メソッド合成による新規クラスの自動生成

丸山勝久[†,☆] 島健一[†,☆☆]

本論文では，オブジェクト指向プログラミングにおいて，既存クラスのメソッドとクラスの変更履歴から，開発者の要求するメソッドを持つ新規クラスを自動生成する手法を提案する．継承を用いてプログラムを作成した際，親クラスとその後継者クラスが持つメソッド間の変更差異を特定し，その履歴を親クラスに蓄積する．開発者が参照しているクラスにおいて，開発者の要求した呼び出しメソッドが定義されていない場合，このメソッドに関連する変更差異を持つ類似クラスを見つけ，その差異を参照クラスに存在するメソッドと合成することで，呼び出しメソッドを生成する．類似クラスは，ライブラリにおけるクラス階層関係とメソッドの名前およびシグニチャの等価判定に基づき決定する．また，変更差異の特定と合成は，オブジェクト指向プログラム対応の区間限定スライシングによる機能分割とクラス依存グラフの同形写像比較による等価機能特定を組み合わせたメソッド合成アルゴリズムにより実現する．本手法により自動生成された新規クラスを再利用することで，継承におけるメソッドや変数の追加および再定義という開発者の負担が軽減される．

Generating New Classes by Method Integration

KATSUHISA MARUYAMA[†,☆] and KEN-ICHI SHIMA[†,☆☆]

The proposed mechanism automatically generates new classes from classes existing in a user library by using their modification histories. To generate classes that are likely to meet user requirements and consistent with the existing classes, the history of modifications between methods with the same interface of a parent class and its heir is stored. If the required method is not defined in the existing class which a user is referring to, the past modifications of classes similar to the referenced class are applied to the referenced class. Classes are determined to be similar based on their positions in a class hierarchy tree. Specifying different fragments in modifications and integrating these fragments with the referenced class are achieved by a method integration algorithm based on bounded slicing for object-oriented programs and matching for class dependence graphs. The mechanism enables users to reuse new classes with little or no modification, and thus easily create object-oriented programs.

1. はじめに

オブジェクト指向プログラミングでは，継承（inheritance）を用いることで，既存のクラスから派生させた後継者クラスにおいて，開発者がメソッドや変数を追加および再定義するだけで，要求するプログラムを作成することが可能である[1)]．継承により，ライブラリ内に存在する既存クラスの再利用性は高くなり，開発者が同じコードを繰り返し記述することが避けられる．

しかしながら，後継者クラスで再定義したメソッドが派生元である親クラスのメソッドと関係を持つことがあるため，開発者は再定義メソッドの変更部分が他の部分に与える影響を十分に考慮してコード変更を行う必要があり，その変更は容易であるとはいえない．さらに，継承における再定義の最小単位はメソッドである．このため，要求する機能が既存クラスで定義されているメソッドの持つ機能と微妙に異なる場合，たとえば，メソッド引数の１つの入力条件だけが異なる場合でも，開発者はメソッド全体を再定義することになり，コード変更が要求される場面は多い．このように，継承により既存クラスを再利用できる機会は増加するが，再利用時のコード変更に関する開発者の負担はそれほど軽減していない．

このような問題に対して，我々は，開発者がオブジェクト生成（instantiation）とメソッド呼び出し（メッセージ送信，message passing）のコードを記述するだけでプログラムを作成できるようにすることを目指

† NTT ソフトウェア研究所
NTT Software Laboratories
☆ 現在，NTT メディア技術開発センタ
Presently with Media Technology Center, NTT
☆☆ 現在，NTT 移動通信網株式会社マルチメディア研究所
Presently with Multimedia Laboratories, NTT Mobile Communications Network Inc.

す．本論文では，クラス変更の履歴を積極的に利用することで継承におけるコード変更を自動化し，ライブラリ内に存在する既存クラスから開発者の要求する機能を満たす可能性の高い新規クラスを自動生成する手法を提案する．本手法は，1）新規クラスを生成する際に過去のクラス変更を模倣する，2）独立に変更された2つのプログラムを依存関係に関する無矛盾性を保ちながら合成可能なプログラム合成アルゴリズム[2]に基づく，という点が特徴である．ここで，本論文で扱う変更は，メソッドや変数の追加および再定義というクラス内部のコードだけを書き換えることを指し，クラス階層の再構成を含まない．

過去のクラス変更を模倣するため，開発者が既存クラスを変更した際に，その変更の履歴を既存クラスに蓄積する．次に，開発者が再利用しようとした参照クラスにおいて呼び出しメソッドが定義されていないとき，過去において類似クラスが受けた変更の履歴を参照クラスに適用する．これらの動作は，能動的部品の機能交換変化メカニズム[3]における，区間限定プログラム・スライシング[4]とプログラム依存グラフ（PDG: Program Dependence Graph）[5]の同形写像比較を拡張することで実現可能である．

本手法をオブジェクト指向プログラム開発に導入することで，継承におけるメソッドや変数の追加および再定義というコード変更に関する開発者の負担の軽減が期待でき，開発者が容易にプログラムを作成可能となる．

以下，2章では新規クラス生成手法の概要を示す．次に，3章で本手法を実現するために必要な技術として，プログラム・スライシング，グラフの同形写像比較，メソッド複写を述べる．さらに，4章で本手法を実現する3つの手続きを示し，5章で新規クラスの生成例を紹介する．最後に，6章で本手法に関する考察を行う．

2. 新規クラス生成手法

本論文では，オブジェクト指向プログラム開発において，既存クラスに対する過去の変更は類似のクラスに対しても同様に行われる可能性が高い，という仮定をおく．この仮定は，開発者が新規のプログラムを作成する際に，類似のプログラムに対して過去に行った変更を模倣することから導かれ，新規クラスの生成という目的において妥当である．このような仮定に基づき，本手法は，たとえ開発者の要求するクラスがライブラリ内に存在しない場合でも，開発者の要求を満たす可能性の高いクラス候補を開発者に提供する．

本手法では，開発者の記述したメッセージ送信に対して，そのメッセージを処理可能なメソッドを持つクラスを，開発者の要求を満たすクラスと見なし，要求をメソッドのインタフェースで表現する．メソッドのインタフェースとは，メソッドの名前およびシグニチャ（戻り値の型および引数の数と型）を指す．

類似クラスにおける過去の変更を模倣することで，開発者の要求を満たす新規クラスを自動生成する手法は，次に示す仕組みにより実現可能である．

(1) 親クラスとそのクラスから派生した後継者クラスの機能に関する変更差異を特定し，その差異を変更履歴として蓄積する仕組み．
(2) 開発者が参照している既存クラスに対して，類似な機能を持つ類似クラスを見つける仕組み．
(3) 類似クラスの持つ履歴に含まれる変更差異を，開発者の参照している既存クラスの機能に組み込む仕組み．

機能とは，メソッドにおいて，特定の入力変数の値から着目する出力変数の値を計算する際の動作を指す．(1)と(3)の仕組みを，能動的部品の機能交換変化メカニズムにより実現する．この際，オブジェクト指向プログラムから新規クラスを生成できるようにするため，手続き型プログラムから新規部品を生成する従来の合成アルゴリズム[3]に対して，次の2つの改良を加えた．1）オブジェクト指向プログラム対応の区間限定スライシングを用いて変更コードを抽出する，2）生成した新規クラスにおいて，呼び出しメソッドの同一性と参照可能性を保証するためにメソッドを複写する．

また，(2)の仕組みを，単一継承により階層化されたライブラリにおけるクラスの位置とメソッドインタフェースの同一性に基づくクラス探索手続きにより実現する．ここで，2つのクラスが類似であるとは，それらのクラスが祖先から継承したメソッドや変数を数多く共有していることを指す．

クラス自動生成手法の概要を**図1**に示す．四角形はクラスを表す．楕円は，上記の3つの仕組みをそれぞれ実現する，(1) 変更差異特定部（Specifier），(2) 類似クラス探索部（Finder），(3) 変更差異合成部（Integrator）である．

開発者が，継承により親クラス（*Parent*）から後継者クラス（*Heir*）を派生させ，メソッドや変数の追加および再定義を行った場合を考える．変更差異特定部は，*Perent* と *Heir* のメソッドから変更差異を抽出し，この差異を *Parent* の履歴（history）に蓄積する．次に，開発中のクラス（*Client*）から呼び出されたメソッ

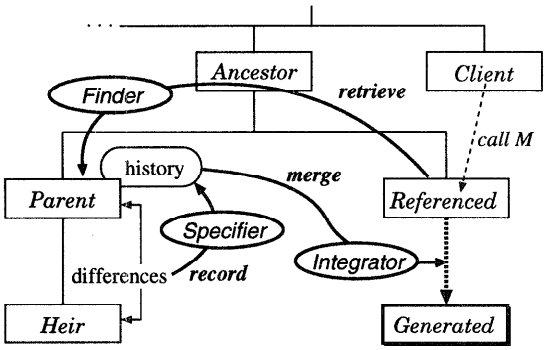


図 1 クラス生成手法の概要
Fig. 1 Overview of our mechanism for generating classes.

ド M が，参照クラス（*Referenced*）において定義されていない場合を考える．参照クラスとは，メッセージの受信先として指定されたインスタンス（オブジェクト）の生成元クラスを指す．実際のプログラミングにおいて，開発者が要求だけから，*Referenced* に存在しないメソッド M を呼び出す文を *Client* に記述することは少ない．しかしながら，開発者が *Referenced* の変更を前提に開発を進めている場合，他のクラスに存在するメソッドの名前と機能を参考にして，*Client* から呼び出すメソッドのインタフェースを（呼び出されるメソッドの実体を記述する前に）決定しておくことは多い．よって，このようなメソッド呼び出し文を実際にソースコード内に記述することを奨励した場合，その記述は多いと考えられる．類似クラス探索部は，ライブラリの階層関係をたどることにより類似クラス（この例では，*Parent*）を見つける．変更差異合成部は，*Referenced* に類似クラスの持つ変更差異の一部を組み込むことで，新規クラス（*Generated*）を生成する．

このように，本手法では，開発者が親クラス *Parent* から後継者クラス *Heir* を派生させた際の変更を模倣することで，既存の参照クラス *Referenced* から新規クラス *Generated* を生成する．*Generated* は，開発者の要求したメソッド M と，M に関連するために変更の影響を受けたメソッドを持つ．開発者は，*Generated* を無変更あるいは少ない変更により再利用することで，作成中のクラス *Client* を実行可能である．

3. 準　　備

本章では，まず，メソッドの合成を実現するための準備として，クラス依存グラフについて述べ，このグラフを用いた区間限定スライシングおよび同形写像比較について述べる．さらに，継承の特性によりクラス生成時に生じる問題を示し，この問題を回避するメソッド複写を提案する．ここで，本手法は，型宣言のある静的型推論機構を持つ単一継承のオブジェクト指向言語を扱う．本論文では，Java 言語[6] の記法を用いる．

3.1 クラス依存グラフ

クラス内部のコードを抽出，比較，合成する際，PDG を拡張したクラス依存グラフ（ClDG: Class Dependence Graph）[7] を用いる．PDG とは，プログラム内の代入文および条件式をラベルとして割り当てた節点の集合と，各節点間のデータ（定義–参照）依存関係（data dependence）および制御依存関係（control dependence）を表す矢印の集合からなる有向グラフである．ClDG において，クラス内の各メソッドは手続き依存グラフ（procedure dependence graph）[8] で表現し，手続き型プログラムにおける関数と見なす．ClDG は，メソッド呼び出しの引数やインスタンス変数の受け渡しを担う入出力変数節点（formal-in 節点，actual-in 節点，formal-out 節点，actual-out 節点）とそれらの節点間のデータ依存関係（parameter-in, parameter-out dependence），メソッド呼び出し関係（call dependence），および呼び出しメソッドにおける引数のデータ依存関係（interprocedural flow dependence）を表す矢印を，PDG に付加することで作成する．ライブラリに存在するクラスのメソッドに関しては，あらかじめ内部の依存解析を行い，メソッド引数に関するデータ依存関係を求めておく．本手法では，スライシングおよび同形写像比較における計算量を抑えるため，これら解析済みメソッドの内部コードを ClDG において展開せず，メソッド呼び出し節点に関する依存関係に置き換える．

3.2 オブジェクト指向プログラム対応の区間限定スライシング

プログラム・スライシング[9] とは，プログラム内の任意の文において着目する変数に影響を与える一部のコードだけを，もとのプログラムから抜き出すことである．抜き出したコードの集まりをスライスと呼び，依存グラフにおける到達可能性の判定により作成できる[10]．

我々は，オブジェクト指向プログラムに対して，ClDG を用いる文献 11) のスライシング技法を採用し，この技法をスライシングの対象区間を指定できるように拡張した区間限定スライシングを用いる．区間限定スライス（bounded slice）[4] $\hat{S}(n_u, n_l, V)$ は，節点 n_u，n_l によって指定された区間内において，着目する変数 v $(\in V)$ に関して依存関係をたどることに

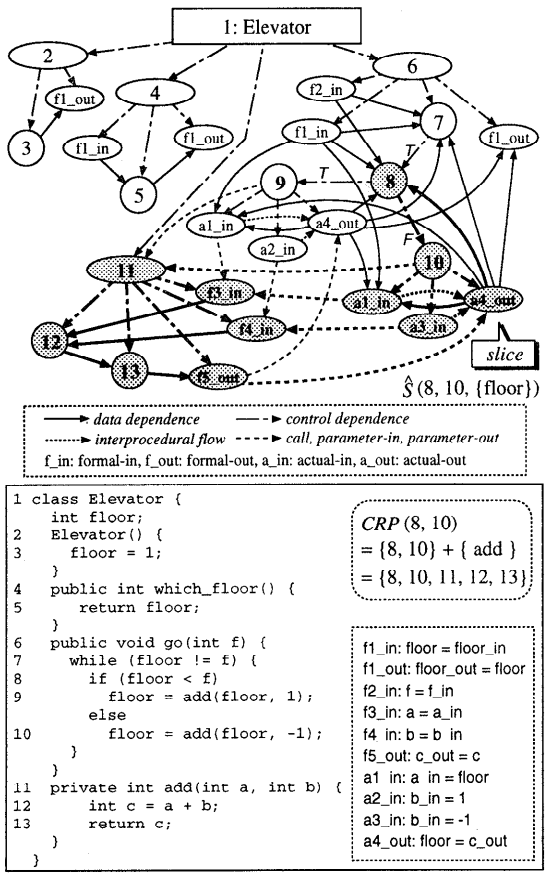


図 2 クラス依存グラフと区間限定スライスの例
Fig. 2 Example of a CIDG and a bounded slice.

よって得られる静的スライスである．区間は，制御フローグラフ（CFG: Control Flow Graph）[12] 上の上限節点 n_u と下限節点 n_l の制約付き到達可能経路[4] に含まれる節点集合 $CRP(n_u, n_l)$ によって定義される．オブジェクト指向プログラム対応の区間限定スライシングは，任意のプログラムの同一のメソッド内において n_u と n_l を自由に設定でき，指定した区間内の依存関係だけに基づきスライスを抽出可能であるという利点を持つ．よって，このような区間限定スライシングをプログラム合成に導入することで，より多くの変更差異を特定および合成可能である．本論文において，オブジェクト指向プログラムの区間限定スライスを単にスライスと呼ぶ．

CIDG と区間限定スライスの例を図 **2** に示す．網掛け節点および太矢印が，節点 10 を実行した後の変数 `floor` の値に影響を与えるスライス $\hat{S}(8, 10, \{\texttt{floor}\})$ である．

3.3 クラス依存グラフの同形写像比較

本手法では，2 つのプログラムの機能に関する等価性を判定するために，グラフの同形写像比較[13] を用いる．この方法では，比較する 2 つのグラフにおいて，一方のグラフの節点と矢印を他方のグラフに写像することで，同形を判断する．

同形写像比較において，グラフ G_1 と G_2 が等価であるとは，次の 4 つの条件を満たすことを指す．

(1) G_1 と G_2 が同数の節点と同数の矢印を持つ．
(2) G_1 の節点から G_2 の節点に，1 対 1 かつ上への写像 φ が存在する．
(3) G_1 の各矢印 $p \to q$ に対して，矢印 $\varphi(p) \to \varphi(q)$ が G_2 に存在し，対応する矢印の種類（データ依存，真あるいは偽の制御依存）が等しい．
(4) G_1 の各節点 p に対して，p のラベルと G_2 内の節点 $\varphi(p)$ のラベルが等しい．

同形写像比較において，同一のクラスから生成したインスタンスは同じメソッドや変数を持つので，その機能は等価であると見なせる（同一の基本データ型から生成したインスタンスも，その機能は等価であると見なす）．しかし，実際に比較を行う際には，インスタンスを参照する変数名の違いによって，条件 (4) が満たされないことがある．さらに，変数名が同じでも，その変数の指しているインスタンスの機能が異なる場合がある．

そこで，機能の等価性を判定可能とするために，比較前に CIDG の各節点のラベルにおける変数名を型に基づき置換する．まず，比較する 2 つの CIDG に対して，型が一致する変数名の対を置換変数の候補として求める．このとき，比較対象のクラス（CIDG）に宣言を持たない変数については，その変数が指すインスタンスが同一機能の場合に必ず同じ変数名で参照されるため，置換対象候補としない．また，比較対象のクラスに型が異なる同名の変数が存在する場合，クラス名を変数名の前に付加することで，それぞれ別の変数名に変更する．次に，置換変数の候補に基づき一方の CIDG において変数の置換を行った際，2 つの CIDG の各節点のラベルが一致するかどうかを検査する．CIDG のすべての節点のラベルにおいて成立する置換変数の対応関係 φ_v が存在するとき，実際に変数名を対応関係 φ_v に基づき置換する．以上より，上記の条件 (4) を，以下のように再定義する．

(4) G_1 の各節点 p に対して，φ_v により変数名を置換した p のラベルと G_2 内の節点 $\varphi(p)$ のラベルが等しい．

変数名を置換する様子を図 **3** に示す．クラス A あるいは B で宣言される変数に対して，型に基づき置換変数

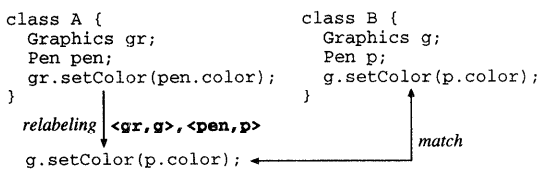


図 3 変数名の置換
Fig. 3 Relabeling for variable names.

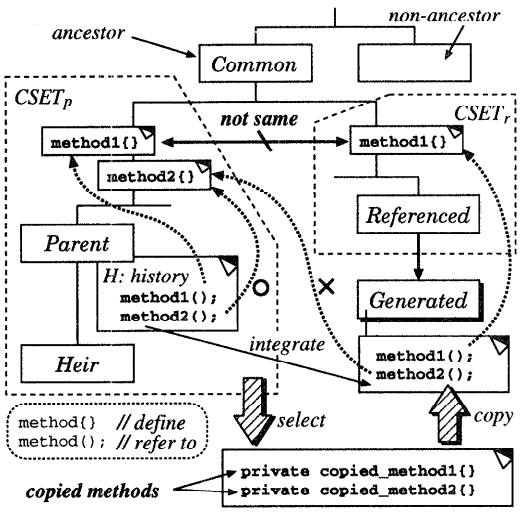


図 4 メソッドの呼び出しに関する問題
Fig. 4 Problems of calling methods.

の対応関係の候補を求めると，$\varphi_v = \{\langle \mathtt{gr}, \mathtt{g}\rangle, \langle \mathtt{pen}, \mathtt{p}\rangle\}$ となる．これらの候補に従ってクラス A において実際に置換を行った場合，2 つのラベルは一致する．変数 `color` については，比較するクラス A および B のどちらでも宣言されていないため置換対象としない．

3.4 メソッドの複写

親クラスと後継者クラスから求めた変更差異を単純に参照クラスに合成すると，オブジェクト指向の持つ継承の特性より，参照インスタンスを明に指定しないメソッド呼び出しに関して 2 つの問題が生じる．

これらの問題を説明するため，クラス間のメソッド呼び出しの様子を図 4 に示す．*Parent*, *Heir*, *Referenced*, *Generated* は，それぞれ図 1 における親クラス，後継者クラス，参照クラス，新規生成クラスを指す．`method{}` はメソッドの定義，`method();` はメソッドの呼び出しを表す．*Common* は *Parent* と *Referenced* の共通の祖先において最下位の（*Parent* あるいは *Referenced* に最も近い）クラスを指す．H は *Parent* と *Heir* の変更差異を集めた履歴を指す．いま，*Common* から *Parent* へ到達する経路上にあるクラスの集合（*Common* を除く）を R_p，*Common* から *Referenced* へ到達する経路上にあるクラスの集合（*Common* を除く）を R_r とおく．*Common* が *Parent* あるいは *Referenced* と同一であるとき，*Parent* あるいは *Referenced* は，R_p あるいは R_r から削除する．$CSET_p$ は R_p 内に存在するクラスの子孫クラスの集合，$CSET_r$ は R_r 内に存在するクラスの子孫クラスの集合である．$CSET_p$ や $CSET_r$ を単純に *Parent* あるいは *Referenced* の祖先クラスだけとせず，R_p あるいは R_r の子孫クラスの集合としたのは，呼び出されたメソッドが抽象メソッドで，対応する具象メソッドが抽象メソッドを含むクラスの子孫で定義されている可能性があるからである．

メソッド呼び出しに関する 2 つの問題を次に示す．

同一性に関する問題 変更差異 H と新規クラス *Generated* から呼び出される同名のメソッドが同一のものを指していないことがあり，これらのメソッドの機能が等しいとは限らない．このため，H に含まれる機能が新規クラスに正しく合成されず，H 内に含まれていた文と実際に *Generated* から呼び出されたメソッド内の文が矛盾することがある．たとえば，*Generated* におけるメソッド `method1();` の呼び出し先は，$CSET_r$ 内に存在するメソッド `method1{}` であり，H から呼び出される $CSET_p$ 内に存在するメソッド `method1{}` と実体が異なる．この場合，H に含まれる `method1{}` の機能が *Generated* に正しく合成されない．

参照可能性に関する問題 参照クラス *Referenced* から呼び出し不可能なメソッドが変更差異 H に含まれる場合，実行不可能なメソッドの呼び出しが新規クラス *Generated* に存在することになる．たとえば，$CSET_p$ 内に存在するメソッド `method2{}` が *Referenced* に対して非公開である場合，*Generated* はこのメソッドを呼び出すことはできない．

呼び出しメソッドの型が静的かつ一意に決定可能な場合を考える（型が動的に特定される場合については，6 章で考察する）．この場合，上記の問題を回避するためには，実際にメッセージを処理するメソッドの実体が必ず *Generated* に存在するように，*Parent*, *Heir*, *Referenced* から呼び出されるすべてのメソッドの実体を，新規クラス *Generated* に複写すればよい．なぜなら，実際にメッセージを処理するインスタンスは，呼び出されたメソッドの実体を含むクラスから生成したものであるからである．

しかしながら，メソッドを複写することは，新規生成クラスのソースコードの量の増加につながる．さらに，メソッドを複写すると，祖先クラスにおける複写対象

元メソッドへの変更が複写メソッドを含む新規生成クラスに伝搬しなくなる．このように，*Parent*, *Heir*, *Referenced* から呼び出されるすべてのメソッドを無制限に複写することは，新規生成クラスに対する理解あるいはクラス階層の管理という面で得策ではない．

そこで，本手法では，同名の呼び出しメソッドの実体が必ず等しくなる，かつ，新規生成クラス *Generated* から呼び出し可能なメソッドを，クラス階層木における *Parent* と *Referenced* の位置関係に基づいて決定しておき，メソッド複写の対象から除く．いま，図4において，$CSET_p$ あるいは $CSET_r$ のどちらにも含まれないクラスを考える．これらのクラスは，*Common* の祖先であるクラスと祖先でないクラスに分けられる．*Common* の祖先クラスで定義されている公開（public）メソッドおよび保護（protected）メソッドは，*Generated* から呼び出し可能である．さらに，これらのメソッドが *Parent*, *Heir*, *Generated* から呼び出された場合，そのメソッドの実体は必ず同じになる．すなわち，呼び出しメソッドの同一性および参照可能性は保証される．ここで，公開メソッドとはどのクラスからでも参照可能なメソッドを，保護メソッドとはその子孫クラスからのみ参照可能なメソッドを指す．*Common* の祖先でないクラスについては，そのクラスで定義されているメソッドの呼び出し時に必ず参照インスタンス名が指定されるため，同一性および参照可能性に関する問題を引き起こさない．*Parent*, *Heir*, あるいは *Referenced* が非公開メソッドの呼び出しを含む場合には，呼び出しメソッドが *Generated* から呼び出し不可能なので必ず複写する必要がある．非公開メソッドとはそのメソッドが定義されるクラスからのみ参照可能なメソッドを指す．

以上より，$CSET_p$ あるいは $CSET_r$ のどちらかに存在するクラスで定義されているメソッドに関してのみ，呼び出しメソッドの同一性と参照可能性を保証し，できるだけ複写数が小さくなるように複写メソッドを決定すればよい．本手法では，変更履歴 H を参照クラス *Referenced* に合成する際，$CSET_p$ および $CSET_r$ を決定し，次に示すメソッドの呼び出しに対してのみ，メソッド複写を行う．

複写操作 C1 $CSET_p$ に含まれるクラスで定義されており，参照インスタンスを指定せずに呼び出された公開メソッド．

複写操作 C2 $CSET_p$ に含まれるクラスで定義されている保護メソッド．

複写操作 C3 *Parent*, *Heir*, あるいは *Referenced* で定義されている非公開メソッド．

ただし，同名のメソッドが変更差異の合成により新規に生成されている場合，そのメソッドの複写は行わない（新規生成メソッドでない場合，複写を行う）．複写対象のメソッドが抽象メソッドの場合，そのメソッドに静的に束縛される実体を複写する．また，複写されたメソッドが，さらに上記のメソッドに対する呼び出しを含む場合，メソッド複写を再帰的に行う．すべての複写メソッドは，他のクラスから呼び出されることがないため，非公開とする．さらに，メソッド名の衝突をさけるため，複写メソッドの名前は，もとのメソッド名に接頭語（たとえば，"copied_"）を付けたものとする．複写メソッドで利用されるインスタンス変数が新規クラスにおいて宣言されていない場合，その変数の宣言部も同時に複写する．図4では，$CSET_p$ に含まれるクラスのメソッド `method1{}` と `method2{}` が，*Generated* に複写される．

4. 新規クラス生成手法における各手続き

本章では，新規クラスを生成する3つの手続きについて詳細を述べる．

4.1 変更差異特定部

変更差異特定部は，継承において開発者がメソッドの追加および再定義を行った際，**図5** に示す動作を行う．

親クラス *Parent* と後継者クラス *Heir* において，インタフェースが一致する2つのメソッド（再定義メソッドと再定義されたメソッド）`method` から，スライス S_{Pi} と S_{Qj} を抽出する．次に，S_{Pi} および S_{Qj} の区間限定補スライス[3] S^*_{Pi} および S^*_{Qj} の ClDG に関する同形写像比較を行う．区間限定補スライス（以後，単に補スライスと呼ぶ）とは，もとのプログラムにおける抽出対象区間の節点からスライス S_{Pi} と S_{Qj} に

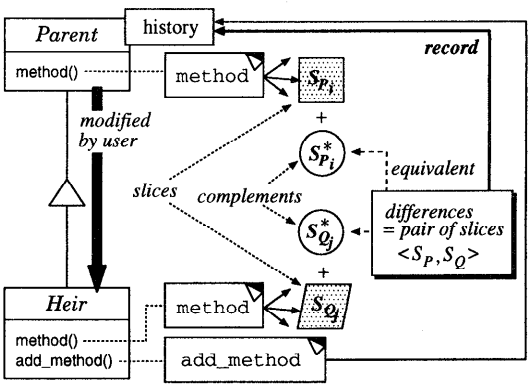


図5 変更差異の特定

Fig. 5 Procedure for specifying the differences.

含まれる節点を取り除いたものであり，スライスおよび区間外の全節点を1つの節点に集めることで作成する．比較する2つのClDGにおいて変更部分がそれぞれのスライス S_{Pi} および S_{Qj} に閉じ込められた場合，無変更部分を表す補スライス S^*_{Pi} と S^*_{Qj} が等価になる．よって，同形写像比較により等価な補スライスを見つけることで，もとのメソッドのClDGの一部を変更部分として特定可能である．最後に，変更部分を内包するスライス S_P と S_Q の対 $\langle S_P, S_Q \rangle$ を，それらに含まれる節点の対応関係とともに親クラスの履歴に蓄積する．節点の対応関係は，スライス S_P と S_Q の部分スライス（部分グラフ）どうしの同形写像比較により得られる．メソッド **add_method** は，*Heir* だけに含まれるため，メソッド全体が変更部分となる．

4.2 類似クラス探索部

類似クラス探索部は，参照クラスに開発者の要求する呼び出しメソッドの定義が存在しないとき，参照クラスと合成可能な変更履歴を持つ類似クラスの探索を行う．

本手法では，メソッドのインタフェースにより開発者の要求を定義しているため，要求メソッドとインタフェースが一致するメソッドを履歴に含むクラスを類似クラスの候補とする．しかしながら，インタフェースの一致だけに基づき類似クラスを特定すると，合成対象の候補が大量に見つかる恐れがある．さらに，これらの候補のほとんどは，要求メソッドの生成に役立たない可能性が高い．なぜなら，メソッドの合成を成功させるためには，参照クラスと類似クラスが同様の変更を受け入れる，つまり同様のプログラム構造（ClDG）を含む同名のメソッドを持たなければならないからである．

そこで，本手法では，メソッドの合成の試行回数を減少させるため，クラス階層における参照クラスとの位置関係を利用して類似クラスの探索範囲を限定する．いま，クラス階層木内の2つのクラスに対して，共通の祖先クラスまでの継承段数（階層数）が少ないほど継承する共通な性質は多い．よって，これらのクラスにおいて，同じインタフェースを持つメソッドが再定義されている可能性，さらに，それらのメソッドが同様の変更を受ける可能性が高いと判断できる．このことより，クラス階層木において，参照クラスと近い位置に存在するクラスほど，参照クラスと合成可能な変更差異を履歴に含む可能性は高いといえる．

類似クラス探索部の動作を**図6**に示す．参照クラスは *Referenced*，開発者が呼び出した要求メソッドのインタフェースは **requirement(type)** である．類似

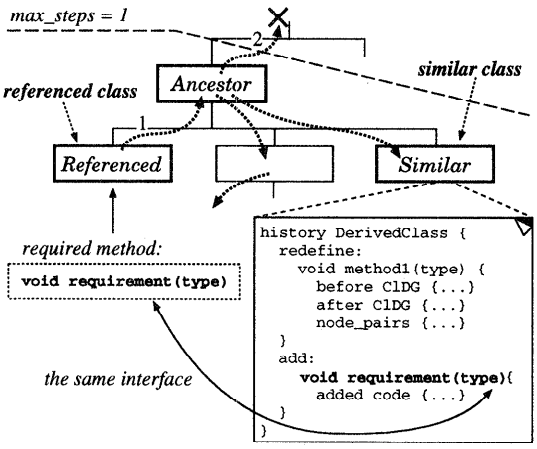


図6 類似クラスの特定
Fig. 6 Procedure for finding similar classes.

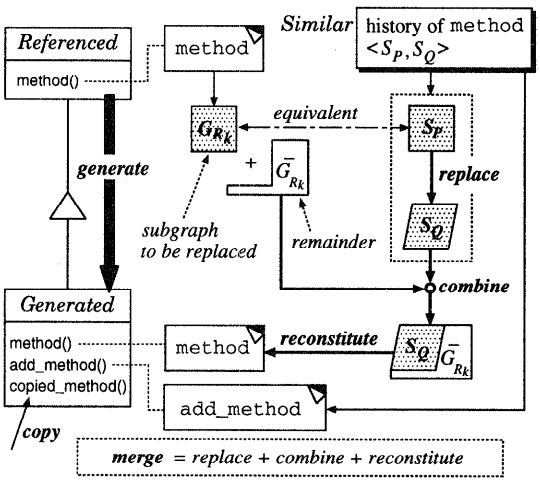


図7 変更差異の合成
Fig. 7 Procedure for integrating the differences.

クラス探索部は，クラス階層における継承関係を *Referenced* から1段階ずつ上がり，到達したクラス（1段階上位の場合，*Ancestor*）の子孫を類似クラスの候補とする．このとき，クラス階層木ごとに探索段数をあらかじめ設定しておく．これらの候補に対して，変更履歴が要求メソッドと同じインタフェースを持つメソッドを含むかどうかを検査することで，類似クラス *Similar* を特定する．図6では，探索段数の最大値を $max\_steps = 1$ と設定したので，*Ancestor* より上位のクラスに対しては探索を行わない．

4.3 変更差異合成部

変更差異合成部は，類似クラスが特定された後，**図7**に示す動作を行う．

変更差異の合成は，参照クラス *Referenced* において，類似クラス *Similar* の履歴に含まれるメソッドと

インタフェースが一致するメソッドに対して行われる．*Referenced* の合成対象メソッド **method** の ClDG において，*Similar* の履歴内の変更前スライス S_P に等価な部分グラフ G_{Rk} を抽出する．この部分グラフを置換対象グラフと呼ぶ．置換対象グラフは，S_P の抽出元メソッドのインタフェースと S_P の各節点のラベルに基づいて抜き出し，S_P との同形写像比較により等価性を検査する．次に，履歴内の節点の対応関係を用いて，置換対象グラフ G_{Rk} を変更前スライス S_P に置換し，さらに，S_P を変更後スライス S_Q に置換する．置換後の S_Q を，参照クラス内の合成対象メソッドにおける無置換部分 $\bar{G}_R$ と合成する．合成後の ClDG（$S_Q + \bar{G}_R$）は，ソースコードに再構成され，新規クラス *Generated* のメソッド **method** となる．また，合成対象メソッドが *Referenced* に存在しない **add_method** については，そのコードがそのまま *Generated* に追加される．最後に，クラス階層木における *Referenced* および *Similar* の位置に基づき，3.4 節で示したメソッド複写操作 C1，C2，C3 を適用し，**copied_method** が複写される．

5. 新規クラス自動生成の例

新規クラスを自動生成する例を，Java プログラムを用いて示す．本例で用いるクラスの関係を図 **8** に示す．

(1) 変更差異の特定

開発者が，継承により，親クラス DisplayArea にスクロールバーの機能を追加して，後継者クラス DisplayAreaScrollbar を作成した場合を考える．DisplayArea のソースコードを図 **9**，DisplayAreaScrollbar のソースコードを図 **10** に示す．Java パッケージ内の Graphics クラスにおいて定義されるメソッド（**setColor**, **drawLine**, **drawString**）と Component クラスにおいて定義されるメソッド（**size**）内部の依存関係は解析済みであるとする．本論文では，メソッド内部のデータ依存関係が分かるように，各節点における定義および参照変数を図 9，図 10 に示す．**line[].X1/Y1/X2/Y2** は，**line[].X1**, **line[].Y1**, **line[].X2**, **line[].Y2** の短縮形である．また，スクリーンへの書き込みを扱うため，描画領域を指す仮想的な変数 **gImage**, **gImage2** を導入した．

DisplayArea と DisplayAreaScrollbar のメソッド **paint** の ClDG と変更差異を図 **11** に示す．メソッド引数およびインスタンス変数に関する入出力節点は省略した．変更差異特定部は，これら 2 つの ClDG から，それぞれスライス S_P $(= \hat{S}(\text{p7}, \text{p11}, \{\textbf{gImage}\}))$

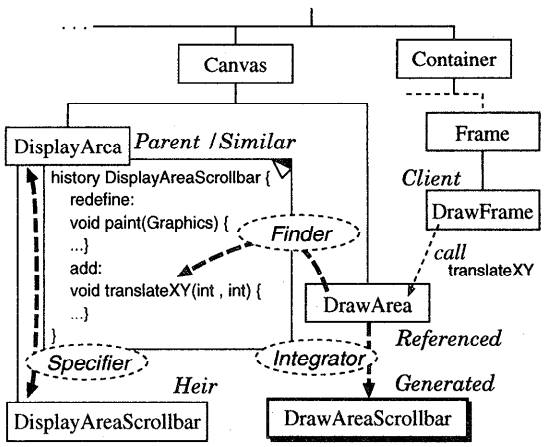


図 8 新規クラス生成の例
Fig. 8 Example of generating new classes.

```
p1 class DisplayArea extends Canvas {
     Line line[]; int lineN;
p2   DisplayArea() {...}
p3   public void paint(Graphics g) {
p4     g.setColor(Color.blue);
p5     int hmax = size().width;
p6     int vmax = size().height;
p7     boolean outRange = false;
p8     for (int l = 0;
p9     l < lineN;
p10    l++) {
p11      g.drawLine(line[l].x1,line[l].y1,
                    line[l].x2,line[l].y2);
p12      if (line[l].x1 < 0 || line[l].x1 > hmax
           || line[l].x2 < 0 || line[l].x2 > hmax
           || line[l].y1 < 0 || line[l].y1 > vmax
           || line[l].y2 < 0 || line[l].y2 > vmax)
p13        outRange = true;
       }
p14    if (outRange)
p15      g.drawString("out of range",5,size().height);
     }
p16  public void redraw() {...}
   }
```

Node	defined variables	used variables
p4	g.color	g
p5	hmax	size().width
p6	vmax	size().height
p7	outRange	
p8	l	
p9		l, lineN
p10	l	l
p11	gImage	gImage, g, g.color, l, line[].x1/y1/x2/y2
p12		l, hmax, vmax, line[].x1/y1/x2/y2
p13	outRange	
p14		outRange
p15	gImage2	g, g.color, size().height

gImage,gImage2 : virtual variables for storing images

図 9 親クラスのソースコード
Fig. 9 Source code of the parent.

および S_Q $(= \hat{S}(\text{q11}, \text{q19}, \{\textbf{gImage}\}))$ を抽出し，それらの補スライス S_P^* と S_Q^* を作成する．いま，S_P と区間外の節点を節点 N_P，S_Q と区間外の節点を N_Q に集約し，S_P^* および S_Q^* 内の節点とそれらの節点どうしを結ぶ矢印を残した 2 つの ClDG（図 11 における右側の ClDG）は，同形写像比較において等価である．よって，網かけ節点で表現されたスライスの対 $\langle S_P, S_Q \rangle$ が DisplayArea と DisplayAreaScrollbar のメソッド **paint** の変更差異となる．$\langle S_P, S_Q \rangle$ におけ

```
q1 class DisplayAreaScrollbar extends DisplayArea {
     int transX, transY;
     int hscmax, vscmax;
q2   DisplayAreaScrollbar(int h, int v) {...}
q3   public void translateXY(int x, int y) {
q4     transX = x;
q5     transY = y;
q6     redraw();
     }
q7   public void paint(Graphics g) {
q8     g.setColor(Color.blue);
q9     int hmax = size().width+hscmax;
q10    int vmax = size().width+vscmax;
q11    boolean outRange = false;
q12    for (int l = 0;
q13    l < lineN;
q14    l++) {
q15      g.drawLine(line[l].x1-transX,line.y1-transY,
                    line[l].x2-transX,line.y2-transY);
q16      if (line[l].x1 < 0 || line[l].x2 > hmax
          || line[l].x2 < 0 || line[l].x2 > hmax
          || line[l].y1 < 0 || line[l].y1 > vmax
          || line[l].y2 < 0 || line[l].y2 > vmax)
q17        outRange = true;
       }
q18    g.color(Color.black);
q19    g.drawString("X="+transX+";Y="+transY, 5, 15);
q20    if (outRange)
q21      g.drawString("out of range",5,size().height);
     }
   }
```

Node	defined variables	used variables
q8	g.color	g
q9	hmax	size().width, hscmax
q10	vmax	size().height, vscmax
q11	outRange	
q12	l	
q13		l, lineN
q14	l	l
q15	gImage	gImage, g, g.color, l, transX, transY, line[].x1/y1/x2/y2
q16		l, hmax, vmax, line[].x1/y1/x2/y2
q17	outRange	
q18	g.color	g
q19	gImage	gImage, g, g,color, transX, transY
q20		outRange
q21	gImage2	g, g.color, size().height

図 10 後継者クラスのソースコード
Fig. 10 Source code of the heir.

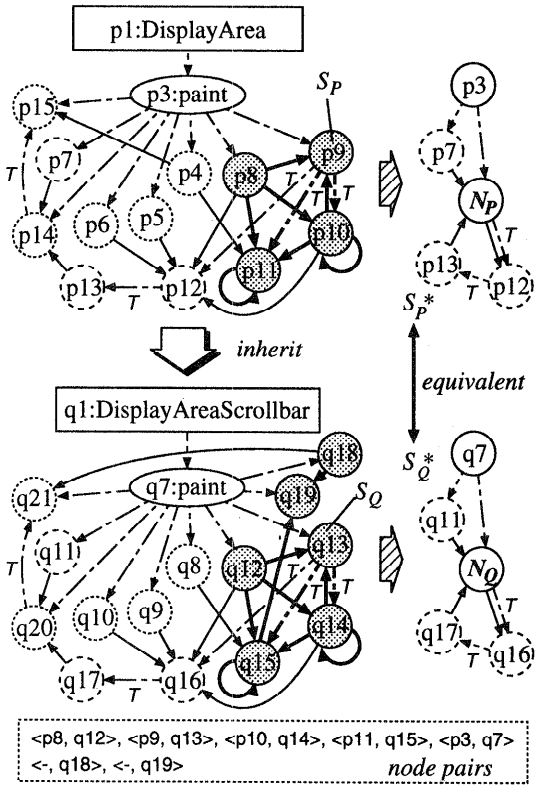


図 11 親クラスと後継者クラスにおけるメソッド paint の ClDG と変更差異 S_P と S_Q
Fig. 11 ClDGs of method paint in the parent and the heir, and different fragments S_P and S_Q.

る節点の対応関係を図 11 の ClDG の下に示す．– は，対応する節点が存在しないことを表す．

(2) 類似クラスの探索

参照クラス DrawArea で定義されていないメソッド **translateXY** に対する呼び出し文を，開発者がクラス DrawFrame に記述した場合を考える．DrawFrame と DrawArea のソースコードを図 **12** に示す．このような状況は，開発者が，他のクラス（たとえば，DisplayAreaScrollbar）に存在する **translateXY** の名前と機能から，DrawArea の派生クラスに **translateXY** を追加する必要があると考えた場合に生じる．この場合，開発者は，**translateXY** の存在を想定して，このメソッドを呼び出す文を DrawFrame に記述する．

本例では，文 **c12** におけるメソッド **translateXY** の呼び出しが失敗し，類似クラスの探索が行われる．類似クラス探索部は，図 8 に示すように，呼び出しメソッド **translateXY** とインタフェースが一致するメソッドを持つ変更差異を DisplayArea の履歴内に見つける．

(3) 変更差異の合成

変更差異合成部では，発見した変更差異を参照クラス DrawArea に合成することで，最初にメソッド **translateXY** を生成する．さらに，変更差異に含まれる他のメソッドに関しても合成を適用し，メソッド **paint** を生成する．**translateXY** については，合成対象となる同一インタフェースを持つメソッドが DrawArea 内に存在しないので，メソッドのコード全体を新規クラスに追加する．一方，**paint** については，変更差異に含まれる **paint** のコードと DrawArea に含まれる **paint** のコードを合成することで生成する．

DrawArea における合成対象メソッド **paint** と本手法により生成した新規メソッド **paint** の ClDG を図 **13** に示す．本例では，DrawArea 内の変数名 **n** を **1** に置換することで，同形写像比較において，G_R と類似クラス DisplayArea における変更前スライス S_P が等価になる．よって，網かけ節点で表現された G_R が置換対象グラフとなる．次に，図 13 に示す節点の対応関係を用いて，G_R 内の節点を $r \to p$, $p \to q$ の順に置換する．さらに，S_Q に含まれていた他の節

```
c1 class DrawFrame extends Frame {
     ControlPanel cp; Scrollbar horz, vert;
     DrawArea area;
c2   DrawFrame() {
c3     cp = new ControlPanel();
c4     area = new DrawArea(cp);
c5     horz = new Scrollbar();
c6     vert = new Scrollbar();
c7     showFrame();
     }
c8   public boolean handleEvent(Event e) {
c9     if (e.target instanceof Scrollbar) {
c10      int horzValue = horz.getValue();
c11      int vertValue = vert.getValue();
c12      area.translateXY(horzValue, vertValue);
c13      return true;
       }
c14    return super.handleEvent(e);
     }
     ...
   }
```

method call

```
r1 class DrawArea extends Canvas {
     Line line[]; int lineN;
     ConrolPanel cp;
r2   DrawArea(ControlPanel c) {...}
r3   public void paint(Graphics g) {
r4     if (lineN > 0) {
r5       for (int n = 0;
r6       n < lineN;
r7       n++) {
r8         g.setColor(line[n].color);
r9         g.drawLine(line[n].x1,line[1].y1,
                      line[n].x2,line[1].y2);
         }
       } else {
r10      g.setColor(Color.blue);
r11      msgX = size().width/2-40;
r12      msgY = size().height/2-5;
r13      g.drawString("Please draw.",msgX,msgY);
       }
     }
r14  public boolean handleEvent(Event e) {
r15    if (e.id == Event.MOUSE_DOWN) {
r16      storeLineData(e.x, e.y);
r17      repaint();
       ...
     }
     ...
   }
```

unimplemented (no method)

Node	defined variables	used variables
r4		lineN
r5	n	
r6		n, lineN
r7	n	n
r8	g.color	g, n, line[].color
r9	gImage	gImage, g, g.color, n, line[].x1/y1/x2/y2
r10	g.color	g
r11	msgX	size().width
r12	msgY	size().height
r13	gImage2	g, g.color, msgX, msgY

図 12 呼び出し側クラスと参照クラスのソースコード
Fig. 12 Source code of the referring class and the referenced class.

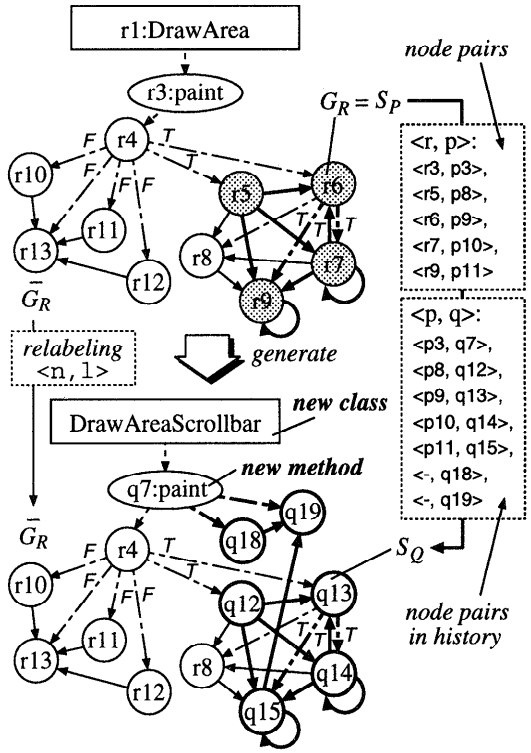


図 13 参照クラスと新規クラスのメソッド paint の ClDG
Fig. 13 ClDGs of the referenced and generated class.

点および矢印を追加し，S_Q と置換対象グラフ G_R の残り $\bar{G}_R$ を合成する．このようにして，新規クラス DrawAreaScrollbar のメソッド **paint** の ClDG が生成される．本手法により自動生成された新規クラス DrawAreaScrollbar のソースコードを**図 14** に示す．

メソッド **paint**（13–31 行目）は，図 13 の下段の ClDG をソースコードに変換したものである．メソッド **translateXY**（8–12 行目）と，このメソッドで用いられるインスタンス変数 **transX**，**transY** の宣言文（2 行目）が，類似クラス DispalyArea の変更履歴から追加された．また，参照可能性を保証するために，メソッド **copied_redraw**（32–36 行目）が DispalyArea から複写（複写操作 C1）され，このメソッドに対する呼び出しが **copied_redraw()**（11 行目）のように書き換えられる．呼び出し文（35 行目）に対応するメソッド **paint** の実体は，同名のメソッドが直前に生成されているため複写されない．

(4) 生成された新規クラスに関する考察

新規クラス DrawAreaScrollbar のソースコードを見ると，開発者の要求したメソッド **translateXY(int, int)** が存在することが確認できる．また，DisplayAreaScrollbar において座標変換を行うように再定義したメソッド **paint** の変更部分（19–21 行目，節点 q15）が，DrawAreaScrollbar の新規メソッド **paint** に反映されているのが分かる．さらに，参照クラス DrawArea のメソッド **paint** において個々の線に色を付ける機能（18 行目，DrawArea の節点 r8）と，類似クラス DisplayAreaScrollbar のメソッド **paint** におけるスクロールバーの現在座標を表示する機能（29–30 行目，DisplayAreaScrollbar の節点 q18，q19）が保存されていることも確認できる．

本手法では，DrawAreaScrollbar の名前（1 行目）およびコンストラクタは（3–7 行目）は自動生成されないため，これらのソースコードは開発者が記述する必

```
1:    class DrawAreaScrollbar extends DrawArea {
2:      int transX, transY;  ——————————  added
3:      DrawAreaScrollbar(ControlPanel c) {
4:        super(c);
5:        transX = 0;
6:        transY = 0;                    implemented by user
7:      }
8: q3   public void translateXY(int x, int y){
9: q4     transX = x;
10: q5    transY = y;                    added
11: q6    copied_redraw();
12:     }
13: q7  public void paint(Graphics g){
14: r4    if (lineN > 0) {
15: q12     for (int l = 0;
16: q13     l < lineN;                   merged
17: q14     l++) {
18: r8        g.setColor(line[l].color);
19: q15       g.drawLine(
20:             line[l].x1-transX,line[l].y1-transY,
21:             line[l].x2-transX,line[l].y2-transY);
22:         }
23:       } else {
24: r10     g.setColor(Color.blue);
25: r11     msgX = size().width/2 - 40;
26: r12     msgY = size().height/2 - 5;
27: r13     g.drawString("Please draw.",msgX,msgY);
28:       }
29: q18   g.setColor(Color.black);
30: q19   g.drawString("X="+transX+";Y="+transY,5,15);
31:     }
32: p16 private void copied_redraw(){
33: p16   Graphics g = getGraphics();
34: p16   g.clearRect(0,0,size().width,size().height);
35: p16   paint(g);                      copied
36:     }
37:   }
```

図 14 新規クラスのソースコード
Fig. 14 Source code of the generated class.

要がある．また，本例で生成した DrawAreaScrollbar を実際に実行すると，図形描画時にマウスで指定した座標と実際に図形が描画される座標にずれがあることに気づく．これは，DrawArea のメソッド **handleEvent**（節点 r14–r17）が，そのまま DrawAreaScrollbar に継承されたために生じる．座標のずれを取り除くには，節点 r16 の **e.x** と **e.y** に対して，メソッド **paint** に適用した座標変換と同様の変更（**e.x** → **e.x+transX**, **e.y** → **e.y+transY**）を適用し，**handleEvent** を再定義する必要がある．

このように，本例では，既存クラスおよび変更履歴内の依存関係を保存，かつ，合成により追加した機能が既存クラスの機能に無矛盾のままで，開発者の要求するメソッドを持つ新規クラスの生成に成功した．この新規クラスは，コンストラクタの追加という少ない変更で実行可能であった．また，要求によっては，一部のメソッドを再定義する必要があることも示した．

本例に関して，より完全な DrawAreaScrollbar クラスの自動生成を実現するためには，コンストラクタどうしの一致あるいは類似を判断可能なようにコンストラクタ名を置換する仕組みや，生成メソッドに関連を持つ複数のメソッドに同様の変更を伝搬させる仕組みが必要である．ただし，単純にコンストラクタ名の文字列置換を行うと，大部分のコンストラクタどうしが一致であると判断されてしまう．また，関連を持つすべてのメソッドに無制限に変更を伝搬させると，生成したクラスが要求を満たさなくなる恐れがある．これらの問題を回避するには，メソッドあるいはコンストラクラのインタフェースだけでは不十分で，それらの内部構造および機能に関する情報を正確に扱う手法が必要である．

6. 議　　論

本章では，関連研究との比較により本洗練手法の有効性を述べ，さらに，今後の課題について考察する．

6.1 関連研究との比較

我々の目的と同様に，コンポーネントウェア[14]や部品組立て型ソフトウェア開発（Component-Based Software Development）[15]の目的は，開発者（エンドユーザ）が部品を組み合わせるだけで，要求プログラムを作成できるようにすることである．これらの研究では，組合せ可能な部品の形態やそれらの部品を利用するアーキテクチャの提案に主眼がおかれ，開発者の要求する部品があらかじめ構築できることを前提としている．しかしながら，開発者のさまざまな要求を完全に予測し，無変更で再利用可能なクラスや部品をあらかじめ構築しておくことは難しい．

オブジェクト指向プログラミングに限定した場合でも，新規クラスを静的に作成するのではなく，オブジェクトコンポジション[16]により既存オブジェクトの機能を動的に合成することで，開発者の要求する機能を実現する方法がある．また，多重継承（multiple inheritance）により既存クラスを拡張する方法がある．しかしながら，これらの方法では，合成対象のオブジェクトや派生元のクラスが適切に設計されている必要があり，開発者の要求する機能が既存クラス（オブジェクト）の機能の組合せだけで実現できる場合にのみ有効である．特に，オブジェクトコンポジションでは，既存クラス（オブジェクト）の機能の一部を洗練するような機能変更は実現できない．また，多重継承においても，既存クラスの組合せだけで開発者の要求が満たせない場合，メソッドの再定義が必要である．

5 章の例では，クラス DisplayArea のメソッド **paint** とクラス DrawArea のメソッド **paint** において，描画する図形（線）に対する色の付け方が異なり，色を指定する文と実際に図形を描画する文が密接に関連している．このような場合，DisplayArea にスクロールバーの機能を追加する際に用いた合成オブジェクトを，単純に DrawArea に合成することはできない．よって，オブジェクトコンポジションでは，開発者の要求する機能を備えたクラス DrawAreaScrollbar の実現

は不可能である．また，多重継承を用いた場合でも，DrawAreaScrollbar において，メソッド **paint** の再定義が必要となる．

我々の提案するクラス生成手法は，継承におけるコード変更を自動化し，開発者の要求するメソッドを持つ再利用可能なクラスを生成する．これにより，オブジェクトコンポジションや多重継承に比べて，次に示す利点を持つ．

(1) 過去のクラス継承における変更を模倣することで新規クラスを生成するため，ライブラリに存在しないクラスを要求した場合でも，その要求を満たすメソッドを持つ再利用可能なクラスを提供できる可能性がある．さらに，同様の変更を適用した多種多様なクラスが，既存クラスから生成されるため，予測できない要求に対して数多くのクラスをライブラリに用意しておく必要がない．

(2) 区間限定スライシングを用いて特定される変更の差分はメソッドより小さいため，微妙に異なる機能を持つさまざまな新規メソッドを生成できる．よって，既存のクラスに対してメソッドや変数を追加するだけでは作成できないクラス，つまり機能の合成だけでは実現できない派生クラスを生成可能である．

(3) プログラム・スライシングによる機能分割手法と PDG の同形写像比較による等価機能の特定手法に基づくメソッド合成アルゴリズムにより，メソッドを合成する．よって，既存クラスの機能の一部を保存し，合成により付加した変更差異の機能が既存クラスのメソッドの機能に無矛盾であることを保証する．

一般に，クラス継承とオブジェクトコンポジションは，相反する再利用技法ではなく，協調して利用可能である[16]．よって，本生成手法は，オブジェクトコンポジションによる方法と置換されるものではない．たとえば，合成だけで実現可能な機能をオブジェクトコンポジションで，洗練により実現可能な機能を持つメソッドを本手法で生成することが考えられる．また，最初に本手法により開発者の要求を満たす可能性の高い新規クラスを生成し，このクラスを修正あるいは洗練することにより，機能合成において利用可能なクラス（オブジェクト）を作成することが考えられる．

6.2 課　　題

(1) メソッドの整合性

本生成手法では，開発者の要求を呼び出しメソッドのインタフェースで表現し，インタフェースの一致により類似クラスを特定する．さらに，新規クラスを生成する際には，類似クラスの変更履歴と参照クラスにおいて，インタフェースが一致するメソッドどうしを合成対象とする．これらの動作は，クラスライブラリにおいてメソッドの整合性が保たれるべき，つまり，類似の機能を持つメソッドは同じ名前およびシグニチャを持つべきである[17]という前提に基づいている．しかしながら，実際のプログラミングにおいて，このようなメソッドの整合性は必ずしも保証できない．

本手法では，インタフェースの綴りを比較することにより，類似メソッドの探索あるいは合成対象メソッドの特定を実現している．よって，上記の前提が成立していないライブラリにおいても，新規メソッドの生成は可能である．しかし，この前提が成立しているかどうかは，本手法の能力に大きな影響を与える．たとえば，同じ機能を持つメソッドが異なるインタフェースを持つ場合，類似クラスの発見が困難になる．また，異なる機能を持つメソッドが同じインタフェースを持つ場合，生成したメソッドが開発者の要求する機能を満たさない可能性は高くなる．

このような問題を解決するため，メソッド・インタフェースの綴りでなく，メソッドの持つ機能の等価性あるいは類似性を判定可能とする手法を検討している．たとえば，インタフェースの緩やかな一致（relaxed match）を判定できる signature matching 手法[18]を，類似クラス探索手続きやメソッド合成手続きに組み込むことで，変更を適用するメソッドの範囲を広げることを考えている．さらに，生成したメソッドが開発者の要求する機能を満たすことを検証するため，クラス内の各メソッドに表明（たとえば，Eiffel 言語[1]における，前提条件，終了条件，不変表明）を付け，メソッド合成と同時に表明の合成を行う手法を考察中である．

(2) 多相性とメソッド複写

多相性（polymorphism）を有するメソッドに対して，3.4 節で述べたメソッド複写操作を単純に適用することはできない．なぜなら，動的束縛（dynamic binding）により実行時に呼び出される可能性を持つ複数のメソッドを同一クラス内に複写することはできないからである．このため，本手法では，複写対象のメソッドを含むインスタンスに対して，その生成元クラスが静的かつ一意に決定可能であることを前提とし，多相性を有するメソッドが複写対象となるとき新規クラスの生成を行わない．複写対象でないメソッドが多相性を有する場合は，新規クラスの生成が行われる．

本手法において，多相性を扱う方法の 1 つとして，多相性を有するメソッドを直接複写するのではなく，

複写メソッドの多相性を保存したままで，束縛される可能性のある関連クラス全体を複写することが考えられる．現在，クラス階層スライシング[19]を用いて，複写対象のクラスを特定する手法を実現中である．

(3) 本手法の実用性

本論文では，本手法によりクラスの自動生成が成功する一例を示しただけである．実際のプログラミングにおいて，どの程度生成に成功するのか，さらに，新規生成クラスが開発者の負担をどの程度軽減可能かを確認するためには，実在するクラスライブラリに対して本手法を適用し，その適用結果を評価することが必須である．

我々は，本手法により生成される新規クラスの有用性をオブジェクト指向メトリクス[20]の視点から評価する予定である．たとえば，新規クラスにおけるメソッドおよび変数の数，追加再定義，継承メソッドの数から，本手法の有効性を確認することが考えられる．さらに，これらの評価基準を本生成手法における類似クラスの特定，あるいは，新規生成クラスの候補から妥当なクラスを選択する際の指標に用いることを考えている．

(4) クラス生成ツールの実現

我々は，手続き型言語 Pascal を簡単化した言語で記述されたプログラムに対して，本手法と同様の仕組みでソースコードを生成するツールをすでに開発した[3),21]．現在，このツールをオブジェクト指向プログラムに適用可能となるように再構築している．現時点での目的は，本生成手法を実現するアルゴリズムの確かさを立証することである．

本手法の計算量を見積もる際に参考となる情報として，手続き呼び出しを含むプログラムを合成するアルゴリズムの計算量が文献 22) に示されている．これによると，SDG (System Dependence Graph)[8] からソースコードを再構成する部分を除いて，合成アルゴリズムの計算コストは SDG を構築するコストに対して多項式時間で抑えられる（ソースコード再構成部分については，NP 完全となることが指摘されている[2)]）．

オブジェクト指向プログラムに対する依存解析は，手続き型プログラムに比べて困難であるため，スライシングおよびグラフの同形写像比較の計算量は多くなることが予測される．さらに，本手法を実現する現時点でのアルゴリズムは，変更差異の特定あるいは合成を，対象となるメソッドから抽出したスライスおよび部分グラフのすべての組合せに対して行うため，その計算量はかなり多い．よって，本手法を実現するツールを構築する際には，計算量に関するアルゴリズムの改良が必要となる．特に，このツールは，プログラムの記述とクラスの生成を繰り返す対話的なプログラミング開発環境で用いられることが多いと考えられ，ツールの性能に関する検討は重要である．

7. おわりに

オブジェクト指向プログラミングにおいて，開発者が容易にプログラムを作成できるようにするためには，少ない変更で再利用可能なクラスを提供する必要がある．本論文では，開発者が再利用している参照クラスで呼び出しメソッドが定義されていない場合，参照クラスのメソッドとクラス変更履歴から，開発者が要求するメソッドを持つ新規クラスを自動生成する手法を提案した．さらに，本手法を実現する，変更差異特定手続き，類似クラス探索手続き，変更差異合成手続きの詳細を述べ，実際にこれらの手続きを用いて新規クラスが生成できることを示した．本論文で取り上げた例では，開発者の要求するメソッドを持つ新規クラスが得られ，このクラスはコンストラクタの追加だけで実行可能であった．このように，本手法により，継承におけるメソッドや変数の追加および再定義という開発者の負担軽減が期待できる．現在，6 章で述べた課題について考察を行っている．

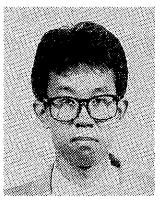
謝辞 ご指導ご討論いただきました NTT 情報流通プラットホーム研究所後藤厚宏リーダーに深く感謝します．

参考文献

1) Meyer, B.: *Object-Oriented Software Construction*, Prentice Hall (1988). 二木厚吉（監訳）：オブジェクト指向入門，アスキー出版局 (1990).
2) Horwitz, S., Prins, J. and Reps, T.: Integrating Noninterfering Versions of Programs, *ACM Trans. Prog. Lang. Syst.*, Vol.11, No.3, pp.345–387 (1989).
3) 丸山勝久，島 健一：ソースコード再利用における能動的部品変化メカニズム，情報処理学会論文誌，Vol.37, No.12, pp.2334–2351 (1996).
4) 丸山勝久，高橋直久：区間設定可能なプログラムスライシングを用いたソフトウェア部品の作成，情報処理学会論文誌，Vol.37, No.4, pp.520–535 (1996).
5) Ferrante, J., Ottenstein, K.J. and Warren, J.D.: The Program Dependence Graph and Its Use in Optimization, *ACM Trans. Prog. Lang. Syst.*, Vol.9, No.3, pp.319–349 (1987).
6) Gosling, J., Joy, B. and Steele, G.: *The Java Language Specification*, Addison-Wesley

(1996).
7) Rothermel, G. and Harrold, M.J.: Selecting Regresion Tests for Object-Oriented Software, *Proc. ICSM*, pp.14–25 (1994).
8) Horwitz, S., Ball, T. and Binkley, D.: Interprocedural Slicing Using Dependence Graphs, *ACM Trans. Prog. Lang. Syst.*, Vol.12, No.1, pp.26–60 (1990).
9) Weiser, M.: Program Slicing, *IEEE Trans. Softw. Eng.*, Vol.10, No.4, pp.352–357 (1984).
10) Ottenstein, K.J. and Ottenstein, L.M.: The Program Dependence Graph in a Software Development Environment, *ACM SIGPLAN Notices*, Vol.19, No.5, pp.177–184 (1984).
11) Larsen, L. and Harrold, M.J.: Slicing Object-Oriented Software, *Proc. ICSE*, pp.495–505 (1996).
12) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley (1986).
13) Horwitz, S. and Reps, T.: Efficient Comparison of Program Slices, *Acta Inf.*, Vol.28, pp.713–732 (1991).
14) 青山幹雄：コンポーネントウェア：部品組立て型ソフトウェア開発技術，情報処理，Vol.37, No.1, pp.71–79 (1996).
15) Brown, A.W. (Ed.): *Component-Based Software Engineering*, IEEE Computer Society Press (1996).
16) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (1994). 本位田真一，本田和樹（監訳）：デザインパターン，ソフトバンク (1995).
17) Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W.: *Object-Oriented Modeling and Design*, Prentice Hall (1991). 羽生田栄一（監訳）：オブジェクト指向方法論 OMT，トッパン (1992).
18) Zaremski, A.M. and Wing, J.M.: Signature Matching: A Tool for Using Software Libraries, *ACM Trans. Softw. Eng. and Meth.*, Vol.4, No.2, pp.146–170 (1995).
19) Tip, F., Chio, J.D. and Ramalingam, G.: Slicing Class Hierarchies in C++, *Proc. OOPSLA*, pp.179–197 (1996).
20) Lorenz, M. and Kidd, J.: *Object-Oriented Software Metrics*, Prentice Hall (1994). 宇治邦昭（監訳）：オブジェクト指向ソフトウェアメトリクス，トッパン (1995).
21) Maruyama, K. and Shima, K.: A Mechanism for Automatically and Dynamically Changing Software Components, *Symp. Software Reusability*, pp.169–180 (1997).
22) Binkley, D., Horwitz, S. and Reps, T.: Program Integration for Language with Procedure Calls, *ACM Trans. Softw. Eng. and Meth.*, Vol.4, No.1, pp.3–35 (1995).

(平成 9 年 12 月 10 日受付)
(平成 11 年 3 月 5 日採録)



丸山 勝久（正会員）

1967 年生．1991 年早稲田大学理工学部電気工学科卒業．1993 年同大学院理工学研究科修士課程修了．同年日本電信電話（株）入社．ソフトウェア研究所において，ソフトウェア再利用，プログラム自動合成，プログラム解析技術，ネットワーク再構成の研究に従事．現在，長距離国際移行本部メディア技術開発センタに所属．情報処理学会 1997 年度山下記念研究賞，1997 年度前期全国大会奨励賞授賞．博士（情報科学）．電子情報通信学会，日本ソフトウェア科学会，IEEE-CS，ACM 各会員．

島 健一（正会員）

1976 年北海道大学工学部電気工学科卒業．1978 年同大学院情報工学専攻修士課程修了．同年 NTT 武蔵野電気通信研究所入所．現在，NTT 移動通信網（株）マルチメディア研究所室長．主に，知識ベース構築用システムの基礎研究，ソフトウェア設計での知識獲得，学習システム等の研究開発に従事．また，WWW，モバイル環境でのユーザモデル研究に興味を持つ．電子情報通信学会，人工知能学会各会員．