ソフトウェア工学 118−22
（1998. 3. 11）

オブジェクト指向設計におけるクリーンルーム手法

友納 正裕

NEC C&C メディア研究所

クリーンルーム手法は手続き型プログラムを対象としており、オブジェクト指向プログラムにそのまま適用することはできない。オブジェクト指向プログラムでは手続きとデータが一体化しており、手続き (メソッド) に対してだけでなく、データ的側面であるクラスに対しても、クリーンルーム手法の中心的枠組みである段階的詳細化と等価性検証が必要となる。本稿では、この点を踏まえて、オブジェクト指向設計に適したクリーンルーム手法を提案する。提案手法では、クラス構成の表現としてクラスグラフを導入し、クラスグラフに対する詳細化オペレータを定義して、詳細化前後での振る舞い等価性を検証する。これにより、クラスおよびメソッドの双方を段階的に詳細化しながら、検証レビューを行うことが可能になる。

An Extension of the Cleanroom Method for Object-Oriented Design

Masahiro Tomono

C&C Media Research Laboratories, NEC Corporation
1-1, Miyazaki 4-Chome, Miyamae-ku, Kawasaki, Kanagawa 216-8555, Japan

The Cleanroom method cannot be applied directly to object-oriented programs, since it focuses on only the procedural aspect of programs and provides no framework for class structure design. This paper extends the Cleanroom method for object-oriented design. The proposed method employs a graph notation to represent the class structure of an object-oriented program, and provides the framework of stepwise refinement on a class structure, defining the conditions to preserve program behaviors after refinement.

1 はじめに

クリーンルーム手法は高品質なソフトウェアを開発するための手法であり、テストとデバッグでバグを取り除くことよりも、設計段階でバグの混入を未然に防ぐことに主眼を置く。クリーンルーム手法の適用により、品質が一桁向上 (リリース後のバグ数の減少) し、生産性も向上したという報告がなされている [6]。クリーンルーム手法は複数の技術の総体であるが、形式検証技術に基づいた設計レビューに大きな特徴がある。形式検証技術は形式理論を用いてプログラム検証を行うものであるが、自動化は難しいため、設計レビューで人間が形式検証技術に基づいて検証を行うことでバグの大幅削減を実現する点がクリーンルーム手法のポイントである。

しかし、従来のクリーンルーム手法は手続き型プログラムを対象としており、オブジェクト指向プログラム設計にそのまま適用することはできない。クリーンルーム手法では、プログラムをボックスで表して段階的に詳細化していくボックス構造分析を用いる。この方法は、手続きの設計には適するが、データ設計には十分ではない。そのため、データと手続きがオブジェクトとして一体化しているオブジェクト指向プログラムには、データの側面をとらえる新しい方法が必要となる。

本稿では、オブジェクト指向設計に適したクリーンルーム手法を提案する。提案手法では、段階的詳細化による設計と形式検証技術によるレビューという枠組みを継承しながら、手続き (メソッド) だけでなくクラス構成に対しても段階的詳細化と検証の枠組みを与える。そこでは、クラス構成の表現としてクラスグラフを導入し、クラスグラフに対する詳細化オペレータを定義して、詳細化の前後での振る舞いの等価性を検証する。これにより、クラスおよびメソッドの双方を段階的に詳細化しながら、検証レビューを行うことが可能になる。

以下、2章において、従来のクリーンルーム手法の概要を述べ、3章でクリーンルーム手法をオブジェクト指向設計に適用するための枠組みについて述べる。4章でオブジェクト指向プログラムの段階的詳細化と検証方法の詳細について述べる。

2 クリーンルーム手法

クリーンルーム手法は、大きく、インクリメンタル開発、段階的詳細化、等価性検証、統計的テストという4つの要素技術から構成される [8]。

(1) インクリメンタル開発

開発すべきソフトウェアをインクリメントと呼ぶ部分ソフトウェアに分割し、インクリメントを単位として累進的に開発を進める。インクリメントは全体機能のサブセットであり、重要度、使用頻度、テスト容易さなどの観点から切り出される。インクリメンタル開発により、部分ソフトウェア単位で後戻りやプロトタイピングが可能になるため、開発プロセスが柔軟になる。

(2) ボックス構造分析による段階的詳細化

プログラムを関数と見なし、入出力を持ったボックスで表現する。ボックスには、ブラックボックス、ステートボックス、クリアボックスの3種類がある。ブラックボックスは仕様に対応し、入出力だけが規定されたボックスである。ステートボックスは内部状態と手続きからなる。ステートボックスの内部状態はブラックボックスへの入力履歴を保存するものであり、ステートボックスの出力は入力と内部状態から決まる。クリアボックスは、ステートボックス内の手続きを詳細化したものである。手続きの詳細化は、構造化プログラムの3つの基本構文である順次処理、条件分岐、ループへの分解により行う [7]。

上記のようなブラックボックス→ステートボックス→クリアボックスという詳細化を繰り返して、すべてがプログラミング言語で記述されたとき、設計・コーディングが終了する。

(3) 関数等価性の検証

ボックス構造分析による各詳細化の過程で、関数等価性の検証を行う。これは詳細化の前後におけるボックスが関数的に等しいかどうかを確認するものである。順次処理、条件分岐、ループへの分解の前後における関数等価性を、ホーア論理と同様の推論規則を適用することで検証する [1]。この自動化は一般に不可能であり、人間が設計レビューの形で行う点がクリーンルーム手法のポイントである。

(4) ユーザシナリオに基づく統計的テスト

作成されたプログラムに対してテストを行う。このテストはシステムテストに相当するもので、ユーザが実際にシステムを使用するシナリオに基づいて行われる。各機能の使用頻度を統計的に求め、テストケースを作成する点に特徴がある。このテストにより、検証レビューで漏れたバグ、モジュール間のインタフェースミスなどが摘出される。なお、単体テストは行わず、検証レビューで代替される。

本稿では、以上4つの要素技術のうち、段階的詳細化と等価性検証に焦点を絞り、オブジェクト指向プログラム設計への適用を検討する。

3 オブジェクト指向設計への適用

3.1 問題点

クリーンルーム手法における検証レビューは、プログラムの表現モデル、詳細化オペレータ、等価性検証の3つの要素から成る。従来のクリーンルーム手法では、各要素として、それぞれ、ボックス構造記法、構造化プログラムの基本構文(順次処理、条件分岐、ループ)への展開、関数等価性を用いている。

オブジェクト指向設計に対しても、同様の枠組みに沿って考えることができる。しかし、従来のクリーンルーム手法で用いた3つの要素は手続き型プログラムを対象としており、オブジェクト指向プログラムにそのまま適用することはできない。オブジェクト指向設計にはクラス設計とメソッド設計の2つの側面がある。メソッド設計は手続きの設計なので、従来のクリーンルーム手法と親和性がよい。だが、クラス設計はデータ設計の側面が強く、従来の枠組みをそのまま用いることはできない。

オブジェクト指向設計にクリーンルーム手法を適用するには、クラス設計に対する検証レビューの枠組みが必要になる。すなわち、クラス設計のためのプログラム表現モデル、詳細化オペレータ、検証方法が必要になる。さらに、クラス構成の詳細化によってメソッド内容が影響を受けるため、クラス設計とメソッド設計が連携可能でなくてはならない。

これらの要件を満たすために、オブジェクト指向プログラム[1]をグラフで表現し、そのグラフに対する詳細化オペレータと等価性検証を導入する。この枠組みは従来のクリーンルーム手法を包含する。概要を次節で述べ、詳細を4章で述べる。

3.2 枠組み

(1) プログラムの表現モデル

クラス構成はクラスグラフで表現する。クラスグラフは、UML [10]のクラス図を簡略化したもので、クラスをノードとし、継承関係および参照関係をエッジとしたグラフである。ノードはボックス構造分析における初期段階のボックスに対応し、詳細化の対象となる(3.3節参照)。

メソッドはメソッドグラフで表現する。メソッドグラフはメソッドの処理の流れを表す制御フローグラフであり、文やブロック(文の集まり)をノードとし、ノード間の遷移関係をエッジとする。ノードはボックス構造分析におけるボックスに対応し、詳細化の対象となる。

[1]本稿では、設計過程の仕様も合わせてプログラムと呼ぶ。

(2) 詳細化オペレータ

クラスグラフおよびメソッドグラフに、それぞれの詳細化オペレータを導入する。クラスグラフの詳細化オペレータには、クラスボックス分割を基本として、参照の張り替え、多相性の利用など良構造化に関するオペレータがある。クラスグラフを詳細化する際、等価性を保つためにメソッドグラフの変換も同時に行う。メソッドグラフの詳細化オペレータは、従来のクリーンルーム手法と同様に順次処理、条件分岐、ループへの展開が基本になる。この他、メソッドを分割するオペレータも用意する。

(3) 等価性検証

等価性検証は、メソッドグラフの振る舞い等価性に基づく。すなわち、クラスグラフおよびメソッドグラフの詳細化の前後で、メソッドグラフの振る舞い(環境に対する入出力)が等価であることを検証する。メソッドグラフの詳細化においては、従来のボックス構造分析と同様の方法で等価性をチェックする。クラスグラフの詳細化においては、等価性を保つためのメソッドグラフ変換が満たすべき条件を確認することで検証を行う。

3.3 設計プロセス

前節で述べた枠組みにしたがって、オブジェクト指向ソフトウェアを設計していくプロセスの概要を図1に示す。最初は、対象システム全体がブラックボックスであり、入出力関係(外部仕様)だけが規定されている(同図(a))。次に、1段詳細化してシステム全体を表すクラス1を作る(b)クラス1は属性とメイン関数だけをもつ。メイン関数はまだブラックボックスである。クラス1は従来のボックス構造分析でのステートボックスに対応し、属性がステートボックスがもつ内部状態、メイン関数が手続きに対応している。次に、メイン関数を分割して詳細化していく。詳細化につれてサイズが大きくなると、いくつかのメソッドに分割する(c)。属性やメソッドが増えると、クラス1を分割してクラス2を作り、クラス1からクラス2に属性とメソッドの一部を移動する(d)。以降、クラス分割やメソッド分割を繰り返して詳細化を進め、最終的なプログラムを得る。

なお、詳細化途中のクラス1やクラス2などは、本来の意味でのクラスと区別するため、クラスボックスとでも呼ぶべきであるが、本稿では簡単のため単にクラスと呼ぶことにする。

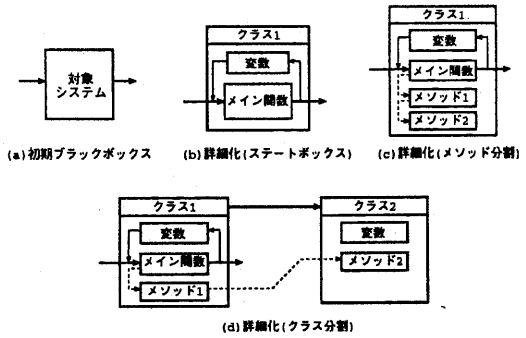

図 1: 設計の流れ

4 段階的詳細化と検証

4.1 表現モデル

クラスグラフおよびメソッドグラフを以下のように定義する。

定義 1 (クラス) クラス c は 3 項組 $< n_c, A_c, M_c >$ である。n_c はクラス名、A_c は属性の集合、M_c はメソッドの集合である。属性は、型、属性名、属性値からなる。基底クラスから継承した属性やメソッドは A_c や M_c に含めないが、オーバーライドしたものは含める。本稿では、属性はすべてインスタンス変数とし、クラス変数は考えない。簡単のため、アクセス権限はすべて public とする。 □

定義 2 (クラスグラフ) クラスグラフ CG は、2 項組 $< CV, CE >$ である。CV はクラスを表すノードの集合であり、CE はのエッジの集合である。ノードは、抽象クラス、具象クラス、基本クラス (Integer や Vector などシステムにあらかじめ組み込まれたクラス) などの種類に分かれる。エッジには、参照関係と継承関係の 2 種類がある。 □

定義 3 (メソッド) クラス c のメソッド $m(\in M_c)$ は 5 項組 $< n, arg, ret, var, body >$ である。n はメソッドの名前、arg は引数の集合、ret は返値、var はローカル変数の集合、$body$ は対応するメソッドグラフである。 □

定義 4 (メソッドグラフ) メソッド m のメソッドグラフ MG_m は 2 項組 $< MV_m, ME_m >$ である。MV_m はノードの集合であり、メソッドの開始ノードと終了ノードを必ず含む。ME_m はエッジの集合である。すべてのメソッドのメソッドグラフの和を MG で表す。$MG =< \bigcup_m MV_m, \bigcup_m ME_m >$ である。 □

簡単のため、本稿では単純な手続き記述言語を想定する。メソッドグラフのノードは、代入文、変数宣言文、if 文、while 文、メソッド呼び出し文、リターン文、メソッド開始文、メソッド終了文のいずれかである。代入文の右辺、メソッド呼び出しの実引数、if 文や while 文の条件部に項を許す。項は定数、変数、式、メソッド呼び出しから再帰的に構成される。また、メソッド呼び出しは値渡し、値返しとする。なお、設計初期には、ノード内容が自然言語で記述されることも多い。

4.2 等価性の定義

等価性検証は、振る舞い等価性に基づいて行う [4]。振る舞いとは、プログラムが環境に与える作用である。環境はプログラム中のすべてのオブジェクト (インスタンス) の集合であり、クラスグラフを反映したオブジェクトのグラフ構造をなす。プログラムの実行によって、オブジェクトの生成 / 削除や属性値の変更が行われ、環境が変化する。

クラスグラフの詳細化における振る舞い等価性は、以下のように定義する。いま、環境 S_1 においてメソッドグラフ MG を実行して環境が S_2 になることを $S_2 = Run(MG, S_1)$ と書く。クラスグラフの詳細化に伴って行われるメソッドグラフの変換を π、クラスグラフの詳細化に伴う環境構造の変化を表す関数を ω とする。以下の式が成り立つとき、クラスグラフの詳細化の前後で振る舞いが等価であるという (図 2参照)。

$$\forall S.\quad \omega(Run(MG, S)) = Run(\pi(MG), \omega(S))$$

メソッドグラフの詳細化に関しては、詳細化対象のノード v に対して Run を適用し、基本構文への展開ごとに等価性を定義する (次節参照)。

従来のクリーンルーム手法で用いていた関数等価性は、ボックスの入出力の等価性であった。振る舞い等価性をこれに対応づけるには、ボックスに対する入力ストリームオブジェクトと出力ストリームオブジェクトを環境に追加し、これらのオブジェクトに対して上式が成り立つことを検証すればよい。

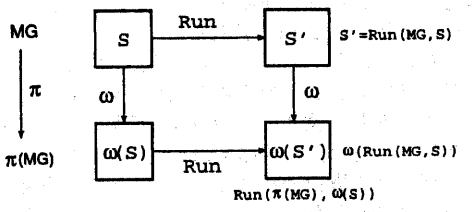

図 2: 等価性の定義

4.3 メソッドグラフの詳細化

従来のクリーンルーム手法に近いため、メソッドグラフの詳細化オペレータをクラスグラフより先に説明する。5つのオペレータがある。

(1) ローカル変数の抽出

メソッドグラフのノードvから、そのメソッド内で用いるローカル変数xを抽出する。vの内容をxを用いた記述v'に変更し、xの変数宣言文v_{dec}をノードv'の前に追加する(図3(a))。vの内容変更の例を以下に示す。たとえば、vが「変数aとbの値を交換する」であるとき、変数xを導入して、v'を「変数aの値をxに代入し、bの値をaに代入し、次にxの値をa代入する」とする。

等価性については、以下の式をレビューで確認する。

$$Run(v', S) = Run(v, S)$$

この式は、環境Sに対してvを実行することとv'を実行することが等しい、ことを述べている。

(2) 順次処理への展開

メソッドグラフのノードvを連続する2つのノードv_1, v_2に分割する(図3(b))。

等価性の式は以下のようになる。

$$Run(v, S) = Run(v_2, Run(v_1, S))$$

この式は、環境Sに対してvを実行することは、Sに対してv_1を実行して得られた状態にv_2を実行することに等しい、ことを述べている。

(3) 条件分岐への展開

メソッドグラフのノードvを条件分岐(if文)に展開する(図3(c))。v_{cond}は条件部、v_tはthen部、v_fはelse部、v_{end}はend-ifに対応する。

等価性の式は以下のようになる。

$$\begin{aligned} &if\ Eval(v_{cond}, S) = true \\ &\quad Run(v, S) = Run(v_t, S) \\ &else \\ &\quad Run(v, S) = Run(v_f, S) \end{aligned}$$

$Eval(u, S)$は、環境Sにおけるノードu(が表す式)の値を表す。この式は、環境Sに対してvを実行することは、Sにおけるv_{cond}の結果がtrueの場合はv_tを実行することに等しく、falseの場合はv_fを実行することに等しい、ことを述べている。ただし、v_{cond}は環境Sに影響を与えないものとする。

(4) ループへの展開

メソッドグラフのノードvをループ(while文)に展開する(図3(d))。v_{cond}は条件部、v_{body}はループ本体、v_{end}はend-whileに対応する。

等価性については、ループが終了することと以下の式をレビューで確認する。

$$\begin{aligned} &if\ Eval(v_{cond}, S_n) = true \\ &\quad S_{n+1} = Run(v, S_n) = Run(v_{body}, S_n) \\ &else \\ &\quad Run(v, S_n) = S_n \end{aligned}$$

$n \geq 0$で、S_0はwhile文に入る直前の環境とする。この式は、環境S_nに対してvを実行することは、S_nにおけるv_{cond}の結果がtrueの場合はv_{body}を実行することに等しく、falseの場合は何も実行しないことに等しい、ことを述べている。ただし、v_{cond}は環境S_nに影響を与えないものとする。

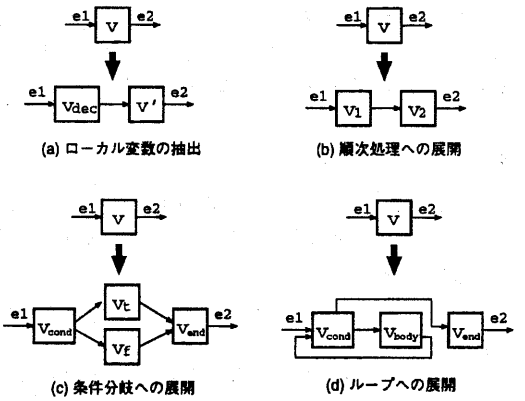

図3: メソッドグラフの詳細化

(5) メソッドの抽出

クラスc_1のメソッドm_1について、m_1のメソッドグラフのノードvの処理内容を新しいメソッドm_2として抽出し、vをメソッド呼び出しノードv_{call}で置換する。v_{call}では、実引数を渡してm_2を呼び、返値を所定の変数に代入する。

変換は次のように行う。まず、新メソッドm_2を生成する。m_2の引数argとして、vに入る前に値が定義され、vの中で定義/参照される変数からクラスc_1の属性を除いたものを設定する。m_2の返値retとして、vの中で値が定義されてvを出てからm_1で参照される変数からc_1の属性を除いたものを設定する。m_2のローカル変数varとして、vだけで定義/参照される変数を設定する。これらをもとに、m_2のメソッドグラフを定義する(図4)。

v_{etr} はメソッドの入口、v_{var} はローカル変数宣言、v' は v で変数名を適宜変換したもの、v_{ret} は返値のリターン、v_{end} はメソッドの出口を表す。次に、メソッド m_2 をクラス c_1 に追加する。最後に、m_1 のメソッドグラフからノード v を削除し、m_2 を呼び出すノード v_{call} をその位置に挿入する。

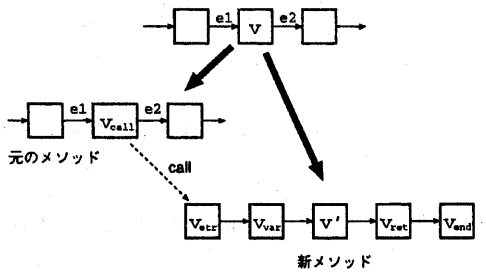

図 4: メソッドの抽出

この変換では等価性は保たれる。上述した m_2 の引数と返値の設定方法により、詳細化の前後で環境 S に与える影響は変わらないためである。

4.4 クラスグラフの詳細化

クラスグラフには種々の詳細化オペレータがある。本稿では代表的な 5 個のオペレータを説明する。

(1) 属性の抽出

クラス c のメソッドグラフのノード v から属性 a を抽出する。c に a を追加する。

属性 a の追加により環境の構成要素が増えるため、等価性は以下の包含関係により定義する。

$$Run(v, S) \prec Run(v', S')$$

S' は属性 a が増えた後の環境である。$S_1 \prec S_2$ は、環境 S_2 に含まれるデータが環境 S_1 に含まれるデータを包含することを意味する。

(2) 基底クラスの抽出

n 個のクラス $c_1, .., c_n$ の共通部分をもつ基底クラス c を生成する。継承によるクラスのコンパクト化や多相性の利用のために行う。

変換は次のように行う。まず、基底クラス c のノードを生成し、クラスグラフ CG に追加する。c の属性は、派生クラス c_i のもつ属性の共通集合の部分集合とする。この部分集合は設計者が選択する。メソッドも同様である。次に、新しいエッジ e_i $(i = 1, .., n)$ を CG に追加する。e_i の起点は c_i、終点は c である。最後に、基底クラスに移動した属性とメソッドを各派生クラス c_i の属性とメソッドから削除する。

この変換では等価性は保たれる。基底クラス c を参照するものはまだなく、派生クラス c_i への参照は変わらないためである。

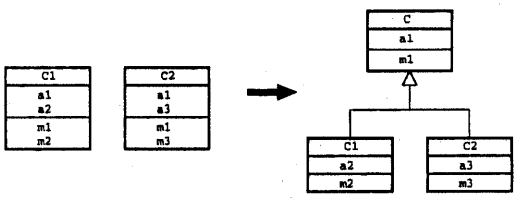

図 5: 基底クラスの抽出

(3) 被委譲クラスの抽出

クラス c_1 を分割してクラス c を生成する。c_1 と c は参照関係をもち、c_1 は c に処理の一部を委譲する。本稿では、c_1 を委譲クラス、c を被委譲クラスと呼ぶ。

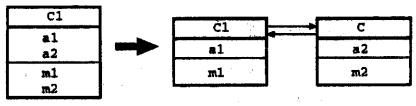

図 6: 被委譲クラスの抽出

クラスグラフの変換は次のように行う。まず、被委譲クラス c のノードを生成し、クラスグラフ CG に追加する。また、c_1 の属性とメソッドの一部を委譲クラス c に移動し、c が c_1 を参照するための属性 a を c に追加する。次に、c_1 から c を参照するための属性 a_1 を c_1 に追加する。最後に、c_1 から c への参照エッジと c から c_1 への参照エッジを CG に追加する。

メソッドグラフの変換は次のように行う。c_1 のインスタンスを o_1、c のインスタンスを o とする。まず、o の生成文、o を o_1 の属性 a_1 に代入する文、o_1 を o の属性 a に代入する文、を表すノードをそれぞれメソッドグラフ MG に追加する。次に、属性アクセス記述の変更を行う。c_1 から c に移動した属性に対するアクセス主体を o_1 から o に変え、c のメソッドにおいて c_1 に残った属性のアクセス主体を o_1 にする。次に、メソッド呼び出し記述の変更を行う。c_1 から c に移動したメソッドに対する呼び出し主体を o_1 から o に変え、c のメソッドにおいて c_1 に残ったメソッドの呼び出し主体を o_1 にする。

等価性検証では、メソッドグラフの変換において以下の条件を確認する。c のインスタンス o がア

クセスされる前に o が生成されること。c_1 のインスタンス o_1 の属性 a_1 がアクセスされる前に、a_1 に o が代入されること。o の属性 a がアクセスされる前に、a に o_1 が代入されること。後者2つの検証には、o や o_1 の代入文ノードが、それらのアクセスするノードすべての支配ノード[2]になっていることを確認すればよい。

(4) 参照の張り替え

クラス c からクラス c_1 への参照を、クラス c_2 への参照に変更する。参照パスが長くなるのを避けるために行う。

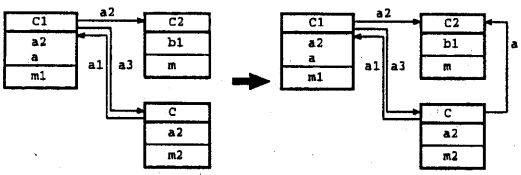

図 7: 参照関係の張り替え

変換は次のように行う。まず、新属性 a をクラス c に追加する。型は最終参照先 (図7で c_2) とする。次に、a に c_2 のインスタンスを代入するノードを追加する。最後に、全メソッドグラフ中で $a_1.a_2$ によるアクセス記述を a によるアクセス記述に変更する。

等価性については、a への値代入が a へのアクセスより先に行われることをレビューで検証する。

(5) 条件分岐から多相性利用への変換

クラス c_0 のメソッド中の条件分岐記述 $b_1, ..., b_n$ を、クラス $c_1, .., c_m$ とその基底クラス c の多相性を利用した記述に変換する (図8)。各 b_i $(i = 1, .., n)$ は分岐処理を m 個もち、b_i の j 番目の分岐における条件部を $cond_j$(各 i で同じ)、実行部を $body_{ij}$ と表す $(j = 1, .., m)$。また、各 $cond_j$ の値はプログラム実行中に変化しないとする。

クラスグラフの変換は次のように行う。まず、各 $body_{ij}$ を c_0 のメソッド m_{ij} として抽出する。次に、被委譲クラス c_j を生成し、m_{ij} を c_j に移動する。このとき、m_{ij_1} と m_{ij_2} のメソッド名が同じになるようにメソッド名をに変える (これを m_i とする)。また、必要に応じて、他のメソッドや属性の一部を c_j に移動する。最後に、クラス $c_1, .., c_m$ の基底クラス c を抽出する。これは、(2) で述べた方法で行う。

メソッドグラフの変換は次のように行う。まず、基底クラス c のインスタンス ob を条件 $cond_j$ に合わせて派生クラス c_j 化して生成するノードを追加する。次に、c_0 のメソッドグラフから条件分岐記述 $b_1, .., b_n$ を削除し、その位置に ob がメソッド m_i を呼び出すノードを挿入する。最後に、c_j に移動したメソッドの呼び出し主体や属性のアクセス主体の変更を行う。これは、(3) で述べた方法で行う。

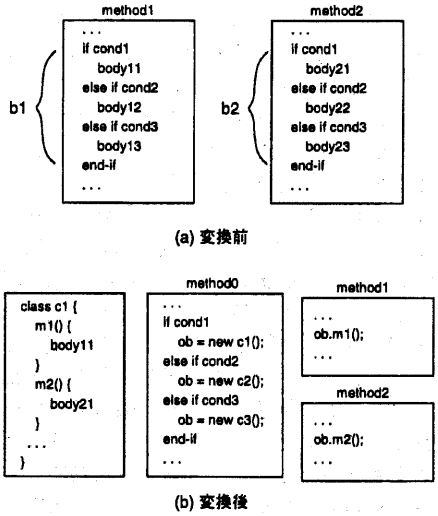

図 8: 条件分岐から多相性利用への変換

等価性検証では、メソッドグラフの変換において以下の条件を確認する。c_j のインスタンスの c_0 への設定がメソッド m_i の呼び出し記述より先に行われること。条件分岐記述 $b_1, ..., b_n$ が1つに集約されて位置が移動するが、その変換によって条件式の値が変わらないこと。

5 考察

従来のクリーンルーム手法との違い：本稿で述べた提案手法と従来手法の本質的な違いは、クラス構成の詳細化にある。文献 [2] において、クリーンルーム手法とオブジェクト指向の関係が論じられているが、そこでは単純にオブジェクトをボックスに対応づけている。ボックスもオブジェクトも内部状態と手続きをもつため、初期段階では対応は一見よい。しかし、詳細化を進める段階になると、ボックス構造分析は手続きの詳細化しか用意していないた

[2] ノード v に対する支配ノードとは、プログラムの開始ノードから v に到達するすべての経路に含まれるノードのことを言う。

め、クラスの詳細化をしようとしたところで行き詰まる。提案手法では、クラスとメソッドの表現モデルを分け、メソッドには従来と同様の詳細化オペレータを用意し、クラスには別の詳細化オペレータを用意することで、この問題を解決している。

オブジェクト指向分析・設計との関係： 3.3節で述べた設計プロセスは、初期ブラックボックスへの入出力をユースケースと捉えると、ユースケース分析 [5] と関連づけられる可能性がある。ユースケースからクラスの役割を抽出し、クラスの詳細化へ結びつける枠組みは興味深い課題である。

プログラム解析の応用： クラスグラフ詳細化に伴って等価性を保つために行うメソッドグラフの変換では、プログラムの静的解析技術が応用できる。たとえば、被委譲クラスのインスタンスの生成 / 定義が参照よりも先であることを保証するには、支配ノードの解析やデータフロー解析が使える。また、変数や式の値の等価性、副作用のチェックなどの支援には、プログラムスライシング技術が有効である [3]。これらに関しては、別の機会に報告する。

クラス構成の変換： クラス構成の変換に関する研究として、FAMOUS [4]、Refactoring [9] がある。FAMOUS は適応的ソフトウェアを実現する枠組みであり、クラス構造を変更したときに手続きや永続オブジェクト構造を変更して振る舞いの等価性を保証する手順を与えている。しかし、FAMOUS における手続き記述はクラス構造を自動的にトラバースできる独自の言語を用いているため、通常のオブジェクト指向言語におけるメソッドとは振る舞いの等価性を保証する方法が異なる。Refactoring は、クラス良構造化のためにクラスを再構成する技術である。本稿で述べたものと同様の変換を提供しているが、自動化を指向しているため制約条件が強く、本稿のものより変換能力は弱い。また、両者ともプログラムの設計手法ではなく再構築手法であり、段階的詳細化、すなわち抽象度の異なるレベルへの変換 (メソッド詳細化全般、属性の抽出など) は用意していない。

6 まとめ

オブジェクト指向プログラムに適したプログラム表現モデル、詳細化パラメータ、等価性検証の方法を提案し、オブジェクト指向設計においてクリーンルーム手法を適用するための枠組みを示した。クラス設計のための表現モデルと詳細化オペレータを導入した点が、従来のクリーンルーム手法と本質的に異なる。今後の課題として、詳細化オペレータの体系化、実プログラムへの適用試行などがある。

謝辞

本研究に際して有意義な討論をしていただいた情報処理学会ソフトウェア工学研究会クリーンルーム WG のメンバの方々、および、NEC C&C メディア研究所基盤ソフトウェア TG の方々に感謝いたします。

参考文献

[1] Dyer,M., Cleanroom Approach to Quality Software Development, John Willey & Sons, Inc., (1992).

[2] Hevner,A.R., Mills,H.D., Box-structured methods for systems development with objects, *IBM Systems Journal*, Vol.32, No.2, pp.232-251, (1993).

[3] Horwitz,S., Reps,T.,and Binkley,D., Interprocedural slicing using dependence graphs, *ACM Trans. Program. Lang. Syst. 12, 1*, (Jan.1990), 26-60.

[4] Hursch,W.L., Seiter,L.M., Automating the Evolution of Object-Oriented Systems, *Proceedings of ISOTAS'96*, (Mar.1996), 2-21.

[5] Jacobson,I.,Christerson,M.,Jonsson,P. and Overgaard,G. Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley,(1992).

[6] Mills,H.D., Dyer,M., and Linger,R.C., Cleanroom Software Engineering, *IEEE Software*, vol.4, no.5, pp.19-24, (Sep. 1987).

[7] Mills,H.D., Stepwise Refinement and Verification in Box Structured Systems, *IEEE Computer*, vol.21, no.6, pp.23-35, (June 1988). pp.232-251, (1993).

[8] 西橋、ソフトウェア開発クリーンルーム手法 – その概要と留意点 –、電子情報通信学会誌 Vol.80, No.5, pp.470-478,(May 1997).

[9] Opdyke,W.F., Refactoring Object-Oriented Frameworks, PhD thesis, University of Illinois at Urbana-Champaign,1992.

[10] Rational Software Corp., UML resource center, http://www.rational.com/uml/index.html.