# 重み付き依存グラフを用いたメソッドの再構成

丸山勝久[†,☆] 島健一[†,☆☆]

オブジェクト指向ソフトウェア開発において，フレームワークを再構成することは，その再利用性をより高くする効果を持つ．しかしながら，再構成操作は手動で行うには複雑である．本論文では，過去のアプリケーション開発時のメソッドの変更履歴に基づく重み付き依存グラフを用いて，フレームワークを自動的に再構成する手法を提案する．本再構成手法では，継承によりクラスを再利用した際，メソッド内部に存在する依存関係が保存あるいは破壊されるかどうかに応じて，依存関係の強さを指す重みを変動させる．重み付き依存グラフの矢印に蓄積された重み値に基づき，もとのフレームワークにおいて，そのまま再利用可能な固定部分と要求に応じて柔軟に変更する可変部分を分離することで，個々の開発者に特化したフレームワークの成熟化を実現する．適切に分離された固定部分と可変部分を含むフレームワークを用いることで，アプリケーション開発における実装の繰返しを軽減できる．本論文における評価実験では，開発者の記述コード量に関して，最大 22%（従来手法に比べて約 2 倍）の減少率を確認した．

# Automatic Method Refactoring Using Weighted Dependence Graphs

Katsuhisa Maruyama[†,☆] and Ken-ichi Shima[†,☆☆]

While refactoring makes application frameworks more reusable, it is complex to do by hand. This paper presents a mechanism that automatically refactors methods in object-oriented frameworks by using weighted dependence graphs, whose edges are weighted based on the modification histories of the methods. To find the appropriate boundary between frozen spots and hot spots in the methods, the value of the weight varies based on whether the dependence in the original methods has been repeatedly preserved or destroyed in the methods of applications created by programmers. The mechanism constructs both template methods that contain the invariant dependence and hook methods that are separated by eliminating the variant dependence. The new template methods and hook methods tailored to each programmer save him/her from writing superfluous code when reusing a framework. Experimental results show a reduction rate of up to 22% in the number of statements a programmer has to write when creating several applications; this percentage is double that achievable by a conventional refactoring technique.

## 1. はじめに

オブジェクト指向フレームワーク（あるいはフレームワーク）[1] は，特定のアプリケーションドメインにおいて共通に利用されるクラスとクラス間の相互作用を内包する[2),3)]．よって，フレームワークを再利用することで，開発者が必要なクラスを記述する手間とクラス間の関係を設計する際の負担を大幅に減らすことが可能である．しかしながら，アプリケーション開発者あるいは利用者が将来要求するであろう機能を予測することは困難であり，必要なクラスおよびクラス間の関係をフレームワーク作成時に完全に定義しておくことは不可能である．よって，フレームワークの再利用性をより高めるためには，開発アプリケーションに応じて，フレームワークを成熟させることが必要である[4),5)]．

このような要求に対して，自動（あるいは半自動）的にフレームワーク（あるいはクラスライブラリ）を再構成する手法が提案されている[4),6)~10)]．特に，文献 4)，9) では，メソッドを分割可能なものとして扱い，きめの細かいフレームワーク成熟化を実現している．文献 4) の手法は，発見的規則により 2 つのクラスにおいて類似の構造を検出し，共通のコードを 2 つのクラスが共有する抽象クラスに移動させる．また，

† NTT ソフトウェア研究所
NTT Software Laboratories
☆ 現在，立命館大学理工学部情報学科
Presently with Department of Computer Science, Ritsumeikan University
☆☆ 現在，株式会社 NTT ドコモ
Presently with NTT DoCoMo, Inc.

文献 9) の手法は，同じメッセージを同じ順序で送信する 2 つのメソッド（式）を共通コードとして抜き出し，クラス階層を再構築する．上記の手法は，フレームワークの構造を簡素化するという点で有効である．しかしながら，これらの手法は，クラスやメソッドの静的構造に基づく一般的な変換により実現されており，プログラマがどのようにフレームワークを利用するのか（したのか）という個々の開発者の特性に関して考慮していない．よって，開発者全体に合わせた汎用フレームワークを提供することはできるが，個々の開発者あるいは開発チームに特化した個別フレームワークを提供することはできない．

本論文では，フレームワークの成熟化を，フレームワークにおいてそのまま再利用する固定部分と要求に応じて柔軟に変更する可変部分の分離でとらえ，これら 2 つの部分を自動的に特定するメソッド再構成手法を提案する．本手法では，現在のフレームワークにおいて，将来もそのまま利用される可能性が高い部分を抽出するために，開発者がアプリケーションを作成する際に行ったメソッド再定義における変更履歴（および利用履歴）を用いる．このようなアプリケーション開発時の履歴は，開発者がどのコードを頻繁に再利用あるいは変更したのか，さらに，開発者はどのようにそれらのコードを再利用あるいは変更したのかという情報を含む．そこで，もとのフレームワーク内に存在する依存関係の強度に対して，履歴情報に基づく重みを導入し，従来のプログラム依存グラフ（PDG: program dependence graph）[11] を拡張した重み付きプログラム依存グラフ（WPDG: weighted program dependence graph）[12] 上に蓄積する．重みは，もとのメソッドを再利用した際にどの依存関係が繰り返し保存されたのか，および，もとのメソッドを再定義した際にどの依存関係が繰り返し破壊されたのかに応じて変動する．本再構成手法は，重み値を指標として既存メソッドにおける固定コードと可変コードを分離し，それぞれに対応するメソッドを構築する．

メソッド変更履歴に基づく重み付き依存グラフを用いた本再構成手法は次に示す特徴を持つ．

(1) 開発者の過去の行動を反映させた再構成メソッドを提供するため，従来の静的構造だけに基づき再構成を行う手法に比べて，個々の開発者に適したフレームワークを提供できる．
(2) 再構成を行う単位が個々の文（文献 9) における式と同じ）であるため，従来のクラスやメソッドを単位とする粒度の粗い再構成手法に比べて，開発者が変更する可変部分を小さく特定できる．
(3) 同一のシグニチャを持つメソッドの動作を変化させないため，フレームワークに対する理解の容易性を損なわずにメソッドを分割できる．

このような特徴を持つ再構成手法を用いることで，開発者は必要最小限の可変部分だけを定義するだけで要求アプリケーションを作成することが可能となり，メソッド変更に対する負担が軽減できる．

以下，2 章では，フレームワークにおける固定部分と可変部分について述べ，メソッド再構成における仮定および本再構成手法の概要を示す．3 章では，重み付き依存グラフと重みの変動操作を定義する．次に，4 章で，重み値を指標とするメソッドの分割および再構築手続きを述べる．5 章では，本手法を用いた評価実験の結果を示し，本手法の有効性について考察する．

## 2. メソッド再構成手法

本章では，フレームワークの成熟化に関する仮定を述べ，本再構成手法の概要を示す．

### 2.1 テンプレートメソッドとフックメソッド

提案するメソッド再構成手法は，継承におけるメソッドの変更履歴を利用する．よって，フレームワークを利用してアプリケーションを作成する際，開発者が既存クラスを拡張および洗練することを仮定している．開発者は，変更が必要なクラスに対して継承を適用し，その派生クラスにおいて既存メソッドを再定義（override）することで，要求する機能を実現する．変更を行う必要がないクラスについては，派生クラスを作らずにそのまま再利用する．このように，フレームワークは，そのまま再利用する固定部分（フローズンスポット：frozen spots）と，要求に応じて変更可能な可変部分（ホットスポット：hot spots）で構成されている[13]．一般的に，フローズンスポットはテンプレートメソッド（template method），ホットスポットはフックメソッド（hook method）で実装する．

いま，フレームワークが特定のアプリケーションに特化したテンプレートメソッドを提供していると，開発者は，少数のフックメソッドを再定義するだけで，要求を満たすアプリケーションを実現可能である．しかし，一方で，テンプレートメソッド全体を再定義しなければならない場面が多くなり，開発者の負担が増加する危険性を持つ．よって，再利用性の高いフレームワークを構築するためには，適切な規模のテンプレートメソッドとフックメソッドを定義することが重要である[13),14)]．

本論文では，再定義されないテンプレートメソッドと，最小の変更をともない再定義されるフックメソッ

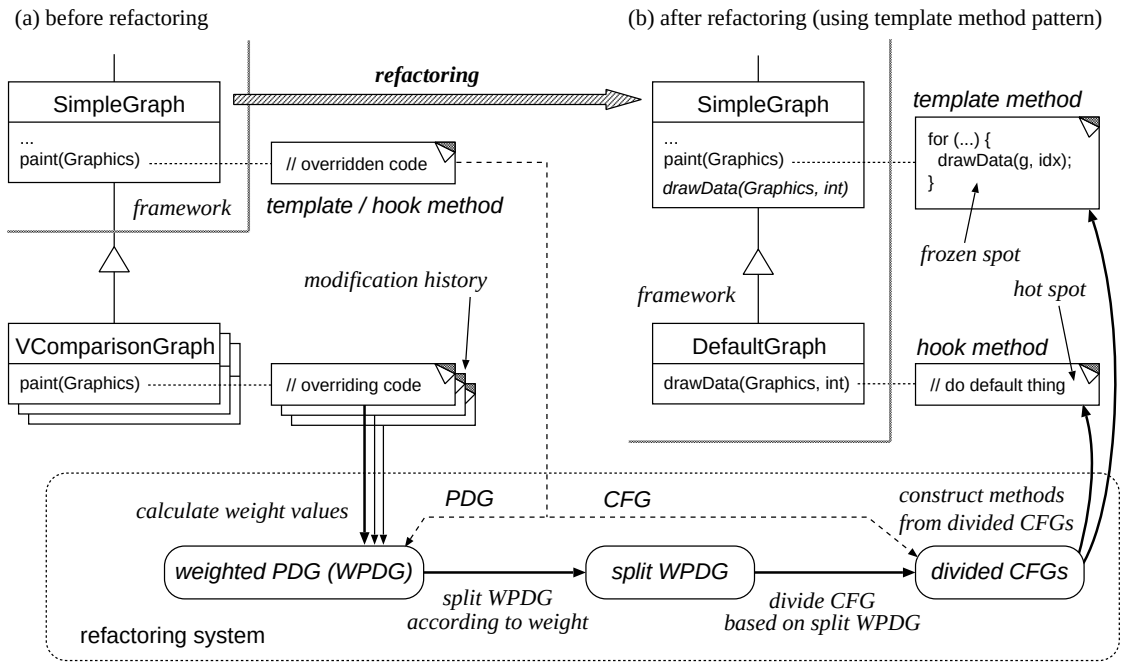



**図 1** 変更履歴に基づくメソッドの再構成
Fig. 1 Overview of the proposed mechanism for refactoring methods.

ドを含むフレームワークを成熟していると定義する．いいかえれば，新規アプリケーションを作成する際に，開発者が記述するコード（文）が少ないものを，より成熟しているフレームワークとおく．我々は，フレームワーク開発者が，どんなに注意深くアプリケーションドメインの特性を解析したとしても，このような成熟化フレームワークを初めから構築することはできないと考えている．なぜなら，フレームワークの利用方法は個々の開発者のプログラミングスタイルに大きく依存し，将来要求されるすべての機能を完全に予測することは不可能であるからである．そこで，開発者が過去に作成したアプリケーションを用いて，既存のフレームワークを段階的に成熟化させる．その際，次に示す仮定をおく．

**仮定** 再定義メソッドにおいて，開発者が過去に繰り返し実装した同一の制御フローは，将来もそのまま実装される可能性が高い．

開発者は過去に変更したメソッドのデータ構造および制御構造に関する知識を蓄積している．このため，過去のアプリケーションと類似のものを開発する場合，同様のメソッド変更を適用することが多い．よって，上記の仮定は妥当である．本再構成手法は，この仮定に基づき，頻繁に再定義されるメソッドを，同一の制御フローを実装するテンプレートメソッドと，その残りを実装するフックメソッドに分割する．

### 2.2 メソッド再構成の概要

本論文で提案するメソッド再構成手法の概要を**図 1** に示す．矢印の左側 (a) が再構成前，右側 (b) が再構成後のクラス関係図〔OMT（object modeling technique）表記[15]〕である☆．

アプリケーション開発者が，クラス SimpleGraph の派生クラス（たとえば，クラス VComparisonGraph）において，メソッド `paint` を頻繁に再定義していた場合を考える．ここで，開発者は，再定義メソッドをゼロから記述することはしないで，再定義されるメソッドを複写して，そのメソッドの一部を変更することで，アプリケーションを作成すると仮定する．さらに，メソッドを複写および変更する際，開発者はタグ付け機能を持つ構文エディタを用いることとする．タグはメソッド内部の各文に割り付けられ，等しいタグは変更前後において同一の文を指す．

メソッド再構成システムに対して，開発者（あるいはフレームワーク保守者）は，いくつかの再定義メソッドのコードを同時に入力，あるいは，メソッドを再定義するごとに 1 つずつ入力する．図 1 では，クラス VComparisonGraph のメソッド `paint` を含む複数の再定義メソッドが同時に入力されている．再構成システムは，入力された再定義メソッドを持つクラスの親

☆ フレームワーク中のクラスは文献 16) を参考にした．

クラス SimpleGraph のコード取り出し，そのクラス SimpleGraph の WPDG の矢印に割り付けられている重みを計算する．次に，重み値に応じて，WPDG を複数の部分グラフに分割する．これらの部分グラフは，フローズンスポットおよびホットスポットの候補となる．さらに，制御フローの一貫性を保ちながら，分割 WPDG に対応させてメソッド `paint` の制御フローグラフ（CFG: control flow graph）[17] を分割し，テンプレートメソッドとフックメソッドを構築する．このようにして，本再構成システムは，クラス SimpleGraph のメソッド `paint` を，再定義メソッドにおいて繰り返し記述されたコードを含むテンプレートメソッド `paint` と，繰り返し変更されたコードを含むクラス DefaultGraph のフックメソッド `drawData` に分割する．

## 3. メソッド変更履歴に基づく重み付け

本章では，重み付きプログラム依存グラフと，重みの変動操作を定義する．

### 3.1 重み付きプログラム依存グラフ

メソッドを再構成するための指標として，2 章の仮定に基づき，再構成対象メソッドを含むクラスのクラス依存グラフ（ClDG: class dependence graph）[18] における節点間の依存関係矢印に重みを割り付ける．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引数節点と，それらの節点間のデータ依存関係を表す矢印，メソッド呼び出し矢印，および，クラス–メンバ関係矢印を，PDG に付加したものである．ここで，本再構成手法は，クラス内部のすべての文に対して静的依存解析が可能なオブジェクト指向プログラムを扱う．本論文では，クラスおよびメソッドのコードを Java 言語[19] により記述する．また，フレームワーク内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関係を求めておき，メソッド引数に関するデータ依存関係に付けかえておく．

本論文では，重みを有する ClDG（あるいは ClDG のメソッドに対応する PDG）を重み付きプログラム依存グラフ（WPDG）と呼ぶ．WPDG は，プログラム内の代入文および条件式をラベルとする節点の集合と，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．これらの依存関係は，再構成対象ソースコードの CFG において，データおよび制御の到達可能性を調べることにより求める．重みは，依存関係矢印の接続元および接続先節点間の依存関係の結合の強さを表す．節点 $p$ から節点 $q$ にデータあるいは制御依存関係が存在するとき，その関係を矢印 $p \to q$ で表現する．さらに，重み $w$ を持つ依存関係矢印を，$p \xrightarrow{w} q$ と記述する．メソッド呼び出し矢印とクラス–メンバ関係矢印は，本再構成手法では用いないため，重みを割り付けない．

図 1 (a) の再構成前のクラス SimpleGraph のソースコードとメソッド `paint` の WPDG および CFG を**図 2** に示す．WPDG の各矢印に割り付けられている数値は，その依存関係矢印の重み値を表す．WPDG および CFG の節点上の数字とコードにおける各文の左側の数字は，節点と文の対応関係を示すタグである．図を明確にするため，メソッド `getXPos`，`getYPos`，`thickness`，`SimpleGraph`，`getFloat` については，WPDG における内部節点を省略した．また，入出力引数に対応する節点はコードに現れないため，各入出力引数節点が属するメソッド呼び出し節点およびメソッド入口節点に縮約した．クラス Graphics で定義されているメソッド `setColor`，`fillRect` は解析済みである．

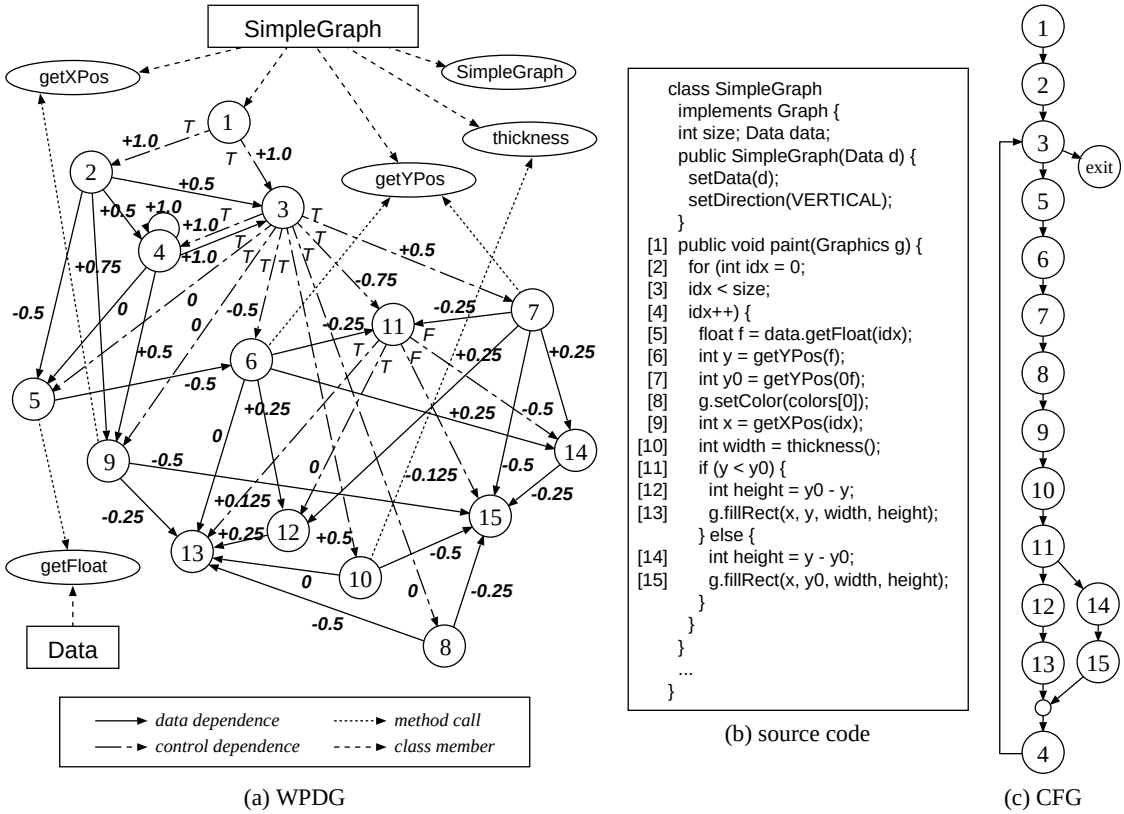
### 3.2 重み変動操作

本再構成手法では，メソッドのコード内部に存在する制御フローを依存関係でとらえる．したがって，過去のメソッド変更において，どの制御フローが繰り返し実装されたのかを特定するために，再定義メソッドの PDG における依存関係矢印の状態を，再構成対象メソッドを含むクラスの WPDG の矢印の重みに反映させる．いま，基底クラスにおいて再定義されるメソッド（再構成対象メソッド）の PDG を $G_B$，派生クラスの再定義メソッドの PDG を $G_D$ と表す．もし，$G_B$ に存在する節点 $p$，$q$ 間の依存関係が $G_D$ において保存された場合，すなわち，開発者が節点 $p$，$q$ 間に同一の制御フローを記述した場合，依存関係矢印 $p \to q$ の重み値を増加させる．逆に，$G_B$ に存在する節点 $p$，$q$ 間の依存関係が $G_D$ において破壊された場合，すなわち，開発者が節点 $p$，$q$ 間に存在する制御フローを変化させた場合，依存関係矢印 $p \to q$ の重み値を減少させる．依存関係の保存あるいは破壊のどちらでもない場合，重み値は変えない．

以上より，$G_B$ を含む WPDG の矢印 $p \xrightarrow{w} q$ における重み $w$ の変動条件と変動操作は次のようになる．

(a) **増加**：論理式 $p \to q \in E(G_D)$ が真のとき，依存関係の強度を強くする（$w_n = w_{n-1} + \delta$）．

(b) **減少**：論理式 $p \to q \notin E(G_D) \wedge (p \in N(G_D) \vee q \in N(G_D))$ が真のとき，依存関係の強度を弱くする（$w_n = w_{n-1} - \delta$）．




(a) WPDG (b) source code (c) CFG

**図 2** 再構成対象メソッドを含むクラスと，その WPDG および CFG
Fig. 2 Example of a class defining a method to be refactored, and its WPDG and CFG.

(c) **変動なし**：(a) および (b) の変動条件の両方とも成立しない，つまり論理式 $p \notin N(G_D) \wedge q \notin N(G_D)$ が真のとき，依存関係の強度を変えない（$w_n = w_{n-1}$）．

$G_B$ および $G_D$ において，同一のタグを持つ節点は，同一の節点を指す．さらに，同一の接続元および接続先節点を結合する矢印は，同一の矢印を指す．$N(G)$ は WPDG（あるいは，PDG）$G$ の節点集合，$E(G)$ は $G$ の矢印集合を表す．$n$ は $G_B$ に対応する再構成対象メソッドを再利用あるいは再定義した回数（重みの変動回数），$\delta$ は重みの変動範囲を決定する正の定数である．重みの初期値は，変動範囲の中間値（$w_0 = 0$）とする．下付きの添字 $n$ を持たない $w$ は，重み $w_n$ の現在の値を表す．

再定義メソッド `paint` のコードと変動操作 (a), (b), (c) による重み変動の例を**図 3** に示す．派生クラス VComparisonGraph において，図 2 に示す基底クラス SimpleGraph のメソッド `paint` を再定義した場合を考える．図 3 の PDG は，図 2 に示す WPDG $G_B$ の一部である．網かけは $G_D$ に含まれる節点を指す．記号 + を有する矢印は $G_B$ と $G_D$ の両方に存在するため，それらの矢印の重みは増加する．逆に，記号 − を有する矢印は $G_D$ に存在しない，かつ，接続元あるいは接続先節点のどちらかが $G_D$ に存在するため，それらの矢印の重みは減少する．記号 $*$ を有

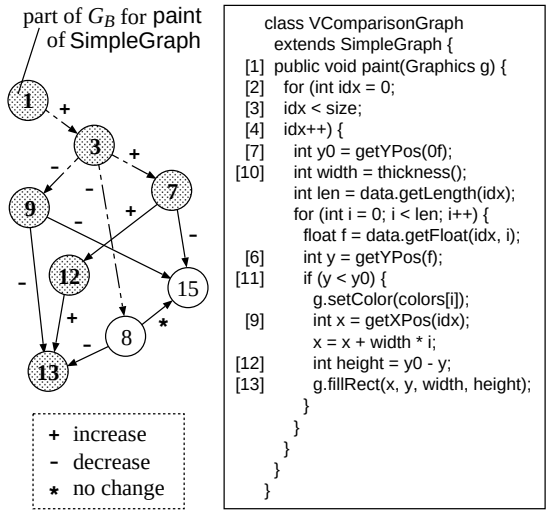



**図 3** 重みの変動操作
Fig. 3 Operations for varying weight.

する矢印は，接続元および接続先節点の両方とも $G_D$ に存在しないため，その矢印の重みは変化しない．

上記の変動操作 (a)，(b)，(c) により計算した重み $w$ の変動範囲は $-\delta n$〜$\delta n$ となり，その最大値および最小値が一定でない．そこで，重みがフレームワーク内のどのメソッドに対しても共通の指標となるように，変動範囲の最大値（$|\delta n| = \delta n$）に対する割合で重みを正規化する．すなわち，正規化後の重みを $W = w/\delta n$ とする．$W$ の値は $-1$ から 1 の間を変動し，その初期値は 0 である．

本再構成手法では，重み $w$ の現在値を WPDG 内部に蓄積し，この値を正規化した重み $W$ を WPDG の依存関係矢印に割り付ける．図 2 に示す WPDG の矢印の数値は $W$ の値を指す．重み $w$ から $W$ への正規化にともない変更（利用）回数に関する情報が失われるが，本再構成手法では重みの閾値により制御フローを区別するだけであるため，このような単純な正規化で十分である．変更回数が少ない場合は再構成の対象としない，あるいは，変更回数に応じて閾値を段階的に変えるというようなメソッド再構成を達成する際には，正規化の方法を改良する必要がある．

### 3.3 依存関係矢印に対する重みの意味

制御フローに着目してメソッドの再構成を行う場合，そのソースコードを CFG で表現し，CFG を操作するのが一般的である．しかしながら，メソッド内部に存在するすべての制御フローが，そのメソッドの CFG に矢印として直接現れるわけではない．たとえば，図 2 (b) のソースコードにおいて，文 [2] から文 [5] に到達するデータ $idx$ に関する制御フローや文 [11] から文 [13] に選択的に到達する制御フローは，図 2 (c) の CFG には直接現れていない．そこで，本論文では，メソッド内部の個々の制御フローを区別し，その利用に関する頻度を蓄積するために，CFG における制御フロー矢印の代わりに，PDG における依存関係矢印を用いる．

データ依存関係 $p \to q$（たとえば，図 2 (a) の矢印 2→5）は，節点 $p$，$q$ 間のデータの受渡しに関する制御フローを暗に指す．すなわち，節点 $p$ における変数の定義が節点 $q$ における変数の利用に到達する経路が，CFG に存在することを指す．同様に，制御依存関係 $p \to q$（たとえば，図 2 (a) の矢印 11→13）は，節点 $p$，$q$ 間の従属に関する制御フローを暗に指す．すなわち，節点 $p$ における述語の評価（真偽）が節点 $q$ の実行に選択的に到達する（到達する場合としない場合の両方の）経路が，CFG に存在することを指す．

WPDG における矢印の重み $W$ は，過去のメソッド変更において，その依存関係矢印の指す制御フローがどの程度保存あるいは破壊されたのかを指す．重み $W$ の値が 1 に近いほど，その依存関係矢印が過去に作成したアプリケーションにおいて繰り返し現れた割合が多いことを意味する．いいかえれば，開発者は，重み値が大きい矢印により結合されているコードを頻繁に再利用し，その矢印の表す依存関係を変化させなかった（保存した）といえる．よって，このような矢印は，再構成対象メソッドにおける不変的な制御フローを表していると見なせる．逆に，重み $W$ が $-1$ に近いほど，その依存関係矢印が過去に作成したアプリケーションにおいて繰り返し消去された割合が多いことを意味する．いいかえれば，開発者は，重み値が小さい矢印により結合されているコードを頻繁に再利用したものの，その矢印の表す依存関係を変化させた（破壊した）といえる．よって，このような矢印は，再構成対象メソッドにおける可変的な制御フローを指していると見なせる．

## 4. 重み値に基づくメソッドの構築

本章では，WPDG における矢印の重み値を指標として，フローズンスポットとホットスポットを特定し，テンプレートメソッドとフックメソッドを構築する手続きについて述べる．

### 4.1 フローズンスポットとホットスポットの特定

2 章において，開発者はフレームワークを過去と同様の方法で利用および変更する可能性が高いという仮定を述べた．この仮定と重みの意味より，重み値が大きい依存関係矢印は，将来のアプリケーションにおいても頻繁に現れると予測できる．すなわち，このような矢印により接続された 2 つの節点（に対応する文）は，それらの間に成立する制御フローを維持したままで再利用される可能性が高い．よって，これらの節点はフローズンスポットに残しておくべきである．逆に，重み値が小さい依存関係矢印は，将来のアプリケーションにおいて，それほど現れないと予測できる．すなわち，このような矢印により接続された 2 つの節点（に対応する文）は，それらの間に成立する制御フローが分断されて再利用される可能性が高い．よって，これらの節点の片方あるいは両方はホットスポットに移動させる方がよい．

本再構成手法では，維持する矢印と消去する矢印を区別する際の重みの閾値を，重み $W$ の中間値（つまり，0）に設定する．ここで，重み値が 0 より大きい（正の値の）矢印を強依存関係矢印，重み値が 0 より小さい（0 あるいは負の値の）矢印を弱依存関係矢印

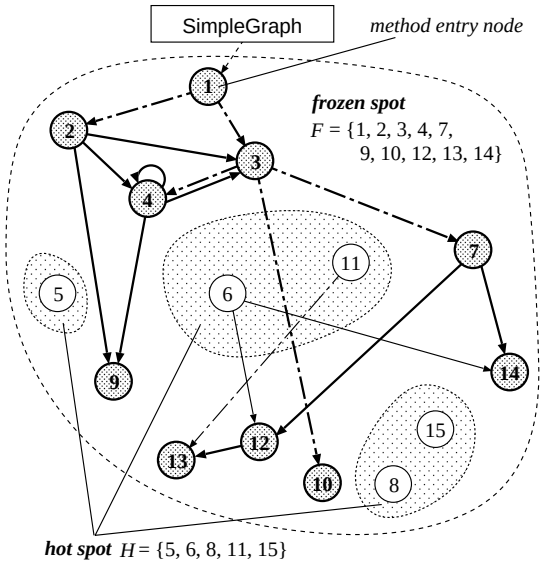



**図 4** 分割後の WPDG
Fig. 4 Split WPDG.

と呼ぶ．まず，再構成対象メソッドの WPDG において，強依存関係矢印を残し，逆依存関係矢印を消去することで，この WPDG を複数の部分グラフに分割する．その後，このメソッドにおいて必ず実行される入口節点から到達可能な節点の集合 $F$ を求め，これをフローズンスポットとする．ホットスポットは，メソッド内部の全節点から $F$ 内の節点を取り除いた際に残る節点の集合 $H$ で表す．図 2 の WPDG に対して弱依存関係矢印を消去した際に生成される部分グラフを**図 4** に示す．網掛けの節点はフローズンスポット $F$ を指す．

### 4.2 フローズンスポットとホットスポットの再編

フローズンスポット節点集合 $F$ とホットスポット節点集合 $H$ から，テンプレートメソッドとフックメソッドを構築する．その際，$F$ に含まれる節点に対応する文はもとのメソッドに残し，これらの文で基底クラスのテンプレートメソッドを構築する．$H$ に含まれる節点に対応する文は，もとのメソッドから新規派生クラスのフックメソッドに移し，このフックメソッドを呼び出す文を，テンプレートメソッドの適切な位置に挿入する．

一見すると，上記の操作は正しいように見える．しかしながら，フックメソッドに移動した文がもとのメソッドにおいて連続していたとは限らないため，再構成前メソッドに存在していた制御フローが再構成後のメソッドにおいて維持される保証はない．不連続な文を 1 つのフックメソッドに集めてしまうと，そのフックメソッドに複数の入口節点が存在することになり，そのようなコードを本再構成手法で扱うプログラム言語で記述することは不可能である．再構成前メソッドに存在していた制御フローを維持したままで，フックメソッドを切り出すためには，1 つのフックメソッドに含まれる文が連続している必要がある．

本再構成手法では，$H$ で構成される 1 つのホットスポットを，制御フローが連続する節点だけを含む複数のホットスポットに分割することで，上記の問題を回避する．ホットスポットを分割する際の規則を次に示す．

**分割規則 1** ホットスポットがフローズンスポットを入れ子とすることはできない．ここで，PDG において，分岐節点 $p$（`if-`, `while-`, `for-condition` 節点）から制御依存関係矢印だけをたどり到達可能な節点の集合を $BranchRegion(p)$ とおく．節点 $n \in H$ に対する $BranchRegion(n)$ 内のすべての節点は，たとえ，それらの節点が $F$ に含まれているとしても $H$ に移動する．

**分割規則 2** 節点 $n \in F$ が $H$ 内の節点間の経路を分断するとき，$H$ 内の節点は CFG 上において，節点 $n$ を通過しないように別々のホットスポットに分割する．$H$ が節点 $n$ により分断されるとは，節点 $p, q \in H$ に対して，$p \Rightarrow^* n \Rightarrow^* q$ となる節点 $n \in F$ が存在することを指す．記号 $\Rightarrow^*$ は CFG の経路を指す．

**分割規則 3** 節点 $n \in F$ が $H$ 内の節点間の経路を選択するとき，$H$ 内の節点は CFG 上において，それぞれの経路に含まれる節点で構成するホットスポットに分割する．$H$ が節点 $n$ により選択されるとは，節点 $p, q \in H$ に対して，$n \Rightarrow p$ かつ $n \Rightarrow q$ となる節点 $n \in F$ が存在することを指す．記号 $\Rightarrow$ は CFG の矢印を指す．

$F$ と $H$ を含む CFG に対して，上記の分割規則 1, 2, 3 を適用することで，再構成対象メソッドのフローズンスポットとホットスポットを再編する．規則 1, 2, 3 に基づき，再構成対象メソッドの CFG を分割するアルゴリズム dividesCFG を**図 5** に示す．関数 $succ(p)$ は節点 $p$ の直後節点，関数 $trueSucc(p)$ は節点 $p$ の真方向への分岐に関する直後節点，関数 $falseSucc(p)$ は節点 $p$ の偽方向への分岐に関する直後節点を返す．関数 $join(p)$ は $trueSucc(p)$ と $falseSucc(p)$ を始点とする経路が合流する節点を返す．ここで，節点 $p$ が $H$ に含まれるとき，関数 $join(p)$ の返す節点も $H$ に含まれることとする．

簡単にいえば，アルゴリズム dividesCFG は，CFG

```
algorithm dividesCFG(G, F, H, F′, H[ ])
input G: a CFG for the method to be refactored;
      F: a set of nodes in a frozen spot;
      H: a set of nodes in a hot spot;
output F′: a set of nodes in a new frozen spot;
       H[ ]: an array of node sets in new hot spots;
declare idx: an integer (an index of array H[ ]);
        m, n: nodes;
  procedure gatherNodes(p)
  input p: a node;
  begin
    if p ∈ H then H[idx] := {p};
      if p is a branch node then
        H[idx] := H[idx] ∪ BranchRegion(p);
        if p is a if-node then
          gatherNodes(join(p));
        else // while- or for-condition-node
          gatherNodes(falseSucc(p));
        fi
      else gatherNodes(succ(p)); fi
    fi
  end // H[idx] is a set of nodes in a path from n
begin
  F′ := a copy of F; idx := 1;
  for each n ∈ H do
    if ∃ m⇒n ∈ edges of G(m ∈ F′ ∧ n ∈ H) then
      // n is the entry node for a new hook method
      gatherNodes(n);
      F′ := F′ − H[idx]; idx := idx + 1;
    fi
  od
end // A − B: difference set
```

**図 5** CFG を分割するアルゴリズム
Fig. 5 Algorithm for dividing a CFG.

において $F$ 内の節点から入ってくる矢印の接続先節点 $p \in H$ と $F$ 内の節点に出ていく矢印の接続元節点 $q \in H$ の間の到達可能経路を抜き出している．個々のホットスポットは，CFG において連続している節点のみで構成され，必ず 1 つの入口節点と 1 つの出口節点を持つ．

### 4.3 ホットスポットの融合

CFG 分割アルゴリズム dividesCFG は，もとの再構成対象メソッドを，内部の制御フローを維持したままで，単一のテンプレートメソッドと複数のフックメソッドに分割する．しかしながら，このアルゴリズムは，規模の小さな（含まれる文の数が少ない）フックメソッドを数多く生成してしまう．再構成後のフレームワークにおいて，1 つのメソッドがあまりにも多くのメソッドと協調していると，そのフレームワークを理解することは困難になる．この問題に対して，本再構成手法では，コード移動（code motion）技法[17),20)]により，できるだけ多くのホットスポットを融合し，新規に生成されるホットスポットの数を抑える．その際，任意のコードを任意の位置に移動させるアルゴリズムは非常に複雑であるため，本再構成手法では制限されたコードだけを移動させるアルゴリズムを用意する．

いま，2 つのホットスポット $H_A$ と $H_B$ を考える．アルゴリズム dividesCFG により，$H_A$ および $H_B$ に対応する部分 CFG は，入口節点と出口節点を必ず 1 つずつ持つ．ここで，$H_A$ の入口および出口節点を $h_A^i$ と $h_A^o$，$H_B$ の入口および出口節点を $h_B^i$ と $h_B^o$ とおく．また，節点 $h_A^o$ の直後の（$h_A^o$ から直接制御が移る）フローズンスポット内の節点を $f_P^i$，節点 $h_B^i$ の直前の（$h_B^i$ に直接制御を移す）フローズンスポット内の節点を $f_P^o$ とおく．もし，節点 $f_P^i$ と節点 $f_P^o$ 間にホットスポットを途中に含まない到達可能経路が存在するならば，その経路に含まれる節点と矢印により構成されるフローズンスポットの一部 $F_P$ は，$H_A$ と $H_B$ にはさまれているという．さらに，もし $F_P$ 内のすべての節点が，1) 節点 $h_A^o$ から入ってくる矢印，2) 節点 $h_B^i$ に出ていく矢印，3)$F_P$ 内の節点を接続する矢印，という 3 種類以外の矢印を持たないとき，つまり，$F_P$ が基本ブロック[17)] のようなもののとき，$F_P$ は $H_A$ と $H_B$ に完全にはさまれている（fully sandwiched）という．

本再構成手法は，ホットスポット $H_A$ と $H_B$ に完全にはさまれているフローズンスポット $F_P$ のコードを移動することにより，$H_A$ と $H_B$ を融合する．次に示す融合前の制御フロー

$$Flow \equiv \underbrace{h_A^i \Rightarrow^* h_A^o}_{H_A} \Rightarrow \underbrace{f_P^i \Rightarrow^* f_P^o}_{F_P} \Rightarrow \underbrace{h_B^i \Rightarrow^* h_B^o}_{H_B}$$

に対して，次に示す 2 つの融合規則が考えられる．

**融合規則 1** 内部の制御フローを維持したまま，$F_P$ 内のすべての節点を節点 $h_A^i$ の直前に移動する．移動後の制御フロー $Flow'_u$ は次のようになる．

$$Flow'_u \equiv \underbrace{f_P^i \Rightarrow^* f_P^o}_{F_P} \Rightarrow \underbrace{h_A^i \Rightarrow^* h_A^o \Rightarrow h_B^i \Rightarrow^* h_B^o}_{H_A + H_B}$$

**融合規則 2** 内部の制御フローを維持したまま，$F_P$ 内のすべての節点を節点 $h_B^o$ の直後に移動する．移動後の制御フロー $Flow'_d$ は次のようになる．

$$Flow'_d \equiv \underbrace{h_A^i \Rightarrow^* h_A^o \Rightarrow h_B^i \Rightarrow^* h_B^o}_{H_A + H_B} \Rightarrow \underbrace{f_P^i \Rightarrow^* f_P^o}_{F_P}$$

融合規則 1，2 に基づき，ホットスポットを融合するアルゴリズム assimilateHotSpots を**図 6** に示す．関

```
algorithm assimilatesHotSpots(G, H[ ], G_C)
input G: a CFG for the original method;
      H[ ]: an array of node sets in the new hot spot;
output G_C: a CFG with the assimilated hot spots;
declare A, B: integers (indices of array H[ ]);
        G_D, G'_D: PDGs;
        G_t: a CFG; F_P: a sub-CFG;
begin
  G_C := a copy of G; G_D := a PDG for G;
  for A := 1 to sizeof(H[ ]) do
    h^i_A := entry(H[A]); h^o_A := exit(H[A]);
    for B := 1 to sizeof(H[ ]) do
      h^i_B := entry(H[B]); h^o_B := exit(H[B]);
      f^i_P := succ(h^o_A); f^o_P := pred(h^i_B);
      extracts F_P in paths between f^i_P and f^o_P;
      if F_P is fully sandwiched
           between H[A] and H[B] then
        G_t := a copy of G_C;
        moves F_P upward to H[A] in G_t;
        G'_D := a PDG for G_t;
        if G_D = G'_D then // graph matching
          G_C := a copy of G_t;
          H[B] := H[B] ∪ H[A]; H[A] := ∅;
        else
          G_t := a copy of G_C;
          moves F_P downward to H[B] in G_t;
          G'_D := a PDG for G_t;
          if G_D = G'_D then // graph matching
            G_C := a copy of G_t;
            H[B] := H[B] ∪ H[A]; H[A] := ∅; fi
        fi
      fi
  od od
end
```

**図 6** ホットスポットを融合するアルゴリズム
Fig. 6 Algorithm for assimilating hot spots.

数 *entry*(*G*) と *exit*(*G*) は，それぞれ部分 CFG $G$ の入口節点と出口節点を返す．関数 *succ*(*p*) と *pred*(*p*) は，それぞれ節点 $p$ の直後節点と直前節点を返す．関数 *sizeof*(*H*[ ]) は，配列 $H[\,]$ の要素の数を返す．

本再構成手法では，部分 CFG $F_P$ を移動した際，もとのメソッドの動作に変化が生じるかどうかを PDG の同値性に基づき判定する．その際，従来の PDG に存在する依存関係に対して，(1) データ依存関係をループ経由（loop-carried）とループ独立（loop-independent）に分類，(2) 定義順序関係（definition-order）を追加，という拡張を行った PDG[21] を用いる．2 つのプログラムに対して，このような拡張 PDG が等価であると，それらのプログラムは等価な動作をする，つまり，与えられた入力に対してつねに同じ出力を計算することが知られている[21]．アルゴリズム assimilatesHotSpots は，融合前後の CFG からそれぞれ拡張 PDG を構築し，グラフマッチングにより PDG の同形性を判定する．融合前後の PDG が同形であると判定された場合のみ，もとの CFG を融合後の CFG と取り替える．このようにして，本再構成手法は，融合後の CFG がもとの CFG と同一の動作を行うことを保証する．

### 4.4 再構成メソッドの生成

本再構成手法は，Template Method パターン[22] により，テンプレートメソッドを含む抽象クラスと，フックメソッドを含む具象クラスを構築する．フローズンスポット節点集合 $F$ 内の節点に対応する文を再構成対象メソッドに残し，抽象クラスの新規テンプレートメソッドとする．ホットスポット節点集合 $H$（正確には，$H[\,]$）内の節点に対応する文を再構成対象メソッドから抜き出し，具象クラスのフックメソッドとする．同時に，抽象クラスにおいて，新規に構築したフックメソッドと結合する抽象メソッドを作成し，この抽象メソッドを呼び出す文をテンプレートメソッドに挿入する．挿入位置は，実際に呼び出されるフックメソッドが，もとの再構成対象メソッドにおいて存在していた場所である．

さらに，もしテンプレートメソッドが，フックメソッド内部で値が定義される変数を利用しているならば，この変数を抽象クラスの保護変数（protected instance variable）として宣言する．同様に，もし異なるフックメソッドが，それらの祖先クラスで宣言されていない変数を共有しているならば，この変数を具象クラスの保護変数として宣言する．保護変数とは，その変数が宣言されているクラスの子孫（自分自身を含む）でのみ定義および利用可能なものである．

**図 7** に，メソッド再構成の例を示す．図 7 (a) は，図 1 (a) に示す再構成前のクラス SimpleGraph のメソッド `paint` の CFG（図 2 (c) と同じ）である．図 4 に示す $F$ と $H$ に対して，フローズンスポットとホットスポットを再編した後の CFG を図 7 (b) に示す．再編後の CFG は，3 つのホットスポット $H_1$，$H_2$，$H_3$ を含む．さらに，ホットスポットを融合した後の CFG を図 7 (c) に示す．本例では，図 7 (b) の CFG において節点 7 が $H_1$ と $H_2$ に完全にはさまれている．節点 7 は節点 5 の直前に移動可能であるため，$H_1$ と $H_2$ が新規ホットスポット $H'_2$ として融合される．同様に，節点 9 と 10 を持つ部分 CFG も節点 5 の直前に移動されて，$H'_2$ と $H_3$ が新規 $H'_3$ として融合される．最終的に，本再構成システムは，図 7 (d) に示すコードを生成する．再構成後のコードは，抽象クラス SimpleGraph の新規テンプレートメソッド `paint` と，その具象クラス DefaultGraph の新規フックメソッド `H3` を含む．このコードをみると，クラス SimpleGraph

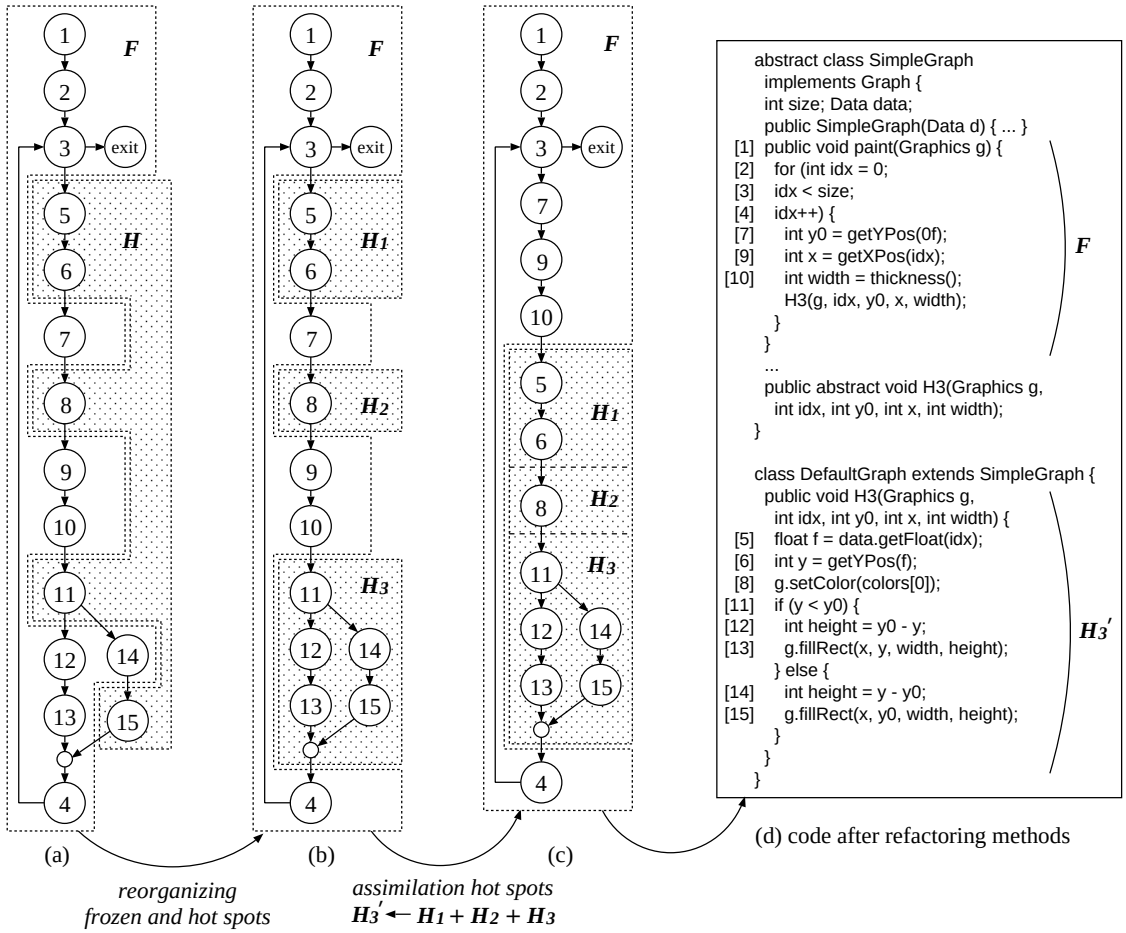



**図 7** 再構成前後の CFG と，再構成後のメソッドのコード
Fig. 7 CFGs before and after refactoring, and the code of the derived methods.

のメソッド `paint` の動作が再構成前後で変化していないことが分かる．

## 5. 評価実験

本再構成手法の特性および有効性を確認するために，実際のフレームワークを用いて，メソッドを再構成する実験を行った．本手法に対する評価項目として，次に示す 3 点を設定した．

(1) 従来の一般的な再構成手法が単純に共有コードを抜き出すのに対して，本再構成手法は重み付き依存関係に基づき共通コードを抜き出す．これらの手法で，再構成後のメソッドに違いがあるのか．
(2) 再構成後のメソッドを用いてアプリケーションを作成した場合，新規コードを記述する手間がどの程度削減されるのか．
(3) 再構成前のメソッドより再構成後のメソッドを含むフレームワークの方が，容易に利用および理解可能であるか．

**図 8** に示すフレームワークに対して，クラス SimpleGraph のメソッド `paint` を分割して，テンプレートメソッド `newTemplate` とフックメソッド `newHook` を作成する．このフレームワークは，2 次元グラフを表示するアプリケーションを作成するためのものである．本実験で用いたメソッドを**表 1** に示す．図 8 のフレームワークおよび表 1 のメソッドは，文献 16) のフレームワークにおけるコードを一部書き換えることで作成した．

本実験では，再構成対象メソッドを VB とし，VB に対する変更履歴として 4 つの再定義メソッドの集合を用意した．変更履歴は，M1 = {VB, HB}，M2 = {VB, VC, VS}，M3 = {VB, VC, VS, HB, HC, HS}，M4 = {VB, VC, VS, HB, HC, HS, VL, HL} である．M1 と M2 はそれぞれ，類似しているメソッ

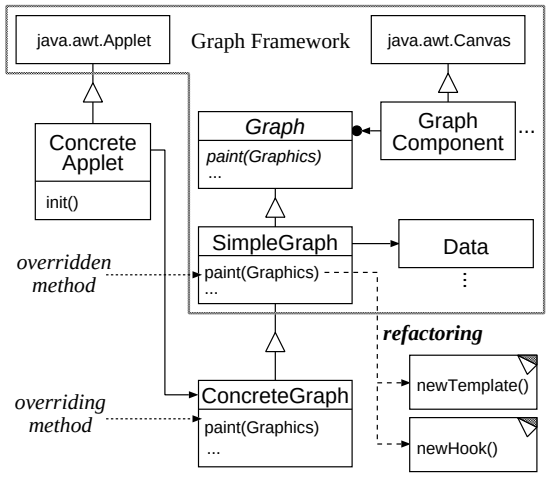



**図 8** 実験用フレームワーク
Fig. 8 Experimental framework.

**表 1** 実験で用いたメソッド
Table 1 Experimental methods.

| メソッド | 節点数 | メソッドの説明 |
|---|---|---|
| VB | 15 | Vertical Bar graph |
| VC | 18 | Vertical Comparison graph |
| VS | 19 | Vertical Stack graph |
| VL | 23 | Vertical Line graph |
| HB | 15 | Horizontal Bar graph |
| HC | 19 | Horizontal Comparison graph |
| HS | 19 | Horizontal Stack graph |
| HL | 22 | Horizontal Line graph |

ドだけを要素とする．M3 は，やや類似しているメソッドを集めたものである．M4 は，他のメソッドとそれほど類似していない 2 つのメソッド VL と HL を，M3 に追加したものである．

さらに，本実験では，提案手法を従来手法と比較するため，変更履歴に含まれるすべての再定義メソッドに共通して現れるコードを単純に抽出することで，メソッド VB の再構成を行った．たとえば，変更履歴 M1 を用いた場合，メソッド VB と HB に共通するコードを再構成後のテンプレートメソッドとした．本実験で用いたメソッドのコードにおいて，メソッド呼び出し式の引数に現れる式は，あらかじめ別の文として引数の外部に抜き出してある．よって，比較のために用意した一般的再構成手法は，文献 4)，9) の手法とほぼ同じ結果を示す．

再構成前後のメソッドにそれぞれ含まれる節点（文）の数を**表 2** に示す．T は再構成後の新規テンプレートメソッド，H1 と H2 は新規フックメソッドを指す．上付きの添字 c は，従来手法により構築されたメソッドを表す．さらに，再構成後のフレームワークを用いて，表 1 のメソッドを含むアプリケーションを実際に作成する際，開発者が記述する節点（文）の数を，表 2 の “VB，VC，VS，VL，HB，HC，HS，HL” 欄に示す．記号–は，アプリケーション作成時に，そのメソッドを利用しなかったことを指す．

### 5.1 再構成メソッドの違いに関する考察

表 2 において，本手法による再構成メソッド T，H1，H2 の節点数と，従来手法による再構成メソッド $T^c$，$H1^c$，$H2^c$ の節点数を比較する．変更履歴 M2，M3，M4 を用いた場合，対応するテンプレートメソッドどうしの節点数が異なる（たとえば，M2 に対して，T は 12 節点，$T^c$ は 7 節点）ことより，2 つの手法により構築されたメソッドは明らかに異なる（変更履歴 M1 に対しては，同一のメソッドが構築された）．さらに，すべての変更履歴 M1，M2，M3，M4 に対して，メソッド T の節点集合が，対応するメソッド $T^c$ の節点集合を必ず包含するという結果を示した．

このような結果の主な理由は，本再構成手法が，変更履歴に含まれる再定義メソッドにおける節点（コード断片）を，それがすべての再定義メソッドに含まれていなくとも，積極的に集めることにある．正確にいえば，本再構成手法は，変更履歴内の再定義メソッドの半分以上に存在する制御フローを，再定義メソッド全体の共通制御フローであると見なす．このため，すべての再定義メソッドに含まれていない節点でも，テンプレートメソッドに含まれることがある．逆に，従来手法では，すべての再定義メソッドに必ず含まれる節点だけが，テンプレートメソッドに含まれる．

さらに，本再構成手法は，変更履歴 M3 に対して変更履歴 M4 が非類似メソッド VL と HL を含むにもかかわらず，M3 と M4 において同一のテンプレートメソッド T を構築した．これに対して，従来手法では，M4 のとき，$T^c$ の節点数は極端に少なくなっている．このような結果は，本再構成手法が，たとえ変更履歴内に他のメソッドと類似していないメソッドが含まれていたとしても，ある程度の規模を持つテンプレートメソッドを構築できることを示している．

### 5.2 コードの再利用に関する考察

メソッド再構成は，開発者が繰り返し記述した共通コードを既存メソッドから抜き出すため，フレームワーク内のコードの再利用性を高めるという効果が期待できる．本実験では，開発者が過去に記述したコードは新規アプリケーションを作成する際にも頻繁に記述される可能性が高いと仮定し，再定義メソッドを含むアプリケーションを作成する際に，開発者が記述しなければならなかった節点（文）の数を調べた．

**表 2** 再構成前後のメソッドに含まれる節点数とアプリケーション作成時に記述した節点数
Table 2 Number of nodes typically written in methods compared to number written when the proposed mechanism is used.

| 変更履歴 | – | M1 | | | | M2 | | | | | M3 | | | | M4 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 手法 | – | 提案 | | 従来 | | 提案 | | | 従来 | | 提案 | | 従来 | | 提案 | | 従来 | | |
| メソッド | VB | T | H1 | $T^c$ | $H1^c$ | T | H1 | H2 | $T^c$ | $H1^c$ | T | H1 | $T^c$ | $H1^c$ | T | H1 | $T^c$ | $H1^c$ | $H2^c$ |
| 節点数 | 15 | 11 | 5 | 11 | 5 | 12 | 3 | 2 | 7 | 9 | 8 | 8 | 7 | 9 | 8 | 8 | 5 | 1 | 11 |
| VB | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| VC | 18 | 18 | – | 18 | – | 18 | – | – | 0 | 15 | 0 | 12 | 0 | 15 | 0 | 12 | 0 | 0 | 15 |
| VS | 19 | 19 | – | 19 | – | 19 | – | – | 0 | 16 | 0 | 13 | 0 | 16 | 0 | 13 | 0 | 0 | 16 |
| VL | 23 | 23 | – | 23 | – | 23 | – | – | 23 | – | 23 | – | 23 | – | 23 | – | 0 | 2 | 20 |
| HB | 15 | 0 | 5 | 0 | 5 | 15 | – | – | 0 | 12 | 0 | 9 | 0 | 12 | 0 | 9 | 0 | 0 | 12 |
| HC | 19 | 19 | – | 19 | – | 19 | – | – | 0 | 16 | 0 | 13 | 0 | 16 | 0 | 13 | 0 | 0 | 16 |
| HS | 19 | 19 | – | 19 | – | 19 | – | – | 0 | 16 | 0 | 13 | 0 | 16 | 0 | 13 | 0 | 0 | 16 |
| HL | 22 | 22 | – | 22 | – | 22 | – | – | 22 | – | 22 | – | 22 | – | 22 | – | 0 | 2 | 19 |
| 合計 | 135 | 125 | | 125 | | 135 | | | 120 | | 105 | | 120 | | 105 | | 118 | | |
| 減少率 | – | 10 (7%) | | 10 (7%) | | 0 (0%) | | | 15 (11%) | | 30 (22%) | | 15 (11%) | | 30 (22%) | | 17 (13%) | | |

表 2 において，再構成後のメソッド T，H1，H2 を用いた際に記述した節点数の合計は，再構成前のメソッドを用いた際に記述した節点数の合計より少ないか等しい．節点数の変化は，135 → 125，135 → 135，135 → 105，135 → 105 となり，それぞれの減少率は 7%，0%，22%，22%である．個々の結果に注目すると，変更履歴 M1 および M2 による再構成では，減少率はわずかである．これは，変更履歴に含まれるメソッドが類似しているとき，本再構成手法は過度に大きなテンプレートメソッドを生成する傾向にあるからである．変更履歴 M1 および M2 に比べて，M3 および M4 に対しては，本再構成手法は従来手法に比べてかなり良い結果を示した（減少率がそれぞれ 2 倍，1.7 倍）．これは，互いに類似していないメソッドを用いて再構成を行った場合でも，本再構成手法は適切な規模の（小さすぎない）テンプレートメソッドを生成可能であることによる．

本実験より，本再構成手法（および従来手法）を用いることで，新規にコードを記述する開発者の負担が軽減されることが確認できた．再構成に用いるメソッドが互いにきわめて類似している場合には従来手法が有利であるのに対して，それらのメソッドがそれほど類似していない場合には本再構成手法が有利である．

### 5.3 利用および理解容易性に関する考察

メソッド再構成の目的は，コードの再利用性だけでなく，設計の再利用性も向上させることである．フレームワーク内のクラス（あるいはオブジェクト）が適切なインタフェースや関連を持つ場合，開発者はそのフレームワークを容易に理解できる．オブジェクト指向ソフトウェアに関する設計メトリクスは従来から提案されている（たとえば，文献 5)，23) がある）．しかしながら，それらのメトリクスは，個々の開発者に特化したフレームワークを提供するという目的において，どの制御フローをメソッド内部に隠蔽するべきであるか，あるいは，どの制御フローを交換可能にするべきであるかという指標を与えてはくれず，本再構成手法の効果を評価するのに適さない．そこで，本論文では，メトリクスを用いずに，再構成前後でのコードの変化という観点から，本再構成手法の設計再利用に関する効果を考察する．

どのメソッド再構成においても，もとのメソッド VB の動作は再構成前後で変化していないことが確認できた．これは，メソッド再構成における必須の条件である．さらに，すべての再構成において，VB 内のループ構造を実装する文（図 2 (b) の文 [2]，[3]，[4]）が新規テンプレートメソッド T に残り，ループ内部の文が新規フックメソッド H1 あるいは H2 に移動していた．このことより，再構成後のメソッドを含むフレームワークは，メソッド再構成を行わないときと比べて，より高い柔軟性を持つといえる．なぜなら，ループ内部のコード断片がループを制御するコード断片と分離され，ループ内部のコードの交換が容易になったからである．

さらに，本再構成手法は，いくつかのアプリケーションで同じように利用される文（図 2 (b) の文 [7]，[9]，[10]）を，これらの文がもとのメソッド内部で連続していないにもかかわらず，同時にテンプレートメソッド T に移動させていた．これは，本再構成手法が，メソッド内部に明示的に存在する制御フローに直接着目するのではなく，内部に存在する依存関係に基づき共通の制御フローを発見することによる．このように，開発者やフレームワーク保守者が見落としがちな暗黙の共通制御フローを見つけ，その制御フローをテンプレートメソッドに隠蔽することは，フレームワークの

理解性の向上につながる．

以上より，本再構成手法は，フレームワークの利用および理解容易性という観点からも有効であるといえる．本再構成手法は，再構成前に存在していたメソッドの動作を保存したままで，開発者のメソッド変更履歴に基づく新たなインタフェースを提供する．また，どのコード断片が再構成可能であるのか，および，それらのコード断片がどのように再構成されるのかを実際に示すことで，開発者やフレームワーク保守者がフレームワークを再設計することを支援する．

## 6. おわりに

メソッド再構成はフレームワークをより再利用性の高いものに変えるという効果を持つ．しかしながら，その操作は人手で行うにはかなり複雑である．このため，メソッド再構成を自動化する手法およびツールは必須である．本論文では，過去のアプリケーション開発時のメソッド変更履歴に基づき，メソッド内部に存在する依存関係の強度に対して重みを割り付ける手法を示した．さらに，この重みを指標としてフローズンスポットとホットスポットを特定し，メソッドを自動的に再構成する手法を提案した．提案手法は，メソッド内部の制御フローをデータ依存関係と制御依存関係でとらえ，その制御フローの利用頻度という従来とは異なる観点でフレームワークを成熟化する．また，本再構成手法を用いることで，アプリケーション開発における実装の繰返しが軽減できることを，評価実験により示した．

提案した再構成手法については，その基本メカニズムを確立した段階であり，現実のフレームワークに対する妥当性や有用性に関しては，議論の余地が多く残されている．現在，以下の課題に対する考察を進めている．

(1) 本論文では，比較的単純なフレームワークを用いた実験結果を示したにすぎない．一般のフレームワークに対しても，同様の結果が得られるのかどうかを調べるためには，たくさんの種類のフレームワークに対して，評価実験を行う必要があり，本再構成手法を実装したツールは必須である．現在，Java ソースコードから WPDG を作成するツールや重みを計算するツールなど，いくつかのツールの実装を完了し，それらのツールを統合しているところである．さらに，フレームワーク全体の広域依存解析など計算量の大きい処理に対して，アルゴリズムの改良を行っている．

(2) フレームワークのフローズンスポットは，そのフレームワークを用いて開発するアプリケーションの性能や制約を内包している．逆に，ホットスポットは，変更が必要な部分を単純に表したものではなく，フレームワーク設計者が変更が行われると考えている部分や，変更してもかまわないと決定した部分を指している．さらに，デザインパターン[22]に関しては，そのパターンに対する目的や適切な利用方法が想定されている．本再構成手法は，このようなフレームワークが本来内包している性質を破壊することがある．この問題に対して，我々は，フレームワークの性質に応じて重み値の変動を調整することを考えている．また，ホットスポットの抽出にユースケース[24]を用いた文献 14) の手法を組み込むことで，要求や役割の変更に基づく重みを用いてメソッドを再構成することを考えている．

**謝辞** ご指導ご討論いただきました NTT 情報流通プラットフォーム研究所後藤厚宏リーダーならびに内藤昭三主任研究員に深く感謝します．

## 参考文献

1) Wirfs-Brock, R.J. and Johnson, R.: Surveying Current Research in Object-Oriented Design, *Comm. ACM*, Vol.33, No.9, pp.104–124 (1990).
2) Jonson, R.E.: Documenting Frameworks Using Patterns, *Conf. Object-Oriented Prog. Syst. Lang. Appl.* (*OOPSLA*), pp.63–76 (1992).
3) Koskimies, K. and Mössenböck, H.: Designing a Framework by Stepwise Generalization, *European Softw. Eng. Conf.* (*ESEC*), pp.479–498 (1995).
4) Opdyke, W.F.: Refactoring Object-Oriented Frameworks, Technical Report, Dept. Computer Science, University of Illinois at Urbana-Champaign (1992).
5) 荒野高志，中西弘毅，藤崎智宏：フレームワーク成熟化段階における抽象化の評価メトリクス，信学論，Vol.J79-D-I, No.10, pp.719–728 (1996).
6) Lieberherr, K., Holland, I. and Riel, A.: Object-Oriented Programming: An Objective Sense of Style, *Conf. Object-Oriented Prog. Syst. Lang. Appl.* (*OOPSLA*), pp.323–334 (1988).
7) Casais, E.: An Incremental Class Reorganization Approach, *European Conf. Object-Oriented Prog.* (*ECOOP*), pp.114–131 (1992).
8) Jonson, R.E. and Opdyke, W.F.: Refactoring and Aggregation, *Intl. Symp. Object Tech. Adv. Softw.* (*ISOTAS*), pp.264–278 (1993).
9) Moore, I.: Automatic Inheritance Hierarchy Restructuring and Method Refactoring, *Conf. Object-Oriented Prog. Syst. Lang. Appl.*

(*OOPSLA*), pp.235–250 (1996).
10) Dicky, H., Dony, C., Huchard, M. and Libourel, T.: On Automatic Class Insertion with Overloading, *Conf. Object-Oriented Prog. Syst. Lang. Appl.* (*OOPSLA*), pp.251–267 (1996).
11) 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).
12) 丸山勝久，島　健一：部品変更履歴に基づく重み付き依存グラフを用いた部品の洗練，情報処理学会論文誌，Vol.39, No.3, pp.725–740 (1998).
13) Pree, W.: *Design Patterns for Object-Oriented Software Development*, Addison-Wesley (1994). 佐藤啓太，金澤典子（訳）：デザインパターンプログラミング，トッパン (1996).
14) 友納正裕，中島　震：ハイパーメディア向けオブジェクト指向フレームワークの構築，情報処理学会オブジェクト指向'96 シンポジウム論文集，pp.175–182 (1996).
15) 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).
16) 戸松豊和：Java プログラムデザイン，SOFTBANK BOOKS (1997).
17) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers*: *Principles, Techniques, and Tools*, Addison-Wesley (1986).
18) Rothermel, G. and Harrold, M.J.: Selecting Regression Tests for Object-Oriented Software, *Intl. Conf. Softw. Maintenance*, pp.14–25 (1994).
19) Gosling, J., Joy, B. and Steele, G.: *The Java Language Specification*, Addison-Wesley (1996).
20) Click, C.: Global Code Motion/Global Value Numbering, *Conf. Prog. Lang. Design. Impl.* (*PLDI*), pp.246–257 (1995).
21) Horwitz, S., Prins, J. and Reps, T.: On the Adequacy of Program Dependence Graphs for Representing Programs, *Symp. Prin. of Prog. Lang.* (*POPL*), pp.146–157 (1988).
22) Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995). 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).
23) 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).
24) Jacobson, I., Christerson, M., Jonsson, P. and Övergaard, G.: *Object-Oriented Software Engineering*: *A Use Case Driven Approach*, ACM Press (1992). 西岡利博，渡邊克宏，梶原清彦（監訳）：オブジェクト指向ソフトウェア工学 OOSE：use-case によるアプローチ，トッパン (1995).

(平成 11 年 2 月 4 日受付)
(平成 12 年 4 月 6 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，立命館大学理工学部情報学科助教授．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受賞．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，(株) NTT ドコモ MM 事業本部担当部長．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，大規模 WWW の構築，モバイル環境でのユーザモデル研究，位置情報研究に興味を持つ．電子情報通信学会，人工知能学会各会員．