ソフトウェア工学 105－6
（1995．9．8）

再利用性を高めるためのクラスの変形

三ッ井 欽一
mitsui@trl.ibm.co.jp

日本アイ・ビー・エム株式会社 東京基礎研究所

本論文では、再利用性を高めることを目的としたクラスの変形の指針となる4つのルールを議論する。クラスの設計は一般に自由度が高すぎ、また設計者は継承を過大に使用している。また、クラスは十分に分割されていない。その結果、プログラムは理解しにくく、再利用においても柔軟性に欠ける。オブジェクト指向プログラムの再利用のための変形は幾つか議論されているが、改良の過程そのものは依然として明確にはなっていない。我々の提案するルールはクラスの設計者が利用できる明確な改良の指針を目指したものである。

Reorganizing Classes for High Reusability

Kin'ichi Mitsui

IBM Research, Tokyo Research Laboratory

This paper discuss four rules for designing highly reusable classes. This discussion is motivated by the fact that designers are given too much freedom and tend to use inheritance excessively, and that classes are usually insufficiently decomposed, and are consequently hard to understand and inflexible for reuse. Several transformations of object-oriented programs intended for design refinement have already been discussed. However, the refinement processes required of designers are still unclear. Our rules offer designers concrete guidance in understanding these processes.

1 はじめに

再利用性の高いソフトウェアを設計するのは難しいとよく言われる。オブジェクト指向技術は、この問題を解決するものとして高い期待がある。しかし、設計の問題は依然として、設計者のスキルに大きく依存しているのが現状である。

多くの場合、再利用性は、評価と改良を繰り返すことによってのみ得られる。Johnson と Foote[JoFo88] は、再利用性を高める設計指針として、例えば、「大きなクラスは分割せよ」といった幾つかのルールを整理している。しかし、今のところ、より明確で標準的な設計指針はなく、各設計者がアドホックに設計を行っている。そのため、再利用性が設計者のスキルに依存してしまう。また、この改良の過程は、ソフトウェア全体の理解とエラーを入れない注意深さを必要とするので、設計者にとっても負担が大きい。

Opdyke[Op92] は、挙動を保存するオブジェクト指向プログラムの変換を Refactoring と呼び、その形式化を行い、上記の改良の過程の機械化のための指針を与えようとしている。しかし、改良の過程は、全部が機械化できるほど単純なものとは考えられず、Refactoring の議論の中でも、どのような場合にどのような変換規則を適用すべきであるかという点は、まだ議論の余地を残している。

このような状況の中で、我々は、プログラマが行なっている改良の過程を分析、整理することが重要であると考えている。また、得られた設計の指針は、機械によるサポートが得られるように形式化されていることが望ましい。本論文では、プログラム開発中に見られる再利用性を高くすることが期待できるクラスの変形操作の幾つかを議論する。

2 例

プログラマは、繰り返しによるプログラム開発の際に、プログラムの改良のための変形を多く行っている。ここでは、例を用いて、設計上起こるクラス設計の問題点やそれに対処する変形のうちあまり自明でない例を挙げ、このようなクラスの変形の過程を明確にし整理することが重要であることを指摘したい。

なお、本論文では、静的な型づけを行なうプログラム言語を前提にしている。以降用いる用語は C++[StEl90] のものを用いることがある。例えば、メソッドをメンバー関数、再定義 (override) 可能なメソッドを仮想関数、インスタンス変数をメンバー変数、スーパークラス／サブクラスは、それぞれ基本クラス／導出クラスと呼ぶ場合がある。

例 1

アイコンのドラッグ操作に見られるようなアニメーションを行なう描画オブジェクトを考える。このオブジェクトは、背景画の一部保存、描画、背景画の修復を繰り返すことにより背景画上に動画を描く。今、背景画の保存、修復の方法として複数の方法を使い分けたいとする。(例えば、イメージ保存と排他 OR 描画等) そこで、クラスは、図 1のような設計になる。抽象クラスのレベルで、一般的なアニメーション管理を記述し、具体的な描画方法は、描画オブジェクトの種類により異なることになる。

```
class AnimationObject {
 public:
  ...
  virtual void saveBackground();
  virtual void draw();
  virtual void restoreBackground();
};
```

図 1:

ここで、別な要求としてアニメーション中の何らかの挙動 (例えばマウスカーソルの形を変える等) を記述できるように拡張するために、図 1の定義を拡張して図 2のようし、アニメーションの直前と直後に呼び出される仮想関数を追加したとする。

さて、この設計にどのような問題があるのだろうか。例えば、あるアニメーションオブジェクトにおいて、特定の描画方法で、特定のアニメーション中の挙動を表現するためにクラスを二つ作成して図 3(a) のような継承構造になったとする。しかし、描画方法とアニメーション中の挙動は独立に設定できるとすると、この設計では別の描画方法が同じアニメーション中の挙動を共有することができない。そこで図 3(b) のように多重継承による構造に変更

```
class AnimationObject {
 public:
  ...
  virtual void saveBackground();
  virtual void draw();
  virtual void restoreBackground();
  virtual void beginAnimation();
  virtual void endAnimation();
};
```

図 2:

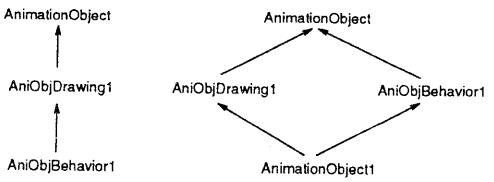

図 3:

する。しかし、この多重継承の設計では、機能の組合せごとに新しいクラスを生成する必要があり、クラスの数の組合せの問題を起こす可能性がある。クラスおよび継承は、設計上の何らかの抽象を表現したものと考えられるが、この組合せのためのクラスや継承は、設計の本質的な部分ではなく、プログラムを必要以上に複雑にしている。この問題は、後述するように機能の組合せをクラスの組合せではなく、クラスのインスタンスの組合せにすることにより解決できる。この例では、抽象クラスを二つに分け、継承階層が二つになるようすればよい。

例 2

上位のクラスがインスタンス変数を持つ場合で不都合が起きる場合がある。いま、図4のようなクラス間の関係があり、サブクラスA'のメソッドが導出クラスB'のメソッドにアクセスする必要があるとする。この場合、A'はB'への参照のために冗長なインスタンス変数を持つか、または、A'はBからB'への型の下方キャスト (downward cast) をする必要がある。下方キャストは、言語によって

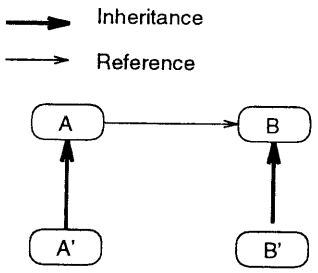

図 4:

は制限されたり安全でなかったりする。この問題に対処するには、AからBへの参照をインスタンス変数ではなくメソッドで表現し、実際のインスタンス変数は下位のクラスに移動する。

例 3

我々の分析 [MiNa94] によると、クラス間の静的な依存関係グラフにおいて強連結成分 (strongly connected component) を計算すると、それが非常に大きい場合が多く観察される。静的な依存関係とは、強い型付けをする言語の場合で、あるクラスの実現から別のクラスのインタフェースへの参照がある、または、あるクラスが別のクラスから継承する、ことと定義する[1]。

強連結成分の意味するところは、それに含まれる任意のクラスどうしは、直接または間接の依存関係をもっており、それらは一つの単位として利用されるということである。言い替えれば、強連結しているクラス群の一部のみを利用することはあり得ないということである。強連結成分が大きすぎるということは、再利用の単位になりうるものが複数混在している可能性があるということである。そのようなことは、再利用を十分意識せずに実現されたプログラムにおいてはよく起こることである。例えば、プログラムの様々な部分からアクセスされる大域変数を集めたクラスを定義すると、このクラスを介して見かけ上多くのクラスが依存関係を持ってしまうことがある。それぞれの部分を個別に再利用したければ、結局このクラスを分解しなければな

[1]オブジェクト指向言語では、呼び出されたメソッドのシグニチャとメソッドの本体の結合は動的に行うことができる。このように実際の制御の流れは静的依存関係と異なる場合がある。これを動的な依存関係として区別する。

らない。

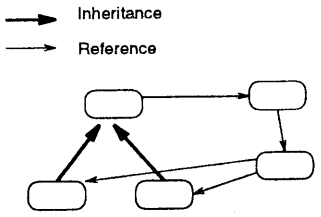

図 5: 強連結したクラス群と継承

強連結成分に関して観察される興味深い例は、その中に含まれるクラス間に継承関係がある場合である（図 5)。継承関係は、抽象度の違いを表しており、抽象度の高いクラス及びそれに関連したクラス群が実現している機能を、より具体的なクラスが具体化することにより再利用する、と考えるのは自然であろう。ところが、この両方のクラスが同一の強連結成分に含まれるということは、抽象クラスから具体クラスへの静的な依存関係があることになる。すなわち、抽象クラスは具体クラスから十分分離されていない。これはいづれかのクラスが十分に分割されていないことを示しており、実際のプログラムの中でもしばしば観察される。継承関係は、何らかの抽象を表現しており、それがプログラムの静的な構造上明確に分離できるべきである、というのが我々の主張である。

このような場合のクラスの変形は、抽象度の異なるものが混在するクラスをさがし、それを分割し、継承または集約関係で分割したクラスを関連づけることである。ただし、これは労力のかかる作業である。

分析

多くのオブジェクト指向方法論において、分析の初期段階においてオブジェクトの見つけ方が論じられている。オブジェクトは観察される現実世界の対象に対応させられることが多い。一つのオブジェクトのクラスには、再利用の観点からすると複数の異なる抽象が含まれていることがある。しかし、この初期段階では、積極的に適当な抽象を見つけクラスを分割することはない。

理想的には、分析、設計を繰り返すことにより、アプリケーションの各所またはやや異なるアプリケーションでたびたび用いることができる、あるいは、将来の変更に対して安定している抽象が明かになる。そこで、クラスの、抽象として利用できる部分とそうでない部分を分離する。ところで、そのように分解した結果が、十分でなかったり、あるいは、分解のしかたに問題があったりすると、上述した例のような問題を起こすことがある。その結果、再利用が困難であったり、再利用時に問題が発覚したりする。

3 クラスの変形の指針

ここでは、クラスの改良のための変形を考慮する際の指針となるようなルールを幾つか提案する。

3.1 多相性と継承

ルール：継承は、多相性を利用している場合に限定する。

このルールは、クラス間に継承関係がある場合、サブクラスにおいて少なくとも一つのメソッドが上位クラスのメソッドを再定義していることを要請する。

継承は、再利用の観点において強力な機構である。特に多相性 (polymorphism)[2]を用いることにより、いわゆるフレームワーク [De89] と呼ばれる一般化したソフトウェア部品を再利用時に具体化するという方法でプログラムを構成することができる。継承は様々な使い方がある [HaOB87] が、多相性をもちいない場合は、集約 (aggregation) と呼ばれる形に書き換えることができる。一般に、プログラマは継承をオブジェクト指向概念の必須の要素と考えて多用する傾向があるが、継承の利用を多相性に限定することにより、再利用時にプログラマが下すべき決定の数を減らし、プログラムをより標準化するのに役立つ。この結果、プログラムの理解もより容易になると期待できる。

なお、同様の議論は、[Car91] における多重継承の有効性の議論の中にも見られる。

[2]多相性は通常メッセージとメソッドの動的結合という形で利用される。それ以外でも、multi-method[Bo86] の機構のない言語ではしばしば型の下方キャストを必要とするが、このときも多相性を用いていると考える。

3.2 多相性の一貫性

ルール：ある継承階層に互いに独立な多相性の利用が含まれないように継承階層を分割する。

このルールは、直観的には、ある継承階層中に多相性の利用により再定義されるメソッドが複数あったときに、それぞれのメソッドの定義の間に強い依存関係がなく、それぞれのメソッドの定義の組合せが可能な場合に、この組合せを単一の継承階層上で表現すべきでないことを要請する。

前述の例1の問題は、クラス中の仮想関数が複数のグループに分かれ、各々の導出クラスにおける実現がグループ間では独立に決めることができたために起きた。これに対処するために、グループごとに別の継承階層を持つように設計を再構成することができる。

まず、ある継承階層中に複数種類のメソッドのシグニチャがあった場合、その間の依存関係を決める。例1は、仮想関数 saveBackground の実現が決まれば、仮想関数 restoreBackground の実現も一意に決まる、と仮定できたとして、これを依存関係と呼ぶ。この例の場合、逆方向の依存関係もある。この依存関係のグラフにおいて強連結成分によりグループ分けをする。一方向のみの依存関係の場合は、別グループに分ける。例1では、メソッドはふたつのグループに分けられる。

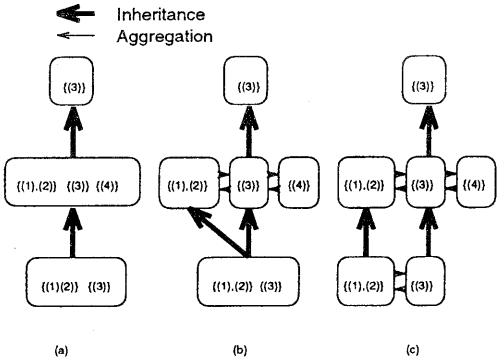

図 6: 継承階層の分割

図6は、やや複雑な例でどのように変形がなされるかを示している。矩形はクラスを、(1) から (4) はメソッドシグニチャを、{} はメソッドシグニチャ間の依存関係でのグループを表している。図 6(a) では、3つのクラスが継承階層をなしており4種類のメソッドが定義されている。メソッドは3つのグループに分かれる。図 6(b) では中段のクラスをグループごとに3つに分解している。分解されたクラスどうしは参照し合う可能性があるのでその場合は各クラスに相手のクラスを参照するインスタンス変数を追加し、オブジェクトの初期化の時点でその変数の初期化をする。図 6(c) では、更に下段のクラスの分解を同様に行なっている。ここで、クラス間の参照のうち冗長なものは取り除く。継承は上位のクラスの分解に応じて適当に設定される。この例では、変形の結果、3つの継承階層が形成された。

さて、あるクラスにこのメソッドのグループが複数含まれていても、先の組合せの問題が起こらない場合がある。それはメソッドの定義本体の集合が継承階層上で分岐しない場合である。このような場合にはメソッド間に依存関係がなくてもクラスを分割しないとすると、そのメリットは、クラスの数が増えることによる複雑さをさけることができることである。したがって、このようなオプションは考慮に値する。

メソッド間の依存関係や定義本体の継承階層における分岐可能性は、継承の上位クラスの設計者から導出クラスの利用者に対する一種の制約条件の表明とみなすことができる。従来の設計では、そのような制約条件が明確に表明されない結果、利用者がクラス階層を拡張していく段階で組合せの問題や継承階層が複数の抽象を含むことによる理解しにくさの問題が起こる。上位クラスの利用者は、必ずしも上位クラスの設計者とは同じでなく、上位クラスの設計変更の権利を持っていないのが普通である。したがって、上位クラスの設計者が注意深く設計を行ない、その意図するところを明確に伝え、クラス階層が無秩序に拡張されるのを避けなければならない。

3.3 インスタンス変数のメソッド化

ルール：あるクラスとそのインスタンス変数のクラスが含まれる継承階層どうしが独立でないときはインスタンス変数へのアクセスをメソッド呼び出しに置き換える。

例2の問題は、上位のクラスがインスタンス変数の型を決定してしまっていることに起因する。参

照を上位のクラスでは下位クラスで再定義されるメソッドにし、実際のインスタンス変数の割り当てと変数の型の決定を最下位のクラスに移せば解決できる。もちろん、導出クラスどうしでの参照が無いならば、上位のクラスにインスタンス変数を割り当てても問題はない。このような場合、それぞれのクラスが含まれる継承階層どうしは独立であると呼ぶ。

上位のクラスの設計者は、インスタンス変数を通して参照関係にある継承階層どうしが独立であるかを注意深く検討してクラスの設計をしなければならない。

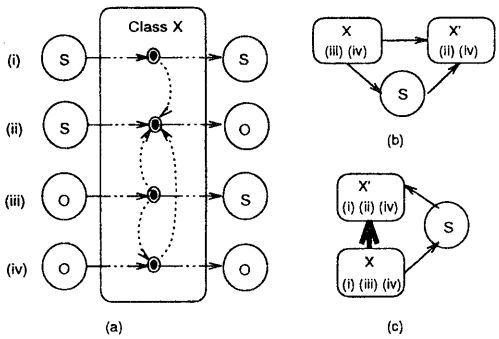

図 7: 静的依存関係のサイクルの除去

3.4 サイクルの無い静的依存関係

ルール： クラス間の静的依存関係のサイクルが無くなるまでクラスを分解する。

例3の問題が示すように、再利用性を考えれば、一般にクラス間の静的依存関係グラフの強連結成分が大きくなりすぎないようにすべきである。極端な場合として、強連結成分の大きさを1、すなわち、依存関係のサイクルが無い場合を考えてみる。

クラス間に静的依存関係のサイクルが無いと、プログラムの構造が、よりアプリケーションに依存した部分からより一般的な部分へと展開する階層構造になり、サイクルがある場合に比べて理解しやすくなる。また、例3で議論した継承階層の上位にある抽象度の高いクラスからその下位にある抽象度の低いクラスへの静的依存関係はサイクルを生ずるが、このルールによりそのようなものが無いことを保証できる。

もちろん、小さい強連結成分は（理解しやすさの意味で）許容できる場合も多いが、このルールは、上記のような利点を考えると、クラスを改良する過程で用いる標準として考慮に値すると考える。

このクラスの分解は、次のように行なうことができる。一般に、ある強連結成分に含まれる任意のクラスは、分解することによりその強連結成分からはずすことができる[3]。これを図7で示す。

Xは強連結成分の中の任意のクラスとする。(i) から (iv) は、そのクラスのメンバー（メソッドまたはインスタンス変数でクラス内に点で表されている）が4種類に分類できることを表している。クラスの外のSおよびSとメンバーとの間の矢印は同じ強連結成分に含まれる別のクラス（群）との直接または間接の参照関係があることを表している。Oは同じ強連結成分に含まれる別のクラス（群）との直接または間接の参照関係が無いことを表している。この場合、強連結成分外のクラスとの参照関係があっても無くてもよい。クラス内の矢印は、メンバー間での可能な参照関係を表している。

もしも、(i) に含まれるようなメンバーが無い場合、図7(b) のように、新しいクラスX'を生成し、(ii) に分類されるメンバーをX'に移す。(iv) に含まれるメンバーは、X、X'のどちらに含めても良いが、X、X'間の依存関係にサイクルができないように、かつ、クラス間を交差する参照が最小になるように振り分けるのが妥当であろう。この変形の結果、図7(b) に示すようにXおよびX'は元の強連結成分には含まれない。

もしも、(i) に含まれるようなメンバーがある場合、その幾つかのメンバーへの静的なメソッド呼び出しを動的結合によるメソッド呼び出しに変更する。すなわち、いま、(i) に含まれるあるメソッドが選択されたとすると、これを二つに分離し、一方はSからの呼び出しを受けるシグニチャのみからなるものとし (ii) に含め、もう一方は、メソッド本体がSへの呼び出しをもつものとして (iii) に含めることにする。この操作を (i) に含まれるメンバーが無くなるまで繰り返すと、前の (i) に含まれるメンバーが無い場合のクラスの変形に帰着できる。このクラス変形の様子を図7(c) に示す。ここで、XとX'の間に継承関係（太い矢印）が導入されている

[3]但し、異なるクラスのメソッド間で相互再帰呼び出しがある場合は例外である。

ことに注意する。

さて、上記の操作において、静的なメソッド呼び出しを動的結合によるメソッド呼び出しに変更する対象となったメソッドや変形の対象としたクラスXの選択は、設計者に委ねなければならない。この選択の順序によって結果として得られるクラスの構造は異なったものになる。クラスの変形の妥当性としては、この分解により設計者の意図する抽象度の違いがクラスの違いにも反映され、かつ、必要以上に分解しすぎることがないことが要求されよう。クラスの単位が細かくなり過ぎるとプログラムのモジュール性が不明瞭になり、プログラムの複雑さを増すことになってしまう。抽象度の違いをプログラムに記述する方法がない以上、上記の変形操作は、完全に自動化することはできない。むしろ、分解の候補の探索をある程度機械が自動的に行ない、設計者が対話的に選択をするというかたちが現実的であろう。

4 議論

オブジェクト指向プログラムの再利用性を高めるための改良には、名前の付け方から構造的な再構築まで様々なものがありうるが、本論文で述べた指針の焦点は次のようにまとめることができる。

- 継承の使い方を制限することにより、それぞれの継承階層の意図するものが利用者に理解しやすいようにプログラムを構成しなおす。また、上位のクラスの設計者が、下位のクラスの作成者に対する幾つかの表明をすることで、クラス階層が無秩序に拡張されていくのを防ぐ。

  一般に、いわゆる組み立て (construction または composition) 方式の再利用に比べてサブクラス方式は難しいと言われる [TGP89]。理由は、サブクラスをどのように定義すべきかが十分に文書化されない傾向があり、それを理解するために継承内部の制御フローを追跡するのが容易でないためである。継承の利用を制限し不要な複雑さを避けること、また、サブクラスの定義における制約を上位クラスの設計者が表明することでこの問題に対処する。

- クラスは、プロトタイプ時や開発の初期段階では、直観的ではあるが比較的大きな単位にまとめられることが多い。やがて、一つのクラスが、複数の抽象を含むことが明かになるとクラスを分解したり、再設計したりするわけであるが、設計者には労力を伴う作業であり、また明確な指針がないので分解や再設計の善し悪しは設計者のスキルに依存する。我々は、クラスの分解の標準的な指針となるものを探している。その一案として、クラス間の静的な依存関係に着目してクラスの分割の十分さを検討する方法を提案する。

5 関連する研究

Johnson と Foote[JoFo88] は、再利用性の高いオブジェクト指向プログラムの設計のための１３の一般的ルールを示している。Casais[Cas91] も述べているように、これらのルールは一般的な指針は示しているが、実際どのような場面で、それらのルールが必要になるか、あるいは、具体的にどのようにクラスを変形するかは必ずしも全て明確にされているわけではない。

デザインパターン [GHJV94] は、再利用性も含め良く設計されたオブジェクト指向プログラムに繰り返し見られるパターンをカタログ化したものである。設計者がこのような知識を習得することは重要であるのは言うまでもないが、本論文等で指摘するような言わば「悪い」パターンを明確にすることも現実には意味があるだろう。また、デザインパターンは、機械的なサポートを目指したものではなく、通常、プログラムには明示的に表されない設計者の意図をまとめたものである。我々は、可能な限り機械的なサポートを追求することにも興味がある。

Opdyke[Op92] は、基本的で単純な Refactoring 操作の他に、特に再利用性と関連のある３つの操作、抽象クラスの抽出、Case 文のクラス階層への変形、継承と集約に関する変形、を議論している。また、Casais[Cas91] は、冗長性などを考慮した継承階層の再構成等を綿密に議論している。我々も同様に、再利用性を高めるクラスの改良の過程をできるだけ形式的に整理し、ツールによる機械的なサポートの可能性を追求している。クラスの改良の過程は、様々なものが考えられ、いろいろな観点から整理していくことが重要であると考えている。

6 まとめ

本論文では、再利用性を高めることを目的としたクラスの変形の指針となる4つのルールを議論した。再利用性は、継承の使われ方を制限し、集約をバランスよく利用することによるプログラムの理解性を上げることと、静的依存関係のサイクルをなくすようにクラスを分解することにより改善される。クラスの再構成の過程は、設計者にとって労力のかかる作業である。しかしながら、完全な自動化は困難である。したがって、この過程を支援するツールへの期待が高まる。そのためには、クラスの再構成の過程の十分な形式化が重要であり、これは我々の課題でもある。

参考文献

[Bo86] Bobrow, D. G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F., "CommonLoops: Merging Lisp and Object-Oriented Programming," OOPSLA 86, 1986.

[Car91] Carroll, M., "Using Multiple Inheritance to Implement Abstract Data Types," The C++ Report, Vol. 3, No. 4, April 1991.

[Cas91] Casais, E., "Managing Evolution in Object-Oriented Environments: An Algorithmic Approach," PhD thesis, University of Geneva, 1991.

[JoFo88] Johnson, R. E. and Foote, B., "Designing Reusable Classes," Journal of Object-Oriented Programming, pp. 22-35, June-July 1988.

[De89] Deutsch, L. P., "Design Reuse and Frameworks in the Smalltalk-80 Programming System," Software Reusability, Vol II, pp. 55-71, ed. Biggerstaff, T. J. and Perlis, A. J., ACM Press, 1989.

[GHJV94] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[HaOB87] Halbert, D. C. and O'Brien, P. O., "Using Types and Inheritance in Object-Oriented Programming," IEEE Software, 4(5), pp.71-79, 1987.

[Op92] Opdyke, W. F., "Refactoring Object-Oriented Frameworks," Doctoral Dissertation, University of Illinois Urbana-Champaign, 1992.

[StEl90] Stroustrup, B. and Ellis, M. A., *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[TGP89] Taenzer, D., Ganti, M., and Podar, S., "Problems in Object-Oriented Software Reuse," ECOOP 89, pp. 25-38, 1989.

[MiNa94] 三ッ井欽一, 中村宏明, オブジェクト指向プログラムの視覚化技法, 情報処理学会研究報告, 94-SE-99-17, 1994.