修 士 論 文

メタパターンを用いたJavaソースコードにおける協調クラス群の抽出に関する研究

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

金　旭東

2006年3月

修 士 論 文

メタパターンを用いたJavaソースコードにおける協調クラス群の抽出に関する研究

指導教官　落水浩一郎 教授

審査委員主査　落水浩一郎 教授
審査委員　二木 厚木 教授
審査委員　鈴木正人 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

410038 金　旭東

提出年月: 2006 年 2 月

概 要

本研究では、メタパターンを用いて Java ソースコードにおけるクラス間の参照関係を解析することにより協調クラス群を抽出する方法を提案する.

本研究では、UML 図面中の要素の機能を実現するため実装された Java ソースコード中の関連クラスの固まりを協調クラス群と定義する. また、協調クラス群と別の協調クラス群との間に協調関係が生じる場合、２つの協調クラス群に含むすべてのクラスを１つの協調クラス群として定義する.

協調クラス群を抽出するには, ソフトウェアの設計時構造と実装時構造の差異を考慮する必要がある. 一般に UML 図面中のある要素を実装した場合, 必ずしも単一クラスとはならない. 多くの場合は一個以上のクラス群として実現される.

参照には協調に関与するものとそうでないものがある. 協調関係の判定には, メタパターンの構成要素であるテンプレートメソッドとフックメソッド間の関係を用いる. テンプレートメソッドとフックメソッドを用いた協調の型を定義し, それを利用して協調クラス群を抽出するアルゴリズムを提案する.Java ソースコードから協調クラス群を自動抽出するツールの構築を行った.

目 次

第 1 章　はじめに **1**
1.1 背景 1
1.2 研究の目的 1
1.3 本論文の構成 2

第 2 章　協調クラス群の抽出における問題点 **3**
2.1 ソフトウェア共同開発における変更作業支援のワークフローを自動生成する研究 3
2.2 設計時構造と実装時構造の差異 5
2.3 協調要素である参照とそうでないものの区別 7

第 3 章　デザインパターンとメタパターン **8**
3.1 デザインパターン 8
3.2 デザインパターンのカタログ 9
3.3 メタパターン 10
3.3.1 テンプレートメソッドとフックメソッド 11
3.3.2 テンプレートクラスとフッククラス 12
3.3.3 メタパターンの分類 13
3.4 メタパターンによる GoF デザインパターンの分類 15

第 4 章　アプローチ **17**
4.1 3 つの協調構造 17
4.2 3 つの協調構造から見たクラス間の協調関係 17

第 5 章　協調クラス群を抽出するアルゴリズム **21**
5.1 協調関係の判定法 21
5.2 アルゴリズム全体図 22

第 6 章　協調クラス群の自動抽出を実現するシステム設計 **26**
6.1 システムの設計 26
6.2 本システムの実行例 27

第 7 章　関連研究 **31**
7.1　静的解析と動的解析を用いたデザインパターン検出手法に関する研究 . . . 31
7.2　本研究との比較 31
7.3　考察 32

第 8 章　おわりに **33**
8.1　まとめ 33
8.2　今後の課題 33

謝辞 **34**

図 目 次

2.1 変更作業支援ワークフロー 4
2.2 設計時構造と実装時構造の差異 6

3.1 テンプレートメソッドとフックメソッド 11
3.2 1:1 結合メタパターン 13
3.3 １：N 結合メタパターン 13
3.4 統合メタパターン 14
3.5 1：1 再帰的統合メタパターン 14
3.6 1：N 再帰的統合メタパターン 14
3.7 1：1 再帰的結合メタパターン 15
3.8 １：N 再帰的結合メタパターン 15

4.1 統合的協調構造 18
4.2 結合的協調構造 19
4.3 再帰的結合協調構造 20

5.1 継承グループにおけるクラスの探索手順 22
5.2 テンプレートクラスメソッドとフックメソッドの対応関係１ 23
5.3 テンプレートクラスメソッドとフックメソッドの対応関係 2 24
5.4 協調クラス群を抽出するアルゴリズム 25

6.1 ASTExplorer より Java クラスを構文木 26
6.2 ステートパターンのよる協調クラス群 29
6.3 抽出された協調クラス群 30

表 目 次

第1章　はじめに

1.1　背景

ソフトウェア共同開発においては, すでに作成した成果物を参照しつつ, 新しい成果物を生成するため, 成果物間にさまざまな依存関係が存在する. このことは開発時の修正作業やバグ発生に伴って生じる変更作業を困難にする. さらに, 通常, 成果物間の依存関係は設計者やプログラマ自身により定義されるが, 試行錯誤を伴う設計活動ではその管理は容易ではない.

そこで, 落水研究室では, このような依存関係を自動生成することで, 変更に要するコストの軽減を図るための研究を進めている. 具体的には,UML および Java 言語を対象に, 依存関係を自動抽出し, それを利用して変更作業支援のワークフローを自動生成する研究を進めている. この研究は二つの部分に分けられる. 一つは UML 要素間に依存関係を自動生成する研究であり, すでに一定に成果を得ている. もう一つは,UML 要素に対応する Java ソースコードにおける協調クラス群を抽出する研究である. 本研究では, 後者を対象とする.

1.2　研究の目的

本研究では,UML のモデル要素と対応する,Java ソースコードにおける協調クラス群を抽出することを目的とする. プログラマがどのような実装方針を採用したかを判断するのは困難なので,Pree のメタパターンを用いて, テンプレートメソッドとフックメソッドの協調関係により形成されるクラス群が本研究で定義した協調クラス群と近似すると考える. 具体的には, オブジェクト間の参照関係をメタパターンを用いて解析することにより協調クラス群を抽出する方法を提案する. 参照には協調に関与するものとそうでないものがある. 協調関係の判定には, メタパターンの構成要素であるテンプレートメソッドとフックメソッド間の関係を用いる. テンプレートメソッドとフックメソッドとの協調の型を定義し, それを利用して協調クラス群を抽出するアルゴリズムを開発する. 最後に, 抽出された協調クラス群の有効性を確認する.

1.3　本論文の構成

本論文では,2 章では, 落水研究室で行われているソフトウェア共同開発のおける変更作業支援のワークフローを自動生成する研究の現状を述べ, その中での本研究の位置づけを明確にする. また, 協調クラス群の抽出における問題点を洗い出す.3 章では, 本研究で用いるデザインパターンとメタパターンについて紹介する.4 章では, 本研究のアプローチである, メタパターンを用いて, どのようにクラス間の協調関係を判断するかを述べる.5 章では, これまでに提案した協調関係の判断を利用して協調クラス群を抽出するアルゴリズムについて述べる.6 章では, 協調クラス群の自動抽出を実現するためのシステムの設計について述べて, 実例を適用した結果を用いて開発したツールの有効性について検証する.7 章では, 関連研究を取り上げ, 本研究との比較や優位性の検証を行う. 最後に,8 章の本研究のまとめと, 今後の課題について述べる.

第2章　協調クラス群の抽出における問題点

本章では, 変更作業支援のワークフローを自動生成する研究の現状を述べ, その中における本研究の位置付けを明確にするともに, 本研究を進める上で考慮すべき問題点を述べる.

2.1　ソフトウェア共同開発における変更作業支援のワークフローを自動生成する研究

ソフトウェア共同開発においては, 複数の人間の共同開発により, 中間成果物として文書やソースコードが生成される. 文書またはソースコードは, これらのある部分が変更された場合, 他の部分にこれが波及し, 同時に変更される必要性が生じることがある. 文書内やソースコード内, あるいは文書とソースコード間の整合を保つためには, 変更を加えた場合, その波及部分にもその都度対処しなければならない. 開発時の修正作業やバグ発生の伴う中間成果物の変更作業において, 変更対象となる文書やソースコードを変更波及への対処も含めて漏れなく適切に変更する作業は容易ではない. さらに, 複数の変更作業が同時並列に発生し, しかも複数の人間が関係する作業となる場合が多く, 作業効率を低下させる.

そこで, 落水研究室では,UML 図面と Java ソースコードを対象とし, 安全で効率的な作業変更を支援する情報モデルおよび操作モデルの定義と, 両者の自動生成技術の開発を進めている. ここで, 情報モデルとは, 変更作業を効率的に支援するのに必要なデータベースを意味し, 操作モデルとは, 変更手順を支持するワークフローを意味する. 一般に, 文書またはソースコードの変更作業は依存関係に基づいて実施される. ここで依存関係とは,「B が A に依存する」とき,「A を変更した場合,B を見直す必要がある」という関係である. 依存関係は,UML 図面の構成要素間,Java ソースコードの構成要素間に設定する必要があり効率的でない. 各種文書やソースコードの構成要素型を依存関係で結合したメタモデルを定義し, それを利用して, 文書やソースコードの実例に, 依存関係を自動的に付加する方式（Metamodel-based translation）により作業の自動化がはかれる. このような依存関係を自動抽出し, これを利用して変更作業支援のワークフローを自動生成する研究を行っている.　変更要求の発生に伴いワークフローは動的に生成され, 作業者はワークフローに従って, 安全かつ効率よく変更作業を進めうる. このとき一般には複数のワークフローが同

時並列的に存在し, しかもそれらのワークフローは文書やプログラム部分を共有する場合が多い. このような共同作業を支援するためには, 単に変更順序を指示するだけでなく, アクセス権, 所有者, 共有文書に関する同期制御等の情報を付加し状況に応じて適切な指示を行う操作モデルを定義する必要がある. 順序制御, アクセス制御, 同期制御機能を有するワークフローモデルとして操作モデルを定義し, そのような操作を解釈実行する実行エンジンを開発する.

変更作業支援のワークフローの全体像を図 2.1 に示す.

図 2.1: 変更作業支援ワークフロー

図 2.1 において，通し番号は開発の手順を示している．以下に各手順について概要と，開発が進行しているものについては状況を説明する.

(1) UML1.5 版を対象としてメタモデルを定義する.

(2) UML 図面群に依存関係を付加する変換プログラムを開発した. 変換プログラムは依存関係付き UML 図面群を XML データベースとして出力する.

(3) メタパターンを用いてデザインパターンを解析し,Java クラス群の協調構造を特徴づけるテンプレートメソッドとフックメソッド間の構造的関係を定義する.

(4) Java ソースコード中のすべてのクラスに対して, １つクラスの立場からこのクラスと協調関係があるすべてのクラスを協調クラス群として自動抽出するアルゴリズムを開発する.

(5) 上記のクラス群と UML 要素を対応付けるアルゴリズムを開発する. ここまでのプロセスにより,UML 要素群および協調 Java クラス群に依存関係を付加した XML データベースが設計?実現される.

(6) チームが採用されている作業分担, 作業手順, 責任範囲, コミュニケーション手段に関するルールを保持する作業モデルを開発する.

(7) 上記のデータベース内容を入力して変更作業支援ワークフローを生成するプログラムを開発する. 変更要求一つ一つに対して生成されるワークフローは一般には複数の担当者によって実行される. このため, ワークフローはその実行順序に関する情報だけでなく, アクセス権, 所有者, 共有文書に関する同期制御などの情報をもつ必要がある. このような情報は共同作業チームによってさだめられることが多い. この作業モデルと依存関係付き XML データベースをもとに, ワークフローモデルを定義し（操作モデル）, 生成実行手段を定義する.

(８) 変更作業においては一般に複数のワークフローを同時並列的に実行する必要がある. 順序制御, アクセス制御, 同期制御などの操作を解釈実行するワークフロー実行エンジンを開発する.

本研究は, 図 2.1 における (3),(4), すなわち,UML の要素に対応する,Java ソースコードにおける協調クラス群を抽出する部分を担当する.

2.2 設計時構造と実装時構造の差異

本研究の目的である,UML の要素と対応できる,Java ソースコードにおける協調クラス群を抽出するには, ソフトウェアの設計時構造と実装時構造の差異を考慮する必要がある. 一般には UML 図面中のある要素を実装した場合, 必ずしも単一のクラスとはならない. 多くの場合, 図 2.2 に示すように一個以上の協調するクラス群として実現される.

図 2.2 において,UML 図面上の ClassA を実現しているソースコード中の対応部分が ClassA,ClassB,ClassB1,ClassB2 であるとする.

このような差異が発生する理由としては，次のようなものが挙げられる.

- 設計者（個人，組織）のイディオム
- デザインパターンの使用
- 拡張性・再利用性など，オブジェクト指向プログラミングにおけるメリットの確保

図 2.2: 設計時構造と実装時構造の差異

2.3　協調要素である参照とそうでないものの区別

参照には協調に必要な参照と不必要な参照がある. 図 2.2 において,ClassA の ClassB への参照を協調に関与すると判定し,ClassB の ClassC への参照は協調に関与しないと判定することが必要となる.「ClassA から生成されるオブジェクトが ClassB 1 および ClassB2 から生成されるオブジェクトの参照を持ちメッセージを送れる」ということと,「ClassB 1,ClassB2 が ClassC への参照を持ちメッセージを送れる」ことの間には違いはないが,「ClassA にテンプレートメソッドがあり ClassB にはフックメソッドがあるが,ClassB と ClassC の間にはそのような関係はない」ということで協調に関与するか否かを判定する.

第3章　デザインパターンとメタパターン

本章では, まず, メタパターンへの理解を深めるため GoF のデザインパターンについて説明し, メタパターンの重要な概念であるテンプレートメソッドとフックメソッドについて述べる. 次に,Pree による, テンプレートメソッドとフックメソッドを用いたメタパターンの分類について説明し, メタパターンによる GoF のデザインパターンの分類について述べる.

3.1　デザインパターン

デザインパターン [5] は, 我々の身のまわりで何回も起きる問題, および, それぞれの問題に対する解法のポイントを記述している. これらの解法は何度も使うことができる. 同じ問題に対する同じ解法を何度も何度も最初から考えなおさずに済む. 一般にデザインパターンは次の４つの基本的な要素を有している.

1: 「デザインパターン名」は, 設計問題とその解法およびその結果を 1,2 語で記述した名称である. デザインパターンに名前を付けることで, 設計における用語の語彙を増やすことになる. それによって高い抽象レベルで設計することが可能となる.

2: 「問題」は, どのような場合にパターンを適用すべきかを記述したもので, 問題とその文脈を説明する. オブジェクトとしてアルゴリズムをどのように表現するかというような具体的な設計問題を記述する場合もあれば, 柔軟さに欠ける設計の徴候になるクラスやオブジェクト構造を記述する場合もある. また, デザインパターンを適用する際に満たさなければならない条件のリストを含む場合もある.

3: 「解法」は, 設計の要素, それらの関連および責任, 協調関係を記述する. 解法は特定の具体的な設計や実装は記述しない. なぜならば, デザインパターンはさまざまな状況に適用できるテンプレートのようなものだからである. その代わりにデザインパターンは設計問題を抽出的に記述し, クラスやオブジェクトなどの要素の配置によってどのようにそれらの問題を解決するかを示している.

4: 「結果」は, デザインパターンを適用する際の結果やトレードオフを記述する. 設計上の代替案の評価やパターンを適用する場合のコストや有効性を把握する際に, 結果は極めて重要である. デザインパターンの結果は, そのパターンがシステムの柔軟性, 拡張性, 移植性に与える影響も含まれる.

デザインパターンは, 一般的な設計構造のキーとなる側面に名前を付け, 抽象化, 識別化し, 再利用可能なオブジェクト指向設計を生み出すに有用となるようにしたものである. また, デザインパターンはそれにかかわっているクラスやインスタンス, それらの役割や, 協調関係, 責任の分担を規定する. そして, それぞれのデザインパターンは, オブジェクト指向設計における特定の問題や課題に焦点を絞っている.

3.2 デザインパターンのカタログ

本節では,Erich Gamma のデザインパターンのカタログの特徴について説明する. デザインパターンカタログのそれぞれのパターンは, 非形式的なテキスト, オブジェクトモデル記法, 実装ヒント, そしてコーディング例で記述されている. また, 対応するクラス/オブジェクト図が添えられた形で, パターンに関与するものとそのパターンと協調して働くものについて非形式的記述がなされ抽象化されている.

デザインパターンカタログには細粒度で以下のように分類されている.

1: 抽象カップリングによるパターン

抽象カップリングは, クラス A がクラス B を参照するときクラス B が抽象クラスでなければならない.

- Builder:複雑な構造を持ったものを作り上げるとき, まず全体を構成している各部分を作り, 段階を踏んで組み上げていくパターンである.
- Bridge:2 種類の拡張が混雑するプログラムを機能の階層と実装の階層に分け, その間に橋渡しを行うパターンである.
- State：状態をクラスで表現し, 状態に応じた switch 文を減らすパターンである.
- Strategy：アルゴリズムをごっそり切り替えて改良しやすいようにするパターンである.
- Mediator：複数のクラスが互いに相談しあうのではなく, 窓口役のクラスを１つ用意し, その相談役とだけやりとりすることでプログラミングをシンプルにするパターンである.
- Visitor：構造を渡り歩きながら同じ操作を繰り返し適用するパターンである.
- Abstract Factory：工場のように部品を組み合わせてインスタンス生成を行うパターンである.
- Command：要求や命令を形にしてクラスで表現するパターンである.
- Proxy：本当に目的のものが必要になるまでは代理人を使って処理を進めるパターンである.

- Prototype：ひな型となるインスタンスをコピーしてインスタンスを作るパターンである.
- Adapter：異なるインタフェース（API）を持つクラスの間をつなげるパターンである.
- Flyweight：複数箇所で同じものが登場するときに, それらを共有して無駄を無くすパターンである.
- Observer：状態が変化するクラスと, その変化を通知してもらすクラスを分けて考えるパターンである.
- Iterator：複数の要素が集まっている中から要素を１つ１つ取り出していくパターンである.

2: 再帰構造に基づくパターン

再帰構造とは, 関連クラスの中に, クラスからそのクラス自身への参照を含むことをいう.

- Decorator：飾り枠と中身を同一視して, 飾り枠を何重にも重ねるパターンである.
- Composite：容器と中身を同一視して, 再帰的な構造を構築するパターンである.
- Interpretor：文法法則をクラスで表現するパターンである.

3: そのほかのパターン

- Template Method：スーパークラスで処理の骨組を定め, 具体的な処理をサブクラスで行うパターンである.
- Factory Method：スーパークラスでインスタンスの作り方に骨組を定め, 具体的な作成そのものはサブクラスで行うパターンである.
- Interpretor：文法法則をクラスで表現するパターンである.
- Singleton：インスタンスが 1 つしか存在しないパターンである.
- Memento：現在の状態を保存し, 必要に応じて復帰させ, アンドゥを行えるようにするパターンである.

3.3 メタパターン

Pree[1] はデザインパターンを抽象化してメタパターンをまとめた. メタパターンは, どのようなデザインパターンでもメタレベル分類し記述することのできるエレガントで強力なアプローチである.

単一クラス, あるいは相互作用するクラスのグループから構成されるデザインパターンを設計するために必要なメタパターンは, テンプレートメソッドとフックメソッドから形成される.

3.3.1 テンプレートメソッドとフックメソッド

図 3.1 の例を用いて, テンプレートメソッドとフックメソッドを説明する. テンプレートメソッドとは, 具象メソッド, 抽象メソッド, フックメソッドの任意の組み合わせを呼び出すメソッドで, その詳細を実装することなくアルゴリズム輪郭だけを決定するメソッドである.3.1 における M1 はテンプレートメソッドである.

フックメソッドとは, サブクラスでオーバーライトされることを想定したメソッドである. 抽象メソッド, 具象メソッド, テンプレートメソッドのどれでもフックメソッドになりうる. 図 3.1 において, クラス B の M 2 と M3 がフックメソッドである.

図 3.1: テンプレートメソッドとフックメソッド

テンプレートメソッド M1() は抽象メソッド M2() と具象メソッド M3() を呼び出す. フックメソッド M2() は要求を満たすために変更しなければならないので, グレースポットで表現される. また具象メソッド M3() も置き換え可能なのでグレースポットとして表現される. サブクラス B1 では,M2() をオーバーライドして, テンプレートメソッド M1() を適

合する. クラス B のテンプレートメソッド M1() はソースコード変更することなくアプリケーションに適合される. もちろん, テンプレートメソッド M1() のホットスポットをアプリケーションに適合させるために,B のサブクラスでフックメソッド M3() をオーバーライドすることもできる.

テンプレートメソッドとフックメソッドは, 1 つにクラスの中での柔軟性を実現するためだけに役に立つものではない. テンプレートメソッドとフックメソッドが 1 つのクラスに存在する場合は, サブクラスを定義するのみ, このクラスの振る舞いを変更することができる. 実行時にそのような適合を可能にするには, テンプレートメソッドとフックメソッドを別々のクラスとしなければならない. これによりクラス間にまたがる柔軟性を拡張する必要がある.

3.3.2 テンプレートクラスとフッククラス

フックメソッドを含むクラスを, 対応するテンプレートメソッドを含んでいるクラスのフッククラスと呼ぶ. テンプレートメソッドを含むクラスをテンプレートクラスと呼ぶ. 以後, テンプレートクラスを　T と記し, フッククラスを　H と記す.

テンプレートメソッドとフックメソッドの場合と同様に, 状況によって, クラスはテンプレートクラスと見なされたり, フッククラスと見なされたりする. テンプレートクラスのテンプレートメソッドがほかのクラスのメソッドのフックメソッドとして使われているならば, テンプレートクラスがフッククラスになる.

オブジェクト間でメッセージ送信できるための事前条件は, オブジェクト間に参照が存在することである. テンプレートクラスのオブジェクトと, フッククラスのオブジェクトのカップリングのためにもこれが必要である. テンプレートメソッドはフッククラスのオブジェクトにメッセージを送るので, 多くの場合, テンプレートクラスのオブジェクトはフッククラスのオブジェクトの参照を保持しなければならない. テンプレートクラス T のオブジェクトがフッククラス H のオブジェクトを参照するには以下の 3 つある.

(1) クラス T の中の静的な H 型のインスタンス変数による場合

(2) メソッドのパラメータでオブジェクトの参照を渡すことによる場合

(3) クラス H のオブジェクトを参照するグローバル変数による場合

テンプレートクラスは具象クラスだけでなく, 一般にテンプレートクラスにはテンプレートメソッド以外のメソッドもあるのでこのクラスにテンプレートメソッドと関係ない抽象メソッドがあるときは抽象クラスになる. フックメソッドは抽象メソッドの場合も具象メソッドの場合もあるのでフッククラスも抽象クラスか具象クラスか両方あり得る.

3.3.3 メタパターンの分類

Pree によるメタパターンの分類を (1) から (7) に示す. 分類において結合, 再帰という用語が使用されているが以下の意味を持つ.

T と H を 3.3.1 節で説明した 3 つの参照方法により結び付けることを結合といい, クラスからそのクラス自身への参照を行うことを再帰という.

メタパターンはテンプレートクラスとフッククラスの間に継承関係があるか否かによって大きく二つに分けられる.

1: T と H の間に継承関係がない場合

(1) T のオブジェクトが H のちょうど一つのオブジェクトを参照する場合 1：1 結合メタパターンという.

図 3.2: 1:1 結合メタパターン

(2) T のオブジェクトが H のいくつかのオブジェクトを参照する場合は１：N 結合メタパターンという.

図 3.3: １：N 結合メタパターン

(3) T と H が一つのクラスからなっているがテンプレートメソッドとフックメソッドが同じ名前ではない場合は統合メタパターンという.

(4) T と H が統合され, 対応するテンプレートメソッドとフックメソッドも同じ名前のメソッドに統合されているようなただ一つメソッドだけが存在する,T のオブジェクトが H のちょうど一つのオブジェクトを参照する場合 1：1 再帰的統合メタパターンという.

T のオブジェクトが H のいくつかのオブジェクトを参照する場合は１：N 再帰的統合メタパターンという.

図 3.4: 統合メタパターン

図 3.5: 1：1 再帰的統合メタパターン

図 3.6: 1：N 再帰的統合メタパターン

2: T と H の間に継承関係がある場合

(5) T のオブジェクトが H のちょうど一つのオブジェクトを参照する場合 1：1 再帰的結合メタパターンという.

図 3.7: 1：1 再帰的結合メタパターン

(6) T のオブジェクトが H のいくつかのオブジェクトを参照する場合は１：N 再帰的結合メタパターンという.

図 3.8: １：N 再帰的結合メタパターン

3.4 メタパターンによる GoF デザインパターンの分類

上記のメタパターンを利用した GoF のデザインパターンの分類結果を表 3.1 に示す. 表 3.1 において,3 つのデザインパターン（Facade, Singleton, Memento）における協調はテンプレートメソッドとフックメソッドを含まない異なる協調構造である. 本稿の検討対象外となり今後の課題である.

また,「１：N 再帰的統合メタパターンに属するデザインパターン」は存在しないので実際には６つのメタパターンを利用することになる.

表 3.1: メタパターンによる GoF デザインパターンの分類

第4章　アプローチ

本章では, 上記のメタパターンを協調クラスの抽出に用いるため, さらにメタパターンを 3 つの協調構造にまとめる. そして, 3 つの協調構造から見たクラス間の協調関係について述べる.

4.1　3 つの協調構造

テンプレートクラスのオブジェクトがフッククラスのオブジェクトをいくつ参照するかについては, 協調に関与する参照を判断する立場からは無関係なので注目しない. このような視点からすると,6 つのメタパターンの構造的特徴を表 4.1 のように 3 つにまとめることができる.

表 4.1　メタパターンによる 3 つの代表的な構造へ分類

4.2　3 つの協調構造から見たクラス間の協調関係

上記の 3 つの協調構造は GoF のデザインパターンで使われているクラス間の協調関係をまとめた構造である. 本研究では, まず,GoF のデザインパターンで使われている協調ク

ラス群を抽出するアルゴリズムを開発し, その次に, 一般的な協調クラス群の抽出にも対応できるアルゴリズムに発展されるアプローチを取る. 以下に,3 つの協調構造を図示する.

1. 統合的協調構造：図 4.1 に統合的協調構造の一例を示す. 統合的協調構造においては, テンプレートメソッドとフックメソッドは, 一つのクラスに存在する.

図 4.1: 統合的協調構造

2. 結合的協調構造：図 4.2 に結合的協調構造の一例を示す. テンプレートメソッドとフックメソッドが異なるクラスに存在する.

3. 再帰的結合協調構造:図 4.3 に再帰的結合協調構造の一例を示す. テンプレートメソッドとフックメソッドが異なるクラスに存在し, テンプレートクラスとフッククラスの間に再帰的関係がある.

図 4.2: 結合的協調構造

図 4.3: 再帰的結合協調構造

第5章　協調クラス群を抽出するアルゴリズム

本章では, 協調関係の判定法と Java ソースコード中のすべてのクラスに対し協調関係があるクラス群を抽出するアルゴリズムについて説明する.

5.1　協調関係の判定法

本節では, テンプレートメソッドとそれに対応するフックメソッドが存在する否かにより協調関係を確認する方法について説明する.

1. テンプレートメソッドの探索

   (1) Java ソースコード中の任意のあるクラスについて, まず, 継承関係があるか否かをチェックする. 継承関係がない場合はそのクラスのメソッドが同じクラスのメソッドを利用しているか, または, ほかのクラスのメソッドを利用しているかをチェックする.

   ある場合は：　ステップ 2 でフックメソッドの探索を行う.

   ない場合は：　このメソッドでが、協調関係がないと判定する.

   (2) 継承によりグループになったクラス群に対して (図 5.1), グループのすべてのクラスのメソッドに対して 1 － (1) のチェックを行う.

2. フックメソッドの探索

   (1) 同じクラス内のメソッドを利用した場合, 利用されたメソッドがそのクラスのサブクラスにオーバーライトされていないかをチェックする.

   オーバーライトされていたら：　協調関係があると判定する.

   オーバーライトされていなかったら（1 － (2) の場合のみ）: 1 － (2) に戻り探索を続ける.

   (2) ほかのクラスのメソッドを参照する場合：参照されたメソッドがオーバーライトされてないか（図 5.2 のように）それともほかのクラスのメソッドをオーバーライトしているか（図 5.3 のように）をチェックするそうであれば：協調関係

があると判定する. そうでなければ（1 − (2) の場合のみ）: 1 − (2) に戻り探索を続ける.

5.2 アルゴリズム全体図

図 5.4 の点線に囲まれた部分は,Java ソースコードにおけるすべてのクラスに対してテンプレートメソッド探索を行えるための部分である. 図 5.4 の下半分は, テンプレートメソッドとそれに対応するフックメソッドが存在するか否かにより協調関係の確認を行う部分である. 図 5.4 の中の使われているアルファベット I, J, K は以下の意味を持つ

I：ソースコード内の継承グループ数

J：継承グループ内のクラス数

K：クラス内のメソッド数

C：探索中の Java クラス

図 5.1: 継承グループにおけるクラスの探索手順

図 5.2: テンプレートクラスメソッドとフックメソッドの対応関係 1

図 5.3: テンプレートクラスメソッドとフックメソッドの対応関係 2

図 5.4: 協調クラス群を抽出するアルゴリズム

第6章　協調クラス群の自動抽出を実現するシステム設計

本章では, これまでに述べた手法を利用して,Java ソースコードにおける協調クラス群を抽出するシステムの設計と実行例を述べる.

6.1　システムの設計

本システムは, 2 つに分けてシステムを構築する. 一つ目は, 対象となる Java ソースコードに対し, 構文分析をおこなう,2 つ目は, 訪問者パターン [5] を使って構文分析された Java クラスの情報（図 6.1）を得て, 本アルゴリズムを実現する.

1 : Eclipse プロジェクトに含まれている ASTExplorer を利用して,Java クラスに対し AST 階層構造を構文解析する. その結果, 対象となる Java ソースコードの各クラスことに以下のように分析される.

図 6.1: ASTExplorer より Java クラスを構文木

AST 階層構造の最上位には,ASTNode がある.Java 構成体もこれによって表現される.

図 6.1 での,ClassType には, クラスの継承, 実装を調べるための情報が入っている.Field には, 他のクラスのインスタンス変数の情報が入っている.Method には, 他のクラスの参照を行っている情報が入っている.

2 : 上記の構文分析情報の結果により開発したアルゴリズムを実現するため, 構文分析されたこれらの情報を読み取るため各クラスことに訪問しなければならない.

ASTNode では, 訪問者パターン [5] を使って, ノード・ツリーを辿ることができる.

Org.eclips.jdt.core.dom.ASTVisitor から派生したクラスを作り, そのインスタンスをノードのメソッド accept( ) に渡す. このメソッドが呼ばれると, ツリーの各ノードうち, 現在のノードから先のノードが「訪問」受けることになる. 各ノード・タイプに対して,visit( ) と endVisit( ) という, 1 つのメソッドがある. パラメータ・ノードのタイプは, 訪問を受けているノードに対応する.

これにより, 本研究で開発されたアルゴリズムを実現するための Java クラス情報を得る.

6.2 本システムの実行例

本研究で, 開発したアルゴリズムとシステム構築の有効性を確認するため以下の 4 つのテストを手作業で行った.

1 : GoF のデザインパターンの抽出

表 6.1 GoF デザインパターンの抽出結果

図 5.4 に示したアルゴリズムの妥当性を確認するため, まず, デザインパターンに適用してみた. 結果を表 6.1 に示す.Factory Method, Flyweight, Abstract Factory については抽出できなかった.Factory Method と Abstract Factory については, 生成されるオブジェクト群を併せて解析する必要があり, また,Flyweight については異なる協調構造が使われている. この 3 つのデザインパターンについては今後の課題である.

2: 協調要素である参照とそうでないものの区別

文献 [4] で与えられたエレベータ制御システムの仕様に基づく, 北陸先端科学技術大学院大学での講義に使うため, 開発されたソースコードに対し本アルゴリズムで解析した.

その結果,ElevatorController クラスから ArrivalSensorEvent への参照はクラス間の協調関係の判定法によって切り捨てられ, 図 6.2 の点線に囲まれたステートパターンによる協調クラス群を抽出した.

3: 一般的な協調クラス群の抽出

デザインパターンを含む一般的な協調関係の抽出能力を確認するため, 文献 4 で与えられたエレベータ制御システムの仕様に基づく, 北陸先端科学技術大学院大学の講義で開発されたソースコードを本アルゴリズムで解析した.

その結果, 図 6.3 において,ElevatorController クラスと DoorClosingToMoveUp クラスに関してそれぞれ協調クラス群を抽出できた.ElevatorController クラスと協調するクラス群は, 図 6.3 において黒く塗りつぶしたクラス群であり,State パターンに対応する. DoorClosingToMoveUp クラスと協調するクラス群として図 6.3 全体が抽出される. これは定義された協調クラス群と一致するデザインパターン以外の協調が抽出された例である.

この結果から, 本アルゴリズムが一般的な協調クラス群の抽出にも適用できることがわかる.

4: 異なる実装による異なる協調クラス群

UML 図からどのような構造のソフトフェアを実装するかは, 実装者の意図によって異なるものである. すべての協調クラス群を抽出するためには, 異なる実装による異なる協調クラス群の抽出も可能でなければならない. 図 6.3 とは実装の構造が異なるソースコードを作成し解析してみた. 図 6.3 のプログラムをシステムの機能に影響を与えない前提で, 以下のように書き換える. 元のソースコードでは, DoorClosing-ToMoveUp クラス ElevatorController クラス DoorInterface のインスタンス変数を通じて DoorInterface のサブクラスを参照すること DoorInterface にメッセージを送るようになっていた. これを DoorClosingToMoveUp が ElevatorController へメッセージを送るように委託し, そして ElevatoerController で DoorInterface メッセージを送るように構造を変えることも可能である. この場合は図 6.3 の一点鎖線で囲んだ

部分と黒く塗りつぶした部分が抽出される, これも定義された協調クラス群と一致する.

図 6.2: ステートパターンのよる協調クラス群

図 6.3: 抽出された協調クラス群

第7章　関連研究

本章では, 本研究に関連する研究「静的解析と動的解析を用いたデザインパターン検出手法」を紹介し, 本研究との比較について述べる.

7.1　静的解析と動的解析を用いたデザインパターン検出手法に関する研究

堅田 [3] は, ソフトウェアの設計時に無意識にデザインパターンが使われていた場合, 使われているデザインパターンがドキュメント等に明示されていない状況においてプログラム理解を支援するために, 既存のプログラムにおいてデザインパターンを検出する手法を提案した. 提案した手法では静的解析と動的解析を併用することで検出を行った.

静的解析では, 対象プログラムのソースコードからクラスの継承関係や定義されているメソッドなどの情報を抽出し, これらを Prolog の事実として保持する. また, 静的解析のみでは得ることのできない, 対象プログラムの動的な情報を得るために, 動的解析を行った.

動的解析では, 対象プログラムを実行し, 実際に実行されたメソッドに関する情報を Prolog の事実として保持する. これらの情報は, オーバーライドしているメソッドが呼ばれたときに実際に呼び出されたメソッドや, プログラム実行中に生成されたオブジェクトが実際にどのクラスのインスタンスとして生成されたか, といった情報を調べるために用いる.

Prolog 事実として抽出された対象プログラムの情報に対して,Prolog の規則として記述されたデザインパターン検出条件とのマッチングをとる. 個々のデザインパターン検出条件は, 対象プログラムの情報からデザインパターンの条件を満たすクラス構造の存在を推論するように記述されている. 対象プログラムの情報に対して, この検出条件んをクエリとして発行することによって, 1 つの解としてデザインパターンを形成しているクラス構造を得る方針を取っている.

7.2　本研究との比較

1: 動的解析

堅田は, 静的解析で得ることのできない, 対象プログラムの動的情報を動的解析を行った. 動的解析を行うためメソッドレベルでクラス間の関係を分析し, デザインパター

ンを形成しているクラス構造を検出する方法をとっている.

2: メタパターンによる GoF デザインパターンの分類

堅田は,Pree のメタパターンを用いて, 動的解析するクラス間の関係をテンプレートメソッドとフックメソッドの関わりという観点からデザインパターンの構造を解析した.

以上を踏まえた上で, 堅田の提案手法と本研究の相違点は, 堅田はメタパターンによる GoF デザインパターン検出条件を階層化し, デザインパターンの特徴まとめることである. デザインパターンが持つ同じ特徴を表現するためメタパターンを利用した.

本研究では, メタパターンをデザインパターンの特徴を表現することでなく, デザインパターンのクラス間の協調関係を解析するため利用した. これにより動的分析を行った.

7.3 考察

堅田の手法は、ソースコードからデザインパターンを検出することにより、既存のオブジェクト指向ソフトウェアの中でどのようなデザインパターンが使われているかを知ることができ、ソフトウェアの理解を支援するためである. 対象プログラムのテストにより、使われているすべてのデザインパターンを発見することができた。本研究では、オブジェクト間の参照関係をメタパターンを用いて解析することにより協調クラス群を抽出する方法を提案した. 参照には協調に関与するものとそうでないものを区別するため、メタパターンの構成要素であるテンプレートメソッドとフックメソッド間の関係を用いた.GoF のデザインパターンで使われているすべてのテンプレートメソッドとフックメソッドの協調の型を定義し、それを利用して協調クラス群を抽出した.

本研究で開発したアルゴリズムの妥当性を以下の 3 つの実験より確認した.

(1) GoF のデザインパターンに本アルゴリズムを適用した結果 23 個デザインパターン中 20 個抽出できた. 抽出できなかった Factory Method、 Flyweight、 Abstract Fctory については、生成されるオブジェクト群を併せて解析するなどの必要があり今後の課題である.

(2) デザインパターンを含む一般的な協調関係の抽出能力を確認するため北陸先端大の講義で使われているエレベータ制御システムの Java ソースコードに本アルゴリズムを解析した結果、一般的な協調クラス群の抽出にも適用できることがわかった.

(3) UML モデル要素を実装するには、実装者の意図によって異なりますので、それに対応する協調クラス群の抽出も可能であることを確認できた.

第8章　おわりに

8.1　まとめ

本研究では, メタパターンを用いて協調に関連する参照のみを取り出すアルゴリズムを提案した.Gamma のデザインパターンは 7 つのメタパターンに分類される. 我々は, さらに 7 つのメタパターンを, テンプレートメソッドとフックメソッド間の構造的関係を利用して, テンプレートメソッドとフックメソッドが協調しあう構造的な特徴により三つの代表的な型に分類した. この三つの代表的な型に対応する Java の言語的特徴を整理し,Java プログラム内の協調するクラス群を抽出できることを確認した.

8.2　今後の課題

本研究の検討対象外となっている,3 つのデザインパターン（Facade, Singleton, Memento）における協調は今後の課題としてさらに, アルゴリズムの開発を行う必要がある.

すべての協調クラス群に対応できるアルゴリズムにするため, さらに異なる協調構造も検討する必要がある.

謝辞

本研究を行うにあたり, 終始変わらぬ御指導を賜りました落水浩一郎教授に心から深く感謝申し上げます.

本研究の審査委員として, 二木厚吉教授, 鈴木正人助教授にはご助言, ご意見をいただき深く感謝致します.

本研究を進めるにあたり, 種々の有益なご助言をいただきました藤枝和宏博士を始め, ご助言, ご感想を寄せていただきました落水研究室の早坂良氏, 小谷正行氏に深く感謝申し上げます.

最後に, 勉学への理解とともにこの日に至るまで多大な援助をいただいた家族, そして白山ロータリーのクラブの皆様に深く感謝致します.

参考文献

[1] Wolfgang Pree, “ Design Patterns for Object-Oriented Software Development ”by the ACM Press, a Division of the Association for Computing Machinery, Inc.(ACM). 1995.

[2] 小谷 正行, 落水 浩一郎, ”依存関係を用いた UML 文書間の波及解析法”, 電子情報通信学会ソフトウェアサイエンス研究会, SS2004-62, 2005.03.

[3] 堅田 淳也, 小林 隆志, 佐伯 元司, ”静的解析と動的解析を用いたデザインパターン検出手法 ” 電子情報通信学会, 2005-4.

[4] Hassan Gomma, “ Designing concurrent, distributed, and real-time application with UML ”, Addision Wesley, Inc. 2000.

[5] Erich Gamma,Richard Helm, Ralph Johnson, John Vlissides,Design Patterns – Elements of Reusable Object-Oriented Software（1997 年 Addison Wesley 刊）.

参考文献