特別研究報告

題目

コードクローン解析に基づくデザインパターン適用候補の検出手法

指導教員
井上 克郎 教授

報告者
吉田昌友

平成 20 年 2 月 19 日

大阪大学 基礎工学部 情報科学科

平成 19 年度 特別研究報告

コードクローン解析に基づくデザインパターン適用候補の検出手法

吉田昌友

内容梗概

デザインパターンとは，オブジェクト指向プログラムの設計において発生する特定の問題に対して，熟練したプログラマが用いてきた典型的な解決策をいう．だが，実際のソフトウェア開発では，デザインパターンを利用すべき場合でも，利用していないことがある．そのため，デザインパターンを適用することで既存のソフトウェアの設計品質を高める方法が提案されている．しかし，大規模ソフトウェアのソースコードからデザインパターンが適用可能な部分を手作業で発見するには，大きなコストが必要となる．

本研究では，ソースコードからデザインパターンが適用可能な部分を自動的に検出することで，既存のソフトウェアに対するデザインパターンの適用を支援する手法を提案する．具体的には，類似コードに対してデザインパターンの 1 つである Factory Method パターンの適用を支援した．Factory Method パターンは，オブジェクトを生成する構文の代わりに，オブジェクトを生成して返り値とするメソッドを定義することで，拡張性の高い設計を実現できることが知られている．類似コードに対して Factory Method パターンを適用することで，類似コードを除去し，かつ拡張性を向上させることができる．提案手法は，コードクローン検出ツール (類似コードを検出するツール) が出力した類似コードの位置情報と，構文解析を利用することで得られたクラス継承関係から，Factory Method パターンが適用可能な部分を検出する．

提案手法の有効性を確認するために，オープンソースソフトウェアを対象として Factory Method パターンが適用可能な部分を検出した．実際に，検出された部分に対して Factory Method パターンを適用し，適用の前後でソフトウェアの外部的動作が変化していないことをテストによって確認した．

主な用語

デザインパターン
リファクタリング
コードクローン

目 次

1　まえがき

近年，ソフトウェアの大規模化・複雑化に伴い，ソフトウェアの保守作業がますます困難になってきている．保守作業の例として，ソフトウェアの利用開始後に明らかになったバグの修正，将来バグを引き起こす原因になり得る問題への対応，環境変化に伴う機能の追加・変更などの作業が挙げられる．保守対象のソフトウェアが大規模の場合，ソフトウェア全体を理解することが困難であるため，ソフトウェアの保守作業にかかるコストが増大する．

ソフトウェアの保守性を高める手段としてリファクタリング [29] が挙げられる．リファクタリングとは，“ 外部的振る舞いを保ちつつ，理解や修正が容易になるように，ソフトウェアの内部構造を変化させること ”[15] である．例えば，プログラムの処理の中で類似コード [15] であるコード片をメソッドとして抽出し，類似コードを抽出したメソッドの呼び出し文に置換するなどである．

Kerievsky は，デザインパターンを利用するリファクタリングを提案している [25]．デザインパターンとは，オブジェクト指向プログラムの設計において発生する特定の問題に対して，熟練したプログラマが用いてきた典型的な解決策をいい，Gamma らが定義している 23 種類のデザインパターンが有名である [16]．Kerievsky は文献 [25] の中で，27 種類のデザインパターンを利用するリファクタリングを提案している．例として，Factory Method パターン [16] を適用する “ Factory Method によるポリモーフィックな生成の導入 ”[25] を挙げる．デザインパターン適用前は，兄弟クラス (共通の親クラスを持つクラス対) に，オブジェクトを生成する構文を除いて内容が等しいメソッドが存在し，それらのメソッドが類似コードになっているという問題がある．その類似コードを除去するため，オブジェクトの生成を行うメソッド (Factory Method[16]) が共通の親クラスに定義してあり，Factory Method を子クラスごとにオーバーライドして生成するオブジェクトの型を変える構造にする．その結果として内容が揃った重複するメソッドを，親クラスに引き上げる．以上のようにソースコードを変更することで，兄弟クラスで重複するメソッドを共通の親クラスにまとめることができ，類似コードを除去することができる．

特定の問題に対しては，デザインパターンを適用することでソフトウェアの設計を改善できるのだが，実際のソフトウェア開発では，デザインパターンを適用すべき場合でも，適用していないことがある．そのため，既存のソフトウェアに対してデザインパターンを適用するリファクタリングを行えば，その設計品質を高めることができる．しかし，大規模ソフトウェアのソースコードをプログラマが調査し，デザインパターンの適用が可能かどうかを判断して，デザインパターンを適用するリファクタリングを行うには，大きなコストが必要である．

そこで，ソースコードからデザインパターンが適用可能な部分を自動的に検出することで，

既存のソフトウェアに対するデザインパターンの適用を支援する手法を提案する．本研究ではデザインパターンの適用を検討する部分として，ソフトウェアの保守作業を困難にする類似コードに着目した．類似コードの存在が保守作業を困難にする理由は，例えば，あるコード片にバグが存在した場合，そのコード片の類似コード全てに対して修正の是非を検討する必要が生じるからである．類似コードを除去するデザインパターンの適用を支援することを考え，具体的には，類似コードに対してデザインパターンの 1 つである Factory Method パターンを適用する“ Factory Method によるポリモーフィックな生成の導入 ”[25] を支援した．提案手法は，コードクローン検出ツール (類似コードを検出するツール) が出力した類似コードの位置情報と，構文解析を利用することで得られたクラス継承関係から，Factory Method パターンが適用可能な部分を検出する．提案手法の有効性を確認するために，オープンソースソフトウェアを対象として Factory Method パターンが適用可能な部分を検出した．実際に，検出された部分に対してリファクタリングを試みたところ，リファクタリングを行うことができた．

以降，2 章では，デザインパターン，類似コードを除去するリファクタリング，デザインパターンを利用するリファクタリング，コードクローン (類似コード) 及びコードクローン検出ツール CCFinder[24] について説明する．3 章では，コードクローン検出法を利用してデザインパターンの適用を支援する手法を提案し，その手法の具体的な実装方法について述べる．4 章では，オープンソースソフトウェアを対象に適用実験を行った結果について説明し，5 章では，本研究についての考察を述べる．6 章では，デザインパターンを既存のソースコードに適用する関連研究について述べ，最後に 7 章では，本研究のまとめと今後の課題について述べる．

2　準備

2.1　デザインパターン

デザインパターンとは，“種々の状況における設計上の一般的な問題の解決に適用できるよう，オブジェクトやクラス間の関係を記述したもの”であると定義されている [16]．一般的に，デザインパターンの記述には次に示す 4 つの要素がある [16].

パターン名 デザインパターンを習得している技術者同士であれば，設計構造の詳細を説明する代わりにパターン名を使えばよいので，技術者同士の意思疎通が容易になる．

問題 どのような場合にデザインパターンを適用すればよいかを示す.

解法 設計上の問題を抽象的に記述し，クラスやオブジェクトをどのように組み合わせればよいかを示す.

結果 デザインパターンを適用した場合の結果やトレードオフを示す.

デザインパターンを学ぶことで，過去に対処された設計上の問題と同じような問題に直面した時に過去に用いた解法を再利用することができる．

以下に，デザインパターンの例をいくつか示す (オブジェクト指向プログラミング言語，特に Java を例にとって述べる).

Template Method パターン

処理の流れは共通しているが処理の一部が異なるメソッドが，複数存在する場合に適用できる．まず，親クラスで，共通した処理の流れをメソッドとして定義し (定義されたメソッドを Template Method という)，さらに，異なる処理を (抽象) メソッドとして定義して，Template Method(図 1 の sharedMethod) が異なる処理を行うメソッド (図 1 の method1, method2) を呼び出す構造にする．次に，子クラスごとに継承した異なる処理を行うメソッドをオーバーライドすれば，目的の処理を行うことができる．結果として，共通した処理を親クラスにまとめることで，類似コードを減らすことができる．ただし，子クラスでオーバーライドしなければならないメソッドが多いほど扱いにくくなるので，オーバーライドしなければならないメソッドの数は最小限にとどめる必要がある．

Factory Method パターン

生成しなければならないオブジェクトの型が何種類かあり，子クラスごとに生成するオブジェクトの型を変える場合などに適用できる．まず，親クラスで，生成するオブジェクトに

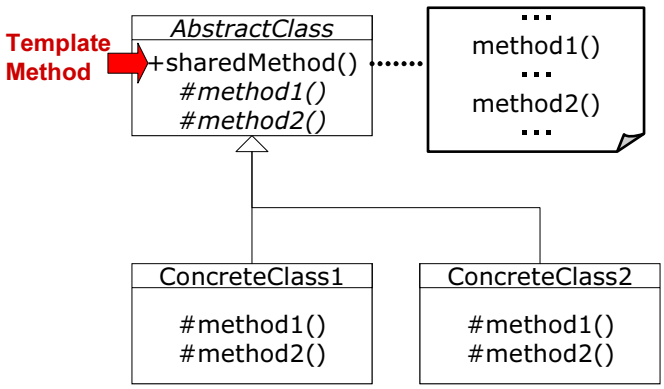


図 1: Template Method パターン

共通する型のオブジェクトを生成して返り値とするメソッド (Factory Method) を定義する．次に，子クラスで Factory Method をオーバーライドして目的のオブジェクトを生成する構造にする．オーバーライドされた Factory Method は多態性を利用して，返り値の型であるクラス (図 2 の Product) を継承したクラス (図 2 の ConcreteProduct クラス)，もしくは返り値の型であるインタフェース (図 2 の Product) を実装したクラス (図 2 の ConcreteProduct クラス) のオブジェクトも返り値とすることができる．Factory Method の返り値の型のクラスを継承 (インタフェースを実装) した新しいクラスのオブジェクトを生成するようにソフトウェアの機能を変更する場合でも，Factory Method のシグニチャは変わらないので，拡張性の高い設計になる．

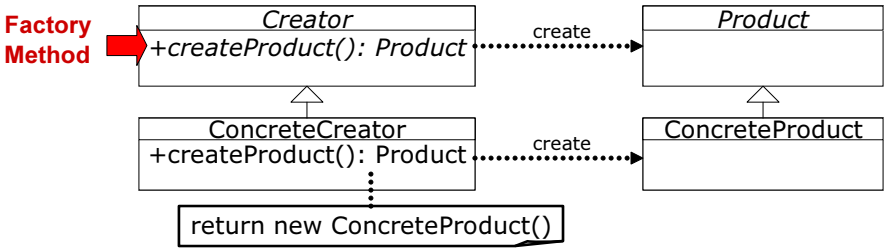


図 2: Factory Method パターン

2.2 リファクタリング

リファクタリング [29] とは，“ 外部的振る舞いを保ちつつ，理解や修正が容易になるように，ソフトウェアの内部構造を変化させること ” であると定義されている [15]．Fowler は文献 [15] の中で，リファクタリングを検討すべき部分にあらわれる特徴を Bad Smell と呼び，

その代表例として類似コード (Duplicated Code) を挙げている．類似コードを取り除く手法として，次のような対処方法がある (以下，2.1 節と同様)．

メソッドの抽出 (Extract Method)

ひとまとめにできるコード片がある場合に，そのコード片を抽出して新たなメソッドとして定義し，抽出されたコード片を抽出先のメソッドの呼び出し文に置き換える．特に類似コードに限ったリファクタリングではないが，類似コードで最も単純な例は，同じクラス内に存在する複数のメソッドに同じコード片がある場合である (図 3 参照)．

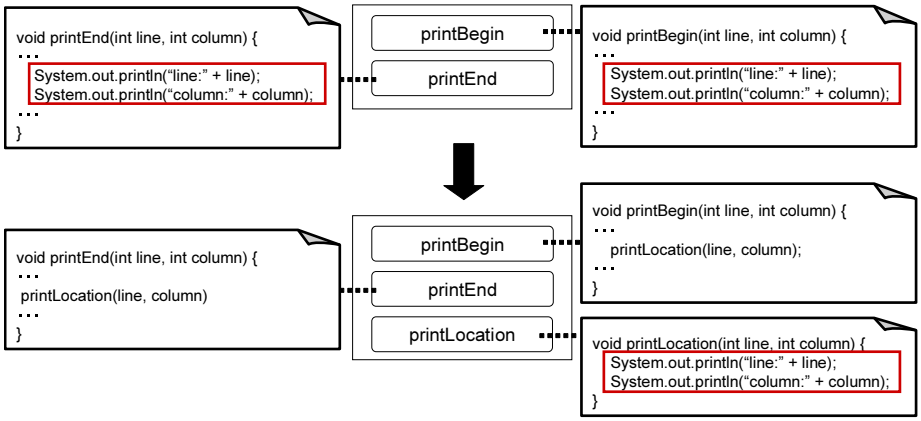


図 3: 類似コードを対象としたメソッドの抽出

メソッドの引き上げ (Pull Up Method)

同じ結果をもたらすメソッドが複数の子クラスに存在した場合，それらを親クラスに引き上げる．最も単純な例は，複数のメソッドがまったく同じ内容を持つ場合である (図 4 参照)．類似コードが兄弟クラス (共通の親クラスを持つクラス対) に存在した場合には，メソッドの抽出を行ってから，メソッドの引き上げを行えばよい．

スーパークラスの抽出 (Extract SuperClass)

似通ったフィールドやメソッドを持つ複数のクラスがある場合に，新たに親クラスを作成して，そのクラスに共通のフィールドやメソッドを移動すると，類似コードが除去される (図 5 参照)．

リファクタリングは，特に機能追加や，バグ修正，コードレビューの際に行うのがよいとされている．いずれもソースコードの理解が必要な作業であり，リファクタリングを行うこ

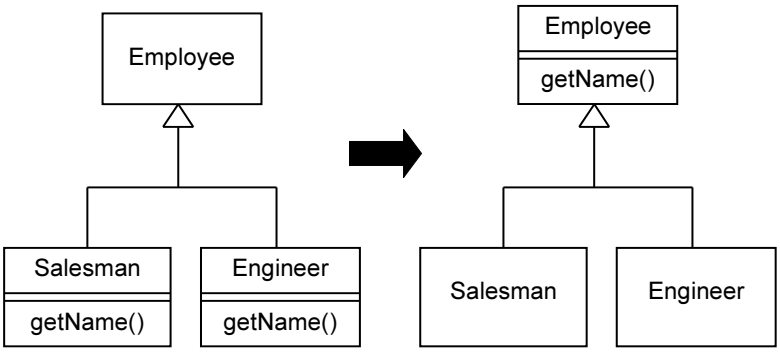


図 4: 類似コードを対象としたメソッドの引き上げ [15]

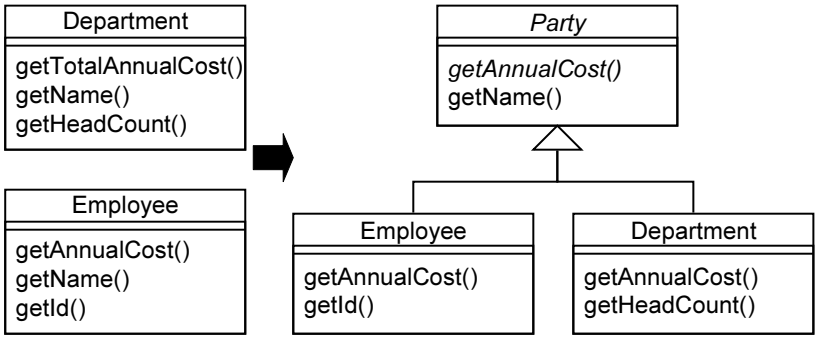


図 5: 類似コードを対象としたスーパークラスの抽出 [15]

とで，ソースコードの理解が深まり，バグが混入しにくくなり，バグを発見しやすくなる．しかし，リファクタリングとは，あくまでもソフトウェアを理解しやすく，変更を容易にするために行うことであり，機能追加とは区別されなければならない．

2.3 デザインパターンを利用したリファクタリング

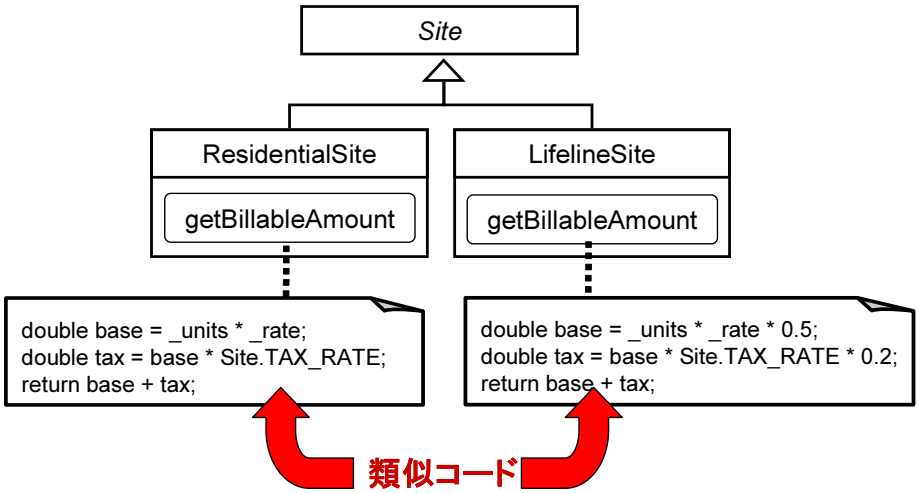
Kerievsky は文献 [25] の中で，デザインパターンは，新たにソフトウェアを設計する時の初期段階よりも，既存のソフトウェアの設計を改善するために利用する方がよいと主張している．類似コードを取り除くためのデザインパターンを適用するリファクタリングには，次のような例がある (以下，2.1 節と同様)．

Template Method の形成 (Form Template Method)

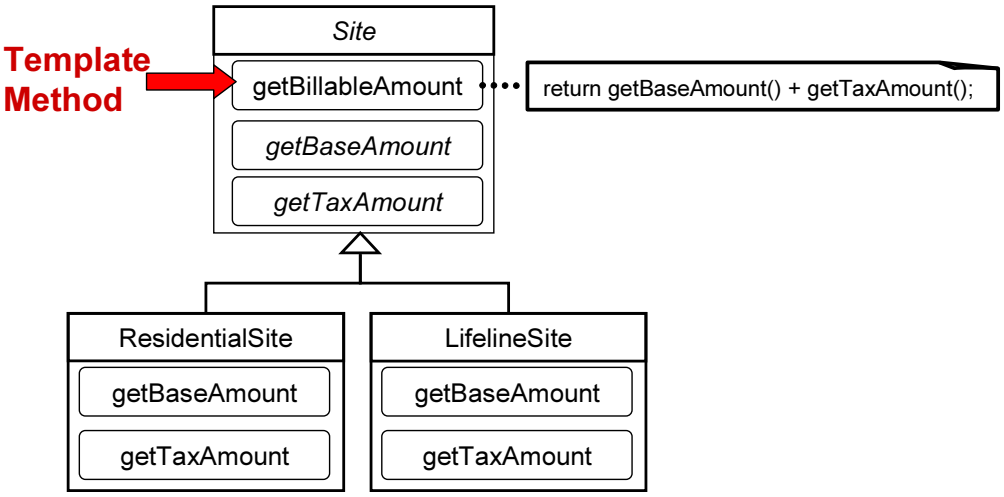
デザインパターン適用前は，兄弟クラス (共通の親クラスを持つクラス対 (図 6 の ResidentialSite クラスと LifelineSite クラス)) に，類似した内容のメソッド (図 6(a) の 2 つの getBillableAmount メソッド) が存在し，それらのメソッドが類似コードになっている．まず，それらのメソッド間で異なる処理を，共通のシグニチャを持つメソッド (図 6(b) の base の値を返す getBaseAmount メソッドと tax の値を返す getTaxAmount メソッド) として抽出することで，類似コードになっているメソッドの内容を揃える．次に，類似コードになっているメソッドを共通の親クラス (図 6 の Site クラス) に引き上げる．この引き上げたメソッドが Template Method[16] である．最後に，Template Method が呼び出すメソッド (図 6(b) の getBaseAmount メソッドと getTaxAmount メソッド) を共通の親クラスに抽象メソッドとして宣言する．共通した処理の流れが親クラスでメソッドとして定義され，処理が異なる部分は子クラスごとにメソッドをオーバーライドして変えていることになるので，Template Method パターンを適用した形になる．以上のように変更することで，兄弟クラスで重複していたメソッドを共通の親クラスにまとめることができ，類似コードを除去することができる (図 6(b))．また，Site クラスに子クラスを追加して同様の処理を行う場合，デザインパターン適用前では重複するメソッドが増えてしまうが，デザインパターン適用後は 2 つの抽象メソッドを新たに実装すればよいので，拡張性の高い設計になる．

Factory Method によるポリモーフィックな生成の導入 (Introduce Polymorphic Creation with Factory Method)

デザインパターン適用前は，兄弟クラス (図 7 の DOMBuilderTest クラスと XMLBuilderTest クラス) に，オブジェクトを生成する構文以外は内容が等しいメソッド (図 7(a) の 2 つの testAddAboveRoot メソッド) が存在し，それらのメソッドが類似コードになっている．まず，



(a) デザインパターン適用前



(b) デザインパターン適用後

図 6: Template Method の形成 [15]

それらのメソッド間で異なるオブジェクトを生成する構文を，共通のシグニチャを持ち，オブジェクトを生成して返り値とするメソッド (図 7(b) の createBuilder メソッド) に置換することで，類似コードになっているメソッドの内容を揃える．オブジェクトを生成して返り値とするこのメソッドが Factory Method[16] である．次に，類似コードになっているメソッドを共通の親クラス (図 7(b) ではスーパークラスの抽出を行い，新たに AbstractBuilderTest クラスを親クラスとして作成している) に引き上げる．最後に，Factory Method を共通の親クラスに抽象メソッドとして宣言する．Factory Method の実装を子クラスごとに変えていることになるので，Factory Method パターンを適用した形になる．以上のように変更することで，兄弟クラスで重複していたメソッドを共通の親クラスにまとめることができ，類似コードを除去することができる (図 7(b))．また，OutputBuilder クラス (DOMBuilder クラスと XMLBuilder クラスの親クラス) に子クラスを追加して同様の処理を行う場合，デザインパターン適用前では重複するメソッドが増えてしまうが，デザインパターン適用後は AbstractBuilderTest クラスに子クラスを追加して Factory Method を新たに実装すればよいので，拡張性も向上している．

2.4 コードクローン

あるトークン列中に存在する 2 つの部分トークン列 α，β が等価であるとき，α と β とは互いにクローンであるという [34]．またペア (α，β) をクローンペアと呼ぶ．α，β それぞれを真に包含する如何なるトークン列も等価でないとき，α，β を極大クローンと呼ぶ．また，クローンの同値類をクローンセットと呼ぶ (図 8 参照)．ソースコード中でのクローンを特にコードクローン (類似コード) という．

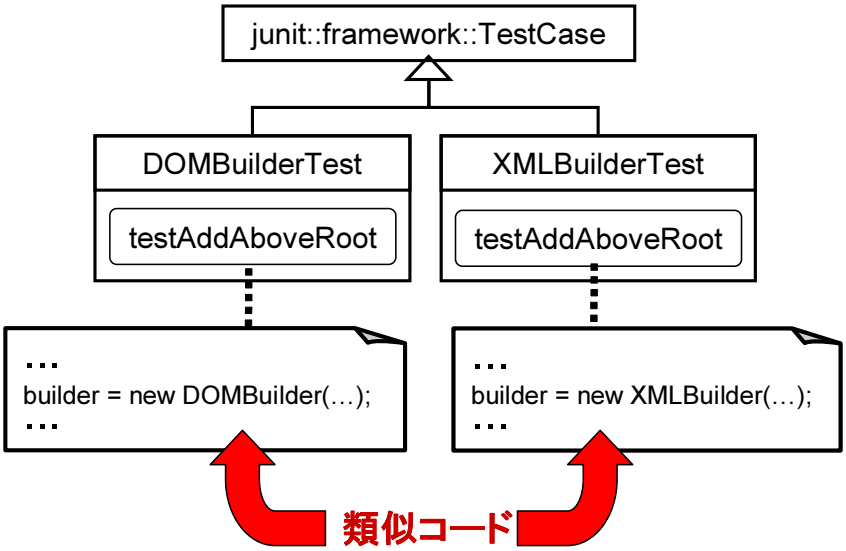
コードクローンは以下のような原因で，ソフトウェアの中に作り込まれ，発生する [9][24]．

既存コードのコピーとペーストによる再利用 近年のソフトウェア設計手法を利用すれば，構造化や再利用可能な設計が可能である．しかし，ゼロからソースコードを書くよりも既存コードをコピーして部分的な変更を加える方が信頼性が高いということもあり，実際には，コピーとペーストによる場当たり的な既存コードの再利用が多く存在する．

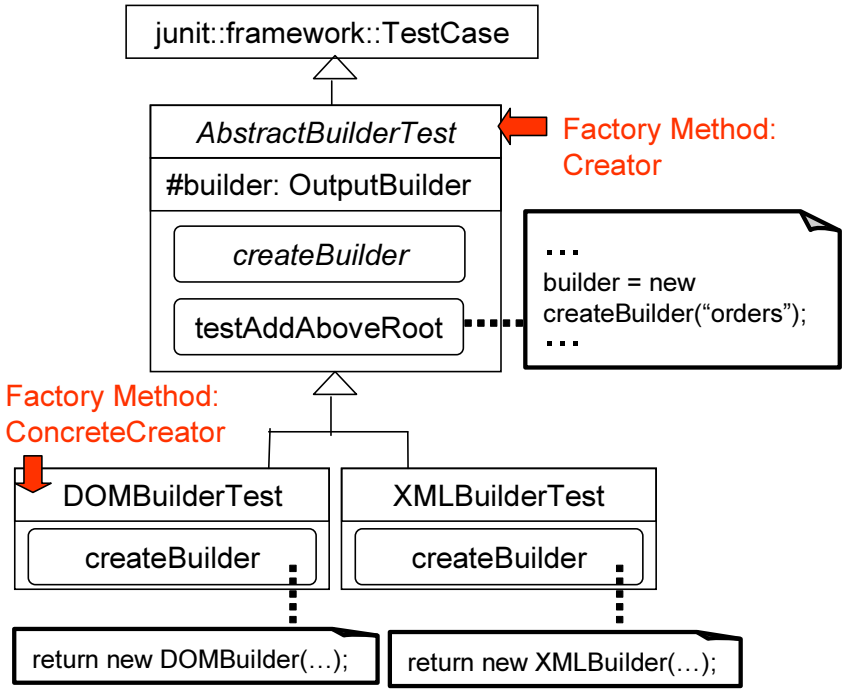
コーディングスタイル 規則的に必要なコードはスタイルとして同じように記述される場合がある．例えば，ユーザインタフェース処理を記述するコードなどである．

定型処理 定義上簡単で頻繁に用いられる処理，例えば，給与税の計算や，キューの挿入処理，データ構造アクセス処理などである．

プログラミング言語に適切な機能の欠如 抽象データ型や，ローカル変数を用いることが不可能である場合には，類似したアルゴリズムを持った処理を繰り返し書かなければな



(a) デザインパターン適用前



(b) デザインパターン適用後

図 7: Factory Method によるポリモーフィックな生成の導入 [25]

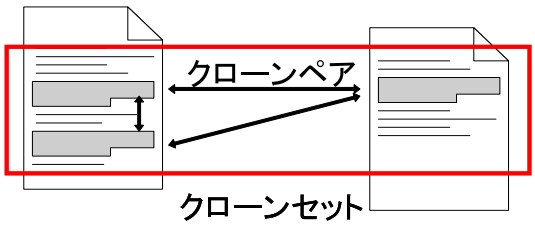


図 8: クローンペアとクローンセット

らないことがある．

パフォーマンス改善 リアルタイムシステムなど時間制約のあるシステムにおいて，インライン展開などの機能が提供されていない場合に，特定のコード片を意図的に繰り返し書くことによってパフォーマンスの改善を図ることがある．

コード生成ツールの生成コード コード生成ツールにおいて，類似した処理を目的としたコードの生成には，識別子名等の違いはあろうとも，あらかじめ決められたコードをベースにして自動的に生成されるため，類似したコードが生成される．

偶然 単純に偶然一致してしまう場合もあるが，大きなコードクローンになる可能性は低い．

プログラムにコードクローンが存在すると，一般的にそのプログラムの保守作業が困難になるといわれる．例えば，あるコード片にバグが存在した場合，そのコード片のコードクローン全てに対して修正の是非を検討する必要が生じる．特に大規模ソフトウェアを対象とした保守作業において，大量のソースコードからコードクローンを探し出し，その一つ一つに対して修正の是非を検討するには，大きなコストが必要である．

このようなコードクローンによる問題に対処する方法としては，以下の方法がある [32]．

- コードクローン情報の文書化を行うことで変更の一貫性を保つ
- コードクローンを自動的に検出する

しかし，コードクローン情報の文書化は，コードクローンに関する全ての情報を常に最新の情報に保たなければならない非常に手間がかかる作業であるため，現実的には困難である．そこで，これまでにさまざまなコードクローン検出手法やツールが提案されている (節 2.5，2.6 参照)．

2.5 CCFinder の概要

CCFinder[24] は，コードクローン検出ツールの 1 つであり，単一または複数のファイルのソースコード中から全ての極大クローンを検出し，それをクローンペアの位置情報として出力する．CCFinder の持つ主な特徴は次の通りである．

細粒度のコードクローンを検出 字句解析を行うことにより，トークン単位でのコードクローンを検出する．

大規模ソフトウェアを実用的な時間とメモリで解析可能 例えば 10MLOC のソースコードを 68 分 (実行環境 Pentium3 650MHz RAM 1GB) で解析可能である [34].

様々なプログラミング言語に対応可能 言語依存部分を取り替えることで，様々なプログラミング言語に対応できる．現在は，C/C++，Java，COBOL/COBOLS，Fortran，Emacs Lisp に対応している．またプレーンテキストに対しても，分かち書きされた文章として解析可能となっており，未対応の言語に対しても完全一致判定によるコードクローンは検出することができる．

実用的に意味を持たないコードクローンを取り除く

- コードクローンは小さくなればなるほど偶然の一致の可能性が高くなるが，最小一致トークン数を指定することができるため，そのようなコードクローンを検出しないようにできる．
- モジュールの区切りを認識する．

ある程度の違いは吸収可能

- ソースコード中に含まれるユーザ定義名，定数をパラメータ化することで，その違いを吸収できる．
- クラススコープや名前空間による複雑な名前の正規化を行うことで，その違いを吸収できる．
- その他，テーブル初期化コード，可視性キーワード (protected, public，private 等)，コンパウンド・ブロックの中括弧表記等の違いも吸収することができる．

CCFinder のコードクローン検出手順 (ソースコードを読み込んで，クローンペア情報を出力する) は大きく 4 つの過程から成り立っている．

ステップ 1(字句解析) ソースコードを字句解析してトークン列に変換する．入力ファイルが複数の場合には，個々のファイルから得られたトークン列を連結し，単一のトークン列を生成する．

ステップ 2(変換処理) 実用上意味を持たないコードクローンを取り除くこと，及び，些細な表現上の違いを吸収することを目的とした変換ルールに従ってトークン列を変換する．例えば，この変換により変数名は同一のトークンに置換されるので，変数名が付け替えられたコード片もコードクローンであると判定することができる．

ステップ 3(検出処理) トークン列の中から指定された長さ以上一致している部分をクローンペアとして全て検出する．

ステップ 4(出力整形処理) 検出されたクローンペアについて，元のソースコード上での位置情報を出力する．

2.6 その他のコードクローン検出手法

CCFinder 以外にも，これまでにさまざまなコードクローン検出手法やツールが提案されている [1][3][4][5][6][7][8][9][13][26][27][28]．それぞれの手法やツールの特徴は次のようになっている．

CloneDR[9]

抽象構文木 (AST) の節点を比較することによって，コードクローン (類似部分木) の検出を行う．また，一部が異なるコードクローンも検出することが可能であり，検出したコードクローンを自動的に等価なサブルーチンやマクロに置き換えることも可能である．検出対象言語は，C/C++，COBOL，Java である．

Dup[3][4][5]

ユーザ定義名のパラメータ化を行った後，行単位の比較を行いコードクローンを検出する．マッチングアルゴリズムには，サフィックス木探索 [20] を用いているため線形時間で解析可能である．

Duploc[13]

前処理として，空白やコメント等を取り除いた後，行単位 (のハッシュ値) での表検索を用いた比較によってコードクローンを検出する．また，コードクローンの散布図等の GUI を備えたツールであり，ソースコード参照支援を行う．検出対象言語は，C，COBOL，Python，Smalltalk である．

JPlag[28]

ソースコードを字句解析し，トークン単位での比較を行う．プログラム盗用の検出を目的として開発され，プログラム間の類似率を検出する．検出対象言語は，C/C++，Java である．

Komondoor らの手法 [26]

関数等にまとめるのに適したコードクローンの抽出を目的として，プログラム依存グラフ (PDG) 上での各節点を比較することでコードクローン (同型 (isomorphic) 部分グラフ) を検出する．文字列比較や抽象構文木等を用いた検出方法では不可能であった非連続コードクローンや，対応行の順番が異なるクローン，互いに絡みあったクローン等を検出可能である．[26] で作成されたツールの検出対象言語は，C である．

Krinke の手法 [27]

AST や Traditional PDG に似た Fine-grained PDG というグラフ上での類似 (similar) 部分グラフ (同型部分グラフではない) を検出することで，コードクローンが存在すると思しき場所を検出する．試作ツールの検出対象言語は，C である．

SMC[6][7][8]

まず，特徴メトリクスによってコードクローンと思しきメソッドに絞り込む．次に，絞り込まれたメソッドのペアに対し，表検索を用いることでメソッド単位のコードクローンを検出する．特徴メトリクスによって絞り込まれているため，実用上ほぼ線形時間で解析可能である．また，検出されたペアのメソッドは，特徴に応じて 18 種類に分類される．さらにそれぞれの分類については共通メソッドへ書き換える指針が示されている．

MOSS[1]

検出アルゴリズムは公開されていない．JPlag 同様，プログラム盗用の検出を目的として開発された．検出対象言語は，Ada，C/C++，Java，Lisp，ML，Pascal，Scheme である．

いずれの手法，ツールにおいても提案者によってコードクローンの定義が微妙に異なり，検出されるコードクローンが異なる．つまり，コードクローンの定義とは検出アルゴリズムそのものによって定義される．Burd らは，全ての面において他のツールよりも優れているツールはなく，使う場面に応じて，適切なツールを選ぶ必要があると述べている [10]．

今回，トークン単位での類似コードを検出する CCFinder を用いた理由は，リファクタリングについての研究で利用されている実績があるためである [21].

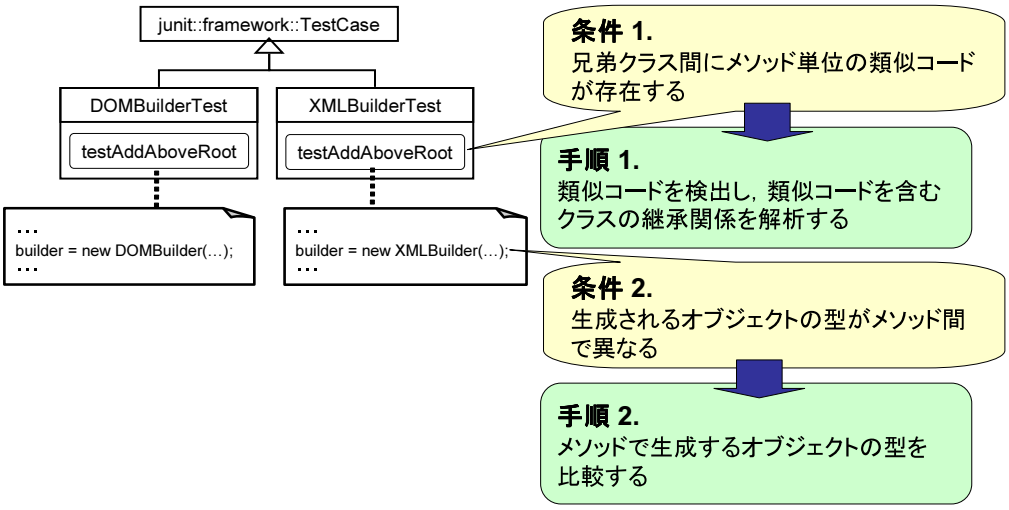


図 9: 提案手法

3 提案手法

本研究では，ソースコードから図 9 の構造を持つ部分を自動的に検出することによって，“ Factory Method によるポリモーフィックな生成の導入 ” を支援する手法を提案する．

まず，図 9 の構造を持つ部分を自動的に検出するために，自動的に判定できる条件を定める．前述の “ Factory Method によるポリモーフィックな生成の導入 ” が掲載されている文献 [25] を参考に，以下の条件を定めた．

条件 1. 兄弟クラス間にメソッド単位の類似コードが存在する．

条件 2. 生成されるオブジェクトの型がそれらメソッド間で異なる．

条件 1. を定めた理由は，図 9 の例は “ Template Method の形成 ” [25] の特殊な事例だからである．兄弟クラス中のメソッド単位の類似コード (図 9 の 2 つの testAddAboveRoot メソッド) では，大部分の処理は共通しているが，一部分 (オブジェクトを生成する構文) が異なっている．その異なる部分を，共通のシグニチャを持つメソッド (図 7(b) の createBuilder メソッド) に置換すれば，メソッド単位の類似コードの内容が揃う．それらのメソッドを共通の親クラスに引き上げてまとめることで，類似コードを除去することができる．この引き上げたメソッド (図 7(b) の AbstractBuilderTest クラスに存在する testAddAboveRoot メソッド) が Template Method[16] である．条件 2. を定めた理由は，生成されるオブジェクトの型がメソッド間で一致していてメソッド間に実質の差異がないので，メソッドの引き上げのリファクタリングを行う方が適切である部分と，生成されるオブジェクトの型がメソッド間で異なるので Factory Method パターンの適用が適切である部分とを判別するためである．

次に，以上の条件を満たすコード片を検出するため，以下の手順を定めた．

手順 1. コードクローン検出ツール CCFinder[24] で類似コードを検出し，類似コードを含むクラスの継承関係を解析する．

手順 2. 手順 1. で検出したメソッドで生成するオブジェクトの型を比較する．

Java で書かれたソースコードを対象として，デザインパターンが適用可能な部分を検出するツールを Java で実装した．ツールは，ソースコードとコードクローン検出ツール CCFinder の出力ファイルとを入力とし，デザインパターンが適用可能な部分として，Factory Method パターンが適用可能なメソッド単位の類似コード群とそれらメソッドに共通した親クラス名を出力する (図 10 参照)．

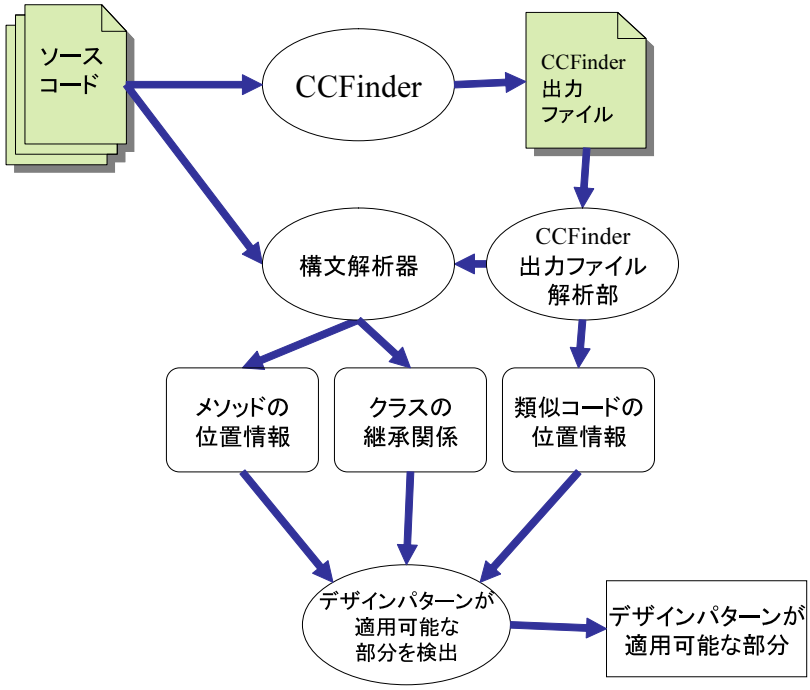


図 10: 実験用ツールの構成

CCFinder の出力ファイルを解析することで，類似コードの位置情報を取得すると同時に入力ファイルのパスを構文解析器に伝達する．構文解析器では，オブジェクトを生成する構文を含むメソッドの位置情報とクラスの継承関係を取得する．構文解析器には JavaCC[18] を使用した．以上の処理で取得した情報から，兄弟クラスに存在する，オブジェクトを生成する構文を含む，メソッド単位の類似コード群を検出する．それらが，Factory Method パターンが適用可能であると判定される部分である．処理の都合上，生成するオブジェクトの

型が一致していて，実質差異のないメソッドも検出されるので，Factory Method パターンの適用が適切な部分とメソッドの引き上げのリファクタリングが適切な部分とを分けて出力している．

4　適用実験

適用実験の内容は以下の 2 つである．

- 実際のソフトウェアにおいて，提案手法で定めたデザインパターンが適用可能である条件に合う部分がどの程度存在するかの調査
- ツールが検出した部分の一例に対して実際にデザインパターンの適用が可能かどうかの調査

3 章で述べたように，提案手法の実現には CCFinder を用いている．実験では，CCFinder が検出する類似コードの最小トークン数をデフォルトの 30 トークンに設定した．今回対象としたオープンソースのソフトウェアは次の通りである．

ANTLR2.7.4[17] Java，C#，C++等に対応したコンパイラ・コンパイラ

Ant1.7.0[2] ビルドツール

Azureus3.0.3.4[12] Peer to Peer ファイル共有ソフト

JBoss3.2.6[22] アプリケーションサーバ

jEdit4.3[14] テキストエディタ

JHotDraw7.0.9[23] 図形描画ソフト

SableCC3.2[31] AST を生成できるコンパイラ・コンパイラ

Soot2.2.4[19] Java バイトコードのコンパイラ，最適化ツール

WALA1.1[33] Java バイトコード及びその関連言語の分析ツール

4.1　デザインパターンが適用可能である条件に合う部分の数の調査

ツールの実行結果を表 1 に示す．“ Factory Method ”の列は，Factory Method パターンの適用が可能であるとツールが判定した部分の数である．“ Pull Up Method ”は，Factory Method パターンの適用は可能であるが，生成されるオブジェクトの型が類似コード間で一致しているので，類似コード間に実質の差異がなく，メソッドの引き上げを行う方が適切であるとツールが判定した部分の数である．

数万から数十万の 9 つのソフトウェアから，1 つにつき最大で 14 個の部分が検出された．

表 1: ツールの実行結果

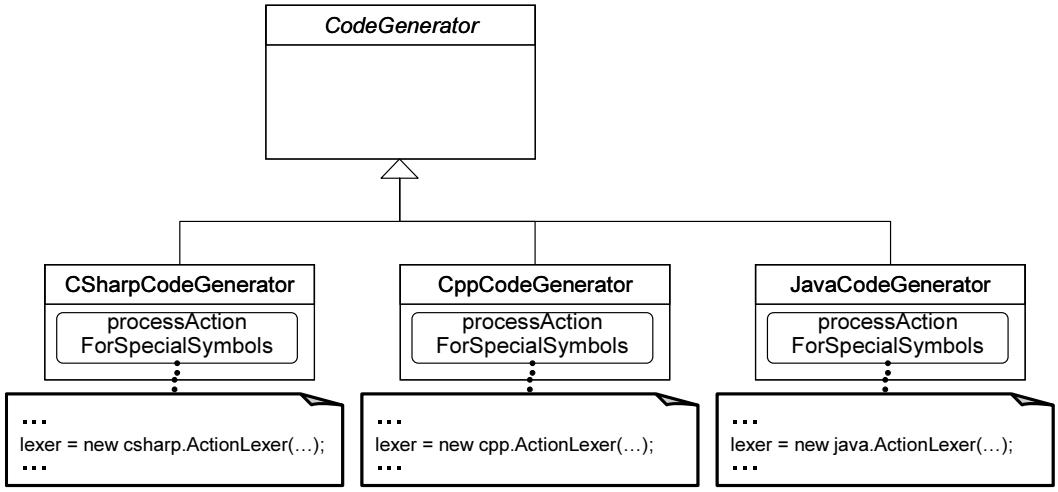
対象ソフトウェア	対象ソースコードの行数	対象クラスの数	対象インタフェースの数	Factory Method	Pull Up Method
ANTLR	32K	167	25	1	9
Ant	198K	994	74	2	15
Azureus	538K	2226	774	14	34
JBoss	679K	3372	909	10	44
jEdit	168K	922	63	0	1
JHotDraw	90K	487	52	1	24
SableCC	35K	237	5	1	1
Soot	352K	2298	250	9	61
WALA	210K	1565	289	6	16

4.2 実際にデザインパターンの適用が可能かどうかの調査

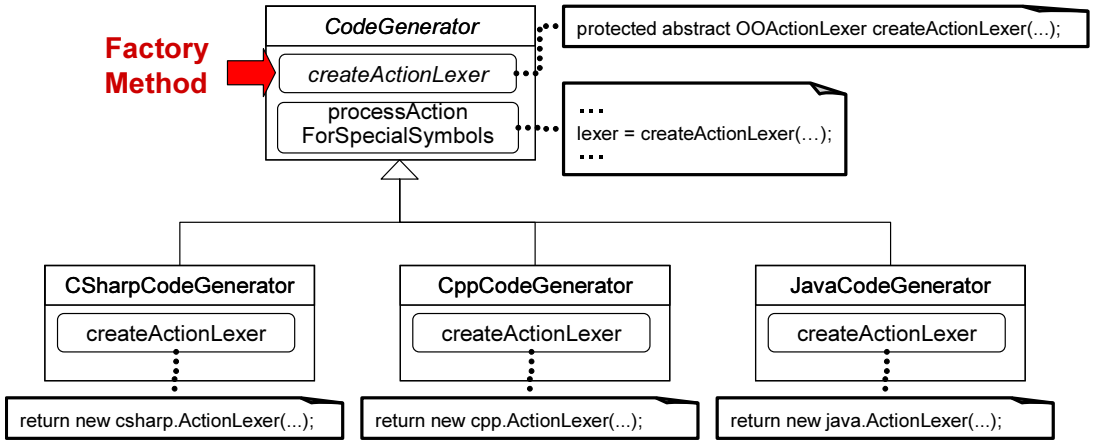
ANTLR からの検出部分に対し，実際に Factory Method パターンの適用を行った．ツールは ANTLR のメソッドのうち，図 11(a) の 3 つの processActionForSpecialSymbols メソッドを Factory Method パターンが適用可能な部分として検出した．processActionForSpecialSymbols メソッドを CodeGenerator クラスに引き上げ，まとめることで，類似コードの除去が可能である．

図 11(a) の 3 つの processActionForSpecialSymbols メソッドには，それぞれ csharp.ActionLexer 型，cpp.ActionLexer 型，java.ActionLexer 型のオブジェクトを生成する文が存在する．まず，Factory Method がオブジェクトの生成を行う構造に変更するには，上記の 3 種類のオブジェクトに共通の型 (csharp.ActionLexer クラス，cpp.ActionLexer クラス，java.ActionLexer クラスに共通する親クラスかインタフェース) が存在する必要があるので，共通の型が存在するかソースコードを参照して確認した．その結果，共通の型は存在しないことが判明したので，新たに OOActionLexer インタフェースを定義した．csharp.ActionLexer クラス，cpp.ActionLexer クラス，java.ActionLexer クラスが OOActionLexer インタフェースを実装する構造に変更し，Factory Method の定義が可能になった．

次に，processActionForSpecialSymbols メソッド内のオブジェクトを生成する文をそれぞれ抽出して，OOActionLexer 型のオブジェクトを生成して返り値とする createActionLexer メソッドに全て置換した．そして，createActionLexer メソッドを共通の親クラスである



(a) Factory Method パターン適用前



(b) Factory Method パターン適用後

図 11: ANTLR に対する Factory Method パターンの適用

CodeGenerator クラスに抽象メソッドとして定義した．CodeGenerator クラスを継承するクラスは，図 11 の 3 種類のクラス以外にも存在し，それらのクラスでも createActionLexer メソッドを実装する必要があるので，null を返り値とするメソッドとして実装した．

最後に Template Method の形成を行い，processActionForSpecialSymbols メソッドを共通の親クラスである CodeGenerator クラスに引き上げ，類似コードを除去した．ソースコードの変更の前後でソフトウェアの外部的動作が変化していないことを 83 個のテストケースを用いて確認した．

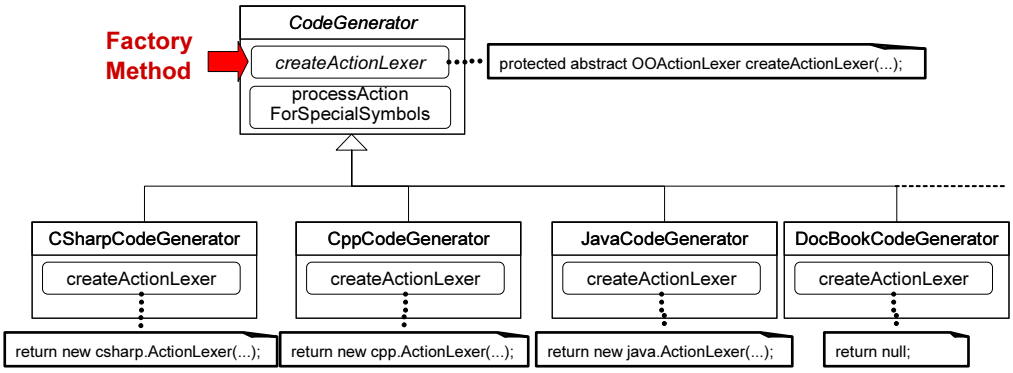
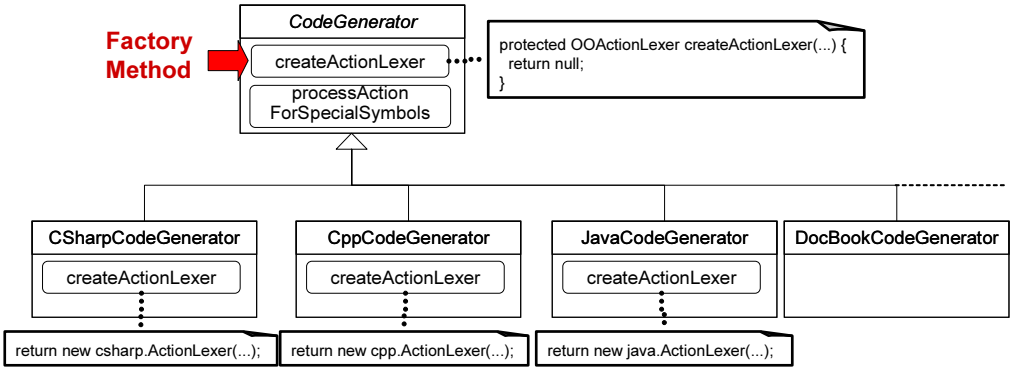


図 12: 適用実験での Factory Method パターン適用後の構造 (一部抜粋)

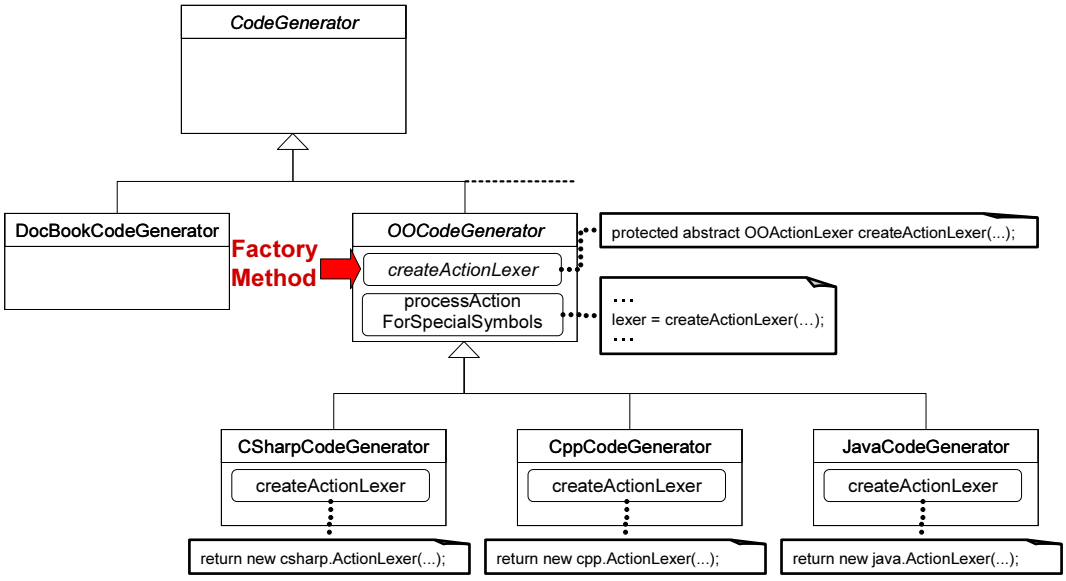
5 考察

Factory Method パターンを ANTLR に適用した結果，対象クラスの類似コードを除去することができた．しかし，CodeGenerator クラスに定義した Factory Method が原因で，Factory Method パターンとは関係のない子クラスでも Factory Method をオーバーライドする必要がある構造になってしまった (図 12)．抽象メソッドではなく，null を返り値とする具象メソッドとして Factory Method を CodeGenerator クラスに定義し，子クラスでオーバーライドする構造 (図 13(a)) や，CSharpCodeGenerator クラス，CppCodeGenerator クラス，JavaCodeGenerator クラスにスーパークラスの抽出を行い，新たに共通の親クラスを作成した構造 (図 13(b)) に変更した方が理解しやすい構造になるかもしれない．

また，CodeGenerator クラスの子クラスすべてに processActionForSpecialSymbols メソッドが存在することがソースコードを参照することで判明した．Factory Method パターンに関係のない子クラスでは，processActionForSpecialSymbols メソッドをオーバーライドして処理内容を相当に変更する構造になる．ただし，Factory Method パターンに関係のない子クラスの processActionForSpecialSymbols メソッドは使用されていないことが，ソースコード中のコメントから判明したので，削除すれば現在の仕様では問題にならなくなる．



(a) Factory Method を具象メソッドにする



(b) スーパークラスの抽出を行う

図 13: ANTLR の構造の変更案 (一部抜粋)

6 関連研究

6.1 JIAD

JIAD[30] は，論理学の推論を用いてデザインパターンが適用可能な部分を検出するツールである．デザインパターンごとに，適用が可能なソースコードに現れる特徴に沿って命題を定め，構文解析を行うことで得られる情報，例えば，メソッドの所属するクラス，メソッドの呼び出し関係などの情報を定めた命題に与えて推論を行う．さらに，ツールによって得られた情報を自動リファクタリングツールに入力することで，自動的にデザインパターンをソースコードに適用することができる．ただし，人の手によるリファクタリングは支援していない．また，デザインパターンを適用した結果やトレードオフについては言及していない．

対して本研究では，ソースコードから検出した類似コードを解析することで，デザインパターンが適用可能な部分を検出している．本研究で利用している文献 [25] では，適用した結果についても述べている．

6.2 Cinnéide らの手法

Cinnéide らは，デザインパターンをソースコードへ適用する作業を自動化する手法の提案をしている [11]．その手法に従い試作されたツールでは，ソースコードと，その変更に関係のあるクラスの名前や新たに必要となるインタフェースやメソッドの名前などを入力することで，ソースコードを構文解析して得られる情報を利用し，指定された部分へのデザインパターンの適用は適切か判断した上で，デザインパターンを適用したコードを出力する．本研究の手法は，ソースコードから検出した類似コードを解析して，デザインパターンが適用可能な部分を検出するが，ソースコードを変更することはしない．

彼らの手法は，既存のソフトウェアに新たな機能を追加することが既存の構造では難しい場合に，デザインパターンを適用して拡張が容易になるように構造を変更するために使用されることを想定している．そのため，ツールの使用者が変更対象のプログラムの構造やデザインパターンを適用したい部分のクラスの関係などについて知っている場合を想定している．対して本研究の手法は，変更の対象であるソースコードの理解や保守を容易にするために使用されることを想定している．結果としてソフトウェアの機能を拡張することも容易になるが，対象のソフトウェアのソースコード全体からデザインパターンが適用可能な部分を検出するので，ある機能の拡張を容易にすることに絞ったデザインパターンの適用には不向きである．また，変更の対象であるソフトウェアの構造について詳しく知らなくても使用できる．

7　まとめと今後の課題

本研究では，コードクローン検出法を用いて既存ソフトウェアのソースコードからデザインパターンが適用可能な部分を自動的に検出し，既存ソフトウェアに対するデザインパターンの適用を支援する手法の提案を行った．具体的には，CCFinder が出力した類似コードの位置情報と構文解析を利用することで得られたクラス継承関係とから，Factory Method パターンが適用可能な部分を検出する．

この提案手法を Java で実装し，Java で書かれたオープンソースソフトウェアを対象に適用実験を行った．その結果，Factory Method パターンが適用可能な部分を検出できることが確認できた．

本研究の提案手法を利用すれば，既存ソフトウェアに Factory Method パターンを適用して類似コードを除去し，かつ拡張性を高める作業にかかるコストが減少する．そのため，設計品質の高いソフトウェアを開発しやすくなることが期待できる．

今回実装したツールでは，Factory Method パターンを適用することで書き換えられる部分の情報しか出力していなかったので，実際に Factory Method パターンの適用を行う際に，Factory Method パターンの適用で影響を受けるクラスやメソッドを調べるためソースコードを参照する作業が多かった．加えて，Factory Method パターンを適用することが適切か判断するためにも，ソースコードを直接参照する必要があった．この 2 点から，ツールが出力する情報を増やすことで，直接参照しなければならないソースコードの量を減らすことが可能だと考えられる．例えば，Factory Method パターンが適用可能な部分の兄弟クラスすべてに関する情報などを出力する，Factory Method パターンの適用で減らすことができるソースコードの量を出力するなどが考えられるが，どのような情報を出力すればよいかは実験を通して調査が必要である．また，“Factory Method によるポリモーフィックな生成の導入”以外のデザインパターンを適用するリファクタリングのいくつかもコードクローン検出法を用いた同様の手法で支援可能だと考えられる．以上のことから，今後の課題としてツールが出力する情報の改善，Factory Method パターン以外のデザインパターンが適用可能な部分の検出が挙げられる．

謝辞

本研究において，常に適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授に心より深く感謝いたします.

本研究において，適切な御指導および御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下 誠 准教授に深く感謝いたします.

本研究において，逐次適切な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 石尾 隆 助教に深く感謝いたします.

本研究において，様々な御指導および御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 吉田 則裕 氏，三宅 達也 氏，宮崎 宏海 氏に深く感謝いたします.

最後に，その他様々な御指導，御助言等を頂いた大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様に深く感謝いたします.

参考文献

[1] A. Aiken. A System for Detecting Software Plagiarism (Moss Homepage). `http://www.cs.berkeley.edu/~aiken/moss.html`.

[2] Apache Ant. `http://ant.apache.org/`.

[3] B. S. Baker. A Program for Identifying Duplicated Code. In *Proc. of Computing Science and Statistics*, Vol. 6, pp. 49-57, 1992.

[4] B. S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proc. of WCRE 2000*, pp. 86-95, 1995.

[5] B. S. Baker. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. *SIAM Journal on Computing*, Vol. 26, No. 5, pp. 1343-1362, 1997.

[6] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Measuring Clone Based Reengineering Opportunities. In *Proc. of METRICS '99*, pp. 292-303, 1999.

[7] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Partial redesign of Java software systems based on clone analysis. In *Proc. of WCRE '99*, pp. 326-336, 1999.

[8] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proc. of WCRE 2000*, pp. 98-107, 2000.

[9] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proc. of ICSM '98*, pp. 368-377, 1998.

[10] E. Burd, and J. Bailey. Evaluating Clone Detection Tools for Use during Preventative Maintenance. In *Proc. of SCAM 2002*, pp. 36-43, 2002.

[11] M. Ó Cinnéide, and P. Nixon. A Methodology for Automated Introduction of Design Patterns. In *Proc. of ICSM '99*, 1999.

[12] Azureus : Java BitTorrent Client. `http://azureus.sourceforge.net/`.

[13] S. Ducasse, M. Rieger, and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proc. of ICSM '99*, pp. 109-118, 1999.

[14] jEdit - Programmer's Text Editor. `http://www.jedit.org/`.

[15] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addition Wesley, 1999.

[16] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addition Wesley, 1995.

[17] ANTLR Parser Generator. `http://www.antlr.org/`.

[18] JavaCC, The Java Parser Generator. `http://suntest.com/JavaCC`

[19] Sable Research Group. `http://www.sable.mcgill.ca/`.

[20] D. Gusfield. *Algorithms on Strings, Trees, And Sequences.* Cambridge University Press, 1997.

[21] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Method and Implementation for Investigating Code Clones in a Software System. *Information and Software Technology*, Vol. 49, No. 9-10, pp. 985-998, 2007.

[22] JBoss.com. `http://www.jboss.com/`.

[23] JHotDraw. `http://sourceforge.net/projects/jhotdraw/`.

[24] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654-670, 2002.

[25] J. Kerievsky. *Refactoring to Patterns.* Addition Wesley, 2004.

[26] R. Komondoor, and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In *Proc. of SAS 2001*, pp. 40-56, 2001.

[27] J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proc. of WCRE 2001*, pp. 301-309, 2001.

[28] G. Malpohl, L. Prechelt, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, Vol. 8, No. 11, pp. 1016-1038, 2002.

[29] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[30] J. Rajesh, and D. Janakiram. JIAD: A Tool to Infer Design Patterns in Refactoring. In *Proc. of PPDP 2004*, pp. 227-237, 2004.

[31] SableCC. `http://sablecc.org/`.

[32] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On Detection of Gapped Code Clones using Gap Locations. In *Proc. of APSEC 2002*, pp. 327-336，2002.

[33] WALA. `http://sourceforge.net/projects/wala/`.

[34] 井上克郎, 神谷年洋, 楠本真二,“コードクローン検出法”. コンピュータソフトウェア, Vol. 18, No. 5, pp. 47-54, 2001.