修 士 論 文

Javaソースコードにおける協調クラス群の抽出

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

NGUYEN VAN TUAN

2010年3月

修 士 論 文

Javaソースコードにおける協調クラス群の抽出

指導教員　落水浩一郎 教授

審査委員主査　落水浩一郎 教授
審査委員　鈴木正人 准教授
審査委員　青木利晃 准教授

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

0810021NGUYEN VAN TUAN

提出年月: 2010 年 2 月

概 要

ソフトウェア開発では、設計モデル要素に対応するソースコードにおける協調クラス群を抽出することは安全で効率的に変更作業を支援するために主な役割を持つ。本研究では、ユースケースに対応する Java ソースコードにおけるクラス群を協調クラス群と定義し、強調クラス群を抽出する方法を提案する。デザインパターンに基づいた協調クラス群の抽出法より、メタパターンの構造的特徴に基づいて協調クラス群を抽出する方法がより簡単だが、抽出できない場合が多い。そのため、我々はメタパターンの構造的特徴とデザインパターンの構造の特徴および振る舞いの特徴を利用して、協調クラス群を抽出するアルゴリズムを開発した。Java ソースコードに現れるデザインパターンを利用したクラス群の特徴に基づき、ユースケースを実装したクラス図中の個々のクラスに対して対応をする Java クラス群を抽出し、サブグラフ同型判定アルゴリズムを適用して、対応付ける手法を提案した。小規模なシステムで実験を行い、抽出アルゴリズムの精度は 92.5％以上であった。

目 次

図目次

表目次

第1章　背景

1.1　研究の背景

本章では、研究の背景と目的について述べる。本研究は落水研究室のプロジェクトの一つの部分であるために、研究の背景については、研究の意義および落水研究室のプロジェクトの構成を述べる。

1.1.1　研究の意義

情報システムを開発する際に、膨大な量のソフトウェア図面とプログラムが存在している。また、このソフトウェア図面やプログラムの間に複雑な依存関係があるために、変更作業が困難である。変更に要する労力の軽減、信頼性の向上を保証するために、変更作業を自動化して支援するツールが必要である。

変更作業を自動化して支援するツールにおいては、設計モデルの要素に対応するソースコードにある協調クラス群を自動的に発見することが大切である。具体的には、クラス図における協調クラス群とソースコードの協調クラス群を発見し、対応付けを行う作業を文書やプログラム読むことで発見することは大変困難であるため、変更作業では、実装モデルの要素に対応する協調クラス群を自動的に発見することはコストの軽減と信頼性の向上をよく支援する。

1.1.2　研究の構成

現在、落水研究室では、システムの変更作業を助けるために“ ソフトウェア設計図面やプログラムの変更を助けるシステム ”を開発している。このシステムはソフトウェア図面やプログラムの間に存在する依存関係を自動的に生成する技術を開発して、生成した依存関係に基づいて変更作業支援ワークフローを自動的に生成する技術を開発する。システムの処理の流れは図 1.1 である。

システム処理の流れ図は、システムが三つの部分に分けられている。一つ目の部分は UML 依存関係メタモデルに基づいて UML 図面群に依存関係を追加して、依存関係付け UML 図面群を作成する。この部分は既に開発されている。二つ目の部分は Java プログラムから協調クラス群を自動的に抽出して、UML モデリング要素と Java 協調クラス群の対応を付けて、UML ― Java の対応関係を作成する部分である。最後は上の二つの部分の結果を利用して変更作業支援ワークフローを自動的に生成する部分である。本研究は UML ― Java の対応関係を発見する手法を開発することである。

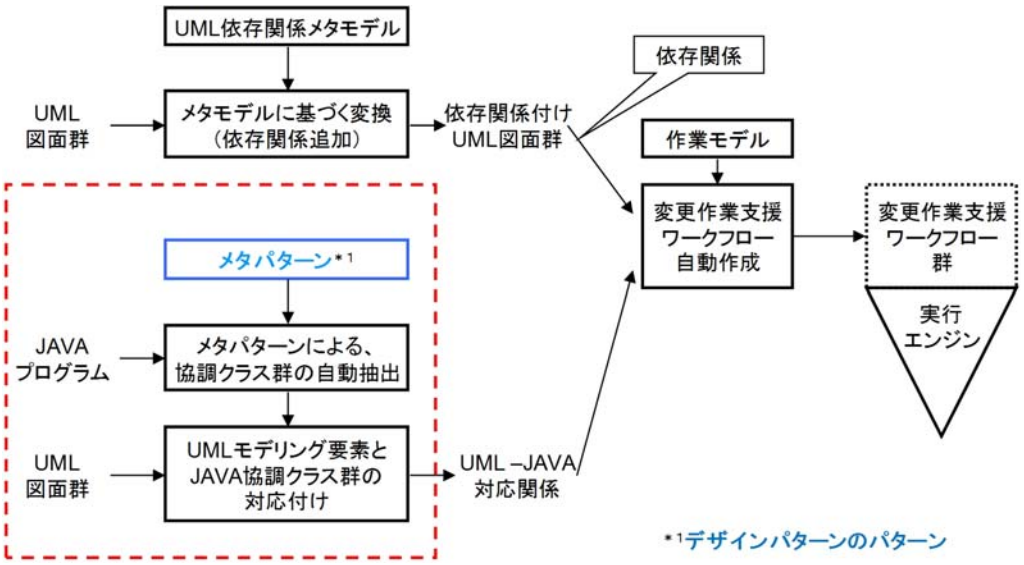

図 1.1: 変更作業支援ワークフローの自動生成

1.2 研究の目的

本研究の目的はユースケースを実現する（クラス図中のクラス群に対応する）Java クラス群を協調クラス群と定義し、それを抽出することである。

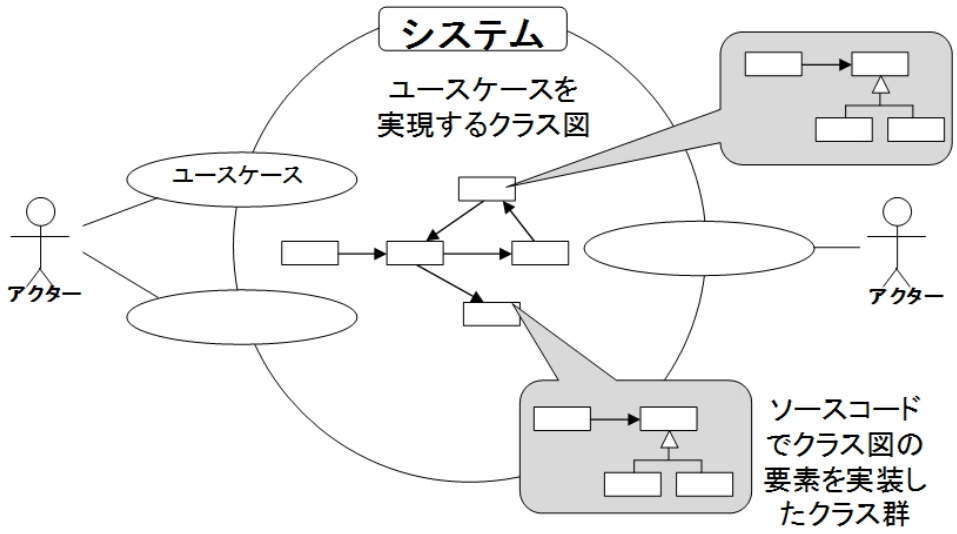

図 1.2: 協調クラス群

具体的には、ソフトウェア開発で、ユースケースを分析してクラス図が作成される。また、クラス図を実装する際に、クラス図の一つの要素が一つの Java クラスで実装されるわけではなく、クラス群で実装される。そのため、ユースケースを実装したクラス群に対応する Java クラス群を抽出するためにユースケースを実装したクラス群の一つひとつ

のクラスに対して対応をする Java クラス群を抽出してから、サブグラフ同型判定アルゴリズムを適用して対応をつける。

第2章　準備

2.1　ユースケース

ソフトウェア工学では、ユースケース [7] はシステムの機能について説明するもので、アクターとシステムの相互作用を表す。他の方法で言うとユースケースはユーザーの観点から連続にアクターによって起動されるイベントを説明する。

アクターはシステムと相互作用する人あるいはものである。アクターはロールであり、システムの個々のユーザーを表すものではない。アクターには、プライマリアクターとセカンダリアクターに分けられている。プライマリアクターはシステムの主要な機能を使うアクターで、セカンダリアクターはシステムを保守、管理、運用するアクターである。

2.2　デザインパターン

2.2.1　デザインパターンとは

ソフトウェア開発では、デザインパターンとは過去のソフトウェア設計者が発見し編み出した設計ノウハウを蓄積し、ソフトウェア設計に良く起きる問題に対する一般的な再利用可能な解法である。デザインパターンはコードに直接変換できるデザインではなく、様々な状況で利用できる問題解決方法で、オブジェクト指向設計におけて、クラスやオブジェクト間の相互関係及び相互作用を表すものである。

信頼できるシステムを最初から開発すると微妙な問題が頻繁に行われて、時間も非常にかかる。実証されたデザインパターンを利用することにより、ソフトウェア開発プロセスを高速化できるし、インプリメントするまで発見できる問題を避けられるし、コードも分かりやすくなる。個々のデザインパターンが異なる特徴を持ち、設計プロセスの柔軟性とカプセル化効果を高め、コードの繰り返しを減らす。これは大規模なシステムで非常に大きな問題である。

デザインパターンは専門家の経験から蒸留されて、それを利用すると成功した設計を繰り返すことである。他の方法で言うとそれは専門家の肩の上に立つことである。一般的にはデザインパターンが4つの基本的な要素（デザインパターン名・問題・解法・結果）を有している。

デザインパターン名は、設計問題とその解法及びその結果を表す名称である。パターン名は１、２語で記述される。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。パターンに関する語彙が増えれば、同僚と話したり、文章に記録したり、自分自身で考え

を整理するのにも役立つ。設計に関して検討するときや、設計上のトレードオフを人に伝えることも容易になる。

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、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。

解法は、設計の要素、それらの関連及び責任、協調関係を記述する。解法は特定の具体的な設計や実装の記述はしない。なぜならば、パターンは様々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、パターンは設計問題を抽象的に記述し、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。

結果は、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、極めて重要である。ソフトウェアに対する結果は、しばしば容量や時間のトレードオフに関するものとなる。言語や実装に関する問題も扱う。また、オブジェクト指向設計においては再利用が重要な要因である場合が多いので、パターンの結果に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。これら の結果で理解や評価もしやすくなる。

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のに有用となるよう にしたものである。また、デザインパターンはそれに関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。すなわち、そのパターンはいつ適用すべきか、設計上のその他の 制約を考慮した上でも適用できるか、そのパターンを使うことによる結果、さらには使う場合と使わない場合のトレードオフを記述しているのである。

デザインパターンを分かるため、デザインパターンの説明書がある。説明書はデザインパターンを利用すればよい背景、問題、推薦解決策を説明する。デザインパターンを文書化するための標準形式がない。よく利用されているのは次のようである。

パターン名と分類：デザインパターンを識別及び参照すると説明及び一意的な名前である。

意図：デザインパターンを利用する目標及び理由を説明する。

他の名：デザインパターンの他の名である。

動機：デザインパターンの利用するべき背景及び状況を説明する。

適用性：デザインパターンがどの状況、背景に適用できるかを説明する。

構造：クラス図と相互作用図を使ってデザインパターンのグラフィック描写を表す。

参加要素：デザインパターンで利用されるクラスリスト及びオブジェクトリストとその役割を説明する。

協調：デザインパターン内のクラスとオブジェクト間の相互作用を説明する。

結果：デザインパターンを利用して発生される結果、副作用、得失を説明する。

実装：デザインパターンの一つの実装を説明する。

コードの実例：プログラミング言語でデザインパターンをどのように利用される一つの実例である。

既知の用途：実際にデザインパターンの利用法の例である。

関連パターン：このデザインパターンと関係がある他のデザインパターンを指して、類似点と相違点を説明する。

2.2.2　デザインパターンの分類

問題の解決方法により、デザインパターンは生成に関するパターン、構造に関するパターン及び振る舞いに関するパターンに分類されている。GoF（Gamma Erich, Richard Helm, Ralph Johnson, John Vlissides）の著作『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』の中で 23 種のデザインパターンを表 2.1 のように分類されている。

2.3　メタパターン

2.3.1　メタパターンとは

メタパターンの意味を理解するために言語自体を分析する。英語では、メタは接頭語で別の概念から抽象化する概念である。それで、この研究の範囲ではメタパターンは複数のデザインパターンを抽象化するパターンである。他の方法で言うとメタパターンはデザインパターンのパターンである。

メタパターンは各種類のフレームワークの開発ツールとして利用されている。メタパターンの作用を理解するために、まず、フレームワークを理解する必要がある。

フレームワークはシステム作成を支援するために協力し合うように設計されたクラス群である。各フレームワークは一定の領域で活動する。例えば、グラフィカル・ユーザー・インターフェース（GUI）を作成するための設計されたフレームワークは GUI を作成する領域内で活動する。フレームワークを設計する際、開発者はフレームワークの利用者が何かに達したいかを明確に知らないため、エンドユーさーが機能を追加できるために余地を残す必要がある。そのため、特定の地域が作成され、それはフレームワークのホットスポットと言われている。いわゆる“ フレームワークのコールドスポット ”はフレームワークの基準機能を実装された領域である。

フレームワークを設計するコツはこれらのホットスポットを見つけることである。これらのホットスポットはユーザーが適応させる領域ため、メタパターンが助けにならない。しかし、これらのホットスポットが認定されてから、メタパターンはフレームワークの周囲を設計することに対して役になる。このタスクを実行するため、William Pree は７種のメタパターンを定義した。詳細は次のようである。

種類	デザインパターン	説明
生成に関するパターン	Abstract Factory	具体的なクラスを指定せずに関連するオブジェクトを作成するためのインターフェイスを提供する。
	Builder	複雑な構造を表現によって各部分に分けるため、同一のプロセスで多様な表現を作成できる。
	Factory Method	オブジェクトを作成するインターフェイスを定義して、サブラスで具体的な作成を行う。
	Prototype	同様のインスタンスを生成するために、原型のインスタンスを複製する。
	Singleton	一つしかないインスタンスが存在するデザインパターンである。
構造に関するパターン	Adaptor	互換性がない二つのインターフェイスを他のクラスを通じて接続する。
	Bridge	橋渡しを通じて、クラスを複数の方向に拡張させる。
	Composite	木構造のような再帰的なデータ構造を表現する。
	Decorator	既に存在するオブジェクトに新機能を動的に追加する。
	Facade	関連するクラス群を簡単に利用するために窓口のような一つのクラスに各手続きを集約する。
	Flyweight	メモリを減らすために多数のインスタンスを共有する。
	Proxy	別の物の代理人として手続きを行う。
振る舞いに関するパターン	Chain of Responsibility	イベントの送受信を行う複数のオブジェクトを鎖状につなぎ、それらの間をイベントが渡されてゆくようにする。
	Command	複数の命令や要求をオブジェクトとしてカプセル化する。
	Interpreter	文法規則をクラス構造で反映する。
	Iterator	オブジェクトの要素に順にアクセス手段を提供する
	Mediator	統一されたオブジェクトを仲介するオブジェクトを提供する。
	Memento	オブジェクトを以前の状態に戻す能力を提供するデザインパターンである。
	Observer	インスタンスを変更しても、他のインスタンスから見える。
	State	内部状態を変化することにより、オブジェクトの動作を変更する。
	Strategy	アルゴリズムの切替えを容易にする。
	Template Method	あるアルゴリズムの途中経過で必要な処理を抽象メソッドに委ね、その実装を変えることで処理が変えられるようにする。
	Visitor	データ構造を保持するクラスと、それに対して処理を行うクラスを分離する。

表 2.1: GoF の 23 種のデザインパターンの分類

2.3.2 フックとテンプレート

メタパターンに対して最も基本的な要素はテンプレートメソッドとフックメソッドとテンプレートクラスとフッククラスである。ここでは、テンプレートメソッド、フックメソッド、テンプレートクラス、フッククラスについて説明する。

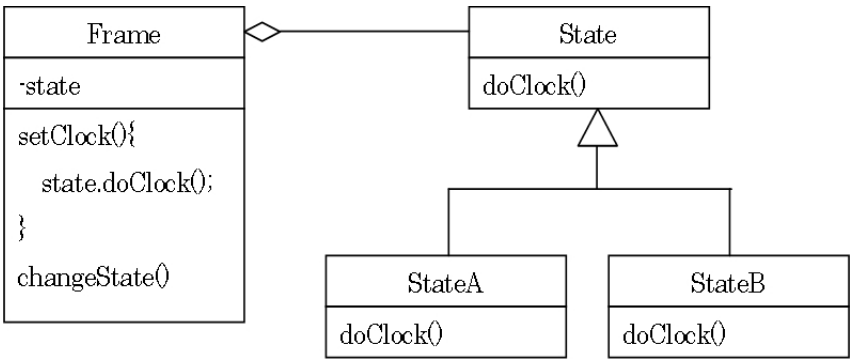

図 2.1: テンプレートとフックの例

フックメソッド：

上記のようにデザインパターンのホットスポットはサブクラスで抽象メソッドのオーバーライドによってカスタマイズされる。このとき、サブクラスでオーバーライドされることを想定し、少なくても他の一つのメソッドに呼び出されるメソッドはフックメソッドである。

図 2.1 では、State クラスの doClock() メソッドは各サブクラス StateA クラスと StateB クラスでオーバーライドされる。doClock() メソッドは Frame クラスの setClock() メソッドに呼び出される。doClock() メソッドはフックメソッドの一つの例である。

テンプレートメソッド：

フックメソッドを呼び出して具体的な機能を実行するメソッドはテンプレートメソッドである。テンプレートメソッドはフックメソッドの呼び出しを通じてパターン内のクラス間の依存関係を維持する。

図 2.1 では、Frame クラスの setClock() メソッドは State クラスのフックメソッドの doClock() メソッドを呼び出し、Frame クラスと State クラスの依存関係を維持する。Frame クラスの setClock() メソッドはテンプレートメソッドの一つの例である。

テンプレートクラス：

テンプレートクラスはテンプレートメソッドを持つクラスである。例では、Frame クラスで、setClock() メソッドはテンプレートメソッドであるため、Frame クラスはテンプレートクラスである。

フッククラス：

フッククラスはフックメソッドを持つクラスである。

例では、State クラスで、doClock() メソッドはフックメソッドであるため、State クラスはフッククラスの例である。

場合によってフックとテンプレートの役割を持つメソッドとクラスがある。テンプレートメソッドとフックメソッドは同じクラスに存在するとそのクラスはテンプレートフッククラスと呼ばれる。あるメソッドはフックメソッドを呼び出すがサブクラスでオーバーライドされるとテンプレートフックメソッドと呼ばれる。

2.3.3 Pree のメタパターン

William Pree はフックメソッドを持つクラス（フッククラス）とテンプレートメソッドを持つクラス（テンプレートクラス）の関係による、デザインパターンを抽象化して、7種のメタパターンをまとめた。この7種のメタパターンについて以下のようである。

（1）1：1結合メタパターン

1：1結合メタパターンでは、フッククラスが一つだけ存在する。テンプレートクラスはこのフッククラスの一つのインスタンスだけを参照する。テンプレートクラスとフッククラスの間継承関係がない。それは実行の時このメタパターンに基づくシステムの振る舞いが変更できる。

図 2.2: 1：1結合メタパターン

（2）1：N 結合メタパターン

図 2.3: 1：N 結合メタパターン

基本的には1：N 結合メタパターンは1：1結合メタパターンと同じですが、テンプレートクラスは一つ以上のフッククラスのインスタンスを参照する。フッククラスとテンプレートクラスの間に継承関係がなく、実行する時、パターンの振る舞いが固定されていない。

（3）1：1再帰的結合メタパターン

1：1再帰的結合メタパターンでは、フッククラスはテンプレートクラスの祖先で、テンプレートクラスはフッククラスの一つのインスタンスを参照する。開発者はこのパターンをインスタンスのチェーンを作成するために利用する。実行する時、ユーザーがパターンの振る舞いを変更できる。

（4）1：N 再帰的結合メタパターン

1：N 再帰的結合メタパターンは基本的に1：1再帰的結合メタパターンと同じように動作する。テンプレートクラスはフッククラスの子である。テンプレートクラスはフッ

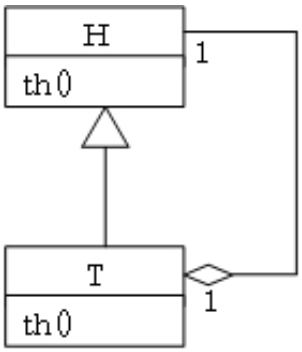

図 2.4: １：１再帰的結合メタパターン

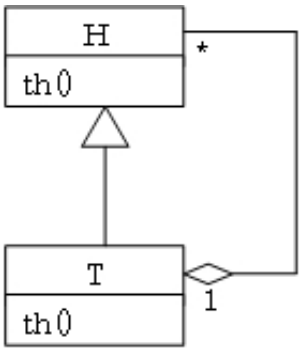

図 2.5: １：N 再帰的結合メタパターン

ククラスの一つ以上のインスタンスを参照するため、開発者はよくこのパターンを利用してインスタンスの木を作成する。実行する時、ユーザーがこのパターンの振る舞いも変更できる。

（５）統合メタパターン

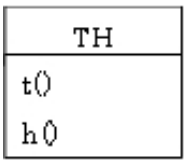

図 2.6: 統合メタパターン

統合メタパターンでは、テンプレートメソッドとフックメソッドは同じクラスに存在する。そのため、パターンの柔軟性が減る。ユーザーはフックメソッドをサブクラスでオーバーライドすることにより望む機能を追加するが、フックメソッドとテンプレートメソッドは同じクラスに存在して、実行する時にパターンの振る舞いを変更できない。

（６）１：１再帰的統合メタパターン

原則として、１：１再帰的統合メタパターンは１：１再帰的結合メタパターンと同じですが、テンプレートメソッドとフックメソッドは一つのメソッドで同じクラスに存在

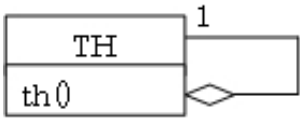

図 2.7: １：１再帰的統合メタパターン

する。

（７）１：N 再帰的統合メタパターン

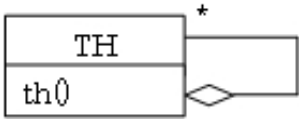

図 2.8: １：N 再帰的統合メタパターン

１：N 再帰的統合メタパターンは原則として１：N 再帰的結合メタパターンと同じである。テンプレートメソッドとフックメソッドは一つのメソッドで同じクラスに存在する。テンプレートクラスはフッククラスの複数のインスタンスを参照するため、このパターンはインスタンスの木を作成するためによく利用されている。

2.3.4　メタパターンによる GoF のデザインパターンの分類

GoF の２３種のデザインパターンは以下の木のように各メタパターンに分類できる。２３種のデザインパターンでは、Visitor デザインパターンは１：１結合メタパターンと１：N 結合メタパターンを含む。Facade デザインパターンと Memento デザインパターンと Singleton デザインパターンはメタパターンを含まないデザインパターンである。

2.4　適合率・再現率

本研究では、協調クラス群を抽出するアルゴリズムの精度を適合率・再現率により、評価する。本章は適合率・再現率について説明する。

適合率は抽出した協調クラス群の中にどれだけに抽出に適合したクラス数を含んでいるかという正確性の指標である。再現率は抽出対象としているクラス群の中に抽出した結果がどのぐらい適合しているかという網羅性の指標である。

N は抽出した結果のクラス数として、C は抽出する対象の正解クラス数として、R は抽出された適合クラス数とすると。

適合率は：

$$\text{Precision} = \frac{R}{N}$$

再現率は：

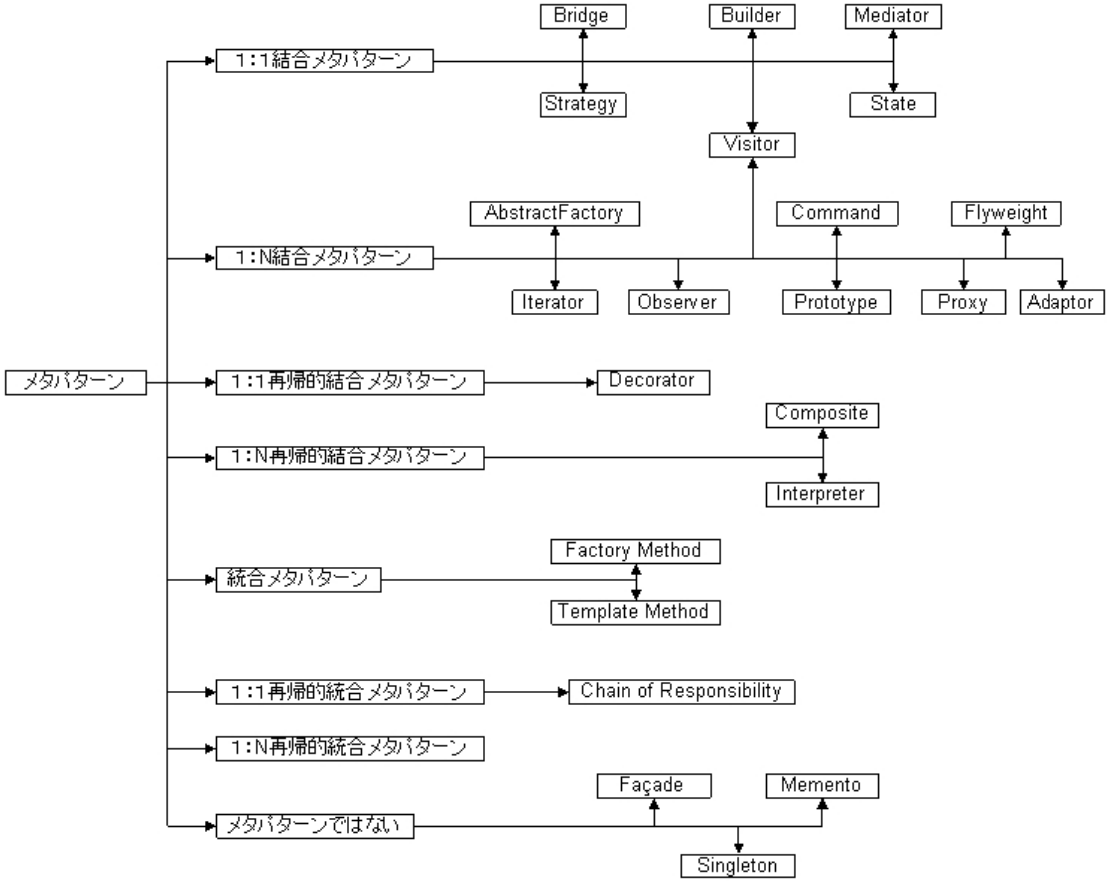

図 2.9: メタパターンによる GoF の２３種のデザインパターンの分類木

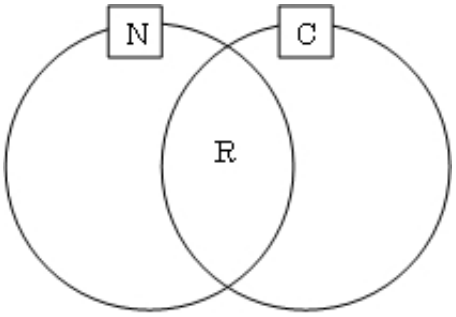

図 2.10: 適合率と再現率

Recall = $\frac{R}{C}$

適合率を上がると再現率が下がる。逆に再現率を上がると適合率が下がる。適合率と再現率を評価するために調和平均（F 値）をまとめる。

F-measure = $\frac{2.precision.recall}{precision+recall} = \frac{2R}{N+C}$

F 値は高ければ抽出アルゴリズムの性能が良いと意味する。

2.5　サブグラフ同型判定アルゴリズム（Jeffrey David Ullman）

サブグラフ同型判定アルゴリズムはグラフH（Vh、Eh）の中にグラフG（Vg、Eg）のグラフ同型を全て抽出するアルゴリズムである。他の方法で言うとグラフHの中ではグラフGと全く同じ部分を抽出するアルゴリズムである。

グラフGのノード数はnとして、グラフHのノード数はmとすると。グラフGはA[n, n]メイトリックスで表せ、グラフHはB[m, m]メイトリックスで表せる。

M’ [n, m]メイトリックスは０値と１値しか持たなくて、一行の値の合計が１で一列の合計が１以下であるメイトリックスと定義する。) M’メイトリックスを利用してBメイトリックスの列と行の順序を変えてCメイトリックスが作成できる。C = [ci, j] = M’(M’B)Tと定義し,Tは転位である。「$\sqrt{}$ i, j($1 \le i, j \le n$):A[i, j]=1 → C[i, j]=1（*）」と言うことが正しいとM’がグラフGとグラフHの部分の同型を表現すると言われる。それはM’[i, j]=1の場合で、グラフHのj番目ノードがグラフGのi番目ノードを対応すると言うことである。

最初Mo[n, m]メイトリックスを次のように作成する。

グラフHのj番目ノードのリンク数がグラフGのi番目ノードのリング数より大きいとMo[i, j] = 1。逆に、Mo[i, j] = 0。

更に、グラフHのj番目ノードがグラフGのi番目ノードを対応できないと明らかに分かれば、Mo[i, j] = 0と設定する。

サブグラフ同型判定アルゴリズムは可能性があるM’メイトリックスを全て（M’[i,j]=1 → Mo[i, j]=1）作成して、条件（*）で同型を検討することにより活動する。

アルゴリズムはグラフGの各ノードの一つずつに対してグラフHにある対応ノードを検索する。対応したら、検索の深さを加えて、次のノードに対して対応ノードを検索する。検索の深さは現在にいくつかの対応ができたかを表現する。アルゴリズムで使う変数dは検索の深さである。配列F[m]は現在どの列らが利用されているかをチェックする配列として（F[i]=1は利用されていることでF[i]=0は利用されてないこと)、配列H[n]がどの検索の深さで列が登録されるかをチェックする配列として、Mdは検索の深さｄでMのコーピである。アルゴリズムは次のようである。

ステップ１：

M = Mo; d = 1; E[1] = 0;

For all $1 \le i \le m$ F[i] = 0;

ステップ２：

M[d, j] = 1 と F[j] = 0 を満たすjが存在しないとステップ７に移動する。

Md = M;

If d = 1 then k = E[1]

Else k = 0;

ステップ３：

k++;

if M[d, k] = 1 or F[k] = 1 then ステップ３に戻す。

For all j # k M[d, j] = 0;

ステップ４：

If d < n then ステップ６に移動する。
Else 条件（*）を利用して同型をチェックする。条件を満たすと出力する。
ステップ５：
j > k と M[d, j] = 1 と F[j] = 0 を満たす j が存在しないとステップ７に移動する。
M = Md;
ステップ３に移動する。
ステップ６：
H[d] = k; F[k] = 1; d++;
ステップ２に移動する。
ステップ７：
If d = １ then 終了。
F[k] = 0; d–; M = Md; k = H[d];
ステップ５に移動する。

第3章　関連研究

Java の協調クラス群を発見することを目標にして、落水研究室ではメタパターンによるアプローチと参照関係の強さによるアプローチがある。

3.1　メタパターンによるアプローチ

更に、2006 年に、金旭東 [1] がメタパターンを用いたクラス群を協調クラス群と定義し、Pree の 6 種のメタパターンを 3 つの協調構造に分類して、各協調構造の特徴による協調クラス群の抽出アルゴリズムを開発し、一定の結果を得た。本章は金旭東の研究についてまとめる。

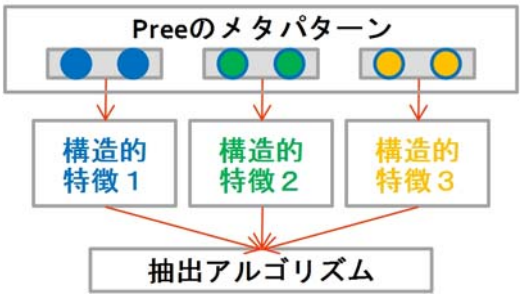

図 3.1: 金のアプローチ

１９９５年に Pree[3] は 7 種のメタパターンを定義した。その中に１：N 再帰的統合メタパターンは GoF の 23 種のデザインパタンに含まれないメタパターンで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であるため、残り 6 種のメタパターンを構造的特徴により、次の表 3.1 のように 3 つの協調構造にまとめた。

次いでに、3 つの協調構造を分析して、テンプレートメソッド、フックメソッド、テンプレートクラスとフッククラス間の協調関係をまとめて、ソースコードにおける関連クラス群の固まりの抽出アルゴリズムを開発した。その抽出アルゴリズムは GoF の 23 種のデザインパターンを含むソースコードで検討して、17 種のデザインパターンを含む関連があるクラス群を抽出できた。結果は表 3.2 に示す。

抽出できなかった 6 種のデザインパターンは Factory Method, Flyweight, Abstract Factory デザインパターンとメタパターンを含まない Facade, Singleton, Memento デザインパターンである。

メタパターン	３つの協調構造
統合メタパターン １：１再帰的統合メタパターン	統合的協調構造
１：１結合メタパターン １：N 結合メタパターン	結合的協調構造
１：１再帰的結合メタパターン １：N 再帰的結合メタパターン	再帰的結合協調構造

表 3.1: メタパターンによる３つの代表的な構造へ分類

メタパターン	GoF のデザインパターン	抽出可・否
統合メタパターン	Template Method	抽出可能
	Factory Method	未対応
１：１結合メタパターン	Builder, Bridge, State, Strategy,	抽出可能
	Mediator, Visitor	抽出可能
１：N 結合メタパターン	Prototype, Adapter, Observer,	抽出可能
	Iterator, Proxy, Command	抽出可能
	Flyweight, Abstract Factory	未対応
１：１再帰的結合メタパターン	Decorator	抽出可能
１：N 再帰的結合メタパターン	Composite, Interpreter	抽出可能
１：１再帰的統合メタパターン	Chain of Responsibility	抽出可能

表 3.2: GoF デザインパターンの抽出結果

金旭東の抽出アルゴリズムにより、対応できない原因は次のようである。

1. メターパターンを含まない各デザインパターンの協調構造を抽出できない。金旭東は各メタパターンの構造的特徴に基づいて、抽出アルゴリズムを開発したため、その 6 種のメタパターンを含むソースコードにおける関連クラス群しか抽出できない。他の方法で言うと、Singleton デザインパタン、Facade デザインパターン、Memento デザインパターン等を抽出可能でない。

2. ソースコードを解析する際、直接にメソッドを呼び出す場合しか解析したが間接にメソッドを呼び出す場合を見落とした。

図 3.2 では、ImageReaderFactory クラスの getImageReader() メソッドは GifReader クラスのフックメソッドを直接呼び出さず、新しいインスタンスだけ呼び出した。たとえば、GifReader クラスのコンストラクタ（構築子、Constructor）を呼び出した。コンストラクタで他のフックメソッド（getDecodedImage）を呼び出した。この場合では GifReader クラスのコンストラクタメソッドがフックメソッドと認められる。

3. 更に、実際には、開発者により、メタパターンを使わないで自由に実装する場合もある。そして、この場合も金旭東の開発したアルゴリズムが対応できない。

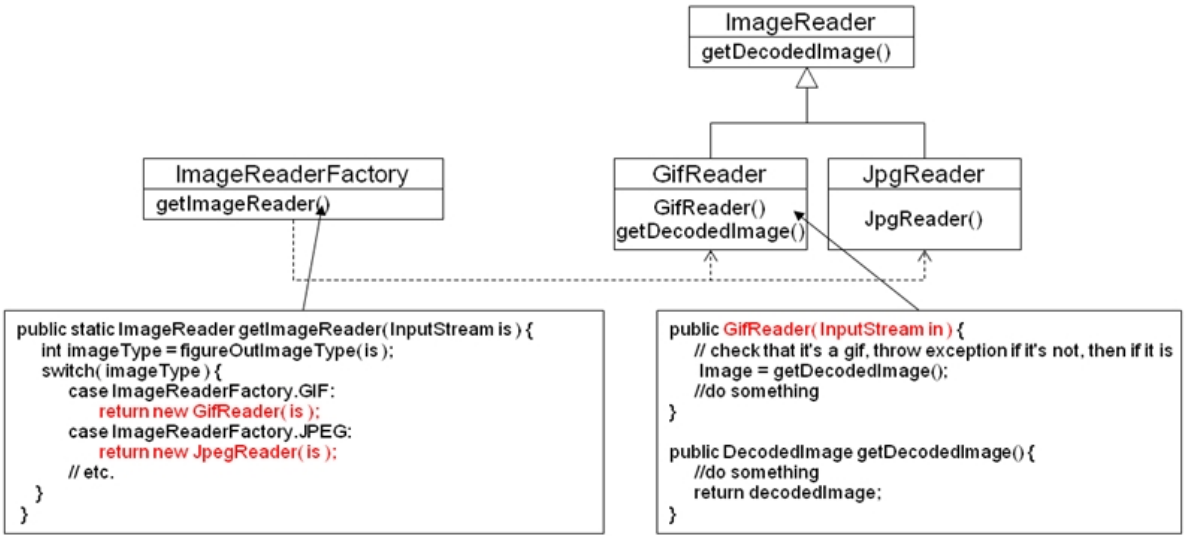

図 3.2: 見落とす場合の例

例えば、社員の情報を処理するために、社員リストというクラスが設計された。分かりやすいため、開発者により、社員リストクラスと社員情報クラスで実装した。社員情報クラスは社員の名前、社員番号、性別等の情報を持ち、社員リストはただ一つのリストの機能を持つクラスである。こう実装したらメタパターンを含まなくて開発したアルゴリズムが抽出できなくなる一つの例である。

3.2 参照関係の強さに基づく抽出アルゴリズム

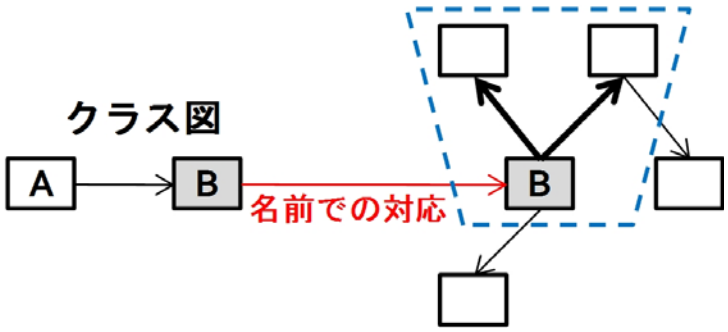

図 3.3: 菅井のアプローチ

2007 年に、菅井拓海 [2] ははクラス図の要素に対応する Java クラス群を協調クラス群と定義して、クラス図の要素の名前と一致する Java クラスがクラス群の中心として、クラス間の参照関係の強さに基づいてクラス群を広めて、抽出するアルゴリズムを開発した。抽出アルゴリズムは次の 4 ステップにまとめる

1. **モデル要素の名前と一致する協調クラス群のコアを見つける。**

2. **コアから、汎化・実現の関係を追跡し，継承グループを作成する。**

3. **継承グループの各要素から追跡ルールに従い関係を追跡する。**

4. **追跡できる関係が無くなるまで、ステップ３を繰り返す。**

この抽出アルゴリズムは各モデル要素の名前は実装したソースコードに全て見つけないと（実装した時、クラスの名前が設計の名前と全部同じでない場合）間違って抽出する。実際には開発者により、全てモデル要素の名前と同じのように実装することはあまりないため、アルゴリズムの応用性が低い。

第4章　研究方法

協調クラス群を抽出するために、我々は先行研究を改良して、ユースケースに含まれる Java クラス群を抽出する。このことを達するために、二つの課題がある。

1. メタパターンで説明できないデザインパターンを解析可能にする。具体的には、メタパターンの構造的特徴を利用すること以外に、新しく「構造の特徴」と「振る舞いの特徴」を導入して、メタパターンを適用するクラス群を抽出する。

2. ユースケースを実装したクラス群に対応する Java クラス群を抽出する。具体的には。ユースケースを実装したクラス図の一つひとつのクラスに対して対応をする Java クラス群を抽出する。その後、サブグラフ同型判定アルゴリズムを適用して対応をつける。

4.1　アプローチ

クラス図を実装する際に、デザインパターンを使って実装することが多いため、協調クラス群を抽出するためにデザインパターンを利用したクラス群を抽出することが大切である。

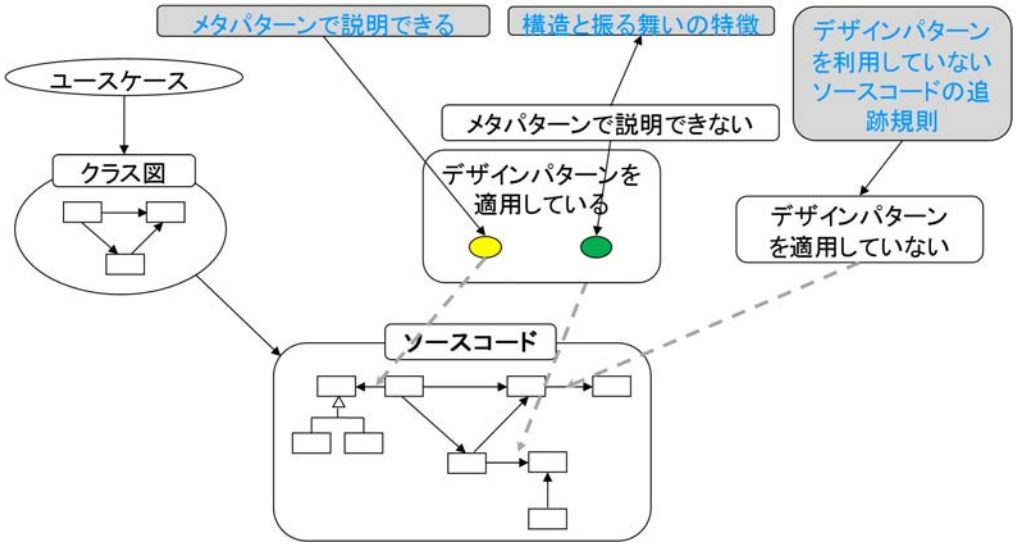

図 4.1: アプローチの概要

デザインパターンの大部分がPreeのメタパターンで説明できるため、デザインパターンを利用したクラス群を抽出するために各デザインパターンを一つずつ抽出することと比べて、メタパターンに基づいて抽出することがより簡単である。その大部分のデザインパターンを利用したクラス群を抽出するアルゴリズムは先行の失敗した場合を調べて、直してから開発する。またメタパターンで説明できないデザインパターンを解析可能にするために、メタパターンで説明できないデザインパターンにたいして、利用したクラス群の構造と振る舞いの特徴をまとめて、抽出アルゴリズムを開発する。また、デザインパターンを使わないで実装する場合があるため、デザインパターンを適用していない箇所を抽出するためにデザインパターンを利用していないソースコードの追跡規則を開発する。

4.2 具体的な解析手順

ユースケースを実装したクラス図に対応するJavaクラス群を抽出するために、次の解析手順を行う。

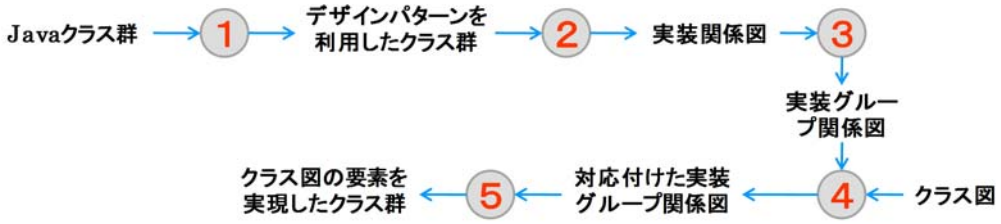

図 4.2: 解析手順

プロセス１：
デザインパターンを利用したクラス群を抽出する。
プロセス２：
デザインパターンを利用したクラス群の特徴に基づいて、実装関係図を作成する。（実装関係については後で説明する）
プロセス３：
実装グループ間の関係を表す実装グループ関係図を作成する。（実装グループについては後で説明する）
プロセス４：
実装グループ関係図の要素とクラス図の要素の対応をつける。
プロセス５：
依存関係の追跡に基づいて、実装グループを結合して、クラス図の要素を実現するクラス群を発現する。
これから、各プロセスについて詳しく説明していく。

第5章　実現

ここでは、前章に載っている解析手順での各プロセスについて詳しく説明する。

5.1　デザインパターンを利用したクラス群の抽出

デザインパターンを利用したクラス群を抽出するために、メタパターンで説明できるデザインパターンを利用したクラス群の抽出とメタパターンで説明できないデザインパターンを利用したクラス群の抽出に分ける。メタパターンで説明できるデザインパターンを利用したクラス群の抽出について、先行研究の失敗した場合を改良してメタパターンを利用したクラス群を抽出するアルゴリズムを開発する。メタパターンで説明できないデザインパターンを利用したクラス群の抽出について、新しく「構造の特徴」と「振る舞いの特徴」を利用して抽出アルゴリズムを開発する。

5.1.1　メタパターンに基づいてデザインパターンを利用したクラス群の抽出

このステップでは、メタパターンに基づいて、デザインパターンを利用したクラス群を抽出することについて説明する。

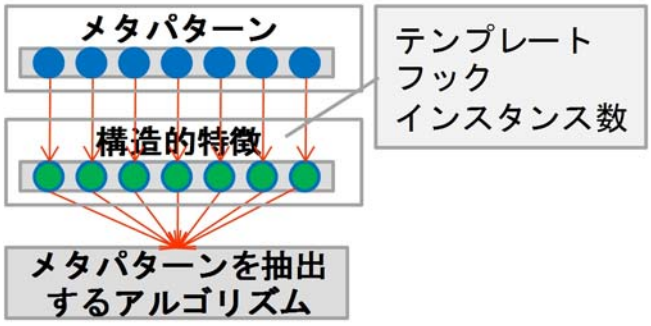

図 5.1: メタパターンの抽出

図5.1のように、一つ一つのメタパターンに対してテンプレート、フック及びインスタンス数により、構造的特徴をまとめて。その構造的特徴に基づいてメタパターンの抽出方法を提案する。Pree の７種のメタパターンの構造的特徴は次のようである。

統合メタパターンの構造的特徴

メタパターンの構造においては、テンプレートメソッドはフックメソッドを呼び出し、このテンプレートメソッドとフックメソッドは同じクラスに存在し、テンプレートメソッドは再帰しない。

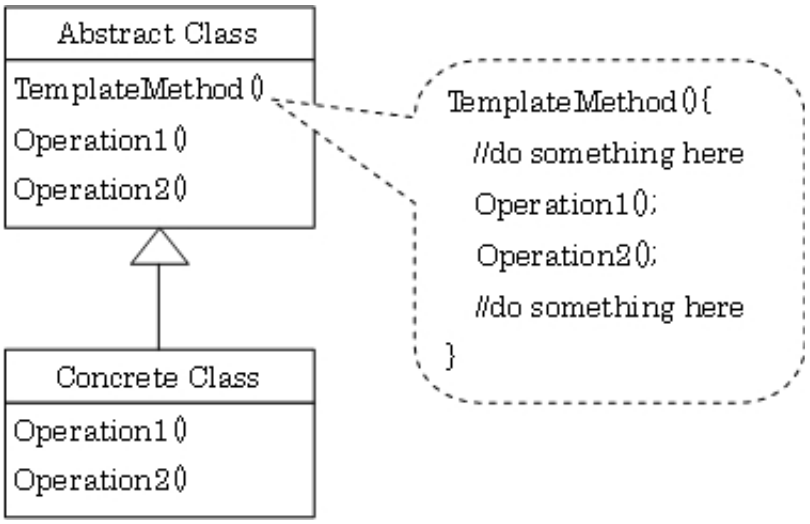

図 5.2: 統合メタパターンの構造的特徴

図 5.2 では、Operation1（）と Operation2（）はサブクラスでオーバーライドされて、フックメソッドである。そのフックメソッドらを呼び出す TemplateMethod() メソッドはテンプレートメソッドで一緒に AbstractClass に存在する。このパターンではテンプレートメソッドは再帰しない。再帰する場合は次の各再帰的統合メタパターンの構造で説明する。

１：１再帰的統合メタパターンの構造的特徴

１：１再帰的統合メタパターンの構造は統合メタパターンの構造と大分同じであるがテンプレートの再帰動作がある。テンプレートメソッドはフックメソッドを呼び出す。このテンプレートメソッドとフックメソッドも同じクラスに存在するが、テンプレートメソッドはクラスの他の一つだけのインスタンスを通じて自分を呼び出す。

図 5.3 では、Operation（）メソッドはサブクラスでオーバーライドされて、フックメソッドである。TemplateMethod（）はフックメソッドを呼び出し、フックメソッドと一緒に AbstractClass クラスに存在する。更に、テンプレートメソッドでは、AbstractClass クラスのインスタンス（Next）を持ち、インスタンスを通じて再帰する。分かりやすいために、この構造は動的なリストの構造を見てもいい。

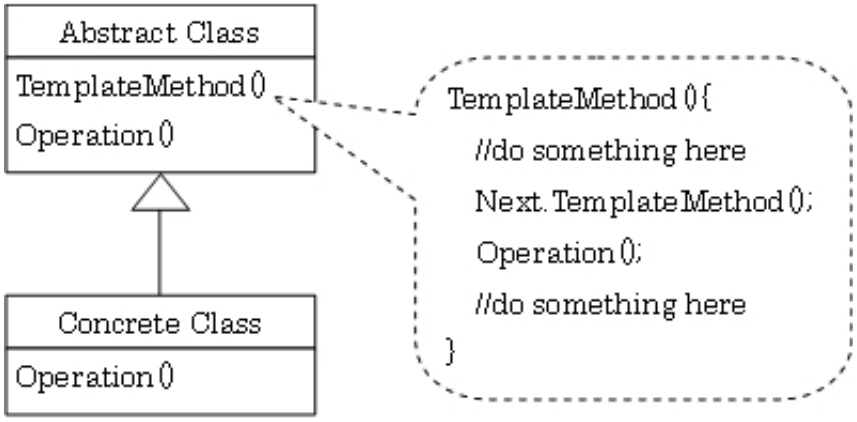

図 5.3: １：１再帰的統合メタパターンの構造的特徴

１：N 再帰的統合メタパターンの構造的特徴

基本的には１：N 再帰的統合メタパターンの構造は１：１再帰的統合メタパターンの構造と同じである。フックメソッドはサブクラスでオーバーライドされ、テンプレートメソッドはフックメソッドを呼び出し、テンプレートメソッドとフックメソッドは同じクラスに存在する。違う点はテンプレートメソッドで一つ以上のクラスのインスタンスを通じて再帰することである。

例えば、１：１再帰的統合メタパターンの構造の例だが、Next と言うインスタンスの代わりに Left と Right と言うインスタンスを使って再帰する。テンプレートメソッドのコードは次のようである。

```
//ソースコード
        Left.TemplateMethod();
        Right.TemplateMethod();
```

図 5.4: コードの例

データ構造では、このメタパターンの構造は木構造のように連想してもいい。

１：１結合メタパターンの構造的特徴

このメタパターンの構造においては、フックメソッドはサブクラスでオーバーライドされる。フックメソッドを呼び出すテンプレートメソッドはフックメソッドを含むクラスと別のクラスに存在する。この二つのクラス間に継承関係がない。テンプレートメソッドはフッククラスの一つだけのインスタンスを通じてフックメソッドを呼び出す。それは１：１関係である。

図5.5では、Stateクラスの doClock（）メソッドは State1 クラスと State2 クラスでオーバーライドされて、フックメソッドである。AbstractClass クラスの setClock（）メソッドは State の一つのインスタンス（state）を通じてフックメソッドを呼び出す。この二つのクラスの間に継承関係がない。これは１：１結合メタパターンの構造の一つの例である。

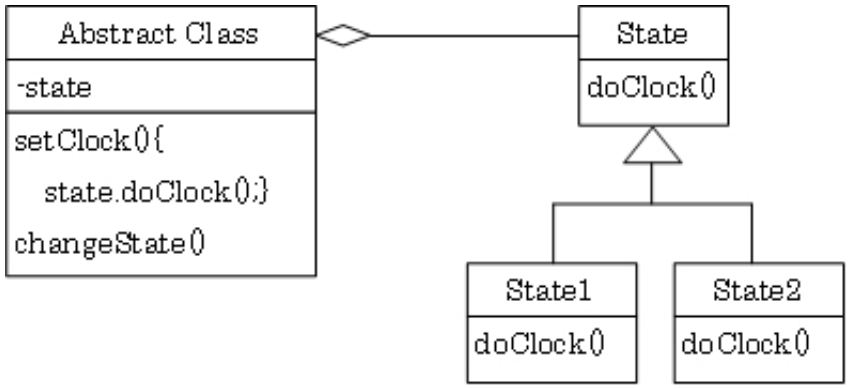

図 5.5: １：１結合メタパターンの構造的特徴

１：N 結合メタパターンの構造的特徴

基本的には、１：N 結合メタパターンの構造は１：１結合メタパターンの構造と同じである。フッククラスはサブクラスでオーバーライドされ、テンプレートメソッドはフックメソッドを呼び出し、フックを含むクラスと別のクラスにそんざいする。しかしそのテンプレートを含むクラスはフックメソッドを含むクラスの一つ以上のインスタンスを参照する特徴がある。

１：１結合メタパターンの構造の例で state だけではなく、テンプレートメソッドは state1.doClock() と state2.doClock() をインプリメントされたら、１：N 結合メタパターンの構造の例である。

１：１再帰的結合メタパターンの構造的特徴

このメタパターンの構造では、フックメソッドはサブクラスでオーバーライドされ、テンプレートメソッドはフックメソッドを含むクラスの一つだけのインスタンスを通じてフックメソッドを呼び出し、テンプレートメソッドを含むクラスはフックメソッドを含むクラスを継承するという特徴がある。次の例でこの特徴を詳しく説明する。

図5.6ではメタパターンの構造の例で Decorator デザインパターンの構造である。Decorator クラスは Component クラスを継承して、Operator() メソッドをオーバーライドする。Decorator クラスの Operator() メソッドは Component クラスのインスタンスを通じて Component クラスの Operator() をよびだす。この場合では、Operator（）メソッドはフックとテンプレートの両方の役割を持つ。

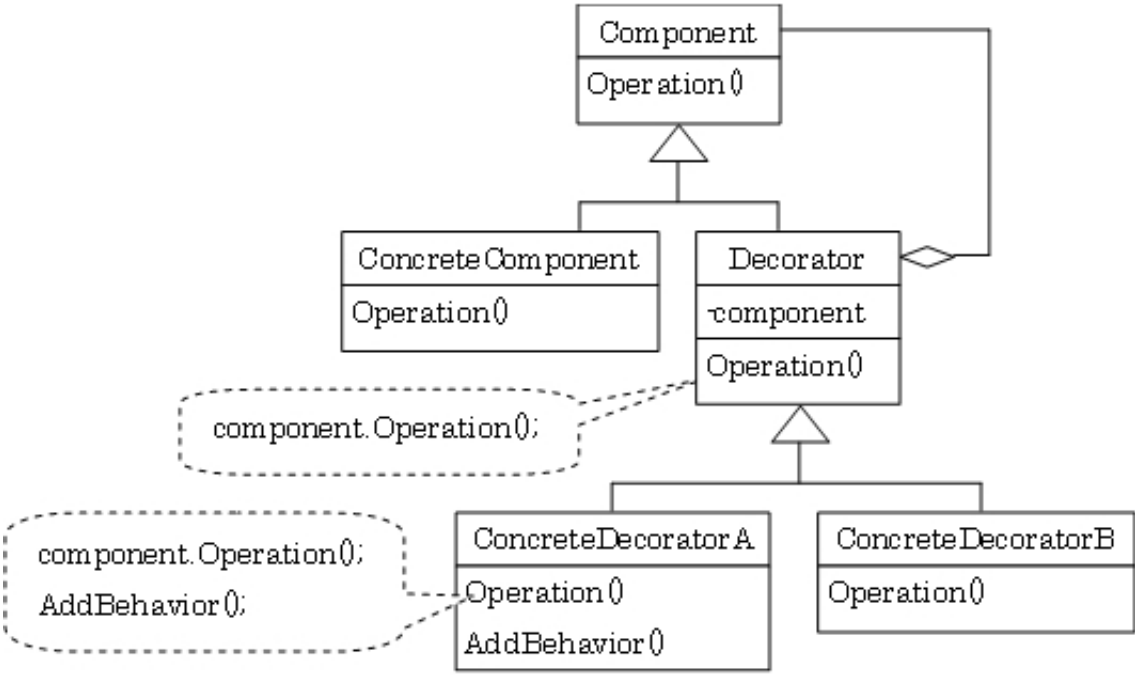

図 5.6: １：１再帰的結合メタパターンの構造的特徴

１：N 再帰的結合メタパターンの構造的特徴

１：１再帰的結合メタパターンの構造のように１：N 再帰的結合メタパターンの構造では、フックメソッドはサブクラスでオーバーライドされ、テンプレートメソッドはフックメソッドを呼び出し、テンプレートメソッドを含むクラスはフックメソッドを含むクラスを継承する。しかし、このメタパターンの構造ではテンプレートメソッドはフックメソッドを含むクラスの一つ以上のインスタンスを通じてフックメソッドを呼び出す特徴がある。

例としては、図 5.7 は Composite デザインパターンの構造である。Composite クラスは Component クラスの子供であり、Operation() メソッドは Composite クラスでオーバーライドされる。Operation（）メソッドでは Component クラスのインスタンスのリストを通じて Component クラスの Operation（）メソッドを呼び出し、Component クラスを参照する。それは１対 N 関係である。

テンプレートとフックを探索

ソースコードにおけるメタパターンの構造を抽出するためにテンプレートメソッドとフックメソッドとテンプレートクラスとフッククラスを検索しなければならない。テンプレートとフックの定義による検索方法は次のようである。

ソースコードを解析して、各ラスのクラス名とメソッドリストと継承するクラスリストと言う情報と各メソッドのメソッド名と呼び出すメソッドリストという情報を取れたとすると。

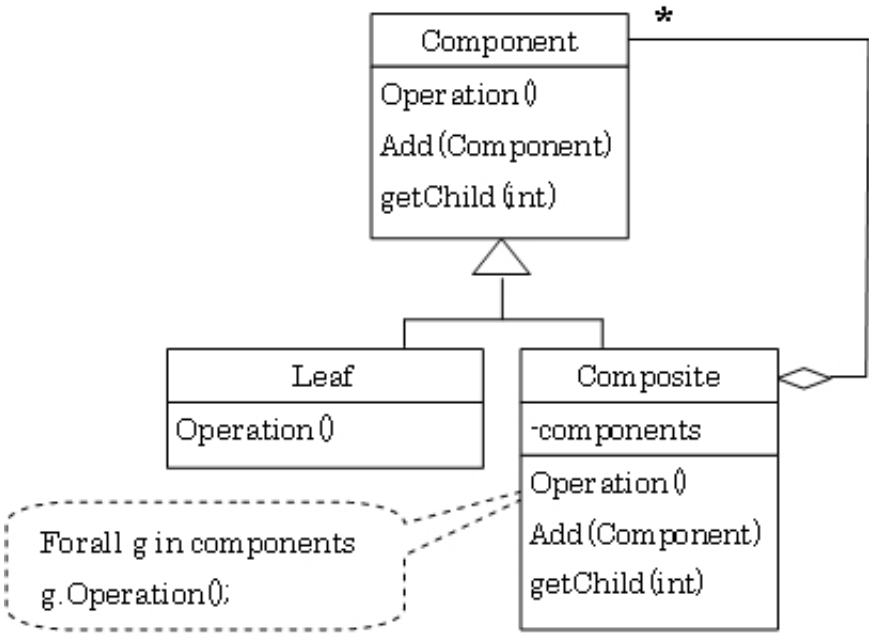

図 5.7: １：N 再帰的結合メタパターンの構造的構造

フックメソッドの検索方法：

Classeslist はソースコードにおけるクラス群で、Methodslist はクラスのメソッド群である。Overridelist はメソッドをオーバーライドするクラスリストである。Inheritlist はクラスを継承するクラス群である。BeCalledlist はメソッドを呼び出すメソッド郡である。Calllist はメソッドの呼び出すメソッド郡である。Callhooklist はメソッドの呼び出すフックメソッド群である。

```
Foreach C in Classeslist {
    Foreach M in C.Methodslist {
        M.Overridelist ← Null
        Foreach Ex in C.Inheritlist
        Foreach M1 in Ex.Methodslist
        If (M.name == M1.name) then {
            If (M.BeCalledlist != Null) then M ← Hook;
            If (M1.BeCalledlist != Null) then M1 ← Hook;
            M.Overridelist←Ex;
            M1.Overridelist←C;
        } //End If
    } //End For
} //End For
```

テンプレートメソッドの検索方法：

```
Foreach C in Classeslist {
    Foreach M in C.Methodslist{
        Foreach Mx in M.Calllist
            If (Mx is Hook) then{
                M ← Template;
                M.Callhooklist ←Mx;
            } //End If
    } //End For
} //End For
```

その後、各クラスに対してメソッドリストにフックメソッドがあるクラスはフッククラスで、テンプレートメソッドがあるクラスはテンプレートクラスでフックメソッドとテンプレートメソッドがあるクラスはテンプレートフッククラスである。

メタパターンの抽出アルゴリズム

総合的にはメタパターンを抽出するアルゴリズムは各メタパターンの構造的特徴によって、図 5.8 のように作成した。

一つのテンプレートクラス C に対して、各テンプレートメソッドを取り出して、テンプレートメソッドの呼び出しリストの中に各フックメソッドを取り出して、テンプレートメソッドを含むクラスとフックメソッドを含むクラスの関係とテンプレートメソッドとフックメソッドの関係に基づいて 7 種のメタパターンを抽出する。

5.1.2 構造と振る舞いの特徴に基づいたデザインパターンを利用したクラス群の抽出

ここでは。メタパターンで説明できないデザインパターンを利用したクラス群を抽出する手法について説明する。一つひとつのメタパターンに対して、構造と振る舞いの特徴をまとめて、抽出するメソッドを作成する。

構造の特徴はクラス間のインスタンス数、結合、インスタンス化と継承関係等の特徴である。

振る舞いの特徴はソースコードのメソッドの定義、変数の定義、戻る値とデータフロー等の特徴である。

デザインパターンを利用したクラス群の構造と振る舞いの特徴をまとめて、一つの特徴に対して抽出メソッドを作成する。これらのメソッドを Java ソースコードに適用して、まとめた特徴があるクラス群（デザインパターンを利用したクラス群）を抽出する。各構造と振る舞いの特徴について、Pattern in Java という本に載っているデザインパターンの抽出メソッドは付録 A で説明している。

分かりやすくために、次の例で説明する。

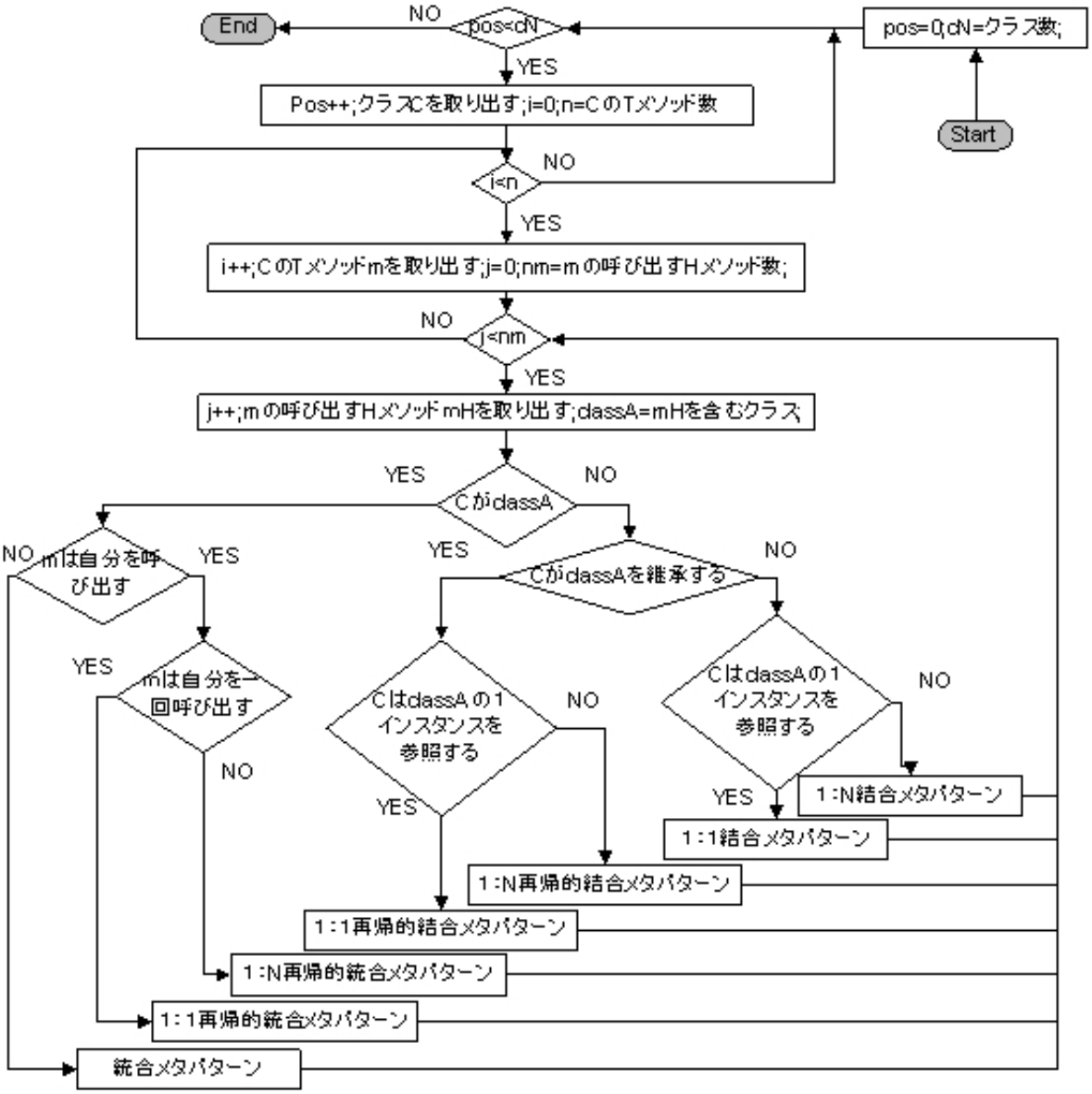

図 5.8: メタパターン抽出アルゴリズム図

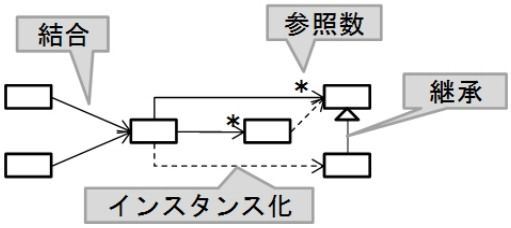

図 5.9: 構造の特徴

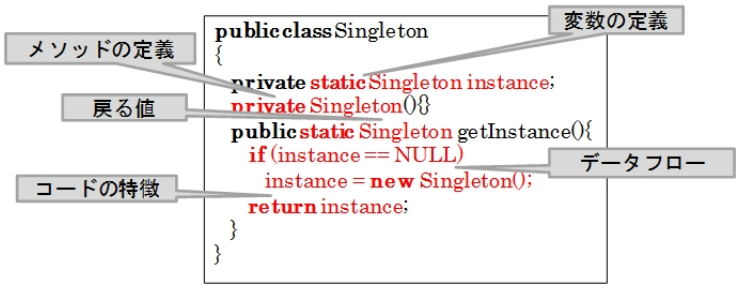

図 5.10: 振る舞いの特徴

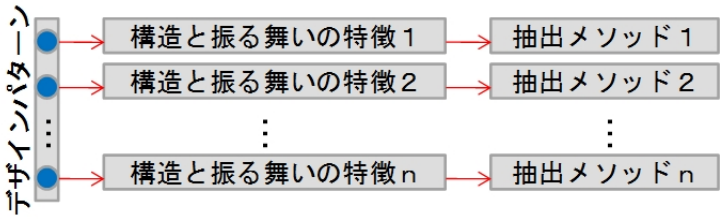

図 5.11: メタパターンで説明できないデザインパターンの抽出手段

GoF の Facade デザインパターンを用いたソースコードを抽出するメソッドは次のようである。

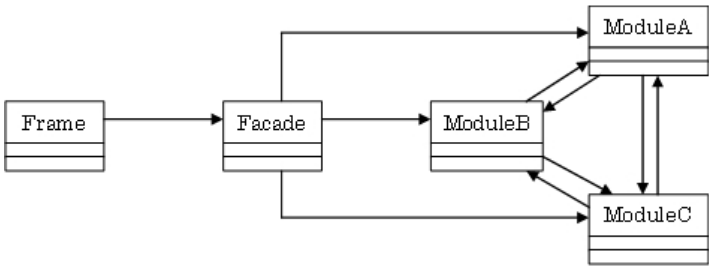

図 5.12: Facade デザインパターンの構造例

Facade デザインパターンを用いたソースコードを抽出するために、一つの C クラスに対して次の手続きを行う。

ステップ１：C クラスの参照するクラス群を A グループにまとめる。

ステップ２：C クラスを参照するクラス群を B グループにまとめる。

ステップ３：A グループの各クラスと B グループの各クラスは関係がないと Facade デザインパターンを用いたクラス群と判断できる。

GoF の Singleton デザインパターンを用いたソースコードを抽出するメソッドは次のようである。一つのクラスは Singleton デザインパターンかどうかを検討すると、振る舞いが次の特徴があるかどうかで検討する。

```
public class Singleton
{
    private static Singleton instance;
    private Singleton(){}
    public static Singleton getInstance(){
        if (instance == NULL)
            instance = new Singleton();
        return instance;
    }
}
```

図 5.13: Singleton デザインパターンのコード例

クラスのコードの中で一つの変数のタイプはそのクラスで static 変数である。その変数は A 変数とする。A 変数は一回だけクラスのインスタンスを設定する。更に、メソッドリストに A 変数を戻すメソッドがある。Constructor は private とクラス名で定義される。

5.2 デザインパターンを利用したクラス群の特徴に基づいく実装クラス図の作成

ここでは抽出したデザインパターンを利用したクラス群の特徴により、Java クラス間の二つのクラスの実装関係（同じクラス図の一つの要素を実装する関係）があるかどうかを判断する。Java クラス間の実装関係を表す図は実装クラス図と定義して作成する。実装クラス図は次のようである。

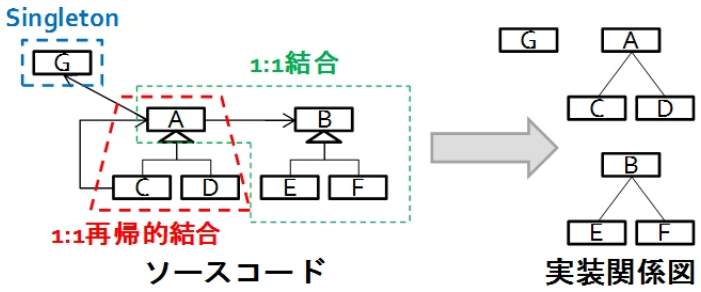

図 5.14: 実装クラス図作成の例

上例では、クラス A とクラス C は同じクラス図の一つの要素を一緒に実装していると判断された。しかし、ここまで、どの要素を実装しているかがまだ分からない。ある要素を実装しているのみと判断している。

5.2.1　メタパターンを含むクラス群の特徴に基づいて実装クラス図の作成

メタパターンの構造的特徴によって、メタパターンにおけるクラス間の実装関係を判断できる。統合メタパターンと１：１再帰的統合メタパターンと１：N 再帰的統合メタパターンを用いたクラス群の全ては同じ要素を実装するため、各クラスの間にどちらでも実装関係があると判断できる。１：１再帰的結合メタパターンと１：N 再帰的結合メタパターンを用いたクラス群はフッククラスとフックメソッドをオーバーライドする各クラスは実装関係があると判断でき、フッククラスとテンプレートクラスは実装関係があるかどうかとまだ判断できない。１：１結合メタパターンと１：N 結合メタパターンを用いたクラス群はフッククラスとフックメソッドをオーバーライドする各クラスは実装関係があると判断できる。

この判断の結果を利用して、実装関係があるとリンクを付けてから、実装関係がないと判断できるとリンクを消して、実装クラス図を作成する。

5.2.2　構造と振る舞いの特徴に基づいて実装クラス図の作成

構造と振る舞いの特徴に基づいて、メタパターンで説明できないデザインパターンを用いたクラス群を抽出できる。デザインパターンの構造と振る舞いの特徴によって、実装関係があるかないかを判断できる。この判断結果はメタパターンによる判断結果を結合して実装クラス図を完全に作成する。

各デザインパターンは個別の特徴を持ち、実装関係を判断するのはデザインパターンによって別々に行う。詳細的には、実験で行ったデザインパターンの特徴による判断法は付録 A に載っている。

5.3　実装グループ関係図の作成

実装クラス図を作成してから、関連がある各クラスを一つのグループにまとめる。このグループは実装グループと定義する。この実装グループらの間に所属するクラスらの間の依存関係によって実装グループ間の関係が形成される。その関係を表す図は実装グループ関係図と定義する。

5.3.1　幅優先探索方法を適用して実装グループの抽出

実装クラス図に幅優先探索方法を適用して、関連あるクラス群を実装グループにまとめられる。幅優先探索方法は次のようである。

Classeslist はソースコードのクラス群で、Groupid はクラスの所属グループの ID（所属しない場合はー１）で、groupnum はグループ数で、classnum はグループのクラス数で、Graph[i,j] はクラス i とクラス j の間の実装関係を表す（０は関係が無い場合）。

groupnum ← 0;

```
Foreach C in Classeslist do
    If (C.Groupid == -1) then{
        groupnum ← groupnum + 1;
        Group[groupnum].classnum ← 1;
        Group[groupnum][classnum] ← C;
        Pos ← 1;
        While (Pos < Group[groupnum].classnum){
            Foreach Cx in Classeslist do{
                If ((Cx.groupid == 0) and (Graph[Group[groupnum][Pos], Cx] != 0))then{
                    Group[groupnum].classnum ←Group[groupnum].classnum + 1;
                    Group[groupnum][classnum] ←Cx;
                    Cx.Groupid ← groupnum
                } //End If
            } //End For
            Pos←Pos + 1;
        } //End While
    } //End If
```

例えば、実装クラス図は下図のようである。二つのクラスの間に線があるのはその二つのクラスはクラス図の同じ要素を実装すると判断された。下図が入力として上のアルゴリズムを実行すると、結果は三つ実装グループが抽出される。A と D と E は一つのグループで B と C と G と H は一つのグループで F は一つのグループである。

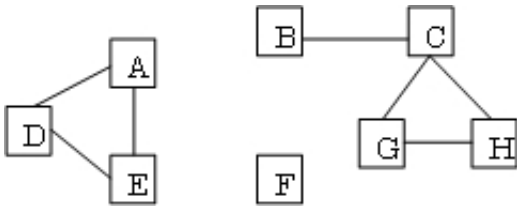

図 5.15: 実装クラス図の例

5.3.2 実装グループ関係図の作成

Java クラスを各実装グループに分け、実装グループ間の関係を抽出して実装グループ関係図を作成する。まず、実装グループ間の関係を次のように定義する。

グループ A がグループ B と関係があるというのはグループ A にあるクラスがグループ B にあるグループのインスタンスを参照することである。一対一関係はグループ A のクラスがグループ B のクラスのインスタンスを一つだけ参照することである。更に、一対多関係は一つ以上のインスタンスを参照することである。

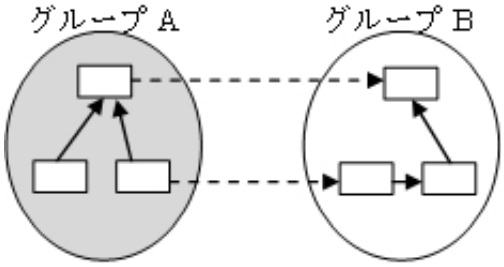

図 5.16: 実装グループ間の関係の例

各実装グループの間の関係を Java クラス間の依存関係に基づいて抽出して、実装グループは実装グループ関係図の要素として実装グループ関係図を作成する。

5.4 実装グループ関係図の要素とクラス図の要素の対応つけ

このステップでは、クラス図と実装グループ関係図を重みつき有向グラフに変換して、サブグラフ同型判定アルゴリズムに基づいて、対応をつける。

5.4.1 クラス図を重みつき有向グラフに変換

クラス図を重みつき有向グラフに変更する方法は次の通りである。
・グラフのノードはクラス図の各要素である。
・グラフのリンクはクラス図にある参照を表す。
・リンクの方向は参照の方法を表す。
・リンクの値は参照の一対一と一対多を表す。（1 が一対一で 2 が一対多)。

5.4.2 実装グループ関係図を重みつき有向グラフに変換

実装グループ関係図を重みつき有向グラフに変更する方法は次の通りである。
・グラフのノードは各実装グループを表す。
・グラフのリンクは実装グループ間の関係を表す。
・リンクの方向は実装グループの参照の方向を表す。
・リンクの値は一対一関係と一対多関係を表す（1 が一対一で 2 が一対多)。
変換後の重みつき有向グラフの形は図 5.17 のようである
開発者はコードを作成する際に設計と全く同じように作成することが珍しい。設計より、新しい参照を追加したり便利のために中間クラスを追加することがある。そのため、できた実装グループ関係図のグラフはクラス図のグラフと違う。一つの実装グループは実際にどのクラス図の要素を対応するかが分かるためにサブグラフ同型判定アルゴリズムを適用して対応をつける。

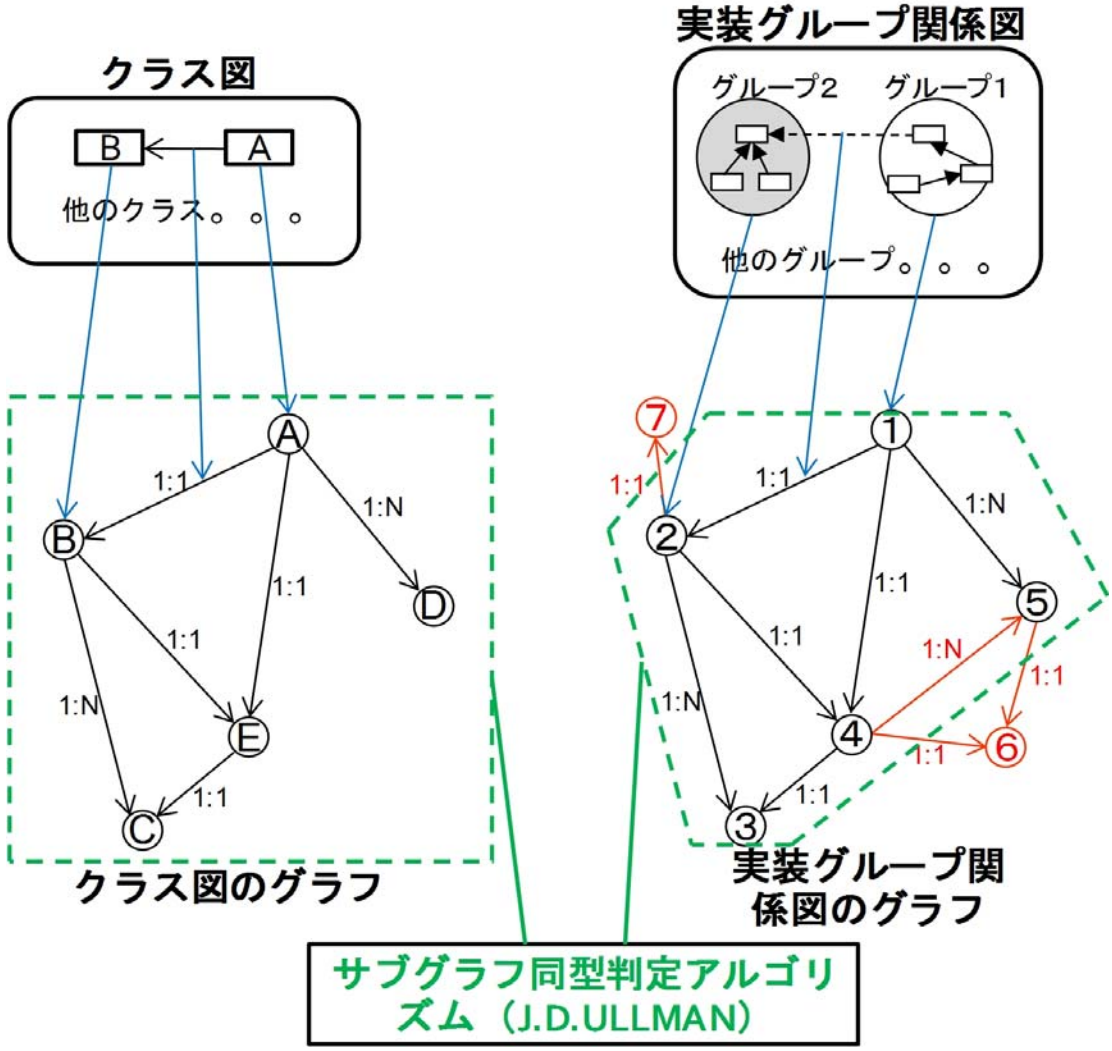

図 5.17: クラス図の要素と実装グループの対応つけ

5.4.3　サブグラフ同型判定アルゴリズムを適用することでの対応付け

ここでは、実装クラス関係図のグラフの中にクラス図のグラフと同型する部分を探索して、クラス図の要素と実装グループを対応付ける。対応付ける意味は対応付けられた実装グループに対応付けられた要素が実装されるという意味である。

1976 年に Jeffery David Ullman はサブグラフ同型判定アルゴリズムを作成した。このアルゴリズムは無向グラフに適用された。本問題に適用するために、我々はこのアルゴリズムの Mo メトリクスの作成条件とアルゴリズムの最初の状態を変更して適用した。変更したのは次のようである。

最初 Mo[n, m] メトリクスを次のように作成する。

次の各条件を検討する。

・条件１：グラフ H の j 番目ノードの値が１であるノードに来るリンク数がグラフ G の i 番目ノードの値が１であるノードに来るリンク数より小さくない。

・条件２：グラフＨのｊ番目ノードの値が２であるノードに来るリンク数がグラフＧのｉ番目ノードの値が２であるノードに来るリンク数より小さくない。

・条件３：グラフＨのｊ番目ノードの値が１であるノードから出るリンク数がグラフＧのｉ番目ノードの値が１であるノードから出るリンク数より小さくない。

・条件４：グラフＨのｊ番目ノードの値が２であるノードから出るリンク数がグラフＧのｉ番目ノードの値が２であるノードから出るリンク数より小さくない。

上の四つの条件を全て満たすと Mo[i, j] = 1。逆に、Mo[i, j] = 0。

更に、グラフＨのｊ番目ノードがグラフＧのｉ番目ノードを対応できないと明らかに分かれば、Mo[i, j] = 0 と設定する。

実際には、クラス図の要素の名前をそのまま使ってソースコードのクラスの名前として実装する場合が多い。そのために、実装グループ（グラフＨのｊ番目ノード）にあるクラスの名前がクラス図の要素の名前（グラフＧのｉ番目ノード）と同じであれば、そのクラス図の要素と実装グループを対応つけられる。その通りに Mo[i, j] = 1 と設定して、Mo[i ’, j] = 0 と Mo[i, j ’] = 0 と設定する (√ｉ ’≠ｉと√ｊ ’≠ｊ)。

このように変更して本問題に適用するとアルゴリズムのスピードが非常に改善された。

アルゴリズムを適用して、出る結果は複数のサブグラフを含むリストである。リストにある各サブグラフに対して、クラス図のグラフとの適合率・再現率のＦ値を計算して、Ｆ値が一番高いサブグラフを対応付けの結果として取り出す。

5.5　依存関係の追跡に基づく、クラス図の要素を実現するクラス群の発現

対応を付けてから、デザインパターンを使わないで実装する場合が残るため、二つの実装グループは一緒にクラス図の一つの要素を実装する可能性がある。このプロセスはその場合を依存関係の追跡に基づいてチェックして、実装グループを結合する。

デザインパターンを使わないで実装する際に、クラス図の一つの要素を実装するクラス群で一つの実装グループは中心として（対応を付けた実装グループ）、他の実装グループは中心に機能を支える役割を持つ（対応を付けられない実装グループ）。機能を支える実装グループは中心実装グループだけと依存関係があり、他の対応を付けた実装グループと依存関係がないという特徴がある。この特徴を利用して追跡方法を次のように開発した。

実装グループグラフで対応がないノードＡは対応があるノードＢに結合できるかどうかの検討方法：

ステップ１：ＡとＢの関係がないと追加できない、終了する。関係がある場合はステップ２に移動する。

ステップ２：グラフからＢを外して、リンクの方法を考えずにＡから行けるノードを全部検索する。行けるノード群に対応があるノードが存在すると追加できない。逆に存在しないと追加できる。終了する。

上の検討方法を利用して、対応がないノードと対応があるノードは各ペアとして検討する。結合できると判断するＡＢペアがあるとグループＡをグループＢに結合して、グ

ラフを変更する。結合できるペアを見つけられなくなるまで繰り返して行う。

例えば実装グループのグラフ図では、ノード７はノード２に結合できると判断できて、クラス図のＢ要素が実装グループ２と実装グループ７で実装されたという結果が得られた。

ここまで、出来上がった結果はクラス図の要素はソースコード上のどのクラス群を使って実装したかがわかる。ユースケースのクラス図の要素に代わって対応する実装グループで表現するとユースケースの協調クラス群の抽出ができる。

第6章　実験

本章では、デザインパターンに対応可能を検討し、開発した抽出アルゴリズムを利用してエレベータ制御システムと ATM システムで実験した結果について述べる。後で、抽出アルゴリズムが失敗した場合について説明する。抽出アルゴリズムの効果を関連知識章で説明した適合率・再現率を利用して評価する。

6.1　デザインパターンに対応する可能の実験

ここでは、メタパターンと構造と振る舞いの特徴に基づいてデザインパターンを含むクラス群を抽出するアルゴリズムの抽出可能の検討について述べる。

現在、GoF の２３種のデザインパターンを含むソースコードで抽出アルゴリズムを検討しただけではなく、他のデザインパターンを含むソースコードでも抽出アルゴリズムを検討した。具体的には、Pattern in Java[5] という本に載っている各デザインパターンを含むソースコードを使って抽出アルゴリズムの性能を検討した。結果は次のようである。

分類	デザインパターン	抽出方法	結果
Fundamental Design Patterns	Delegation	構造の特徴	抽出可能
	Interface	１：１結合メタパターン	抽出可能
	Immutable	振る舞いの特徴	未対応
	Marked Interface	振る舞いの特徴	抽出可能
	Proxy	１：N 結合メタパターン	抽出可能
Creational Patterns	Factory Method	統合メタパターン	抽出可能
	Abstract Factory	１：N 結合メタパターン	抽出可能
	Builder	１：１結合メタパターン	抽出可能
	Prototype	１：N 結合メタパターン	抽出可能
	Singleton	振る舞いの特徴	抽出可能
	Object Pool	振る舞いの特徴	抽出可能
Partitioning Patterns	Layered Initialization	１：１結合メタパターン	抽出可能
	Filter	１：１結合メタパターン	抽出可能
	Composite	１：N 再帰的結合メタパターン	抽出可能

表 6.1: Patterns in Java に載っているデザインパターンの抽出結果 1

分類	デザインパターン	抽出方法	結果
Structural Patterns	Adaptor	１：N 結合メタパターン	抽出可能
	Iterrator	１：N 結合メタパターン	抽出可能
	Bridge	１：１結合メタパターン	抽出可能
	Facade	構造の特徴	抽出可能
	Flyweight	１：N 結合メタパターン	抽出可能
	Dynamic Linkage	１：１結合メタパターン	抽出可能
	Virtual Proxy	１：N 結合メタパターン	抽出可能
	Decorator	１：１再帰的結合メタパターン	抽出可能
	Cache Management	構造の特徴	抽出可能
Behavioral Patterns	Chain of Responsibility	１：１再帰的結合メタパターン	抽出可能
	Command	１：N 結合メタパターン	抽出可能
	Little Language	１：N 結合メタパターン	抽出可能
	Mediator	１：１結合メタパターン	抽出可能
	Snapshot	構造と振る舞いの特徴	未対応
	Observer	１：N 結合メタパターン	抽出可能
	State	１：１結合メタパターン	抽出可能
	Null Object	１：１結合メタパターン	抽出可能
	Strategy	１：１結合メタパターン	抽出可能
	Template Method	統合メタパターン	抽出可能
	Visitor	１：N 結合メタパターン	抽出可能
Concurrency Patterns	Single Threaded Execution	構造の特徴	抽出可能
	Guarded Suspension	振る舞いの特徴	抽出可能
	Balking	振る舞いの特徴	抽出可能
	Scheduler	振る舞いの特徴	抽出可能
	Read/Write Lock	構造の特徴	抽出可能
	Producer-Consumer	構造の特徴	抽出可能
	Two-Phase Termination	振る舞いの特徴	抽出可能

表 6.2: Patterns in Java に載っているデザインパターンの抽出結果 2

ソースコードは本に付けた CD からとったソースコードである。上表の結果はそのソースコードで検討した結果である。未対応のは二つがある。Immutable と Snapshot デザインパターンである。Immutable デザインパターンを含むクラス群を振る舞いの特徴により、抽出可能があるが、動的にソースコードを解析しないと振る舞いの特徴をまとめられないことと分かった。

こういう結果ができて、我々のアプローチは GoF の２３種のデザインパターンを利用するソースコードに限るわけではなく、他のデザインパターンを利用するソースコード

にも適用できる。上表を見ると 41 デザインパターンの中に 26 デザインパターンがメタパターンで説明でき、メタパターンに基づいてそのデザインパターンらを利用するソースコードを対応する可能がある。その他、構造の特徴に基づいて 6 デザインパターンを利用したソースコードを対応する可能があり、振る舞いの特徴に基づいて 7 デザインパターンを利用するソースコードを対応する可能がある。41 デザインパターンの内に 2 デザインパターンしかを対応する可能がない。

次は開発したシステムのソースコードを使って協調クラス群を抽出する精度について述べていく。

6.2 エレベータ制御システムで実験した結果

エレベータ制御システムは北陸先端科学技術大学院大学・情報科学研究科の講義で用いられたシステムである。このシステムのソースコードは Java 言語で実装され、51 クラスを含む。このシステムでは、主なユースケースは４つがあり、Stop Elevator at Floor ユースケースと Dispatch Elevator ユースケースと Select Destination ユースケースと Request Elevator ユースケースである。実験で受けた結果は下表のようである。

ユースケース	適合率	再現率	調和平均
Stop Elevator at Floor	100%	91.7%	95.7%
Dispatch Elevator	100%	90%	94.7%
Select Destination	91.7%	91.7%	91.7%
Request Elevator	91.7%	84.6%	88%

表 6.3: エレベータ制御システムで実験した結果表

適合率が 95.9% で再現率が 89.5% で調和平均が 92.5% である。

6.3 ATM システムで実験した結果

ATM システムは文献 [8] にあるケーススタディの一つである。このシステムを実装したソースコードは 79 クラスを含むシステムである。主なユースケースは Validate PIN for Client ユースケースと Withdraw Funds for Client ユースケースと Query Account for Client ユースケースと Transfer Funds for Client ユースケースと Validate PIN for Server ユースケースと Withdraw Funds for Server ユースケースがある。実験で受けた結果は表 6.4 のようである。

適合率が 98.1% で再現率が 91.3% で調和平均が 94.5% である。

6.4 抽出アルゴリズムが失敗する場合

抽出できない場合を調査して、ほとんどの場合は次のようである。

ユースケース	適合率	再現率	調和平均
Validate PIN for Client	96%	82.8%	88.9%
Withdraw Funds for Client	96.4%	90%	93.1%
Query Account for Client	100%	91.7%	95.7%
Transfer Funds for Client	96.2%	83.3%	89.3%
Validate PIN for Server	100%	100%	100%
ithdraw Funds for Server	100%	100%	100%

表 6.4: ATM システムで実験した結果表

ユースケースに対する機能を実装する時、開発者は中心クラスを継承して別のクラスを作成するが抽出アルゴリズムはその全てのサブクラスは一つのユースケースの機能を実装すると間違ってわかる。例えば、ATM システムでは、Withdraw Funds for Client ユースケースの Transaction 機能を実装するために ATM Transaction クラスの子として Withdrawal Transaction クラスを作成し、Query Account for Client ユースケースの Transaction 機能を実装するために ATM Transaction クラスの子として Query Transaction クラスを作成した。ソースコードを解析した時に抽出アルゴリズムはそのすべてのクラスが ATM Transaction 要素を実装するクラス群と分かるが、ユースケースに対してそれが間違いである。

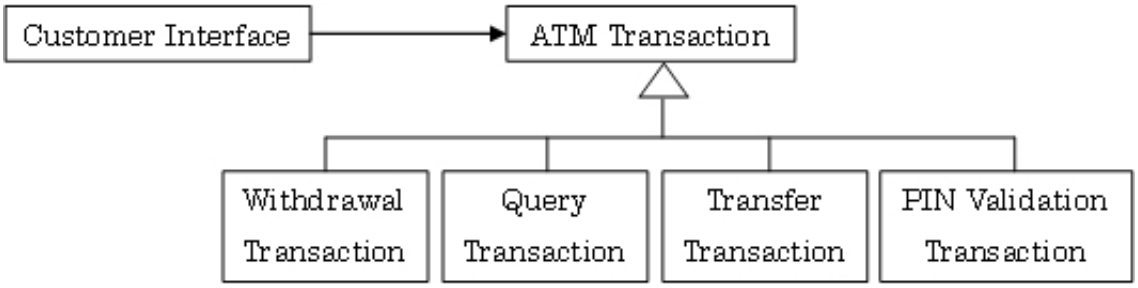

図 6.1: 失敗する場合の例

本論文でこの問題がまだ解決できない。今後の課題である。

6.5 実行時間の評価

実行時間を評価するために、下記のシステムで検討した。結果は次のようである。
環境：
.OS：Windows Vista
.CPU:Intel Core 2CPU @2.00Ghz
.Memory:2GB of RAM
抽出アルゴリズムの正しさを評価する際に、大規模なシステムで検討できなかった理由はシステムの分析と設計図がないためである。分析と設計図が無いと手動で検討する

ソースコード	クラス数	実行時間 (s)
レベータ制御システム	51	0.544
ATM システム	80	0.178
JavaAWT	377	12.13
JHotDraw 7.3.1	585	13.24

表 6.5: 実行時間

ことになり、時間が非常にかかることだけではなく間違いも多く含まれてしまう恐れがある。

第7章　まとめと今後の課題

7.1　まとめ

本研究は、金旭東の提案したアルゴリズムを改善して、メタパターンと構造と振る舞いの特徴により Java ソースコードにあるデザインパターンを利用したクラス群を抽出する方法を提案した。抽出した結果を利用してグラフ化方法とサブグラフ同型判定方法により、クラス図の要素を対応する Java ソースコードにあるクラス群を発見する方法を提案した。ユースケースのクラス図の要素に代わって対応するクラス群を入れることにより、ユースケースに含まれるクラス群を抽出した。

現在、中規模なプログラムを利用して、評価実験を行い、92.5% 以上の精度で抽出できること確認した。先行研究の結果と比べると、より良い結果ができた。

7.2　今後の課題

今後の課題としては、実験で失敗した事例への対応、より多いデザインパターンを利用したソースコードへの対応と Java 以外のオブジェクト指向プログラミング言語への対応が挙げられる。

実験で失敗した事例への対応については、ソースコードを詳しく解析しなければならない、静的解析と動的解析を結合して、研究を進めることが一つの方法と考えている。

より多いデザインパターンを利用したソースコードへの対応については、構造と振る舞いの特徴をより広め、構造と振る舞いの特徴に基づいて抽出できない場合も考えて、研究を発展することが今後の一つの課題である。

Java 以外のオブジェクト指向プログラミング言語への対応については、現在すべてのプロジェクトは Java 言語で実装するわけではなく、他のプログラミング言語で実装する場合が多い。本研究を他の言語に適用できるように広めることが今後の一つの課題である。

謝辞

本研究に関して、親切なご指導をいただいた北陸先端科学技術大学院大学、情報科学研究科、落水浩一郎教授に深く感謝の意を表します。また、北陸先端科学技術大学院大学、情報科学研究科、鈴木正人准教授、青木利晃准教授には、審査の場で助言いただきまして、この場を借りて厚くお礼申しあげます。

参考文献

[1] 金旭東, 早坂良, 小谷正行, 落水浩一郎, メタパターンを用いた Java ソースコードおける協調クラス群の抽出 情報処理学会研究報告, 2005-SE-150, pp101-108, 2006

[2] 菅井拓海, 小谷正行, 落水浩一郎, UML 図面要素に対応する Java 協調クラス群の抽出ツール情報処理学会, 第 155 回ソフトウェア工学研究発表会, 2007.3

[3] Wolfgang Pree, Design Patterns for Object-Oriented Software Developtment A Division of Association for Computing Machinery, Inc.(ACM). 1995

[4] Erich Gramma, Richard Helm, Ralph johnson, John Vlissides, Design Patterns - Elements of Reusable Object - Oriented Software (1997 Addison - Wesley)

[5] Mark Grand, Patterns in Java Volume 1 - A Catalog of Reusable Design Patterns Illustrated with UML (1998 Wiley Computer Publishing)

[6] J.R. Ullmann, An Algorithm for Subgraph Isomorphithms Journal of Association for Computing Machinery, vol.23, pp.31-42, 1976

[7] 藤井拓, 羽生田栄, オブジłクト指向開発自由自在: Rose で実践する OMT 法と Booch 法Lockheed Martin Advanced Concepts Center, Rational Software Corporation 著

[8] Hassan Gomaa, Designing Concurrent, Distributed, and Real-time Application with UML Addison-Wesley, 2000.

付録A

この付録では、Patterns in Java Volume 1[5] と言う本に載っているデザインパターンを利用したクラス群を抽出する方法について述べる。

この本に載っているのは４１種のデザインパターンである。その中に２６種のデザインパターンを利用したクラス群をメタパターンに基づいて抽出できる。そのために、他のデザインパターンを利用するクラス群を構造と振る舞いの特徴に基づいて抽出する方法について説明する。この４１種のデザインパターンは次の表のように分類した。

ソースコード	クラス数
メタパターンで説明できる	Interface、Proxy、Factory Method、Abstract Factory、Builder、Prototype、Layered Initialization、Filter、Composite、Adaptor、Iterrator、Bridge、Flyweight、Dynamic Linkage、Virtual Proxy、Decorator、Chain of Responsibility、Command、Little Language、Mediator、Observer、State、Null Object、Strategy、Template Method、Visitor
構造で説明できる	Delegation、Facade、Cache Management、Single Threaded Execution、Read/Write Lock、Producer-Consumer
振る舞いで説明できる	Immutable、Marked Interface、Singleton、Object Pool、Guarded Suspension、Balking、Scheduler、Two-Phase Termination
その他	Snapshot

表 7.1: Patterns in Java に載っているデザインパターンの特徴による分類

まず、本論文で提案したメタパターンに基づくデザインパターンを利用したクラス群を抽出するアルゴリズムはメタパターンで説明できる２６種のデザインパターンを対応できる。そのため、次は残りの１５種のデザインパターンを利用するクラス群を抽出する方法について説明していく。

構造の特徴に基づくデザインパターンを利用するクラス群の抽出方法

Delegation デザインパターンの場合：一つのクラス A のメソッド Am に対して、他のクラス C のインスタンスを呼び出すことを探索して、あると、クラス C のインスタンスを

通じてクラスCのメソッドCmを呼び出すことがあるかどうかをチェックする。最後にメソッドCmとメソッドAmの定義が全く同じかをチェックする。全ての条件を満たすとクラスAとクラスCはDelegationデザインパターンを利用したと判断できる。この場合では、クラスAとクラスCは実装関係があると判断できる。

Facadeデザインパターンの場合：一つのクラスAに対して、クラスAは参照するクラスらをグループAにまとめて、クラスAを参照するクラスらをグループBにまとめる。グループBとグループAの依存関係がないとクラスAはFacadeデザインパターンを利用したと判断できる。この場合では、クラスAはクラス図の一つの要素を実装すると判断できる。更に、グループBにあるクラスらとクラスA又はグループAにあるクラスらの実装関係がないと判断できる。

Cache Managementデザインパターンの場合：一つのクラスAのメソッドAmに対して、メソッドAmの呼び出すクラスCのメソッドCmとメソッドAmの呼び出すクラスFのメソッドFmを各メソッドの定義が全く同じであると言う条件で検索する。クラスCでは、あるオブジェクトを保存するためのメソッドCm1を検索して、メソッドAmはメソッドCm1を呼び出すとクラスC、クラスA及びクラスFはCache Managementデザインパターンを利用したと判断できる。この場合では、三つのクラスは実装関係があると判断できる。

Single Threaded Executionデザインパターンの場合：このデザインパターンは同時に一つのデータが複数のインスタンスに使われることを避けるために、一つのクラスAのメソッドAmに対して、public synchrinizedで定義されるかをチェックする。次に、複数のインスタンスがAmを呼び出すかをチェックする。この二つの条件を満たすとクラスAがSingle Threaded Executionデザインパターンを利用したと判断できる。この場合では、クラスAはクラス図の一つの要素を実装したと判断できる。

Read/Write Lockデザインパターンの場合：このデザインパターンは同時に複数のインスタンスが一つのデータを変更することを避けるためのデザインパターンである。Single Threaded Executionデザインパターンの一つの場合である。そのため、Single Threaded Executionデザインパターンの特徴でこのデザインパターンを利用したクラスを抽出できる。

Producer-Consumerデザインパターンの場合：一つのクラスAに対して、リストを使うかを確認して、使うと確認できると、リストに値を追加するクラスAのメソッドAaddとリストから値を引き出すクラスAのメソッドApopを検索する。メソッドAaddのみを呼び出すクラスPとメソッドApopのみを呼び出すクラスCを見るけると、クラスA、クラスPとクラスCはProducer-Consumerデザインパターンを利用したと判断できる。クラスPとクラスCとクラスAは実装関係がないと判断できる。

振る舞いの特徴に基づくデザインパターンを利用するクラス群の抽出方法

Immutableデザインパターンの場合：一つのクラスAに対して、クラスの変数を外から変更できないように設定さらた。クラス内で、Constructorのみ変数を変更でき、他のメソッドは変数を変更できないとチェックして、全ての条件を満たすとクラスAはImmutable

デザインパターンを利用したと判断できる。この場合は、クラス A がクラス図の一つの要素を実装していると判断できる。

Marked Interface デザインパターンの場合：一つのクラス A に対して、クラス A は何もせずに {} のみを含むと Marked Interface デザインパターンと判断できる。

Singleton デザインパターンと Object Pool デザインパターンの場合：Object Pool デザインパターンは Singleton デザインパターンを利用するために、Singleton デザインパターンの振る舞いの特徴を使ってこれらのデザインパターンを利用したクラスを抽出できる。Singleton デザインパターンを判断する方法は本研究で振る舞いの特徴に関する章で説明した。特徴は次のようである。一つのクラス A に対して、コードの中で一つの変数のタイプはそのクラスで static 変数である。その変数は V 変数とする。V 変数は一回だけクラスのインスタンスを設定する。更に、メソッドリストに V 変数を戻すメソッドがある。Constructor は private とクラス名で定義される。この場合は、クラス A がクラス図の一つの要素を実装していると判断できる。

Guarded Suspension デザインパターンの場合：一つのクラス A に対して、メソッド Am は synchronized で定義される。メソッド Am の最初では、ある条件として wait(); のみを繰り返して行う。この二つの条件を満たすとクラス A は Guarded Suspension デザインパターンを利用したと判断できる。この場合は、クラス A がクラス図の一つの要素を実装していると判断できる。

Balking デザインパターンの場合：一つのクラス A に対して、メソッド Am は public で定義される。メソッド Am で、synchronized ブロック Amb がある。ブロック Amb はある条件を満たすと return のみを行う。これらの条件を全て満たすとクラス A は Balking デザインパターンを利用したと判断できる。この場合は、クラス A がクラス図の一つの要素を実装していると判断できる。

Scheduler デザインパターンの場合：一つのクラス A に対して、リストを利用するメソッド Am がある。メソッド Am の内で、各 synchronized ブロックがある。このブロックらの内で return 命令と wait 命令を利用する。メソッド Am を呼び出すメソッド B がある。これらの条件を全て満たすとクラス A は Scheduler デザインパターンを利用したと判断できる。この場合は、クラス A がクラス図の一つの要素を実装していると判断できる。

Two-Phase Termination デザインパターンの場合：一つのクラス A に対して、クラス A では thread を使う。クラス A で、ある命令を行う前に thread が interrupt されたかどうかをチェックする命令がある。これらの条件を全て満たすとクラス A は Two-Phase Termination デザインパターンを利用したと判断できる。この場合は、クラス A がクラス図の一つの要素を実装していると判断できる。