーリズムや継承などのオブジェクト指向の基本的概念を知るだけでは十分とは言えない。オブジェクト指

向設計および実装に関する典型的なパターンに関する知識が必要となる。

それらの典型的なパターンを習得には、長期にわたる学習と実践が必要である。クラスライブラリ自体も充実するに従い大規模化し、その知識習得にも長い時間が必要となってきている。

1.1.4 互換性がないクラスライブラリ

プログラマは仕事の内容により、自分の常用しているクラスライブラリとは互換性のないクラスライブラリを求められる場合がある。その場合、新しいクラスライブラリの構造を早急に知る必要が生じる。

1.2 対策方法の現状

本節では、1.1 に述べた現状と問題点を整理し、その対処法の現状を述べる。

1.2.1 問題点の整理

現状と問題点を整理すると以下のようになる。

- オブジェクト指向プログラミングでは、クラスライブラリや、既存プロジェクトで作成したクラス群が、ソフトウエア資源の再利用上重要である。
- クラス群の理解と再利用促進のため、プログラマへの支援が必要である。

さらに開発中などで、実行不能でもプログラマが支援を受けられるように、支援に必要な情報がコードから直接抽出できれば望ましい。

つぎにクラス階層によるクラス群の理解という点からクラス階層について見てみる。

1.2.2 クラス階層によるライブラリの整理

既存コードの再利用では、クラス階層によるライブラリの整理が有用である。

オブジェクト指向プログラミングでは、再利用促進のために継承による差分プログラミングを行う。そして、継承による差分プログラミングの結果、クラスの一般化／特殊化を反映したクラス階層が構成される。

しかし、クラスライブラリ等のクラス群再利用を考えると、クラス階層によるライブラリの整理は、強力ではあるが、十分とはいえない。

個々のクラスは、設計者がある目的を持って構築した協調して振る舞うクラス群の一部である。そのため、クラスを理解する上では、そのクラスだけではなく、そのクラスと協調して動作するクラス群に関する知識が必要となる。

クラス階層は、一般化／特殊化の関係を反映し、その関係の枠内における再利用には有用である。しかし、協調して振る舞うクラス群の一部としてのクラスの再利用を考えたとき、クラス階層によるライブラリの整理だけでは対処し得ない。

1.2.3 クラス群理解とデザインパターン

設計者の目的および動機のクラス群への反映では、設計という行為自体が意志の反映を含む他、意思の反映についての典型的なパターンが存在する。この種の研究には、Erich Gamma らによるデザインパターン (Design Pattern) がある [4]。デザインパターンでは、設計で現れるパターンを分類、整理、命名している。

設計者は、目的および条件から、必要される設計の雛形を、デザインパターンのカタログから選択する。これらは設計の再利用である。そして、デザインパターンによるコード生成は、設計の再利用の結果である。

そのため、デザインパターンはソースコードを整理して、そこに現れるクラス群について理解するという課題には向かない。例えば、あるコードがどのパターンに相当するかを、完全に自動化された手段で特定することできない。クラス間の構造等に現れる特徴から、一部のパターンについて、その使用を推測できるにすぎない [7]。

1.2.4 クラス群からの情報収集

ソースコード等、実装されたクラスライブラリから情報を自動抽出し、再利用コストの削減を目指した研究がある。東田によるオブジェクトパターン Party の研究では、協調して振る舞うクラス群を Party 連鎖として捉え、必要クラスの検索コスト等を抑えることを目指した [6]。ただし、東田の研究では、Party 連鎖に関する情報の抽出、またその表示に関しても、クラスライブラリ設計者の目的、動機の推測しやすさという観点を考慮していない。

1.3 研究の目的

本研究で解決を支援する問題の一例として以下のようなものを挙げることができる。

- オブジェクト指向プログラミング言語導入時の初期学習コストが掛かりすぎる。
- 人員追加に関するオーバーヘッド等を減らしたい。
- ドキュメントの信頼性に関して疑念がある。
- 開発者とメンテナンス者が違う。
- 昔のコードはあるが、担当者がいなくなってしまった。
- 商用として整理し出荷するほどでもないが、再利用したいコードがある。

1.1、1.2 で述べた現状認識および背景を基に、さらに上記の問題に対処するため、プログラマに対する支援ツールを開発する。支援ツールは、クラスライブラリとして整備されたクラス群、また以前のプロジェクトで開発されたクラス群等を対象とする。

対象となるツール開発では以下の２点を目的とする。

- 協調して振る舞うクラス群の関連に関して、その構造理解を支援する。
- クラス群設計者の目的、動機を推測するための補助をする。

また、ツール使用時のコストを抑えるため、クラス関連等の必要情報の抽出法を自動化可能な方式にする。

特に、(1) 東田によるオブジェクトパターン Party に関する成果を導入し、(2) デザインパターン等で知られる成果を導入する。という２点に考慮することにより、協調して振る舞うクラス群に対する再利用のための支援方法を検討する。

1.4 本論文の構成

２章では関連研究として、デザインパターンとオブジェクト指向パターン Party を取り上げる。さらに、デザインパターンに関連した支援ツールの現状について報告する。

３章では、クラスブラウザにおける支援法の検討を行う。

４章では、クラスブラウザの設計を、また５章ではその実装例を紹介する。
６章では、試作ブラウザのクラス群への適用結果と既存ブラウザとの比較を述べる。
７章では、結論と今後の課題を述べる。

第 2 章

関連研究

本章では、オブジェクト指向ソフトウエア開発での資源再利用の現状について報告する。特にソースコードレベルでの再利用について、クラスライブラリや既存ソフトウエアを構成するクラス群に対する再利用について報告する。

2.1 デザインパターン

クラスライブラリや既存ソフトウエアを構成するクラス群は、ある機能を担うために、協調して振る舞うクラス群ということができる。これらクラス群に関する理解には、その設計を知ることが重要となる。本節ではクラス群の設計を理解するためにデザインパターンを利用することについて論じる。

2.1.1 デザインパターンの目的

OOPL による実装例の蓄積につれ、成果物であるクラス群が蓄積された。それらクラス群の分析により典型的に出現するパターンが発見された。クラス群を分類および命名し、対象クラス群を抽象的なパターンとして捉えなおした。

このアプローチでもっとも成功したものとして、Erich Gamma らによる デザインパターン (Design Pattern) が知られている [4]。また、このデザインパターンの成功を受け、コンカレントなシステムを記述する上でのパターンを収集した事例もある [5]。

デザインパターンのアプローチは、クラス群の設計について注目した。そして設計に必要な「目的」「動機」「適用可能性」「構造」「構成要素」「協調関係」という項目によりパ

ターンを分類した。さらに、それらのパターンに「名前」「使用例」等を付け登録し、一種のカタログを作成した。

これらデザインパターンによりクラス群の設計が可能である。目的、必要条件などから、適用可能なクラス群の構造等が明らかになる。また、デザインパターンにより設計者間の情報共有が可能である。さらにクラス群をデザインパターン中で用いられる名前で識別することにより、使用されている構造等を知ることができる。

すなわち、デザインパターンの目的は、(1) クラス群の設計の知識を共有すること、(2) 汎用的な設計の型から、実装というインスタンスを生成ための知識を提供すること、という２点にあると言える。

本研究では、クラスライブラリを利用する上で大きな障害となる、(1) 設計者の目的および動機の理解、(2) クラス群の構造および協調関係の理解、という２つの問題を解決するために、枠組みとしてデザインパターンの利用を検討する。

2.1.2 デザインパターンの概要

本節では、デザインパターンの概要について説明する。概要の説明では、まずデザインパターンを記述するときの一般的フォーマットについて解説し、次にパターンの目的別分類を示す。最後に主要パターンについてその内容を説明する。

1. デザインパターンのフォーマット

   Erich Gamma らによる デザインパターンでは、デザインパターンを記述するため、以下のような項目をもつ記述形式を採用した。これらの項目に沿ってデザインパターンを記述することにより、情報の均質性と、学びやすさ、さらに使いやすさが向上するとしている。

   - **パターン名と分類**

     パターン名とパターンの本質を簡潔に連想させるものである。良い名前を付けることはきわめて重要である。設計用語の語彙に新たに加えられることになるからである。

   - **目的**

     そのデザインパターンが何をするのか、そのデザインパターンの原理と意図はなにか、そのデザインパターンが扱える設計課題や問題は何か、などについて。

- **別名**

  もしあれば、よく知られた別の名前が紹介されている。

- **動機**

  設計問題、および、パターン内のクラスやオブジェクトの構造が問題をどのように解くかを記述するシナリオ。シナリオは、パターンより抽象的な記述を理解するのに役立つ。

- **適用可能性**

  そのデザインパターンはどのような状況で適用できるか、そのパターンが扱う可能性のある粗悪な設計例にはどのようなものがあるか、読者はそれらの状況をどのように認識できるか、などについて。

- **構造**

  そのデザインパターンのクラスを、OMT(Object Modeling Technique) に基づく表記法を用いて図形的に表現したもの。さらに、要求のシーケンスやオブジェクト間の協調関係を表現するためにインタラクションダイアグラムを導入している。

- **構成要素**

  そのデザインパターンに使われるクラスとオブジェクトと、それらの責任分担。

- **協調関係**

  そのデザインパターンに使われているクラスとオブジェクトと、それらの責任分担。

- **結果**

  パターンがその目的に対してどのように貢献するか、パターンを用いた際のトレードオフと結果は何か、システム構造のどの側面を単独で変化させられるのか、などについて。

- **実装**

  パターンを実装するときに注意しなければならない落とし穴、ヒント、技法や、言語に依存した問題について。

- **サンプルコード**

どのようにパターンを実装するかを、C++あるいはSmalltalkで示したコード部分。

- **使用例**

  実際のシステムで使われるパターンの例。異なる分野から少なくとも２つの例を収めている。

- **関連するパターン**

  そのパターンに密接に関連したデザインパターン、そのパターンとの重要な差違、他のどのパターンとともに使うとよいか、などについて。

2. デザインパターンで用いられる設計の原理

   Gamma らのデザインパターンは23のパターンに分類される。これらパターンには、共通する設計原理が存在する。ここでは、この設計原理について説明する。

   デザインパターンでは、以下の原理を用いた設計が多用されている。

   - インターフェースに対してのプログラミング

     プログラミングは、インターフェースに対して行うのであり、実装そのものをプログラミングするのではない。デザインパターンでは、クラスの継承と、そのクラスのインターフェースの継承を区別する。クラスの継承は、実装に対する差分プログラミングである。そして、インターフェースの継承は、クラスインスタンスの変数への代入時など、クラスの利用時に問題となるクラスの型に対する継承である。つまり、実装と、そのインターフェースによってきまる型とを区別している。

     C++などの言語では、クラスの継承とインターフェースの継承を区別していない。そのため、C++で代入互換性のあるクラス型を持たせるためには、同じ抽象クラスから目的クラスをサブクラス化しなければならない。実際、デザインパターンでは、**Command** パターンなど、多くのパターンで、代入互換のために共通の抽象クラスが使用されている。また、型互換を確保するために、クラスを基にした **Adapter** パターンなどが用意されている。

   - クラス継承よりもオブジェクトコンポジションを多用する

機能再利用のための技法として、クラス継承とオブジェクトコンポジションがある。クラス継承による機能再利用は、コンパイル時に静的に確定される。そのため、実行時にその実装方法を変更することができない。また、クラス継承階層は、実装に依存する。よって、サブクラスが問題領域に適合しないときは、親クラスに戻って変更しなければならない。その場合、他のサブクラスに悪影響がおよぶ可能性がある。

こうしたクラス継承での実装依存による、柔軟性、再利用性の制限は、オブジェクトコンポジションにより、解決できる。オブジェクトコンポジションは、オブジェクトからオブジェクトへの参照を通して、実行時に動的に定義される。そのため、実装依存度を低く抑えることが可能になる。

- オブジェクトコンポジションにおける要求の委譲

  クラス階層中での要求の委譲は、サブクラスから親クラスへ行われる。親クラスへの委譲は、Object Pascal などでは、**inherited** 命令を使うことで実施できる。

  対して、オブジェクトコンポジションにおける要求の委譲は、要求を受け取ったオブジェクトが自分自身を委譲対象へ渡す。そうすることにより、委譲したオペレーションが受けてのオブジェクトを参照できる。

  委譲を使用したパターン例としては、**State** パターン、**Strategy** パターンなどがある。

3. パターンの分類

デザインパターンカタログに掲載されているパターンは、23 種類のパターンが登録されている。これらのパターンは、その目的により、「生成に関するもの」「構造に関するもの」「振舞いに関するもの」３種類に分類できる。この分類に沿いパターンの概要を説明する。

- **生成に関するパターン**

  生成に関するパターンは５種類ある。これらのパターンは、オブジェクト生成に関する過程を抽出したものである。これらのパターンでは、**Singleton** パターンを除き、すべて、生成対象のオブジェクトとその生成を要求するクライアントの分離を図ることを目的としている。

分離方法としては、以下の２方法が存在する。

(a) サブクラス化を用いる方法

　クライアントの生成要求を処理するために、生成処理用のオブジェクトを用意する。その生成処理用オブジェクトのインターフェースは、抽象クラスとして定義される。そして、クライアントが要求したオブジェクトを生成できるように、生成処理用クラスを継承してサブクラスを作成する。実際の生成は、そのサブクラスのインスタンスで行う。

　例：**Factory Method** パターン

(b) オブジェクトコンポジションを用いる方法

　オブジェクトコンポジションにより、クライアントは実行時、動的に要求オブジェクトを取得できる。クライアントは、参照する生成処理用オブジェクトを通じて、要求オブジェクトを取得する。その場合、クライアントは、要求するオブジェクトの具体的生成法を知る必要はない。　この方法をとるパターンでは、オブジェクト生成に責任を持つ factory オブジェクトが生成される。クライアントはこの factory オブジェクトに対してオブジェクトの生成を要求する。Abstract Factory パターンでは、複数のクラス型からそれらのオブジェクトを生成するために factory オブジェクトを使用する。Builder パターンでは、生成対象のオブジェクトが多様な構成内容をもつときに、オブジェクト生成に factory(builder) オブジェクトが使用される。また、prototype パターンでは、prototype クラスとそのサブクラスがサポートする自己複製機能を用いオブジェクトが生成される。この場合、prototype 型のオブジェクト自体を factory オブジェクトとみなすことができる。

　例：**Abstract Factory,Builder,Prototype** の各パターン

生成対象オブジェクトとその生成を要求クライアントの分離を行うどちらの方法も、抽象クラスが大きな役割を果している。抽象クラスを用いることにより、実行時に動的に生成対象にあわせ、生成用のオブジェクトを選択できるようになる。

　なお、残る Singleton パターンは、あるクラスに対しそのクラスのインスタンスが１つしかないことを保証するパターンである。

- **構造に関するもの**

  デザインパターンでは、再利用のメカニズムとしてオブジェクトコンポジションを推奨している。

  オブジェクトコンポジションでは、複数のオブジェクトをまとめる、あるいは合成する。そして、その合成等により、オブジェクト群は、協調して振る舞うオブジェクトとして、複雑な機能を得ることができる。オブジェクトコンポジションは、他オブジェクトを参照するオブジェクトを通して、実行時に動的に定義される。そのため、合成の際には、オブジェクト同士がお互いのインターフェースを考慮することが必要となる。

  しかしながら、通常のオブジェクト言語では、オブジェクトの型互換はクラス継承階層のみによって決定されてしまい。実行時にオブジェクトコンポジションにより要求されるクラス型と、実装のための継承階層により決定されるクラス型とが必ずしも一致しない場合が生ずる。

  例えば、プログラムのマルチスレッド化により、既存クラスを独立したスレッドとして動作させたい場合がある。ここでもし、独立スレッドで動作するプロセスのための機構として、スレッドクラスが存在したとする。その場合、別スレッドプロセスに要求されるクラス型はスレッドクラスとなる。よって、既存クラスを別スレッドプロセスとして呼び出すために、スレッドクラスと既存クラスからの多重継承等の対策が必要となる。

  オブジェクトコンポジションでは、このような問題が発生する可能性があるため、利用時には注意しなければならない。そして、クラス群の設計の際には、１つのオブジェクトが、他の多くのオブジェクトとともに使用できるように、注意深くそのインターフェースを設計する必要がある。

  ほとんどの構造に関するパターンは、これらのオブジェクトコンポジションに関する問題の回避と、そのことによる再利用しやすいクラス群の設計を目的の一部としている。

  構造に関するパターンは７個ある。それぞれのパターンは、クラスを基にする **Adapter** パターンを除いて類似した構造を持つ。クラスを基にする **Adapter** パターンを除く構造に関するパターンは、オブジェクトを基にしたパターンである。これらのパターンでは、オブジェクトコンポジションを用いられる。

これらオブジェクトの構造に基づくパターンには、類似した目的等をもつパターンがある。類似パターンごとに各パターンを紹介する。

(a) Adapter および Bridge パターン

Adapter および **Bridge** パターンは、どちらもあるオブジェクトに対して間接的にアクセスできるようにすることで、柔軟性を向上させている。また、さらに両パターンとも、あるオブジェクトに対し、それ自身のインターフェースとは異なるインターフェースから要求を転送することに関連している。

2つのパターン間の違いは目的ある。**Adapter** パターンの目的は、2つの既存インターフェース間の非互換性を解消することに焦点をあてている。対して、**Bridge** パターンは、クラスのインターフェースとその振舞いの実装を橋渡しするものである。このパターンにより、クライアントは常に同じインターフェースで、複数の種類のクラスの実装にアクセスできるようになる。

(b) Composite,Decorator および Proxy パターン

Composite パターンと **Decorator** パターンは、ともに無制限の数のオブジェクトを組織化するために、再帰的なオブジェクトコンポジションを用いている。また、構造的にも類似している。

しかしながら、目的は異なる。**Decorator** パターンは、オブジェクトの機能追加を、動的に行えるようにする。機能追加では、継承によるサブクラス化を行わない。対して、**Composite** パターンは、多くの関連するオブジェクトを一様に扱えるようにする。また複数のオブジェクトを1つのオブジェクトとして扱えるようにするために、クラスの構造化に焦点をあてている。

また、**Decorator** パターンと同様に、**Proxy** パターンは、オブジェクトを合成し、クライアントに対しまったく同じインターフェースを提供する。しかし、**Decorator** とは、ことなり動的に特性を付加することはできない。プロキシーの目的は、対象へのアクセスを代理し、そのアクセスを制御することである。

(c) Facade および Flyweight パターン

残りの **Facade** パターンと **Flyweight** パターンは、それぞれ次のような目的を持つ。

Facade パターンは、サブシステム内に存在する複数のインターフェースに１つの統一インターフェースを与える。サブシステムの利用を容易にするための高レベルインターフェースである。

Flyweight パターンは、多数の細かいオブジェクトを効率よくサポートするためにそれらオブジェクト共有機構を利用する。

- **振舞いに関するもの**

振舞いに関するパターンは、アルゴリズムとオブジェクト間の責任分担を扱う。パターンを構成するオブジェクト群は、アルゴリズムを実行するために協調して振る舞う。本パターンでは、それらのアルゴリズムの実現のためのオブジェクト同士の責任分担を記述する。

振舞いに関するパターンのうち、クラスを基にするものは、クラス間で振舞いを分配するために継承を用いる。また、オブジェクトを基にするパターンは、オブジェクトコンポジションを使用している。それらのパターンでは、協調して振る舞うオブジェクト群同士の結合度を、必要以上に高めない方法を提供する。

なお、振舞いに関するパターンでは、振舞い自体を１つのオブジェクトとしてカプセル化し、要求をそのオブジェクトに委譲するものもある。

これら振舞いに関するパターンの幾つかについて説明する。

(a) Template Method パターン

クラスを基にし継承を用いるパターンである。このパターンでは、アルゴリズムの構造を変えずに、アルゴリズム中のあるステップをサブクラスで定義する。これは、抽象クラスでクラスのインターフェースを記述し、サブクラスでその実装を定義することに相当する。

(b) Interpreter パターン

クラスを基にし継承を用いるパターンである。言語に対して、文法表現と文を解釈するインタプリタを一緒に定義する。

(c) Chain of Responsibility パターン

オブジェクトをチェーン上に結合し、そのオブジェクト間で、要求を処理する責任を次々に委譲する。クライアントと要求処理オブジェクトの直接結合を避ける。

(d) Strategy パターン

アルゴリズムの集合を定義し、各アルゴリズムをカプセル化し、それらを交換可能にする。アルゴリズムの集合は、抽象クラスでそのインターフェースを定義し、ここのアルゴリズムの実装は、そのサブクラスで定義する。また、個々のアルゴリズム実装オブジェクトへの参照をもち、アルゴリズムの使用を制御するオブジェクトも用意する。アルゴリズムとその利用元との結合度を低める。

(e) State パターン

オブジェクトの内部状態が変化したとき、オブジェクトが振舞いを変えるようにする。クラス内では、振舞いを変化を記述せず、状態をあらわすオブジェクトを導入する。通常、状態を表わすオブジェクトのインターフェースを抽象クラスとして定義し、その振舞いをサブクラスで実装する。

(f) Command パターン

要求をオブジェクトとしてカプセル化する。オブジェクト化した要求をキューなどで保存することにより、取消し可能なオペレーションをサポートする。通常、洋弓をあらわすオブジェクトのインターフェースを抽象クラスとして定義し、その振舞いをサブクラスで実装する。

(g) Iterator パターン

集約オ　ブジェクトが内部構造を公開せずに、その要素に対し順次アクセスの方法を提供する。リスト構造などのアクセスに使用する。

(h) Mediator パターン

目的オブジェクト群の相互関係をカプセル化するオブジェクトを定義する。オブジェクト間の関係を Mediator オブジェクトに集約することにより、それ以外のオブジェクト同士の参照を取り除く。結果として、個々のオブジェクト間の結合度を低めることができる。

(i) Memento パターン

カプセル化を破壊せずに、オブジェクトの内部状態を捉えて外面化する。そして、オブジェクトを後にこの状態に戻すことができるようにする。

(j) Observer パターン
あるオブジェクトが状態を変えたとき、それに依存するすべてのオブジェクトに自動的に通知が行く。また、それらの更新のため、オブジェクト間に１対多の依存関係を定義する。通常、依存するオブジェクト群は、更新のためインターフェースをもつ抽象クラスのサブクラスとして、構成する。さらに、更新対象の状態を保持するオブジェクトも定義し、依存するオブジェクト群との相互参照をもたせる。

(k) Visitor パターン
あるオブジェクト構造上の要素で実行されるオペレーションを表現する。このパターンにより、オペレーションを加えるオブジェクトのクラスに変更を与えずに、新しいオブジェクトを定義できる。

2.1.3 デザインパターンの問題点

既存コードに注目し、そこから情報を得ると言う立場に立つと、問題が生ずる。デザインパターンは、あくまで設計のパターンである。そのため、コードからデザインパターンを抽出し、それをもってそのコードを理解するという使用方法には無理がある。

もちろん、オブジェクト指向技術に熟練した技術者が、クラスライブラリを分析すれば、そこからデザインパターンを抽出できる。しかしその場合、パターンを抽出するには、分析者がクラスライブラリを完全に理解しなければならない。そのため、自動化された手段による支援という本研究の目標には、そぐわなくないことになる。

2.2 オブジェクト指向パターン Party

本節では、協調して振る舞うクラス群から、再利用対象のクラスを検索することを目的としたオブジェクト指向パターン **Party** について論じる。

2.2.1 Party の目的

オブジェクト指向パターン **Party** は、クラスの再利用促進のため、東田により提案された [6]。**Party** の目的は、クラスライブラリからの目的クラス検索コスト削減の削減にある。

そのため、主に設計レベル概念を扱うデザインパターンと比べ、実装レベルのクラス管理に利便性を持つように定義されている。さらに、当初より Smalltalk80 の実装レベルのオブジェクト管理機構に使用することを目的としたため、計算機による自動化に適している。

2.2.2 Party の概要

Party は、協調して振る舞うオブジェクト群である。協調して振る舞うオブジェクト群には利用関係がある。実装オブジェクトのあるクラスが他のクラスと協調していれば、それらの間にメッセージ経路が確保されている。そこで、クラス間の協調をクラス間の利用関係と呼ぶ。クラス間の利用関係は以下のように定義される。

> クラスAのオブジェクトからクラスBのオブジェクトへ、少なくとも一度のメッセージパッシングが行われ、その後もメッセージ送受信が行われる状態を維持するとき、A, Bのクラス間にはAからBへの利用関係があるという。

また、この定義により、一つのクラスの利用関係に注目すると、そのクラスを中心として利用関係にあるクラス群が抽出できる。そこで、**Party** を次のように定義する。

> あるクラスとそのクラスの持つ利用関係にあるクラス群を **Party** と定義する。

また、**Party** 間の関係として、以下の２つの関係を定義する。

- 全体／部分関係

  Party(A) part-of Party(B) ： Party(A) の部分構造として、Party(B) が存在する

- 一般化／特殊化構造 (継承関係)

  Party(B) is-a Party(A) ： Party(A) を継承した、Party(B) が存在する

これらの **Party** 間の関係の連鎖をたどることによって、**Party** の連鎖が明らかになる。また、**Party** の連鎖は、協調して振る舞うクラス群を明らかにする。図 2.1に **Party part-of Party** を示す。また、図 2.2に **Party is-a Party** を示す。

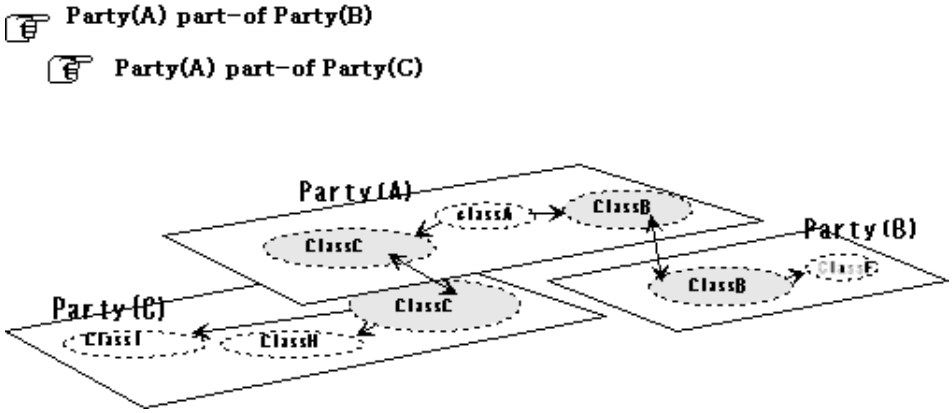

図 2.1: Party part-of Party

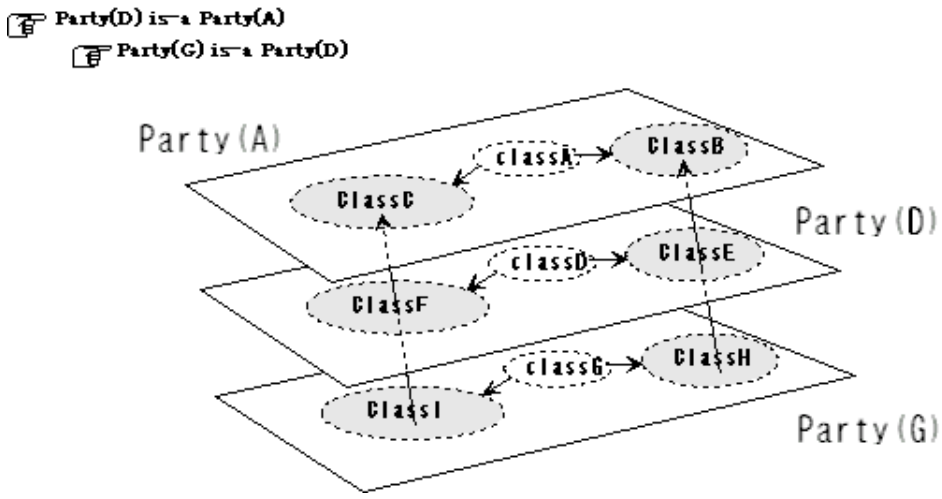

図 2.2: Party is-a Party

2.2.3 Party の利点

Pary is-a Party により、継承関係を扱える。また、**Party part-of Party** により、オブジェクトコンポジションを扱えるようになる。また、**Party** 連鎖をたどり、協調して振る舞うクラス群を認識できるようになる。これらの特徴は、オブジェクトパターン **Party** に、デザインパターンの構造的側面が扱えることを示している。

さらに、**Party** 連鎖等の関係は、計算機による自動抽出が可能である。そのため、プログラマに対する支援を、低コストで行うために有利となる。

2.2.4 Party の問題点

Party を構成するための情報は、ソースコード等から自動抽出できるものであることを前提としている。そのため、コード上に直接現れない情報は利用できない。それらの代表的な情報には、クラス群設計者の目的、動機などを挙げることができる。このことは、デザインパターンを理解するために用いるという目的には、不十分であることを示す。プログラマへの支援という点で、この不十分な部分を補うため、なんらかの対策が必要となる。

また、弱い型付け言語の **Smalltalk**[10] におけるクラス階層とは、実装の継承構造を指す。そして、**Smalltalk** を題材として考案されたオブジェクトパターン **Party** の Party is-a Party も、実装の継承構造を問題としている。そのため、変数への代入互換性等で問題となるインターフェースとしてのクラス型についての配慮がない。

デザインパターンでは、オブジェクトの実装と、その型とを区別して考えることを奨励している。そのため、デザインパターンの中にも、抽象クラスによりオブジェクトの型を決定し、そのサブクラスで実装を提供するという構成をとるものも多い。今後、協調して振る舞うオブジェクト群を考えた場合、**Party** に対し、インターフェースとしてのクラスの型を扱う機構の導入について、検討する必要がある。

2.3 実装レベルでのデザインパターン利用

実装レベルでデザインパターンを利用した例を取り上げ、その問題点を述べる。利用例として２つのものを検討する。第１の例として、既存クラスライブラリからのデザインパ

ターン自動検出をとりある。第２の例として、デザインパターンの実装時利用の支援例を取り上げる。

2.3.1 デザインパターンの自動検出

デザインパターン自動検出の研究は、Kyle Brown によってなされている [7]。この自動検出に関する研究では、リバースエンジリアリングを目的として、Smalltalk クラスライブラリからのデザインパターンの自動検出を試みている。

1. オブジェクト群の関連

   Kyle Brown は、デザインパターンにおける協調して振る舞うオブジェクト群の関連について以下の３つの関係を挙げている。

   - **クラス継承関係による定義**

     多くのデザインパターンは、クラスの継承関係についての定義を含む。一例として、Decorator パターン、また Composite パターンを挙げることができる。これらのパターンでは、抽象クラスとその特殊化した２つの子孫サブクラスでの定義に依存している。

   - **オブジェクト同士の集合および関連**

     集合 (Aggregation) とは、インスタンス変数により、強くオブジェクトが参照されている状態を指し、関連は一つのオブジェクト型一つのオブジェクト型が他のオブジェクト型を知っているという弱い関係である。

   - **メッセージに関する情報**

     多くのパターンは、クラスまたは構造化されたクラス集合のプロトコルに関する情報を含む。これらの例は、Template Method パターンに見ることができる。テンプレートメソッドパターンは、抽象クラスでインターフェースを提供し、その動作を具象クラスで定義している。その場合の抽象クラスのメソッドはプロトコルに関する情報を提供している。

2. 検出可能なデザインパターン

   上記関連については自動検出可能であっても、デザインパターンの自動検出は難しい。なぜなら、**Interpreter** など、多くのデザインパターンでは、その設計原理に

依存しており、その構造には直接依存していない。そのため、パターンの探索を試みるときは、クラス群からその意味を見いだす必要が生じる。

そのため、この研究ではデザインパターンを分析し、構造から検出可能なパターンについてのみ検出を試みている。以下に検出可能とされたパターンを挙げる。

- **Composite,Decorator**

  Composite パターンは、クライアントが個々のオブジェクトとオブジェクトを合成したものとを一様に扱うことを許す。また、**Decorator** パターンは、オブジェクトに新しい責任を追加する方法を提供する。

  それらのパターンに関するクラスの継承・構造図中に注目すると、どちらも抽象クラスと具象クラスとの間で循環路が存在する。ここで、抽象クラスと具象クラスのと関連の数が 1:1 なら **Decorator** パターンであり、1:n なら **Composite** パターンである。

- **Template Method**

  Template Method パターンはオブジェクト指向設計の基本的道具である。このパターンでは、抽象クラスでメソッドのインターフェースを定義し、サブクラスでそのメソッドの振舞いを定義する。親となる抽象クラスの仮想メソッドをオーバーライドしているサブクラスは **Template Method** パターンを構成する。

- **Chain of Responsibility**

  Chain of Responsibility パターンでは、オブジェクトをチェーン上に結合し、そのオブジェクト間で、要求を処理する責任を次々に委譲する。このような構造をとることにより、要求送信側と受信側のオブジェクトの結合を避ける。

  この **Chain of Responsibility** は、 **Decorator** と **Composite** パターンを使った結果として現れる。そのため、これら２つのパターンを合成したものになる。しかしながら、それぞれのオブジェクト間には、呼び出し関係のつながりが存在する。したがって識別には、オブジェクト－メッセージ図の動的解析が必要となる。

- **Strategy,State,Command**

これらのパターンは、異なった局面で、それぞれ別の問題を解決する。しかし、その解となる構造は類似している。これらのパターンでは、クライアントオブジェクトからサーバーオブジェクトへの参照がある。クライアントはサーバーの抽象クラスで用意されたプロトコルを用いて、サーバー抽象クラスのサブクラス群オブジェクトへアクセスする。

このようなパターンは、クライアントのインスタンス変数に注目する。そのインスタンス変数が、サーバーの抽象クラス型オブジェクトを保持する場合は、これらのパターンを使用している。

3. リバースエンジニアリングツール

リバースエンジニアリングツールについて述べる。Kyle Brown によるこの研究では、題材として **Smalltalk** を選択している。そのため、協調して振る舞うクラス群の解析には、動的型付けに対する対策が必要となる。**Smalltalk** のソースコードからではすべての必要な型付け情報を得ることは難しい。

そのため、このリバースエンジニアリングツールでは、実行時の統計的型付 (runtime statistical typing) けと呼ばれるアプローチを採用している。このアプローチでは、アプリケーションプロセスとは別の解析用プロセスを走らせる。そして、実行中の **Smalltalk** アプリケーションをサスペンドし、必要なクラスの型付け情報を得ている。

また、メッセージフローの解析問題については、**Smalltalk** 処理系の機能を用いた。ツールの実装に用いた **Smalltalk** である **Visual Works** のバイトコードを用い、オブジェクト間のメッセージパッシングをキャプチャした。これにより、ソースコード上に明示的に現れないメッセージフローを解析を解決を図っている。

4. パターン探索結果について

Composite,Decorator,Template Method の各パターンについては、自動検出が可能でると報告されている。

2.3.2 デザインパターンの実装時利用支援

デザインパターンによる自動コード生成についての研究が、F.J.Budinsky,M.A.Finnie,J.M.Vlissides および P.S.Yu によってなされている [8]。この研究では、デザインパターンの実装を自動化するアーキテクチャとツールの実装を行っている。

1. デザインパターンに基づく実装

   デザインパターンによる設計に基づいたコーディングでは以下のような問題がある。

   - **サンプルコードの断片から実装までの跳躍が難しい**
     デザインパターンは解決法の記述であるため、カタログのサンプルコードは断片的なものである。そのため、デザインパターンで記述されたサンプルコードからそのパターンを実装するコードを作成するのが困難な場合もある。
   - **コード生成がわずらわしい**
     デザインパターンからのコード生成が、困難でない場合でも、わずらわしいときがある。
   - **パターン変更による再実装**
     使用パターンを変更による設計の変更で、実質的な再実装を必要とするかもしれない。そのようなパターンの変更により、非常に異なったコードを導かれる可能性がある。

   これらのことを解決するために、デザインパターン開発ツールの設計および実装が行われた。

2. ツールの概要

   ツールの情報の表示には、WWW(World Wide Web) ブラウザを使用している。ブラウザでは、パターンについての解説やコード生成ページ等を表示し、ユーザが情報を入力することもできる。これら表示情報の表示情報の生成方法は、Perl Scripts に記述されている。生成時には Perl を用い画面情報、CGI(Common Gateway Interface) を通して表示される。

   この実装支援ツールでは、WWW ブラウザに HTML 化されたデザインパターンの記述を表示する。それら記述は、ハイパーテキスト化されているため、原文の参照

を瞬時にたどることができる。さらに、これらの情報を利用し、コード生成を開始する。パターンの制約等を確認し、使用するパターンを選択する。そしてコード生成に必要な情報を入力すると、機能定義を行うC++コードが生成され、表示される。

この支援ツールにより、パターンとコード生成の融合できるとしている。

3. ツールの問題点

ツールが生成するコードは、基本的な部分だけである。そのため、自動生成されたコードにユーザ独自のコードを追加する必要がある。しかし、一度ユーザがコードを追加すると、その部分に対するツールの支援が難しくなる。このツールでは生成したコア・クラスに対する変更を禁止し、コア・クラスのサブクラスに対する変更のみを許すという対策を取っている。しかしながら、この問題は完全に解決しているとはいえない。

第 3 章

支援法の検討

本章では、ソースコード再利用時のプログラマに対する支援法を検討する。ソースコードの再利用では、協調して振る舞うクラス群に対する再利用性を考慮する。またクラス群に関するプログラマの理解を支援することを検討する。

なお、クラス群に関する理解支援では、(1) 協調して振る舞うクラス群の関連、(2) クラス群設計者の目的および動機を推測のための補助、の 2 点を問題とする。

3.1 対象プログラミング言語

本節では、対象となる OOPL の選択基準を示し、言語の選択する。また、選択言語の概要について説明する。

3.1.1 プログラミング言語選択の基準

以下に、プログラミング言語の選択基準について明らかにする。

1. 選択のための前提事項

   本研究では、ソフトウエア実装時にプログラマする支援をする。支援は既存コードの再利用が目的である。既存コードとは、プロジェクトの成果物としてのクラス群、ソフトウエア自身を構成するクラス群、ソフトウエアから呼び出すクラス群 (標準クラスライブラリなど) などである。

   そのため、対象プログラミング言語は、クラス群を扱える OOPL とする必要がある。

2. オブジェクトの型付けによる選択

   一般にプログラミング言語は、型付けという観点から分類できる。ランボーらのOMT[9] によると、以下のようになる。

   > オブジェクト指向言語によって、型付けの扱いは大きく異なる。型付けといったとき、変数や属性の値が単にオブジェクトであるとだけ知られている (弱い型付け) 場合と、特定のクラスクラスやその子孫に属すると厳密に宣言されている (強い型付け) 場合がある。

   前者の弱い型付け言語には、Smalltalk が含まれ、後者の強い型付け言語にはC++,Java[12, 13] などが含まれる。これらを本研究が必要としている協調して動作するオブジェクト群に関する情報収集という観点から検討する。

   本研究では、協調して振る舞うクラス群に関する関連の情報を必要とする。クラス群からより多くの情報を抽出しなければならない。

   そのため、型付けという観点からプログラミング言語を選択する場合、弱い型付け言語よりも強い型付け言語の方が有利になる。強い型付け言語では、クラス定義において、 保持または参照するクラスを明示的に宣言する。それによりクラス間のさまざまな情報をプログラム実行前に抽出できる。対して、弱い型付けの言語では同様の抽出法をとれない。強い型付け言語によるものと同等の情報を得るためには、実行時の情報までが必要となる。

3. 言語普及度からみた選択

   特に問題がない限り、主として評価対象クラスライブラリ種類の数などから、より使用者数の多いプログラミング言語が望ましい。また、実用という側面を考慮しても、使用者数の多いプログラミング言語を選択することには意義がある。

3.1.2 言語の選択

3.1.1 に示したプログラミング言語の選択基準に基づき、本研究で題材とするプログラミング言語を選択した。選択候補として、Smalltalk80,C++,Java,Object Pascal[14] の４言語を挙げ、それらに対し検討を行った。

東田のオブジェクトパターン Party に関する研究では、題材として Smalltalk80 を取り上げた。しかしながら、Smalltalk80 は弱い型付け言語である。そのため、3.1.1 で検討したように、クラス群から情報を得るという用途には必ずしも適しているとは言えない。本研究では、強い型付けによって収集しえる情報量を重視し、題材として Smalltalk80 を取り上げないこととした。

残る候補言語の C++,Java,Object Pascal, については、それらはすべて強い型付け言語であり、クラスオブジェクトを持たない。オブジェクト指向言語としての基本構成は同等である。そこで、他の観点から選択する。

C++,Java,Object Pascal の３つの言語の中では C++の普及率が高い。そこで、言語普及度からの選択して、第一候補を C++とする。第２候補は C++に構文の近い Java とする。

OOPL として、C++,Java を比較すると大きな違いがある。C++は、Ｃ言語にオブジェクト指向技術を導入したものである。そのため、Ｃ言語との互換性確保等の問題から、関数や手続きなどが残っている。対して、Java は当初から OOPL として設計されたため、オブジェクト指向技術との整合性が高く、クラス群を対象とした支援機構の構築に専念しやすい。

以上のことから、本研究では題材とする言語に Java を選択することにする。

3.1.3 Java の概要

協調して振る舞うクラス群に対するプログラマの理解支援を問題にした場合、Java には、クラス階層によるライブラリの整理という基本的なオブジェクト指向の概念の他に、重要な概念が２つある。その一つはパッケージ化であり、もう一つはインターフェースである。

1. パッケージ化

   クラスは、より大規模なシステムを構造化する手段には適さない。ほとんどのオブジェクト指向言語には、クラス間の可視性を制御できるような分割機構が欠けている [9]。大規模システムのプログラミングを行う場合、作成するクラスに対する適切な可視性の制御が必要となる。可視性の制御により名前の衝突等の問題を回避できる。特に大規模システムのプログラミングでは、作成するクラス数も膨大となるため、名前衝突の危険性も高い。

パッケージはそれらの問題を解決するために、Ada や CLOS 等の言語で導入された。パッケージの導入により、ソフトウエアシステムを構成するクラス群を独立した名前空間として階層的に分割、組織化できるようになった。また、独立した名前空間の確保により、名前の衝突を避けることも可能になった。さらに、名前空間を適切に制御することにより、クラスやパッケージ間の依存関係を制御することもできるようになった。

Java においても、上記の理由を受け、パッケージ機構が採用された。なお、Java では、インターネット等でソフトウエアを配布すること等も考慮し、以下のようなパッケージの命名規則が推奨されている。

- 第 1 レベルをすべて大文字で表記する場合は、その名前をインターネットのトップドメイン名とする。つまり、米国では、COM,EDU,GOV,MIL などであり、日本では、JP となる。
- 上記以降には、インターネットドメイン名を逆順に表記したものとし、そしてその後はドメインの各組織ごとの命名規則に従うものとする。

一例として本学におけるパッケージ名の命名例を挙げると以下のようになる。

JP.ac.jaist.is.ohimizu-lab.kamekame.FCMdraw

クラス群に対するユーザの分類という観点からパッケージを見れば、パッケージはユーザの意志や必要性を反映した分類ともいえる。

2. インターフェース

インターフェースは、抽象クラスや抽象メソッドと同様に、他のクラスが実現することを期待された動作のテンプレート (型枠) を提供するものである。インターフェースにより、クラスの実装型と利用時の型を分離して扱うことができる。

インターフェースはクラス階層と独立した独自の階層を構成できる。このことにより、機能実現の階層と、設計の階層を分離できる。つまり、機能の実現は差分プログラミングによる単一継承のクラス階層によって実現し、設計の実現はインターフェースの多重継承可能なインターフェース階層によって実現できる。

具体的な例として図3.1を示す。ここではマルチスレッドプログラムの実装を行っている。TThread クラスは独立したスレッドで動作するオブジェクトを生成できる。左はインターフェースがない場合、右がインターフェースを使った場合である。どちらも Java と同様に単一継承を前提としたものである。

左側のようにインターフェース機構がない場合は、独立したとしてスレッドオブジェクトを生成するクラスは、すべて TThread クラスから継承しなければならない。そのため、既に独自の機能をもつクラスあっても、そのまま継承し機能を再利用できない (オブジェクトコンポジションを用いた Adapter パターン等を使用する必要がある)。対して、インターフェースを用いる場合は、TThread クラスとは異なったクラス階層にあるクラスも、インターフェースを継承するけで、スレッド型として動作する。

多重継承をサポートする言語の場合は、事情はやや異なる。インターフェースと同等のことは、抽象クラスと具象クラスからの多重継承により実現できる。しかしながら、インターフェースとは違い、抽象クラスはクラス階層とは独立していない。また、多重継承に特有の問題も発生する場合もある。

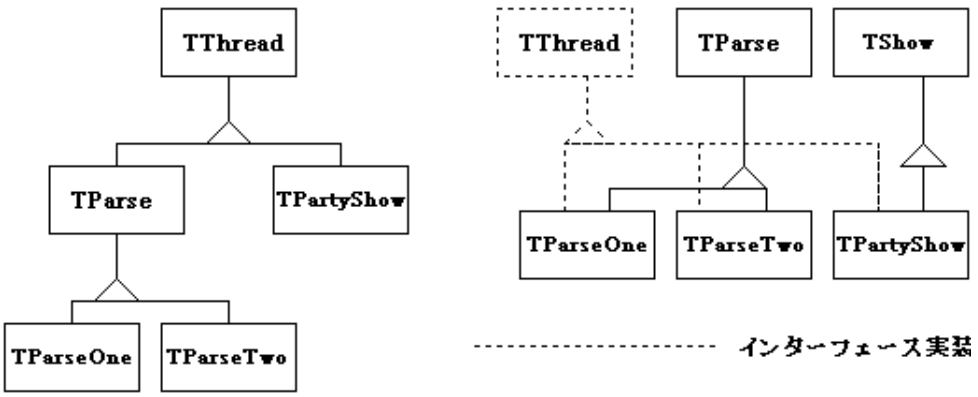

図 3.1: インターフェースの例

3.2 デザインパターンの導入

本節では、プログラマに対する支援法への使用を前提とし、デザインパターンの導入を検討する。まず題材としてとりあげる Java 言語でのデザインパターンの現れ方について検討する。また、デザインパターンを Java 言語で実現するうえで重要なインターフェースについて、議論する。

3.2.1 Java におけるデザインパターンの実装

デザインパターンでは、インターフェースに対してのプログラミングを推奨している。これは、オブジェクトのクラスと、オブジェクトの型とを区別する方法である。すなわち実装に関しては、クラス継承による階層構造で整理する。またオブジェクトコンポジションで必要な型互換確保には、抽象クラスをオブジェクトの型として使用することを勧めている。同一の抽象クラスから派生するクラス群は、同じ型を持ち、インターフェースを継承することになる。

一方、Java では言語仕様として、このようなクラスと型との区別をサポートしている。オブジェクトコンポジションで必要なオブジェクトの型は、インターフェースとしてクラスから独立して定義できる。

Java ではオブジェクトの型は、クラス定義時にインターフェース名として指定できる。指定時には implements 文を用いる。インターフェース実装の書式を以下に示す。

```
class TKame implements TUsagi { ... 定義本体... }
```

上記クラス定義では、**TKame** というクラスに、**TUsagi** というインターフェースを実装している。このインターフェースの実装により、**TKame** クラスのオブジェクトは、**TUsagi** という型を持つことができる。よって、オブジェクトコンポジションで問題となるオブジェクトの変数への代入互換性は、このインターフェースの実装によって確保できる。つまり、クラス階層によって決定される型に関らず、上記、**TUsagi** という型をもつクラスのオブジェクトは以下のように呼び出すことが可能である。

```
TKame     tmpKame = new TKame;
TUsagi    tmpUsagi;
tmpUsagi = tmpKame;
tmpUsagi.hasiru()
```

なお、インターフェースは、クラスの型を提供することを目的とするため、当然、C++の純粋仮想クラスと同様に実装を持たない。またインスタンスを生成できない。しかし、それ以外は、クラスと同様に扱うことができる。例えば、上記のようにインターフェースの型を持つ変数を宣言できる。また、インターフェースの重要な点は、クラス階層構造つまり実装によって決る型の階層とは別に、オブジェクトコンポジションに必要なオブジェクトの型を独自に設計できることである。インターフェースも継承できるので、クラス継承階層とは、独立にインターフェース継承階層を設計できる。クラスとインターフェースの例を如何に示す。

```
public interface TUsagi {

  public void hasiru();
  public void aruku();
  public void tomaru();

}

public class TKame implements {

  private int RunStep = 0;

  public void hasiru() { RunStep = 2; }
  public void aruku() { RunStep = 1; }
  public void tomaru() { RunStep = 0; }
```

```
}
```

この Java の特徴は、デザインパターンでは、抽象クラスをオブジェクトの型として扱うパターンに影響を与える。これらのパターンでは、オブジェクトコンポジションを有効に活用するため、同じ型を持つ必要がある。そのため、呼出し対象となるクラス群を同じ抽象クラスから派生さる必要がある。しかし、オブジェクトのクラスとオブジェクトの型であるインターフェースとが、言語仕様として分離している Java では、そのような配慮は必要とはならない。

以上から、対象言語として Java を選択するにあたり、デザインパターンに対するインターフェースの影響を検討することとする。

3.2.2 デザインパターンの実装に及ぼすインターフェースの影響

いくつかのデザインパターンの実例を取り上げ、インターフェースがパターンの実装に及ぼす影響を検討する。なお、取り上げるパターンは、Kyle Brown の研究で、デザインパターンのパターンの自動検出対象とされたものを中心とする。

1. **Strategy** パターン

   Strategy パターンは、アルゴリズムの集合を定義し、各アルゴリズムをカプセル化することにより、それらを交換可能にしている。このパターンでは、クライアントから独立してアルゴリズムを変更することが可能である。

   C++では、抽象クラスを用い、アルゴリズム集合のインターフェースとして型を定義できる。そして個々のアルゴリズムの振舞いについては、サブクラスで実装を行う。この抽象クラスとそのサブクラスによる構成で、各アルゴリズムの実装クラスが、代入互換のあるオブジェクトの型として定義される。

   このパターンでは、アルゴリズム全体がカプセル化されサブクラスで定義されている。そして、抽象クラス自体は、アルゴリズム集合のインターフェースにすぎない。そのため、Java 言語で実装するときには、Java の言語要素であるインターフェースを用いるのに適している。

このJavaの言語要素であるインターフェースの仕様によって、今まで共通の抽象クラスを必要としたアルゴリズムの集合が、実装に適したクラス階層を構成できるようになる。

以上のことから、Javaにおける **Strategy** パターンの実装では、Java インターフェースを用いられる可能性が非常に高いと考えられる。

なお、同様な構造を持つパターンに、**State** パターンおよび **Command** パターンがある。これらのパターンについても同様な議論が成り立つ。

図3.2と図3.3に抽象クラスを用いた場合と、Java文法要素のインターフェースを用いた場合のそれぞれについて示す。

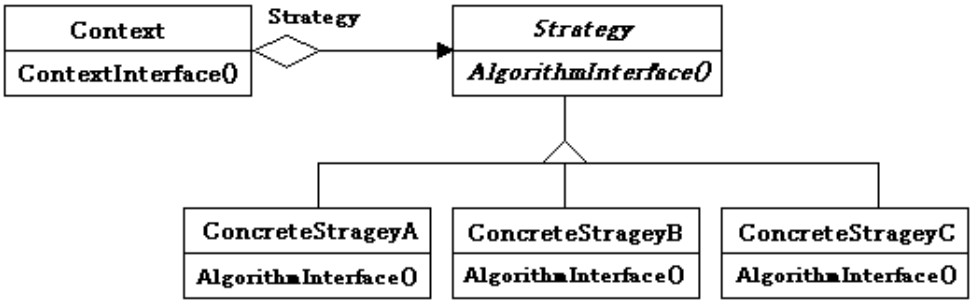

図 3.2: Strategy パターン

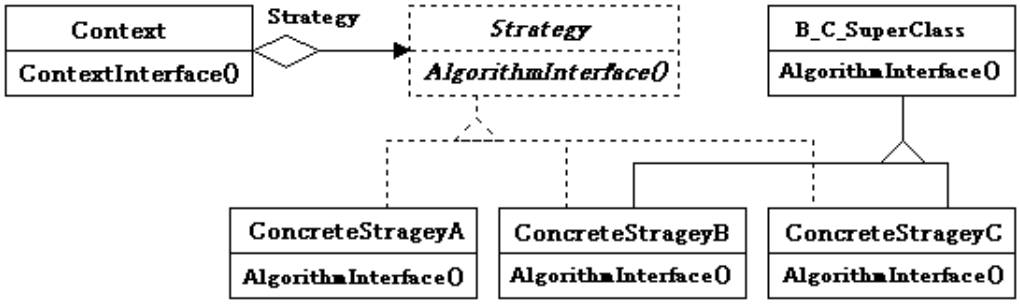

図 3.3: インターフェースを用いた Strategy パターン

2. **Template Method** パターン

Strategy パターンと似たパターンに **Template Method** パターンがある。しかしながら、**Template Method** パターンは、アルゴリズム全体ではなく、アルゴリズムの一部を変更することを主眼としている。

C++で実装する場合は、アルゴリズムのなかで変更を望む部分のみを純粋仮想関数として定義し、変更部分をサブクラスでオーバーライドし実装する。残りの関数は、抽象クラスにおいて実装しておく。

この **Template Method** パターンにおける抽象クラスの使用法は、**Strategy** パターンと違い、インターフェースとしてのものではない。オブジェクト指向概念による一般化／特殊化に基づく、機能と実装の継承である。

よって、**Template Method** パターンの Java 実装においては、Java 文法要素のインターフェースが用いられることはない。Java においても **Template Method** パターンの実装は、通常のクラス継承を用いられることになる。

3. **Composite** パターン

このパターンでは、全体－部分階層を表現するため、オブジェクトを木構造に組立てることができる。クライアントは、個々のオブジェクトとオブジェクトを合成したものとを同様に扱うことができる。

個々のオブジェクトと、それらを合成したものとを同様に扱うためには、それらのオブジェクトの型がすべて代入互換がある型である必要がある。そのため、C++では、それらのオブジェクトをすべて、同じ抽象クラスからの派生クラスとする必要がある。しかしながら、その場合、実装のためのクラス階層が型互換のための階層に依存してしまい、実装の自由度を狭めてしまう。

一方、Java 言語では、このような場合、インターフェースの実装により型に関する解決をおこなう。そのため、**Composite** パターンでは、インターフェースを用いた実装が行われると考えられる。

図 3.4と図 3.5に抽象クラスを用いた場合と、Java 文法要素のインターフェースを用いた場合のそれぞれについて示す。

なお、同様の構造を持つパターンに **Decorator** パターンがある。この **Decorator** パターンにおいても、上記議論が成り立つ。

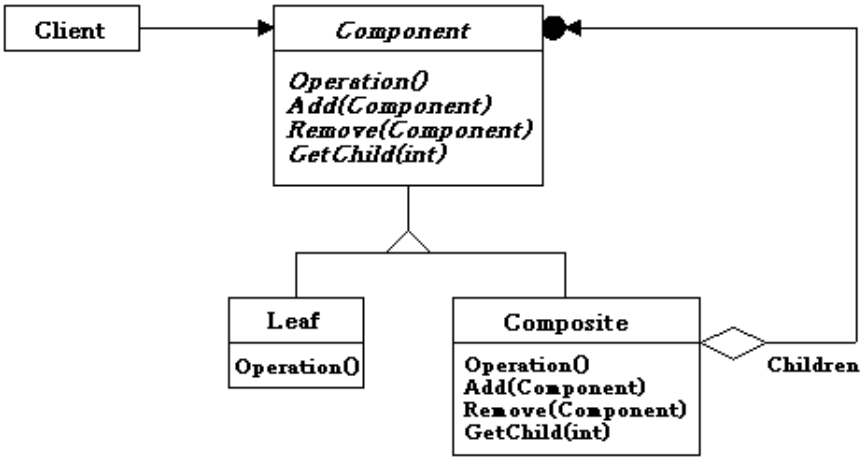

図 3.4: Composite パターン

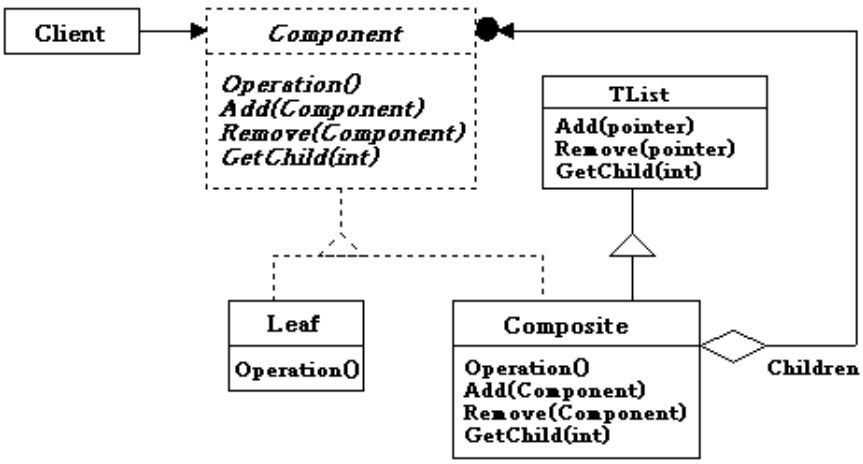

図 3.5: インターフェースを用いた Composite パターン

4. **Chain of Responsibility パターン**

Chain of Responsibility パターンでは、チェーン状に接続されたオブジェクトにそって、次々と要求処理の責任を委譲していく。そのため、要求を処理する可能性のあるクラスは、要求処理のためのインターフェースを備える必要がある。

この構造を Java で実装する場合は、Java 文法要素のインターフェースを用いることが自然である。図 3.6と図 3.7に抽象クラスを用いた場合と、Java 文法要素のインターフェースを用いた場合のそれぞれについて示す。

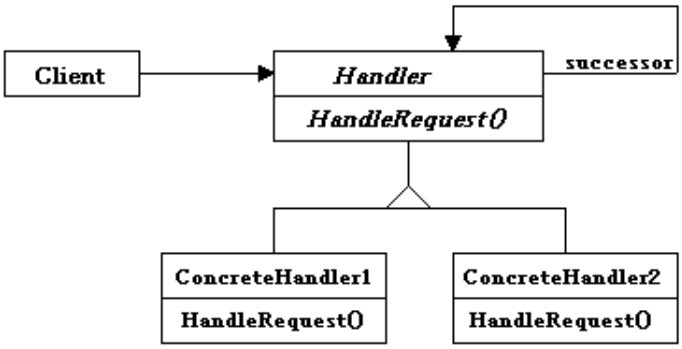

図 3.6: Chain of Responsibility パターン

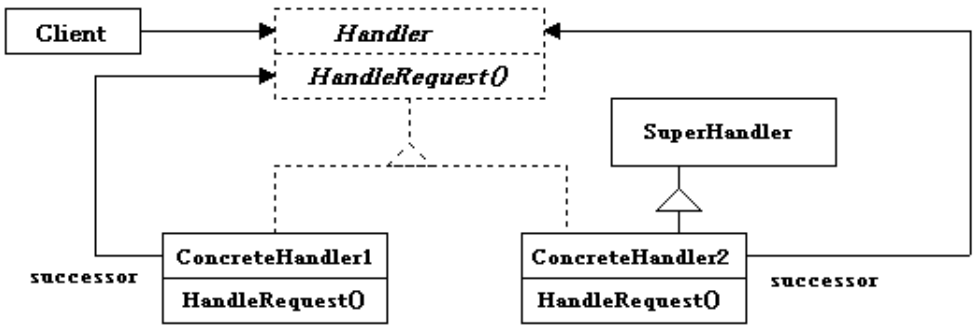

図 3.7: インターフェースを用いた Chain of Responsibility パターン

3.2.3 デザインパターンを考慮した支援法

以上の議論を踏まえ、協調して振る舞うクラス群の理解という観点から、デザインパターンによる支援を検討する。

デザインパターンでは、オブジェクトのクラスとオブジェクトの型とが明確に区別される。この区別を明確にすることが、オブジェクトコンポジションを理解することにつながる。そして、そのことは、協調して振る舞うクラス群として、デザインパターンの実装例を理解するのに役立つ。

Java にはオブジェクトのクラスとオブジェクトの型とを分離するために、インターフェース機構が用意されている。また、このインターフェースは、「デザインパターンの実装に及ぼすインターフェースの影響」で検討したように、デザインパターンに対し、間接的とはいえ、大きな影響を与える。

以上から、協調して振る舞うクラス群の理解に対し次のことを言うことができる。

- デザインパターンの知見を有効に利用するため、オブジェクトのクラスとオブジェクトの型との区別を明確にし、オブジェクトコンポジションを明らかにすることが、プログラマに対する支援として必要である
- そのためには、プログラマに対し、インターフェースの実装を適切に提示することが有効である

3.3 Party の導入

本節では、オブジェクトパターン Party の Java 言語への適用を議論する。まず Java おける Party 連鎖について検討し、次に Java 言語のインターフェースが及ぼす影響について検討する。

3.3.1 Java への Party の適用

本節では、Java 言語に対してオブジェクトパターン Party を適用するときの問題点について検討する。

オブジェクトパターン Party の定義は以下のものである。

あるクラスとそのクラスの持つ利用関係にあるクラス群を **Party** と定義する。

ここでいう利用関係の定義は、以下のものである。

クラスAのオブジェクトからクラスBのオブジェクトへ、少なくとも一度のメッセージパッシングが行われ、その後もメッセージ送受信が行われる状態を維持するとき、A, Bのクラス間にはAからBへの利用関係があるという。

この利用関係はクラスからクラスへのメッセージパッシングを対象としている。しかしながら、Java 言語では、クラスと同様な振舞いをする要素としてインターフェースが定義されている。利用関係の議論では、インターフェースに対する検討が必要になる。

Java 言語では、オブジェクトのクラスとオブジェクトの型を分離するため、インターフェースという要素を導入している。このインターフェースは、オブジェクトの型のみを提供する。クラスはインターフェースを実装することによって、オブジェクトの型として、インターフェースの型をもつことができる。

このインターフェースは、実装をもつことができない。そのため、インターフェース自体は、オブジェクトを生成できない。また、メッセージを受け取って処理することもできない。インターフェース型への参照とは、インターフェース型の変数が保持するオブジェクトへの参照である。これらのオブジェクトは、インターフェースを実装するクラス群より生成される。

インターフェースへの参照では、インターフェースを実装する複数のクラス群が生成したオブジェクトへメッセージパッシングが発生する可能性がある。そのため、これらクラス群を利用関係のあるクラス群として扱うことにする。

すなわち、利用関係に対し、以下の定義を追加する。

インターフェースAを実装するクラス群を、インターフェースの利用関係のあるクラス群とする。すなわちインターフェースとそのインターフェースを実装するクラス群は、利用関係にある。

さらに、Party の定義にインターフェースを追加することとする。

以下の関係を Party と定義する。

- あるクラスと、そのクラスの利用関係にあるクラス群およびインターフェース群
- あるインターフェースと、その利用関係にあるクラス群

なお、インターフェースは実装を持たないので、インターフェースと利用関係にあるインターフェース群はない。

3.3.2 Java における Party 連鎖

ここでは、Party 連鎖の Java に対する有効性と、Java ソースコードからの Party 連鎖の抽出を議論する。

1. Party 連鎖の Java に対する有効性

   Party 間の関係には以下のものがある。

   - 全体／部分関係

     Party(A) part-of Party(B) ： Party(A) の部分構造として、Party(B) が存在する
   - 一般化／特殊化構造 (継承関係)

     Party(B) is-a Party(A) ： Party(A) を継承した、Party(B) が存在する

   前節で、Party に対し、インターフェースの導入を行った。そのため、これらの Party 関係に対しても、インターフェースの影響を検討する必要がある。

   インターフェースが問題となる例は以下の３つである。Party(A) と Party(B) 間の関係において、

   (a) Aがクラスであり、Bがインターフェース

   (b) Aがインターフェースであり、Bがクラス

   (c) AおよびBが、ともにインターフェース

   それぞれの関係についていかに検討を行う。

(a) Aがクラスであり、Bがインターフェースである場合

Party(A) part-of Party(B) および **Party(B) is-a Party(A)** のそれぞれについて検討を行う。

- **Party(A) part-of Party(B)**

  Party 間の全体／部分関係である。この関係は、クラスAにおいてインターフェースBへの参照が発生している場合に生じる。

- **Party(B) is-a Party(A)**

  Party 間の一般化／特殊化構造 (継承関係) である。クラスからインターフェースへの継承関係が生じ得ない。そのため、この関係は発生しない。

(b) Aがインターフェースであり、Bがクラスの場合

Party(A) part-of Party(B) および **Party(B) is-a Party(A)** のそれぞれについて検討を行う。

- **Party(A) part-of Party(B)**

  Party 間の全体／部分関係である。この関係は、クラスBがインターフェースAを実装している場合に生じる。

- **Party(B) is-a Party(A)**

  Party 間の一般化／特殊化構造 (継承関係) である。インターフェースからクラスへの継承関係は生じ得ない。そのため、この関係は発生しない。

(c) AおよびBが、ともにインターフェースの場合

Party(A) part-of Party(B) および **Party(B) is-a Party(A)** のそれぞれについて検討を行う。

- **Party(A) part-of Party(B)**

  Party 間の全体／部分関係である。インターフェースは実装を持つことができない。そのため、インターフェースAからインターフェースBへのメッセージパッシングは発生しない。

  また、インターフェースBがインターフェースAを実装する場合もない。

  よって、この関係は発生しない。

- **Party(B) is-a Party(A)**

Party 間の一般化／特殊化構造 (継承関係) である。この関係は、インターフェースAからインターフェースBへの継承関係がある場合に生じる。

以上のことから、Party に対するインターフェースの導入により、成立しない Party 間の関係が生じることがわかった。

- AおよびBがともにインターフェースである場合、**Party(A) part-of Party(B)** は成立しない。
- AおよびBがクラスとインターフェースに分かれるとき、**Party(B) is-a Party(A)** は成立しない。

しかしながら、本研究で課題の一つとなる、協調して振る舞うクラス群の関係を明らかにするという観点では、十分に使用できると考えられる。そのため、これら成立しない Party 間の関係については、本研究での検討は、これ以上行わないものとする。

2. Party 連鎖の自動抽出

   Java は、強い型付けをもつプログラミング言語である。強い型付けを持つ言語では、変数や属性値が、特定のクラスやその子孫に属すると厳密に宣言される。

   よって、Party 連鎖の抽出に必要な、それぞれのクラスから他クラスに対する関連の情報は、ソースコード中に明示されているといえる。そのため、ソースコードのパースにより Party 連鎖の自動抽出が可能である。

   本研究では、Java プログラムファイルからの Party 連鎖自動抽出を試みることとする。

3.4 クラスブラウザへの支援機能の導入

オブジェクト指向プログラミングの推進を効率よく推進するには、クラスブラウザが必要となってくる。

一般的にブラウザでは、構造に基づいてソースコードを探索、吟味が可能である。また、どのようなクラスがあるか問い合わせたり、それぞれのクラスでどのような操作が定義されているかを調べることができる [9]。

しかし、既存のクラスブラウザでは、(1) デザインパターンによる知見を再利用時や実装時に利用すること、および、(2)Party 連鎖による協調して振る舞うクラス群の検索、という２点については、十分な力を発揮しない。

そこで、本研究では、3.2,3.3 で行ったデザインパターンおよびオブジェクト指向パターン Party に関する議論に基づいて、作成するクラスブラウザの設計に以下の２機能を導入する。

- **オブジェクト指向パターン Party によるブラウジング機構を導入**

  Party 連鎖の表示により協調して振る舞うクラス群の発見支援が必要である。

- デザインパターンに関する支援機能の導入

  プログラマが実装時にデザインパターン等の知見を利用できるような、ブラウザの表示方法が必要である。

第 4 章

クラスブラウジング機構の設計

本章では、クラスブラウザの設計を議論する。クラスブラウザでは、オブジェクトパターン Party およびデザインパターンの知見の導入を図る。そのことにより、協調して振る舞うクラス群に対するプログラマの理解を支援し、再利用性を向上を図る。

4.1 ブラウザの設計方針

デザインパターンの構造面では、以下の 2 点が重要である。

- オブジェクトコンポジションを理解する。
- オブジェクトのクラスと、オブジェクトの型を区別して扱う。

オブジェクトコンポジションについては、Party および Party の連鎖の抽出により、その理解を支援することができる。また、オブジェクトのクラスとオブジェクトの型の分離で重要な役割を果すインターフェースについては、既に、前章において Party に導入した。

これらのことから、クラスブラウザでの Party 連鎖表示よって、デザインパターンの構造面に関する理解の支援が期待できる。そこで、ブラウザの主要な表示内容として、Party 連鎖を採用する。

また、特に、Party 連鎖の表示では、デザインパターンに関する知見利用という観点から、オブジェクトのクラスとオブジェクトの型の区別を明確にする必要がある。そのため、Java インターフェースに関する Party については、表示時に強調することとする。

4.2　クラスおよびインターフェースのブラウジング

本節では、パッケージ構造、クラス継承構造、およびインターフェース継承構造に対するブラウジングについて検討する。

4.2.1　パッケージのブラウジング

Java 言語では、クラスおよびインターフェースの管理構造としてパッケージを導入している。そのため、本ブラウザでもこの管理構造をサポートする必要がある。

パッケージによる管理構造は、ツリー構造で表現することができる。そこで、ブラウザでのパッケージの表示には、ツリー表示を採用することとする。

4.2.2　クラス継承構造のブラウジング

オブジェクト指向プログラミングで、クラスの継承構造を理解することは、差分プログラミングによる再利用を図る上でも大切である。本ブラウザにおいても、クラス継承構造の表示を行う。

Java 言語における継承は、単一継承である。そのため、ある一つのクラスを見た場合、そのクラスが直接継承しているスーパークラスの数は０または１個となる。この特徴により、継承構造をツリー構造として表現できることとなる。そこで、本ブラウザでは、継承構造の表示には、ツリー表示を採用することにする。

4.3　インターフェース継承構造のブラウジング

Java 言語では、オブジェクトのクラスとオブジェクトの型の分離のため、インターフェースを採用した。このインターフェースは、インターフェース自体の継承をサポートしている。また、インターフェースは実装を持たないので、多重継承をおこなっても問題を生じない。そのため、Java のインターフェースの継承は、多重継承をサポートする。

多重継承をサポートするインターフェースが直接継承するスーパーインターフェースの数は、０個以上となり、１つだけとは限らない。そのため、インターフェース継承構造をクラス継承構造と同様に、ツリー表示することはできない。

しかし、本研究では、クラス継承構造ブラウジング方法との整合性を向上し、操作の統一を図るため、ツリー構造による実装を試みた。以下に対策を示す。

- 多重継承の有無に関らず、スーパーインターフェースのないインターフェースからツリー状に継承構造を表示する。
- インターフェースをツリーへ表示するとき、そのインターフェースが多重継承していた場合は、スーパーインターフェース数を同時に表示する。

この対策をとると、多重継承を行っているインターフェースが、インターフェース継承ツリー中の複数の場所に出現することになる。しかし、この問題は、他の出現場所への参照を瞬時に行えるような機能を搭載することにより解決できる。

4.4 Party 連鎖のブラウジング

デザインパターンでは、オブジェクトコンポジションが特に重要である。そこで、ブラウザの表示は次のようにする。

- オブジェクトコンポジションの可能性を表現するため、Party 間の「part-of」関係を中心とした表示とする。
- Party 間の「is-a」関係については、上記関係の表示へ、補助的に導入する。

また、Party 間の「part-of」関係は、ある Party をルートとしたツリーとして表示できる。そこで、本研究にでは、指定クラスからの「part-of」関係による Party 連鎖ツリー表示をサポートすることにする。

さらに、ツリー表示をした個々の Party の中には、「is-a」関係をもつ Party も存在する。その場合は、該当ノード形状として、「is-a」関係の存在を明示し、下位の Party への切り替え表示をサポートすることとする。このことによって、サブクラスの形成する Party 連鎖を知ることができる。

インターフェースに関しては、そのインターフェースに関する Party として、インターフェースを実装するクラス群を表示する。

図 4.1に、Party 連鎖ツリーの概要を示す。また、図 4.2に、「is-a」関係の切り替えを示す。

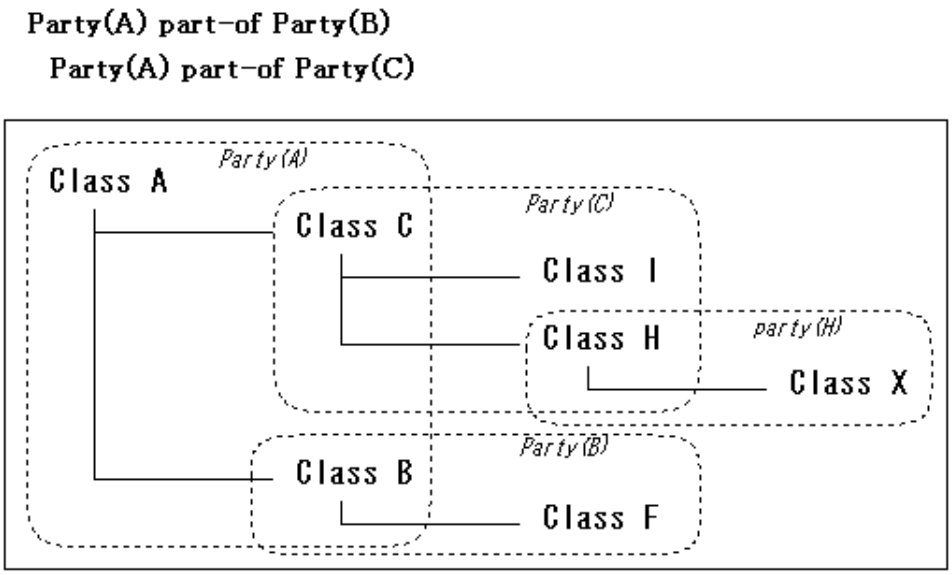

図 4.1: Party 連鎖ツリーの概要

なお、Party 連鎖のルートとなる Party を選択する必要がある。これは、ルートとなる Party を構成するクラス (もしくはインターフェース) を以下の場所から選択することで解決する。

- パッケージツリー
- クラス継承ツリー
- インターフェース継承ツリー
- 他の Party をルートとする Party 連鎖ツリー

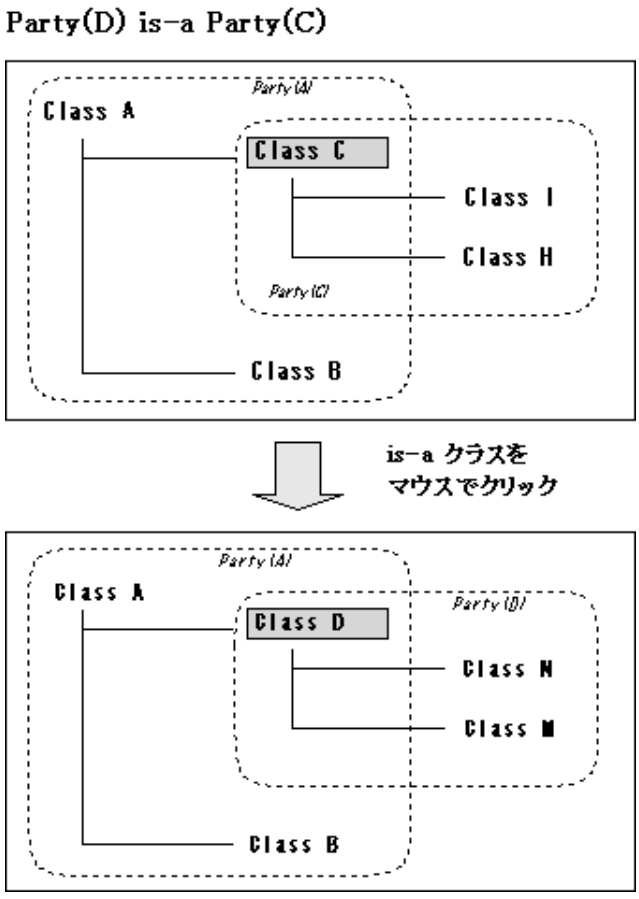

図 4.2: 「is-a」関係の切り替え

第 5 章

クラスブラウザの実装

本章では、クラスブラウザの実装例を紹介する。

5.1 開発環境の構築

クラスブラウザを有効に活用するため、連動して動作する開発環境のプロトタイプを構築した。クラスブラウザは開発環境における主要機能の一つとして位置付けられる。

クラスブラウザ以外に開発環境もつ代表的な機能としては、以下のようなものがある。

- Party 連鎖表示機能と一体化したクラスエディタ
- Java コンパイラ (javac) 呼出し機能
- クラスおよびインターフェースの雛形作成機能

ただし、クラスブラウザの作成が目的であるため、開発環境自体はプロトタイプとして必要なレベルにとどめた。

開発環境プロトタイプの概要を図 5.1に示す。

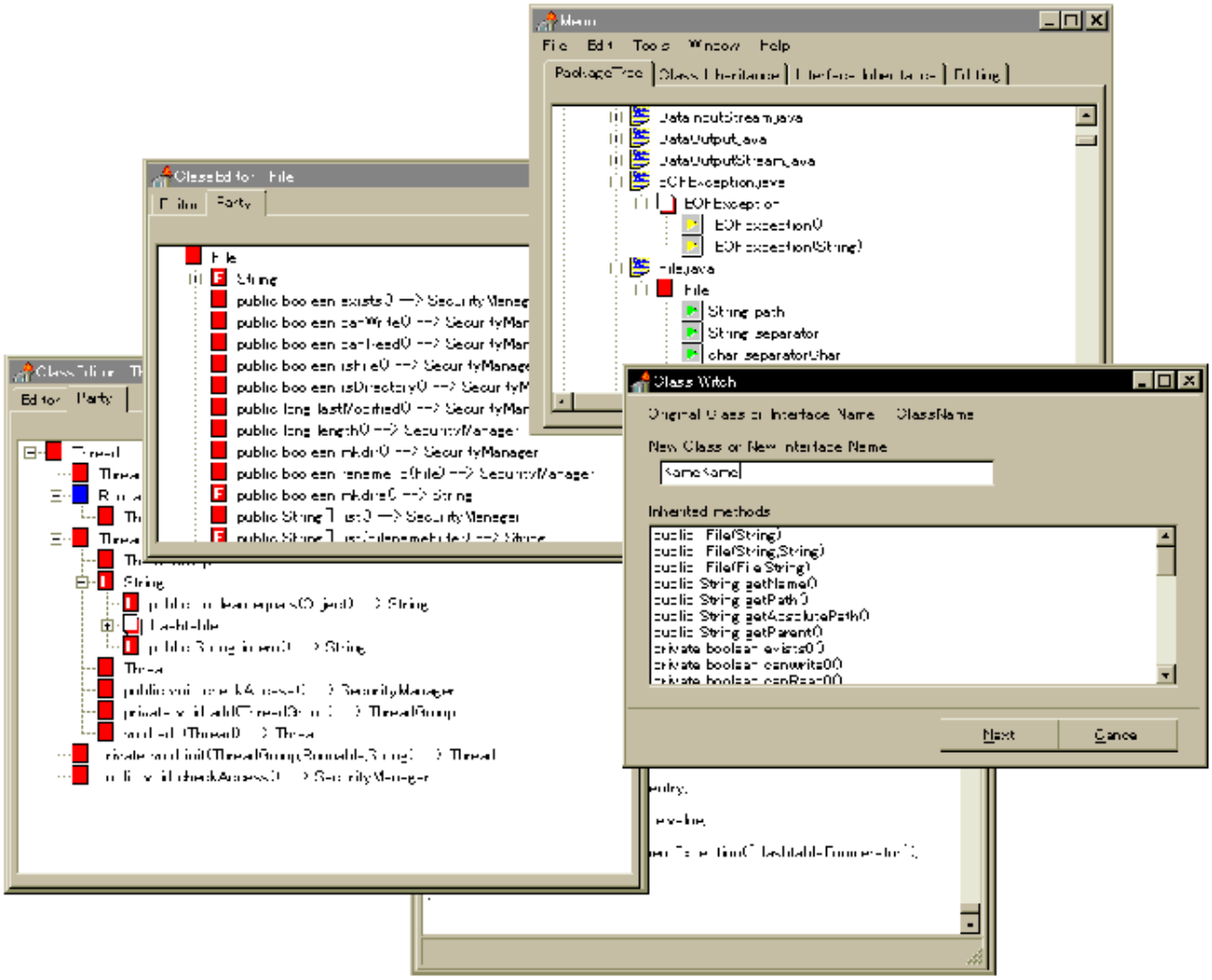

図 5.1: Java 開発環境プロトタイプの概要

図 5.1の開発環境について、説明する。

右上角のあるウインドウがメインウインドウになる。メインウィンドウでは、プログラムファイルの読込み、ツールの環境設定等を行う。また、ブラウジング対象のクラスおよびインターフェースの表示を行う。

残りのウインドウは、メインウインドウにてクラスまたはインターフェースを指定し、クラスエディタを開いたものである。クラスエディタでは、クラスの内容を記述したプログラムコードのほかに、そのクラスより発生する Party 連鎖を表示する。

メインウインドウとなるクラスブラウザを図5.2に示す。なお、このブラウザでは、Party連鎖表示をのぞいたクラスのブラウジングが行える。また、クラスまたはインターフェースを選択し、その選択要素からのParty連鎖表示も可能である。

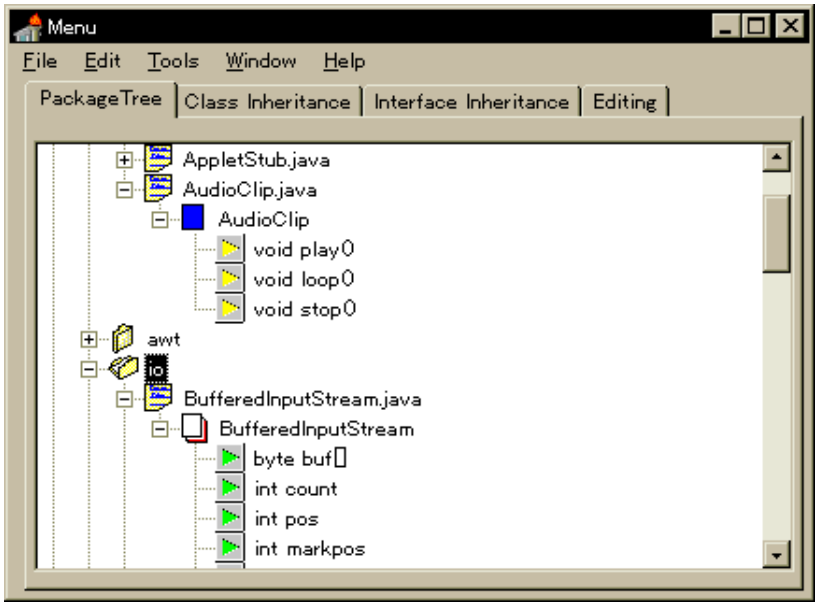

図 5.2: メインクラスブラウザ

以下にメインクラスブラウザに表示する項目を示す。

- パッケージツリー
- クラス継承ツリー
- インターフェース継承ツリー
- エディット中のクラスまたはインターフェースのリスト

これら要素の表示は、メインウインドウ中のタブを選択することによって切り替えることができる。

5.2 クラスブラウザ機能の概要

クラスブラウザとして次の３機能を実現した。

- Java のパッケージ機構に基づいたブラウジング機構
- クラス階層に基づくブラウジング機構
- インターフェース階層に基づくブラウジング機構
- Party 連鎖に基づくブラウジング機構

本節では、それぞれのブラウジング機構について解説する。さらに、クラスエディタについての解説も行う。

5.2.1 パッケージブラウジング

Java パッケージのブラウジングを行う。さらに、パッケージのブラウジングでは、パッケージに属するクラスまたはインターフェースのブラウジングが可能である。

なお、どのパッケージにも属さないクラスまたはインターフェースは、ツリーのルートへ追加される。

図 5.3に実例を示す。

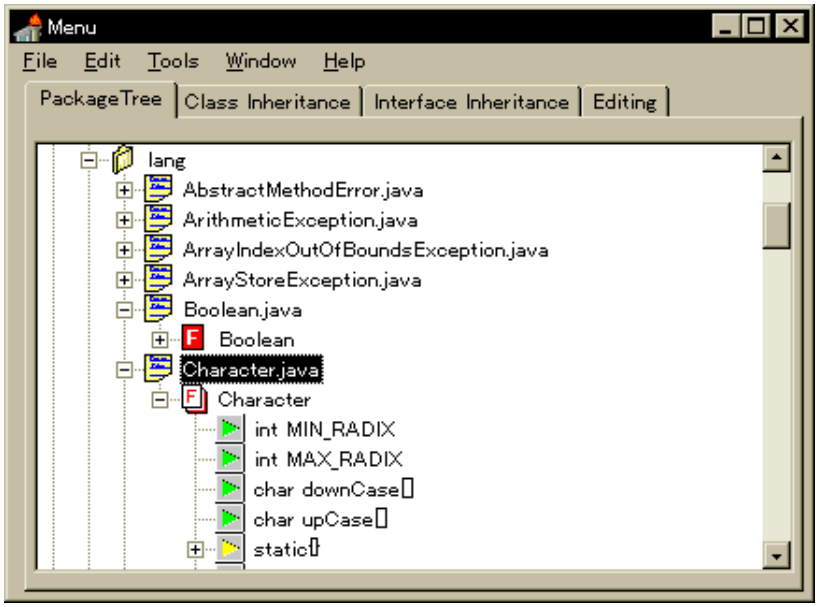

図 5.3: パッケージブラウザ

これより、パッケージツリーの個々の要素について説明する。

- パッケージ構成要素

  パッケージを構成するノードを表わす。例えば、「java.lang.awt」は「java」「lang」「awt」の 3 ノードから成り立つ。

- Java プログラムファイル

  「.java」の拡張子付きで表示される。ひとつのプログラムファイル中に複数のクラスが記述されている場合に対応するために採用した。

  Java の規格上、一つのプログラムファイルにはファイル名と同名の public クラス (または public インターフェース) を、一つだけ記述する。しかし、ファイル外から呼び出されないローカルなクラスは同一ファイル内に複数記述できるため、管理単位のひとつとして Java プログラムファイルが必要となった。

- クラスまたはインターフェース

  ブラウザ中に表示されるクラスおよびインターフェースは色によって区別する。既定値は、クラスが赤色で、インターフェースが青色の表示となる。また、これら

クラスまたはインターフェースは、継承階層中の位置付けにより表示が変わる。この機能により、継承階層中の位置を理解することが可能である。

また、そのクラスまたはインターフェースが Final 宣言をしてある場合は、アイコン中に「F」という文字が表示される。

- メソッド

  クラスまたはインターフェース中のメソッドを表示する。メソッドのスコープおよび型も表示する。

- 変数宣言

  クラスまたはインターフェース中の変数宣言を表示する。変数のスコープおよび型も表示する。

なお、このクラスまたはインターフェースを選択し、マウスを右クリックすると、図5.4のようなポップアップメニューが表示される。メニューには、以下の項目が表示される。それぞれの項目について説明する。

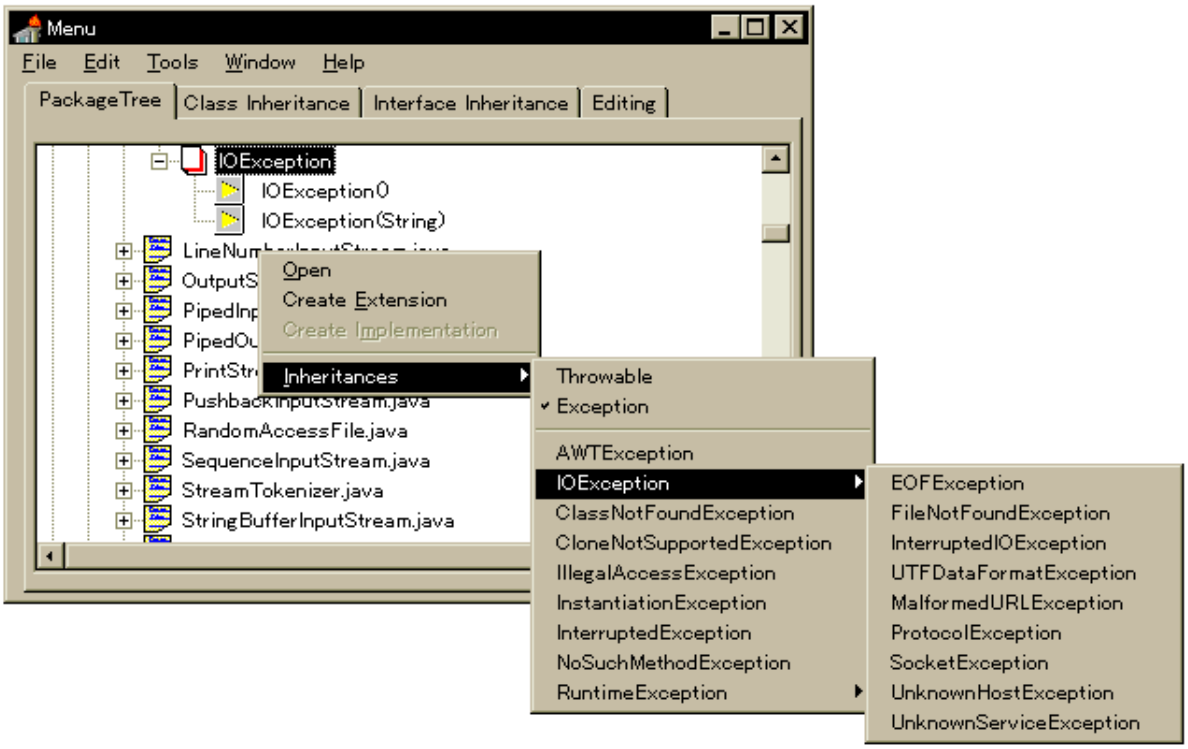

図 5.4: クラス右クリック時のポップアップメニュー表示

- クラスエディタオープンのコマンド

  クラスエディタをオープンし、指定クラスまたはインターフェースののプログラムファイルをオープンする。また、クラスエディタと同時に、ここで指定していしたものからの Party 連鎖を表示する。

- 新規クラスまたはインターフェースの作成コマンド

  指定ノード部に追加する形で、新規クラスまたはインターフェースを生成する。

- 継承構造

  指定ノードからの継承構造をメニュー項目として表示する。表示された項目を選択することにより、その選択したクラスまたはインターフェースをパッケージツリーより検索し、ツリーの該当部分を表示する。

5.2.2 クラス継承ツリー

クラス継承ツリーでは、クラス継承構造を表示する。概要を図 5.5に示す。

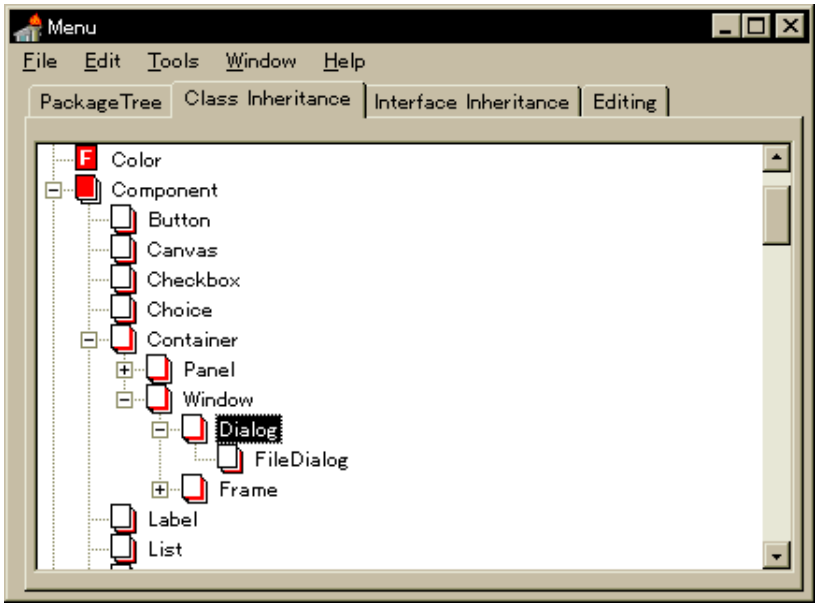

図 5.5: クラス継承ツリーブラウザ

5.2.3　インターフェース継承ツリー

インターフェース継承ツリーでは、インターフェース継承構造を表示する。多重継承がある場合は、その要素のスーパーインターフェース数も同時に表示する。概要を図 5.6に示す。

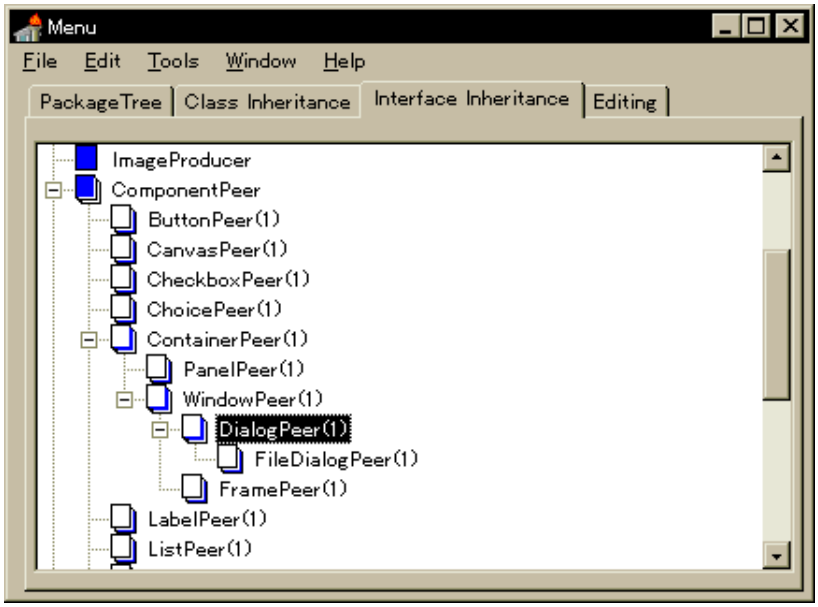

図 5.6: インターフェース継承ツリーブラウザ

5.3　クラスエディタ

メインブラウザでクラスまたはインターフェースを選択することにより表示できる。なお、エディタでは、プログラムコードの他に Party 連鎖についても確認できる。それぞれの切り替えは、エディタ内のタブをクリックすることによって実施できる。

5.3.1 Party 連鎖

クラスエディタ内に Party 連鎖を表示する。概要を図 5.7に示す。

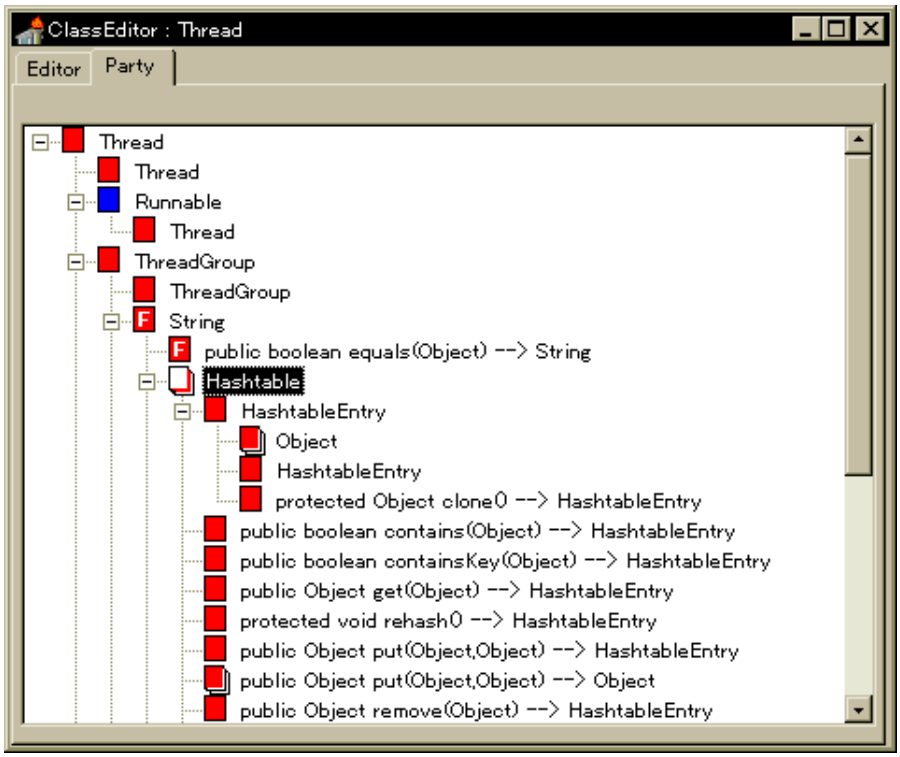

図 5.7: Party 連鎖ブラウザ

なお、「is-a」関係がある Party については、選択後、マウスの右クリック押下すると、図 5.8のように、ポップアップメニューが表示され、「is-a」関係にあるクラスまたはインターフェース群が表示される。この中らか任意の要素を選択することにより、表示が「is-a」関係にある Party 連鎖へ置換され、そこから新しい Party 連鎖をたどることができる。この様子を図 5.9に示す。

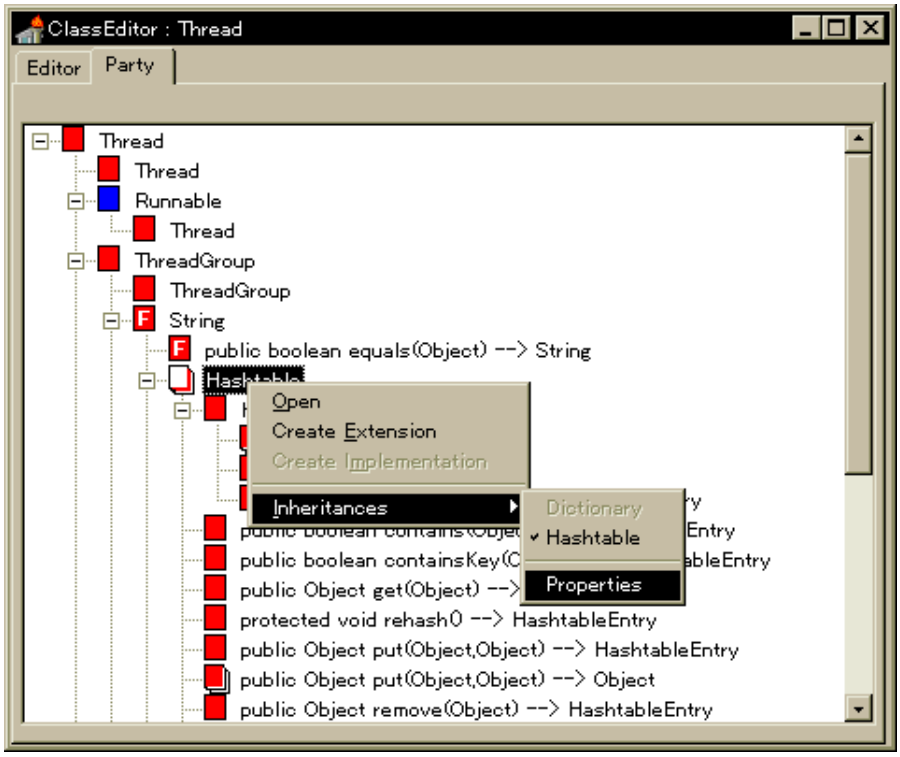

図 5.8: ポップアップメニューの表示

さらに、上記により置換した Party 連鎖は同じ動作により元に戻すことができる。ポップアップメニュー中で、印が付加されている要素が元の Party である。

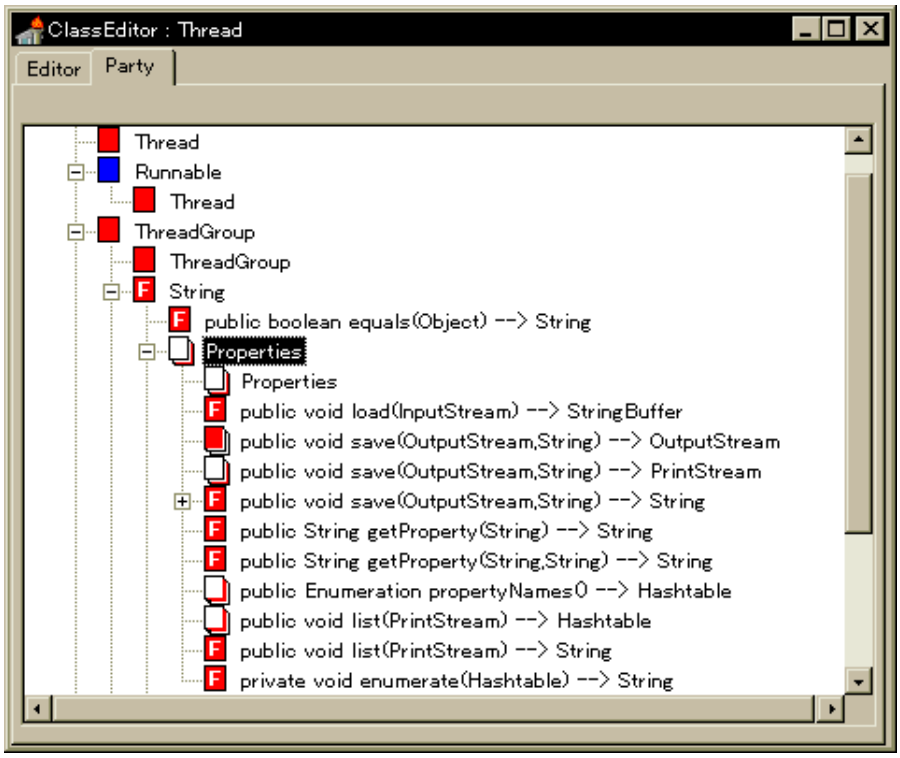

図 5.9: is-a による Party 連鎖の置換

第 6 章

クラス群への適用と評価

本章では、クラスブラウザのクラス群への適用とその評価を行う。まず、ブラウザのクラス群に対する適用を行いその結果を示す。次に、既存ブラウザとの比較をし、本ブラウザが採用した Party 連鎖によるブラウジング法の優位性を示す。

6.1 評価環境

以下の設備と要件で評価を行った。

実験環境	：WindowsNT4.0
開発言語	：Borland Delphi 2.0
評価対象クラス群	：JDK 1.0.1 のクラス群 約 200 クラス

評価項目１	：Party 間の is-a 関係抽出、Party 間の part-of 関係抽出
評価項目２	：Party 連鎖によるクラスブラウジング

6.2 クラス群への適用結果

本研究では、Party の概念を拡張し、Java 言語のインターフェースをサポートした。そのため、まず Party 連鎖におけるインターフェースの現れかたを報告する。

つぎに、Party 連鎖によるクラスブラウジング機能に関して、クラス群の現れかたに関して結果を報告する。

6.2.1 Party 連鎖におけるインターフェース

Party へのインターフェース導入により、Party 連鎖でインターフェースを扱えるようになった。試作ブラウザでクラス群を表示したところ、インターフェースに関する Party の表示に成功した。Party によるインターフェース部の表示では、オブジェクトのクラスとオブジェクトの型の分離についての実例を発見できた。ここで明らかになったクラス群には、同じスーパークラスからの派生クラスでないクラスがあった。実装による生じたクラスの型と役割に応じて付加されたインターフェースの型の実例を見ることができた。

6.2.2 クラス群の振舞いの表示

Party 連鎖により、協調して振る舞うクラス群が表示できた。「Party part-of Party」により、オブジェクトコンポジションが発生しうるクラス間の関係が表示できた。また、「Party is-a Party」による表示の切り替えで、継承関係がある場合でも、協調して振る舞うクラス群が表示できた。

6.2.3 デザインパターンに関る表示

インターフェースでの Party に注目することで、オブジェクトコンポジションを用いたデザインパターンの設計に関する実装が存在すると思われる場所が容易に発見できた。また、その周辺の Party 連鎖を見ることによって、それら構造を理解できた。

しかし、その使用されたデザインパターンの特定には、クラス名を参考にしたり、実際のコードを参照する必要があった。

図に、試作ブラウザー中に現れるデザインパターンの実例と、その構造を示す。

この例では、Composite パターンと Strategy パターンが使用されている。Composite パターンでは、Container クラスから Component クラスへの参照が存在すること、および Container クラスという名前から、Composite パターンであると判断した。また、Strategy パターンでは、LayoutManager インターフェースでレイアウトに関するアルゴリズムを変更可能にしているので、Strategy パターンであると判断した。

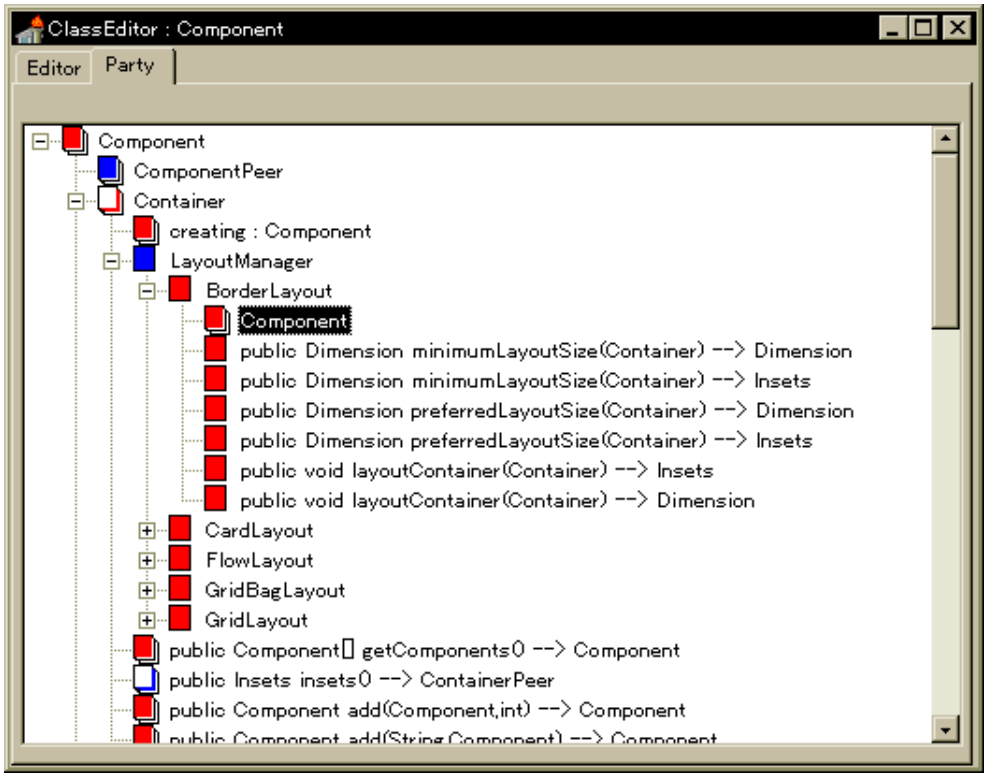

図 6.1: 試作ブラウザでのデザインパターンの現われ方

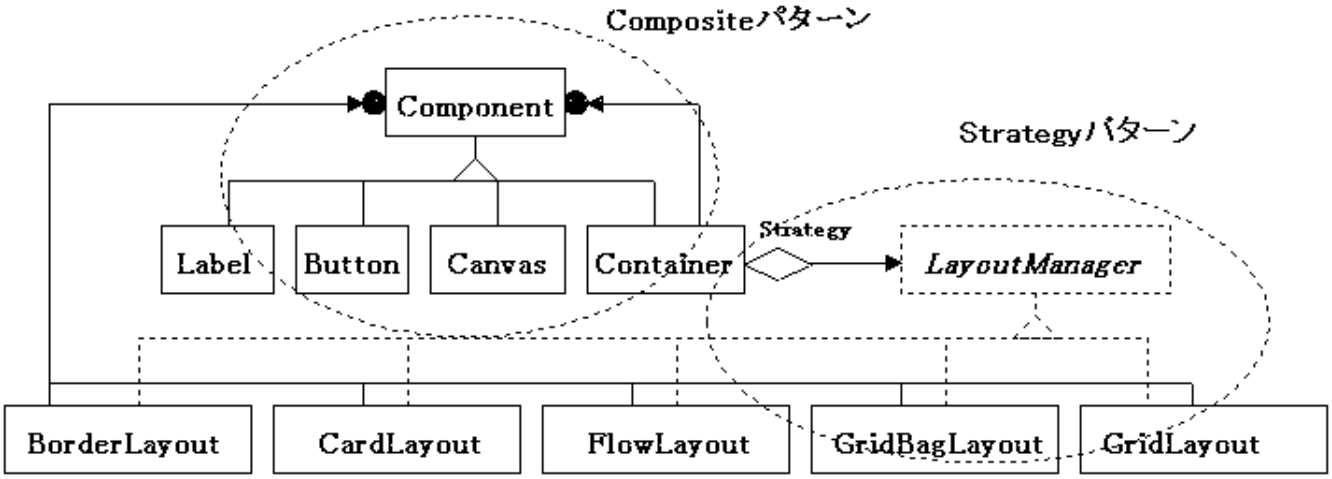

図 6.2: 発見したデザインパターンのクラス構造

6.3 既存ブラウザとの比較評価

既存ブラウザとの比較により、本ブラウザを評価する。
ここでは、比較対照の既存ブラウザとして以下のものを取り上げることとする。

- Taikade[12] 内蔵クラスブラウザ

  Taikade(The Evironment Informally Known As Dejava)。フリーの Java 開発支援環境である。開発元は株式会社 PFU。

- Borland Delphi version 2.0[14] のクラスブラウザ

  Object Pascal の開発環境。視覚的に GUI アプリケーションを実装できる。

- Microsoft Visual C++ version 4.0[15] のブラウザ

  C++開発環境。MS-Windows 系 OS では、標準的な開発環境。

- IBM VisualAge C++ version 3.0[16] のクラスブラウザ

  C++開発環境。VisualAge シリーズとして、同じユーザインターフェースの Smalltalk 等の開発環境もある。OS/2 および MS-Windows 用

以下にブラウザの機能比較を行った。なお、インターフェース継承については、Java 独自の言語要素であるので、比較対照に含めなかった。

なお、以下の表では、機能をサポートしている場合は「○」サポートしていない場合は、「×」となっている。なお、同じ評価段階でも、特別に評価する点がある場合は、＋を付加している。

表 6.1: クラスブラウザの機能比較

↓対象・項目→	クラスリスト	クラス継承グラフ	呼び出しグラフ
Taikade	○+	×	×
Delphi	○	○	×
Visual C++	○	○	○
VisualAge C++	○	○	○
試作ブラウザ	○+	○	○

それぞれの項目ごとに説明する。

1. クラスリスト

   ブラウジング対象のクラスを表示する。「Taikade」「試作ブラウザ」については、Java 言語のパッケージを使用した分類によるブラウジングをサポートしている。このブラウジングにより、クラス群設計者が設計時に意図したクラス分類構造を知ることができる。

   なお、試作ブラウザによるパッケージ構造の表示には、ツリー表示を採用している。そのため、ツリーノードの展開・縮小によって、不必要な部分をユーザに対し隠すことができる。このことは、パッケージ構造の概要を知るうえで、表による一覧表示を採用した「Taikade」と比較した場合、利点の一つとなる。

2. クラス継承グラフ

   クラス継承構造の表示は、オブジェクト指向プログラミングで、差分プログラミングによる実装の再利用を図るうえで重要なものである。そのため、ほとんどのクラスブラウザは、クラス継承グラフの表示をサポートしている。

   このクラス継承グラフによるブラウジングのもつ機能自体は、各ブラウザ間で差はほとんどない。

3. 呼出しグラフ

   本研究で導入した Party 連鎖によるツリーを、呼出しグラフの一種に含めて考えることにより、呼出しグラフという観点から評価をした。

「Visual C++」と「VisualAge C++」については、関数がどの関数を呼び出しているかというようところに注目している。どちらも、関数という単位に重点を置いて呼出しグラフを作成である。このような関数の呼び出しという観点から、呼出しグラフを評価した場合、その表示方法は、「VisualAge C++」が優れている。

ところで、Party 連鎖によるツリー表示は、「Visual C++」や「VisualAge C++」でいうような呼び出しグラフといえない。Party は、協調して振る舞うクラス群を定義している。そのため、「Party part-of Party」および「Party is-a Party」による Party 連鎖を表示すると、協調して振る舞うグラス群が明らかになる。

対して、関数の呼出しグラフを表示した場合は、当然、呼び出した関数を定義するクラスについては知ることができる。しかし、それはあくまで、その関数からの呼出し関係がわかるだけである。つまり、協調して振る舞うクラス群が、全体としてどのような構成になっているかを知るためには適していない。

関数の呼出し関係を把握することは、個々のクラス間の関係のある一面について、その関係を具体的に知るには適している。デバック時などに、そのクラスのある関数が、どのような呼出しグラフを形成するかについて知ることは、有益である。しかし、協調して振る舞うクラス群に関する理解という観点から考えた場合、関数という視点では、協調関係の一部しか知ることができない。

この協調して振る舞うクラス群に関する理解という観点では、前節の適用結果で明らかになったように、Party 連鎖のブラウジングは効果があった。本研究の目的の一つは、「協調して振る舞うクラス群の関連に関して、その構造を理解する」というものである。この目的については、Party 連鎖のブラウジングによって達成されたといえる。すなわち、この研究目的に対する解としては、試作ブラウザが優れている。

また、Party にインターフェースを導入したことにより、デザインパターンの重要な設計原理であるオブジェクトのクラスとオブジェクトの型の区別を認識できるようになった。このインターフェースに関する Party により、クラス設計者の意志に関する以下の２点を明確にする。

- クラス群設計者が同じ役割を担わせようとしたクラス群を特定できる。

- 実装の継承構造とは無関係に、同様な設計目的をもつ他クラスの存在を知ることができる。

以上が明確になることによって、もう一つの研究目的である「クラス群設計者の目的、動機を推測するための補助」が達成できたと考える。オブジェクトのクラスとしての型は差分プログラミングの結果としてのクラス階層によって決定されてしまう。対して、クラスへのインターフェース実装は、役割によって決るオブジェクトの型であり、設計者の明確な意思決定である。

これらの意思決定については、インターフェースを導入した Party により、その糸口をつかむことができる。そしてこの糸口は、クラスブラウザによる Party 連鎖表示によって明らかにでき、さらにデザインパターンで多用されるオブジェクトコンポジションの目的に対する理解補助として活用できることになった。また、この理解が結果的に、デザインパターンの知見を利用するのに役立つと考えられる。

以上のことから、試作ブラウザは、既存ブラウザに比べ、協調して振る舞うクラス群に対する理解に関するプログラマへの支援能力に優れると評価する。

第 7 章

終わりに

本研究では、以下の２点を目的としてクラスブラウザの実装を行った。

- 協調して振る舞うクラス群の関連に関して、その構造理解を支援する。
- クラス群設計者の目的、動機を推測するための補助をする。

上記目的を達成するため、クラスブラウザの設計において以下の決定をした。

- クラスのブラウジングには、Party 連鎖を利用することにした。
- デザインパターンより得られた知見から、オブジェクトコンポジションの理解と、オブジェクトのクラスとオブジェクトの型の区別に関して、ブラウザで支援することにした。
- ブラウザの対象とする言語として Java を選択した。また、そのために Party に対し、Java 言語のインターフェースという要素を導入した。
- ブラウザにおける Party 連鎖表示法を検討・決定した。

以上の決定に基づき、試作クラスブラウザを実装および評価した。

本章では、この試作ブラウザの実装および評価に基づき結論を述べる。さらに、結論を踏まえ将来への展望を述べる。

7.1 結論

クラスブラウザ実装によって以下の結論が得られた。

- クラスブラウザへの Party 連鎖表示の導入により、継承グラフではわからない協調して振る舞うクラス群に関する理解の支援が可能である。
- Java 言語のインターフェースを Party へ導入することにより、設計者の目的、動機を推測するための補助が可能である。
- クラスブラウザによる Party 連鎖の表示は、関数の呼び出しグラフの表示に比較して、協調して振る舞うクラス群に関する理解の支援という点で有効である。

7.2 今後の課題

今後の展望として、以下のような課題を挙げることができる。

- 今回は、対象言語として Java 言語を選択した。しかし、オブジェクト指向開発で用いられる OOPL は、Java だけではない。Java と同じく強い型付けをもつ言語に限っても、「C++」、「Object Pascal」等、使用者の多い言語がある。今後は、これらの言語に対するサポートが必要になってくる。なお、その場合は、今回 Java のインターフェースを Party へ導入することによって解決していたオブジェクトのクラスとオブジェクトの型の分離という点を、表現できるように、今回とは異なる方法で Party を拡張する必要が生じるであろう。
- デザインパターンの設計原理には、「オブジェクトコンポジションによる要求の委譲」というものがある。要求の委譲によってそれらのオブジェクトを定義するクラス間には Party で表現できる関係が生じる。しかし、要求の委譲が行われるかどうかは Party では表現することができない。デザインパターンにおける知見を利用してクラス群を理解するためには、要求の委譲を適切にあつかう手段を検討する必要が生ずるであろう。

第 8 章

謝辞

本研究を行うにあたり終始ご指導頂きました篠田陽一助教授には心からの感謝を申し上げます。さらに研究を開始するにあたり大きな御助力を頂きました落水浩一郎教授、また、研究をまとめあげるにあたり貴重な御意見を頂きました佐伯元司東京工業大学助教授、東田雅宏氏には心からの感謝を申し上げます。また、本研究に関して多くの貴重な御意見を頂きました博士課程後期の藤枝和宏氏、堀 雅和氏、および落水研究室の皆様に厚くお礼もうしあげます。最後になりますが、研究を推進するうえで大きな御助力を頂きました海谷治彦助手には心からのお礼を申し上げます。

参考文献

[1] 山城明弘, 吉田和樹, 入内島裕子, 斉藤悦生：画像ファイリングシステムへの適用, 情報処理 Vol.35 No.5,pp.432–437(1994).

[2] 井上 健:GUI クラスライブラリ構築への適用, 情報処理 Vol.35 No.5,pp.439-443(1994).

[3] 青山幹雄, 本位田真一：オブジェクト指向分析・設計の実際と評価, 情報処理 Vol.35 No.5,pp.451-460(1994).

[4] Erich Gamma, Richard Helm, Ralph Johnshon, Jhon Vlissidies, 本位田真一, 吉田和樹 監訳：デザインパターン (Design Patterns), ソフトバンク (1995).

[5] Doung Lea, Concurrrent Programming in Java, Design Principles and Patterns, Wesley(1996).

[6] 東田雅宏：Party に基づくオブジェクト管理機構の設計と実現, 北陸先端科学技術大学院大学 情報科学研究科 修士論文 (1995).

[7] Kyle Brown, DESIGN REVERSE-ENGINNERING AND AUTOMATED DESIGN PATTERN DETECTION IN SMALLTALK, `http://www2.ncsu.edu/eos/info/tasug/kbrown/thesis2.htm(1996)`.

[8] F.J.Budinsky,M.A.Finnie,J.M.Vlissides,P.S.Yu, Automatic code generation from design patterns, IBM System Journal Vol35,No 2,1996-Object technology.

[9] James Rumbaugh,Michal Blaha,Wiliam Premerlani, Frederick Eddy,and William Lorensen, 羽生田栄一 監訳：オブジェクト指向方法論 OMT, トッパン (1992).

[10] Lewis J.Pinson,Richard S.Wiener, 富士ゼロックス情報システム株式会社 訳:Smalltalk：オブジェクト指向プログラミング, トッパン (1990).

[11] 保木本晃弘：C++ ー大規模システム記述を支援したオブジェクト指向プログラミング言語ー, 情報処理 Vol.36 No.4,pp.315–326(1995).

[12] 有我成城, 衛藤敏寿, 佐藤治, 白神一久, 西村利浩, 村上列：Java 入門, 翔泳社 (1996).

[13] Laura Lemay,Charles L.Perkins, 武舎広幸, 久野禎子, 久野靖 訳：Java 言語入門, プレンティスホール出版 (1996).

[14] Borland International, ボーランド株式会社 編訳：Object Pascal 言語リファレンス, ボーランド株式会社 (1996).

[15] Microsoft Corporation, Visual C++ Books Online 4.2, Microsoft Corporation(1996).

[16] IBM, 日本アイ・ビー・エム株式会社 訳：VisualAgeC++ for OS/2 使用者の手引き バージョン 3.0, 日本アイ・ビー・エム株式会社 (1995).