プログラムコードの静的解析によるインヘリタンス使用法の分類とソフトウェア開発への応用†

北山 文彦††

オブジェクト指向のプログラミングでは，インヘリタンスを用いることによって再利用性や保守性に優れたコードを書くことが可能である．しかし，インヘリタンスの簡単な機構にもかかわらず，それを正しく効果的に使用するのは困難である．この論文では，型をもつコンパイル方式のオブジェクト指向言語を対象にして，インヘリタンスの使われ方を実証的に明らかにし，その結果をもとにして作成したツールのソフトウェア開発への応用について述べる．まず，実際のプログラムの静的な解析を通して，インヘリタンスの使い方として詳細化，多相化，抽象化の3種類のものがあることを示し，それぞれの性質を議論する．次に，この結果を実際のソフトウェア開発に役立てるために，与えられたプログラムのインヘリタンスについて種々の解析をしてユーザにレポートを行う手法を開発し，ツールとして実現した．このツールを用いていくつかの実用規模のプログラムの評価を行い，このようなツールがプログラムの質を向上させるのに役立つことを示す．

1. はじめに

ソフトウェアを効率的に開発するには，プログラム設計論やソフトウェア資産の有効利用の方法論，およびそれに則ったプログラム支援環境を整備していくことが重要である．オブジェクト指向はそのようなパラダイムの1つであり，情報隠蔽やインヘリタンスなどの特徴を持つ．中でも，インヘリタンスはその比較的単純な機構にもかかわらず，プログラムの扱う問題領域を階層構造としてうまく整理し，コードの共有化をすすめ，データの取り扱いの柔軟性を高めることにより，ソフトウェアの保守性や生産性を高めることを可能にする．したがって，インヘリタンスを効果的に用いるためにも，その使い方を明らかにし評価法を確立していく必要がある．インヘリタンスについては一般に言われているような“is-a”関係や“kind-of”関係であるという以外にも，仕様と実現の分離[2),16)]，情報隠蔽との関連[15)]，多重継承を用いた機能の合成[3),10)]，多重継承の是非[5)]，データ抽象と多相化[1),12)]，などの議論が従来からされている．

ソフトウェア開発において，何をオブジェクトにし，それらをクラスとしてどのようにインヘリタンスを用いて組織立てていくかは，それぞれのプログラムの意味的部分に関連し，明解な手続きとして形式化することは難しい．オブジェクト指向分析[4),6)]におけるオブジェクトの発見も同様な問題であり，一般的手続きとしては限界があると思われる．また，プログラムの設計の初期段階から最適なクラスの階層構造を決定するのは難しく，その再構成が頻繁に行われることから，そのことを考慮した設計方法が提案されている[8)]．このような背景から，インヘリタンスとしてどのようなものが望ましいのかを明らかにすることは，ソフトウェアの質を高める修正を行うためにも重要である．このような方法論とともに，インヘリタンス構造の修正を支援するツールも必要であろう．

一方，オブジェクト指向プログラムの再利用性を高めるためにはクラスライブラリの整備が必要である．ライブラリが有効に使われるには，ユーザに対するわかりやすさや使いやすさ，可搬性，コードの機能性，効率，保守性等の質が高いことが重要である．ブラウザなどの支援ツールや文書とともに，コードの質を決定するインヘリタンスの使い方が，ライブラリの質を決定する要因の1つである．したがって，質の高いクラスライブラリを構築していくためにも，インヘリタンスの評価方法は必要となる技術である．

以上の背景を踏まえ，本論文では，オブジェクト指向プログラミングにおけるインヘリタンスの使い方を実証的に明らかにし，その結果をもとにプログラム中のインヘリタンスの使い方の評価を行う方法を提案する．さらに，この手法をツールとして実装し，現実のソフトウェア開発への適用を行い，その有効性を検証する．

従来，インヘリタンスの使用法として，種々のことが言われている[3),9),10),12),13)]が，実際のソースコードから実証的に検証したものはまだない．また，従来のメトリクスにおけるソースコードの行数やコントロー

† Understanding the Use of Inheritance through Static Source-Code Analysis, and Its Application to Software Development by FUMIHIKO KITAYAMA (IBM Research, Tokyo Research Laboratory).

†† 日本アイ・ビー・エム(株)東京基礎研究所

ルフローグラフのように言語やプログラムのセマンティクスと無関係に得られるものと異なり，本論文の評価方法は，(1)オブジェクト指向言語の機構とそのセマンティクスを考慮している，(2)プログラムの問題領域に対応した書き方をしているかを判断する，という点が特徴である．

対象言語は，大規模かつ実用のアプリケーションに使われると思われる，型をもつコンパイル方式のオブジェクト指向言語である．解析例として，C++[7] とCOB[14] で記述されたプログラムを用いた．これらの言語は，クラスを静的な型として扱っており，実行時以前に静的なチェックでそのクラスの性質を調べることが容易である．なお，本論文ではソースコード中の型の情報を用いるため，Smalltalk のように静的な型チェックを行わず型情報が不足する言語には適用不可能である．これは，本論文の方法の限界である．

2章では，インヘリタンスの使い方が安定していると思われるいくつかのプログラムに対し，各インヘリタンスのスーパクラスとサブクラスがそれぞれどのように使われているかを静的かつ定量的に解析する．この結果から，インヘリタンスの使い方として，詳細化，多相化，抽象化の3つの概念が存在することを明らかにし，それぞれの意味と特徴を議論する．3章で，この事実をもとに，統計的手法を用いたインヘリタンスの評価法について述べる．この手法により，インヘリタンスの種類の判別や評価値の計算が可能となる．4章では，この手法をツールとして実装し，実用規模のプログラムに適用を行い，その有効性などを検証する．

2. インヘリタンスの使われ方の分類

2.1 ソースコードの解析

われわれが行ったオブジェクト指向プログラミングの経験があるプログラマへのインタビュー調査の結果や，他の方法論に関する文献[3),9),10),12),13)]から，プログラマがインヘリタンスを使う目的として，コードの共用や再利用，大きなクラスの分解，他人のコードの利用，機能の抽象化，別のクラスの修正，データ構造の融合，機能の合成，といったものがあることがわかった．さらに，どのような目的かによって，クラスの扱い方も異なることが予想できる．クラスの扱い方とは，クライアントのコードの中でそのクラスが型としてどのように使われているかや，インスタンスがどの程度生成されているかということである．

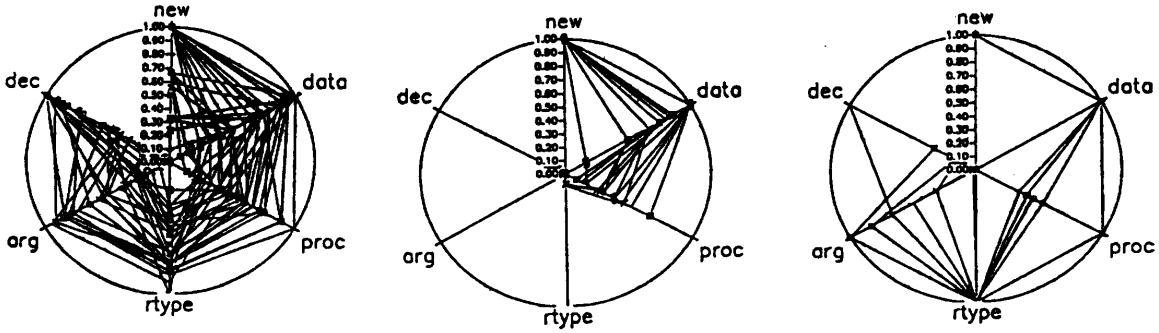
これらの調査から，「インヘリタンスの使い方として異なるものがあり，望ましいインヘリタンスの使い方とはそれぞれの意味に従った適切な使い分けである」，という仮説をたてることができる．そこで，実際のアプリケーションプログラムのソースコード解析から，それぞれの使い方における特徴と目的，意味を明らかにし，この仮説を検証することとした．

対象としたのは**表1**に示したCOBプログラムで，開発が完全に終了しているものを選んだ．これらのプログラムについて前提としたのは，プログラムによって繰り返し検討が行われた結果クラスの階層構造が安定し，上述の望ましいインヘリタンスの使い方がされているということである．

解析は，(1)ソースコードからクラスごとに4種類の数量を静的に抽出し，(2)各インヘリタンスのスーパクラスとサブクラスの数量からそのインヘリタンスの特徴量を計算する，という手順で行った[17]．ここで言うインヘリタンスとは，スーパクラスとサブクラスの対をさす．クラスごとに計量するデータは，(a)インスタンス変数の個数（**data** とする），(b)メソッドの個数（**proc**），(c)インスタンス生成の回数（ここでは，COB の new 関数を呼び出している式の個数とした）（**new**），(d)関数や変数の宣言の型としてクラスが使われる回数（関数の返り値型を **rtype**，引数の型を **arg**，変数の型を **dec**），である．これらは，コンパイラの補助出力等を用いて容易にソースコードから得られ，かつ，クラスのコーディングに関する情報を含むものとして選んだ．特徴量は，(a)から(d)の数量のそれぞれに対し，そのインヘリタンスのスーパクラスの値を p，サブクラスのを c としたときに，$c/(c+p)$ で計算した．ここでは，$c+p$ が0となるようなインヘリタンスは除外した．

表1 測定対象プログラム
Table 1 The subject programs of inheritance analysis.

プログラム名	行数(千行)	クラス数	インヘリタンス数
trans (COB トランスレータ)	18	162	127
prelink (COB ポストプロセッサ)	4	29	19
lisp (LISP インタプリタ)	1	45	41
maze (トイプログラム)	1	7	2
cpp (Cプリプロセッサ)	2	38	25
rcmes (データ処理)	4	31	18
cobenv (COBプログラミング環境)	24	61	32
smake (COBプログラミングツール)	6	54	36
合　　計	60	427	300



(a) 詳細化 (41個)
(a) Specialized (41).

(b) 多相化 (26個)
(b) Polymorphic (26).

(c) 抽象化 (11個)
(c) Abstract (11).

図1 各インヘリタンスグループのポーラ図
Fig. 1 Polar graphs of inheritance.

結果は，インヘリタンスを特徴付ける量として，(a)のインスタンス生成の指標値と，(d)の関数の返り値型としてクラスが使われる回数の指標値を用いると，調査したインヘリタンスが3種類の範疇に明確に分かれることがわかった[17]．すなわち，インスタンスの生成はスーパクラスでもサブクラスでも行われるグループ（**new** は0と1の間），インスタンスの生成はサブクラスのみで行い関数の返り値型としてスーパクラスが使われるグループ（**new** は1，**rtype** は0），インスタンスの生成はサブクラスのみで行い返り値型としてもサブクラスのみが使われるグループ（**new**，**rtype** ともに1），の3グループである．これらは，詳細化インヘリタンス，多相化インヘリタンス，抽象化インヘリタンスと呼ばれる概念に対応する．

図1に各グループごとの(a)，(b)，(c)，(d)の値のポーラ図を示す．各方向で，外側がサブクラスのほうが多いことを，中心がスーパクラスのほうが多いことを表す．タイトル内の数字は，各グループに含まれるインヘリタンスの個数を表す（総数は78）．詳細化のグループ（図1(a)）では，各指標値がまんべんなく散らばっていることがわかる．これは，スーパクラスもサブクラスも同程度に使われることを意味している．多相化のグループ（図1(b)）は，**new** が1，**rtype**，**arg**，**dec** が0というはっきりした特徴をもち，インスタンスの生成はサブクラスで行うが，型としてはスーパクラスのものを使うというパターンを示している．3つめの抽象化のグループ（図1(c)）は，**new** と **rtype** がともに1であり，これは，インスタンスの生成をサブクラスのほうで行い，型（特に関数の返り値の型）としてもサブクラスのものを使うことを意味する．

結果の要約を**表2**に示す．「インスタンス生成」の欄は，そのインヘリタンスのスーパクラスとサブクラスのうち，インスタンスの生成がどちらで行われるかを示している．「型として使用」も同様に，どちらのクラスがより宣言文等における型として使われるかを示している．

表2 3種類のインヘリタンスの特徴
Table 2 Three different kinds of inheritances.

インヘリタンス名	インスタンス生成	型として使用
詳細化インヘリタンス	両方のクラス	両方のクラス
多相化インヘリタンス	サブクラスまたは無	スーパクラス
抽象化インヘリタンス	サブクラスまたは無	サブクラスまたは無

これら3種類のインヘリタンスは，以下に述べるような特徴を持つが，これらは概念的定義であるので，直接これらの特徴を測定してインヘリタンスを判別することは不可能である．しかし，表2に示すように，ソースコードから得られる情報からそれらの判別を推論することが可能である．

2.2 詳細化インヘリタンス

詳細化インヘリタンスは，スーパクラスもサブクラスも同程度にインスタンスは生成され，型としても使われている．これは，スーパクラスとサブクラスは独立して使われ，サブクラスはスーパクラスの仕様とインプリメントを借りているだけであると解釈できる．サブクラスがスーパクラスをさらに特殊化や詳細化するときに使われるので，このインヘリタンスを詳細化と呼ぶ．このインヘリタンスの使い方は，すでにあるクラスをもとに新たなクラスを作りたいときに用いられ，ライブラリや他のプログラムのコードを再利用するときの代表的使い方である．

例えば，ウィンドウのクラスの場合，表示のみ可能

なウィンドウから，入力可能なウィンドウや，さらには，編集可能なウィンドウを詳細化によって作っていくことができる．この場合，サブクラスのコードには，スーパクラスにすでにある表示や入力の機能を書く必要はない．

2.3 多相化インヘリタンス

多相化インヘリタンスでは，インスタンスの生成はサブクラスとして行われるが，型としてはスーパクラスのほうが使われるというものである．スーパクラスとしてオブジェクトを扱うが，どのサブクラスのインスタンスであるかによってそのふるまいがちがうもので，1つの型でオブジェクトに異なるふるまいをさせる使い方である．多相化は，1つのクラスの機能を柔軟に変化させたいときに用いられる．C++ の仮想関数[7]が用いられるのは，このインヘリタンスの使い方においてである．

例えば，コンパイラの構文解析プログラムでは「式」という単一の型で数式の構文解析を行うが，構文解析とともに行われるコード生成などでは，「加算式」や「減算式」などの部分式ごとに異なる処理が行われる．すなわち，このようなプログラムでは，「式」が多相化におけるスーパクラスであり，そのサブクラスの「加算式」や「減算式」などが実際にインスタンスが生成されるクラスである．

2.4 抽象化インヘリタンス

最後の抽象化インヘリタンスでは，インスタンスの生成も型としての使用もサブクラスで行われることがわかる．各サブクラスは実際にクライアントで使われるクラスであるのに対し，スーパクラスはそれらの共通の機能を抽象化したもので，直接クライアントでは使われない．

抽象化の例としては，「整数」や「実数」クラスを抽象化して得られる「数値」や「順序集合」クラスが考えられる．このとき，「数値」クラスや「順序集合」クラスは，クライアントでインスタンス生成が行われたり型として使われることはなく，「整数」や「実数」クラスが使われる．

特殊な場合として，抽象化インヘリタンスが多段になる場合には，スーパクラス，サブクラスともにインスタンスは生成されない（表では，“無”として示した）．これは，両方とも抽象クラスになるためである．このような場合も抽象化インヘリタンスであると考えられる．同様に，多相化についても，そのサブクラスが抽象クラスになっている場合が考えられるので，両方でインスタンスを生成しないことがあるとした．また，抽象化における型としての使用も同様な議論ができるため，「無」のことがあるとした．

2.5 従来のインヘリタンス論との関係

Snyder[15]は，インヘリタンスを使う目的として，コードの再利用という実装上の理由と，スーパクラスのセマンティクスをサブクラスが詳細化する場合をあげている．特に実装という面から，情報隠蔽とインスタンス変数との関連，スーパクラスの操作を除外するようなインヘリタンス，サブタイピングとインヘリタンスの区別，といった点を議論している．本論文ではこのような実装上の問題としてではなく，クラスの外部仕様（セマンティクスも含む）の関係としてインヘリタンスをとらえている．抽象化インヘリタンスでは抽象クラスにおいてコードの共有を行うが，これは外部仕様を共有した結果として行われるものであり，コードを共有すること自体が目的ではない．外部仕様と関係しないコードの共有や再利用のためにはインヘリタンスを使用するべきではなく，インスタンス変数として別のクラスを保持するデリゲーションを用いるべきである．

逆に，Emerald[2]や Misty[16]のように外部仕様のみでインヘリタンスを考え，実装と切り離してしまう考え方もある．このような考え方は，各クラスのインタフェースを規定し，サブタイピングや型チェックをするためのものとしてインヘリタンスをとらえている．しかし，インヘリタンスはクラスのふるまいやセマンティクスも継承するというのが本論文の考え方であり，ふるまいを決定する実装の面も無視すべきではない．インヘリタンスは，クラスを単なる型として規定するだけではなく，クラスのもつ意味や性質も決定しているのである．

多重継承に関しては，mixin[10]や aggregate[3]など機能の合成という点が従来から指摘されている．この場合のスーパクラスは，他の機能と組み合わせるために概念が整理され抽象化されている必要があり，本論文の抽象化クラスにあたる．一方，詳細化の場合，スーパクラスが完成度の高い既存のクラスであるため，それらをサブクラスで組み合わせるような多重継承は行われないと考えられる．多相化の場合も，サブクラスを共有するような複数の型をクライアントで別々に用いるとは考えられず，多相化においても多重継承は使われない．したがって，本論文の手法では，同一のサブクラスに対して複数のスーパクラスがあるときは，異

なる複数のインヘリタンスがあると解釈できるが，これらのインヘリタンスは抽象化であるのが適切な多重継承の使い方であると考えられる．

3. インヘリタンスの評価手法

3.1 評価手法に求められる要件

本論文の考える評価手法は，ソフトウェア開発過程においてソフトウェア品質向上のために用いることをめざしている．したがって，プログラムのどの部分にどれくらい改良の余地があるのかがわかることが望ましい．例えば，各インヘリタンスに対して品質を表す数値を付加することによって，適切でないインヘリタンスの使い方をしているところがわかる．また，インヘリタンスの種類わけにより，クラス階層構造の理解を高めることができる．それにより，プログラマ自身が問題点を発見することを可能にする．

したがって，われわれの目標は，2章で説明した3つのインヘリタンスの定義に即した種類の判別を行うことと，この使い分けに合わせた品質を表す尺度を求めることである．ただし，3つのインヘリタンスの定義というのは概念的なもので，完全な判別や評価を行うのは困難である．定性的なものをできるだけ定量的に近似しようというのが，以下に述べる手法である．

3.2 判別手法の導出

まず，与えられたすべてのインヘリタンスを判別するには，基本的には，表2に示した判定基準に従えばよい．しかし，ソースコードから得られるこのような情報は，直接，インヘリタンスの使い方を定義するものではなく，2章で説明した概念上の定義と合わないこともある．例えば，詳細化インヘリタンスでは，スーパクラスもインスタンスを生成可能，というのが定義であった．しかし，実際には，必要ないためにインスタンスを生成していないことがある．このように，スーパクラスのインスタンス生成はできるのに見かけ上生成されないようなインヘリタンスは，誤判別されてしまう可能性がある．

そこで，このような誤判別をできるだけ避けるため，測定する変数を増やし統計的手法による判別分析を用いることにした．母集団は，2章で述べた詳細化，多相化，抽象化の各インヘリタンスの集合とし，サンプルは，前章のプログラム8本に同様なプログラム5本を追加したものを用いた．これらのサンプルには，576個のクラス，418個のインヘリタンスが含まれる．

表3に，測定した11個の変数の一覧をあげる．ここで「比重」と記述された変数は，2章と同様に $c/(p+c)$ (p はスーパクラス，c はサブクラスの値) で計算したが，今回は $p+c=0$ の場合は除外していない．表2の考察から，このような「無」の場合は「サブクラス」と見なしてもよいことがわかるので，$p+c=0$ のときは「比重」を1とした．これらの変数は，静的解析によって容易に得られるものの中から，判別に寄与するように，3つのインヘリタンスグループの平均が互いに有意の差を持つものを選んだ．

次に，これらの変数を用いて主成分分析を行い，インヘリタンスの判別を行うための成分および判別のための関数を求めた．主成分分析で寄与率の大きい3つ

表3 インヘリタンスの判別分析に用いた変数
Table 3 Variables of inheritance-discrimination analysis.

変数	説明	平均[*1]	標準偏差[*1]
sup	サブクラスに対するスーパクラス数	1.17	0.43
sub	スーパクラスに対するサブクラス数	8.78	9.92
uvar	パブリックインスタンス変数の比重[*2]	0.06	0.23
rvar	プライベートインスタンス変数の比重[*2]	0.37	0.45
msd	メソッド数の比重[*2]	0.44	0.22
new	インスタンス生成箇所数の比重[*2]	0.67	0.43
ret	関数の返り値の型としてクラスが使われた箇所数の比重[*2]	0.15	0.31
arg	関数の引数の型としてクラスが使われた箇所数の比重[*2]	0.15	0.31
dec	変数の型としてクラスが使われた箇所数の比重[*2]	0.16	0.29
ldec	一時変数の型としてクラスが使われた箇所数の比重[*2]	0.30	0.39
reuse	サブクラスで使えるメソッドのうちスーパクラスで定義されたものの割合	0.41	0.30

[*1] インヘリタンスの総数418個の平均と標準偏差である．
[*2]「比重」とあるのは，p をスーパクラスの数 c をサブクラスとして，$c/(c+p)$ ($c+p\neq 0$ のとき)，または1 ($c+p=0$ のとき) で計算した．

表 4 主成分分析による寄与率の大きい3つの成分
Table 4 Three large components of PC analysis.

成分	寄与率	各変数の係数				
		sup	*sub*	*uvar*	*rvar*	*msd*
$Z1$	32.9%	0.160	−0.321	0.236	−0.279	0.195
$Z2$	17.9%	−0.245	−0.113	0.085	0.348	0.618
$Z3$	12.3%	−0.233	0.473	−0.154	0.249	0.006

成分	*new*	*ret*	*arg*	*dec*	*ldec*	*reuse*
$Z1$	0.053	0.400	0.407	0.388	0.434	−0.182
$Z2$	−0.010	−0.068	−0.160	−0.087	−0.060	−0.613
$Z3$	0.710	0.183	0.027	0.185	0.248	0.042

* 省略した4番目以降の成分の寄与率は9%未満
* 変数の値はサンプルの平均と標準偏差から正規化して用いる.

の成分 $Z1$, $Z2$, $Z3$ に対する各変数の係数を表4に示す.

$Z1$ と $Z3$ の値によるクラスタ分析の結果が2章の結果とよく一致すること, および, 主成分分析の係数の考察から, $Z1$ と $Z3$ が詳細化, 多相化, 抽象化の分類に寄与することがわかった. なお, $Z2$ は係数の値からメソッドはサブクラスにあり (*msd* が大), スーパクラスのメソッドが再利用されない (*reuse* が小) ような使い方の程度を表し, サブタイピングのような型としてのインヘリタンスの使い方を表す指標と思われるが, 本論文のインヘリタンスの分類に対しては寄与しない.

図2は, $Z1$ と $Z3$ で各インヘリタンスをプロットしたものである. インヘリタンスの分類によって点の形を変えてある. この図から明らかなように, 各グループを分けるように直線を引くと, インヘリタンスを判別する関数として, 直線(1) ($Z3 = Z1 + 0.5$) と直線 (2) ($Z3 = -Z1 + 0.5$) を設定することができる. これが求める判別式である.

3.3 抽象化度と多相化度の計算

図2の境界(1)(2)上のインヘリタンスは, 使い方があいまいで, 矛盾のあるような使い方をしていると考えられる. したがって, この境界からどれくらい離れているかによって, そのインヘリタンスの使い方の正しさを数値化できる. インヘリタンスの質を表す1つの尺度して, 各インヘリタンスを表す点が(1)の直線よりどれくらい上にあるかを多相化度, (2)の直線よりどれくらい上にあるかを抽象化度として定義し, その直線距離をその値とする. 下側にある場合は負の値とする.

結局, 与えられたプログラムに対し, 次の手順でその全インヘリタンスの種類と, その多相化度, 抽象化度を計算できる. まず, 各インヘリタンスのスーパクラスとサブクラスに対して, ソースコードを静的に解析して表3の11個の変数の値をもとに, インヘリタンスごとに正規化する. 次に, その11個の値に対し, 主成分分析で求めた表4の係数に従って一次変換を行い, その2つの成分の値と先の判別関数(1)(2)を用いて, インヘリタンスの種類の判別と, 多相化度, 抽

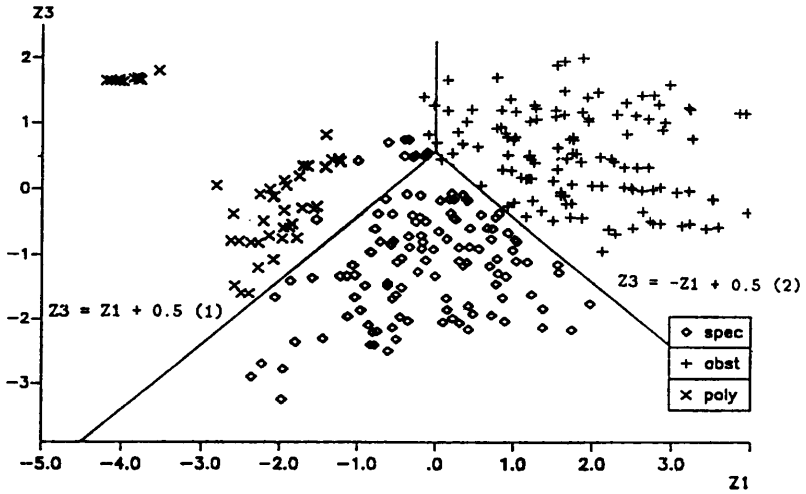


図 2 インヘリタンス判別のための成分による各インヘリタンスのプロットと判別関数
Fig. 2 Plot of inheritances by inheritance-discrimination components and its discrimination functions.

表 5 インヘリタンス判別手法における誤判別の割合
Table 5 Error rate of inheritance-discrimination method.

プログラム名	言語	行数(千行)	クラス数	インヘリタンス数	誤判別個数(割合)
COB translator*1	COB	16	163	133	18(13.5%)
InterViews 2.6*2	C++	24	178	114	6(5.8%)
NIHCL 3.0*3	C++	16	149	65	10(15.4%)
Stream/String	COB	3	14	16	1(6.3%)
PM	COB	8	49	33	9(27.3%)

*1 リリースが表1のものと異なる.
*2 Graphic は除く.
*3 ベクトルは除く.

象化度の計算を行う.

この手法は,インスタンス変数と関数をもつクラスがあり,そのクラスを型として用い,インスタンスの生成を行えるオブジェクト指向言語に適用できる.

3.4 手法の評価

本論文の手法は統計的手順で導いたもので,最初のサンプルを原因とするエラーの可能性がある.そこで,そのエラーがツールとして使えるレベルであるかを別の角度から評価する必要がある.

このようなエラーは,インヘリタンスの誤判別として現れる.そこで,いくつかの既知のプログラムにツールを適用させ,その判別結果の誤判別の割合を調べた.インヘリタンスの種類が正しいかどうかは人間(可能な場合はプログラマ自身)が判断した.

表5に,評価に用いたプログラムとその判断結果を示す.これらのプログラムは評価の導出には用いられなかったもので,ある程度の大きさをもつクラスライブラリである.このようなクラスライブラリは一般的なクライアントコードの部分がないので,判別が難しいと考えられる.ただし,COB のトランスレータはサンプルと同じプログラム(リリースは異なる)であるが,参考のために表に含めた.

ツールと人間の判断の不一致の原因としては,(1)これらのプログラムのインヘリタンスの使い方がもともとよくない,あるいは,(2)判別方法にエラーがある,という2つの理由が考えられる.そこで,誤判別率が高いと判断されたプログラムのソースコードを検討してみると,抽象クラスのインスタンスを特殊な目的のために生成したり,エラーハンドリングのために通常と異なる仮想関数の使い方をしているなど,(1)を原因とするものが数多く存在することがわかった.これらの使い方がよくないものを除くと,(2)を原因とする誤判別の割合は5%前後である.

誤判別はある程度は避けられないものの,局所的に誤まったとしても人間は全体的なコンテキストから推察できるため,問題はないと思われる.例えば,誤判別率 5.8% の InterViews 2.6 の階層構造は,4.3 節でとりあげるように図4に示されているが,全体の構造からどの部分が誤まった判定がされているかはある程度わかる.これは,同じスーパクラスの下のインヘリタンスは同種のものであるというような一般的ルールから明らかである.

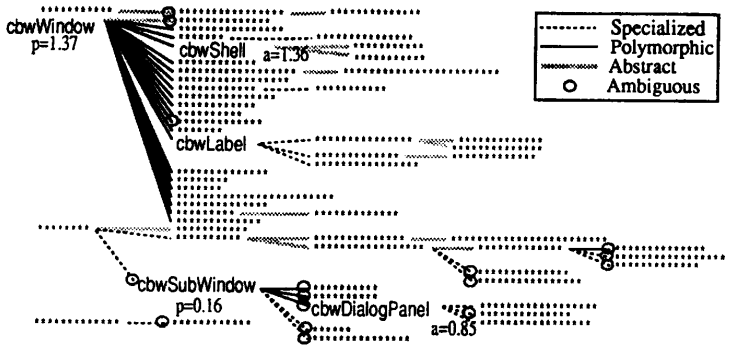
4. インヘリタンス解析のソフトウェア開発への応用

4.1 インヘリタンス評価ツール

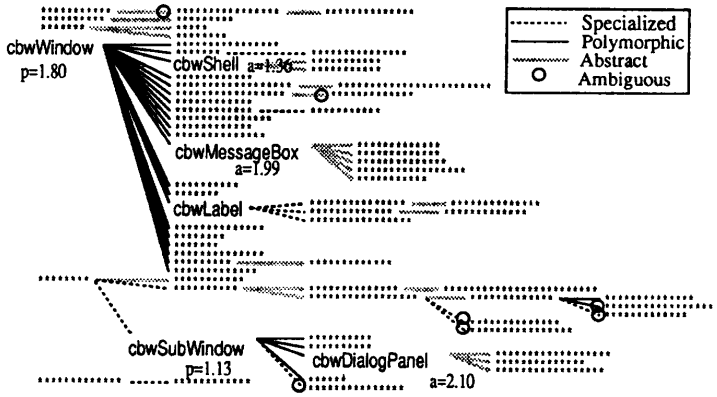
インヘリタンス評価ツールは,今まで述べた評価手法に従って,与えられたプログラム中の全インヘリタンスについて自動的にその使われ方の判別と評価を行うツールである.ツールへの入力は,現在,COB と C++ のソースコードであるが,他の同様な言語にも容易に適用できる.また,対象プログラムが完全に動作している必要はない.出力は,プログラム中の各インヘリタンスの多相化度と抽象化度を付記したインヘリタンスの分類表である.さらに,抽象クラスのインスタンス生成や仮想関数の使用法など,判別の結果と矛盾するコードの書き方をしているものに警告することもできる.これにより,プログラムの設計者は,インヘリタンスの使い方として改良すべき部分を知ることができる.また,インヘリタンスの階層構造を木にして,その種類をつけて出力させることにより,プログラム理解のために役立たせることができる.特にライブラリの場合は,その使い方もある程度はわかる.

ソースコードを解析してクラスごとの情報を抽出する部分だけが言語ごとに必要となるが,それ以降の処理は言語独立に行える.ソースコードの解析は,COB ではコンパイラの付加情報と COB で書かれた

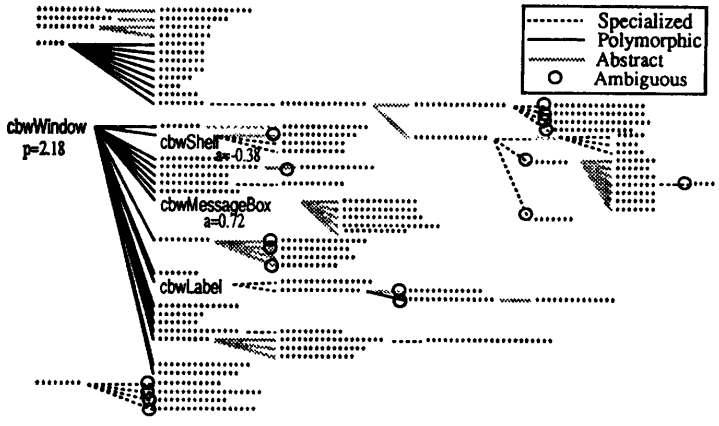
解析プログラム，C++では，Prolog のプログラムデータベースを用いている．その他の処理は UNIX の shell スクリプトと awk プログラムからなる．RS 6000 ワークステーションの上で，2万行程度の InterViews 2.6 の解析に約5分かかるが，実用上問題はないと思われる．



（a） 初期段階のクラス階層構造
(a) Initial class hierarchy.



（b） 3か月後の階層構造
(b) Three-month-old class hierarchy.



（c） 6か月後の階層構造
(c) Six-month-old class hierarchy.

図 3 クラス階層構造の変遷
Fig. 3 History of class hierarchy change.

4.2 プログラム開発過程への適用例

開発過程にあるプログラムとして，Motif 上の GUI ツールの評価を行った．このプログラムはデータのブラウズプログラムで C++ で記述されている．この開発中のプログラムを対象にして，前節のツールを用いて，約3か月おきに3回ソースコードを解析し，時間がたつにつれて評価がどのように変わるかを調べた．プログラムの規模は，初回の測定時点で約 15,000 行，3回めで約 20,000 行である．開発形態としては，プロトタイピング的にプログラムを作成し，仕様を練り直してはコードを書き換えるという方法が用いられた．プログラムは，仕様の変更や性能向上のために書き換えられるが，実際には，クラスの階層構造にも変更が加えられている．

図3は異なる時点でのプログラムのクラス階層構造を並べたものである(時間順に(a)，(b)，(c))．階層構造は木で表示してあるが，ノードはクラス（一部を除いて“*”マークを用いて名前を省略した)，エッジはインヘリタンスを表し，種類により線の種類を変えてある．また，判別関数の境界近くのあいまいなものには丸印を付加してある．ここでは，使い方のあいまいでよくないインヘリタンスの推移がわかるように，多相化度と抽象化度の適当な閾値を一時的に定めた．すなわち，多相化に判別されているが多相化度が 0.5 以下，抽象化に判別されているが抽象化度が 0.5 以下，詳細化に判別されている多相化度と抽象化度が −0.5 以上のものをあいまいとした．さらに，一部のスーパクラスの下には，その下のインヘリタンスにおける多相化度（$\bar{p}$）あるいは抽象化度（$\bar{a}$）

の平均値を付記した．

この図によると，既存のインヘリタンスについては，時間が経過するとあいまいな使い方のインヘリタンスが減り，多相化度や抽象化度が上がっていることがわかる．例えば，ウィンドウを表すクラス **cbwWindow** の直下のインヘリタンスをみると，多相化度の平均が，1.37, 1.80, 2.18 とプログラムが改良されるにつれて上がっている．**cbwWindow** の下のあいまいなインヘリタンスも(a)では2つあったが，(b)と(c)では消滅している．すなわち，**cbwWindow** 下のインヘリタンスは本来の多相化としての性格をより強めるように変更が行われており，この部分のインヘリタンスの質は向上していると言える．(a)と(b)における **cbwSubWindow** や **cbwDialogPanel** も同様である．

以前からあるインヘリタンスは評価値が高くなることが多いが，新しく付け加わったインヘリタンスはあいまいな使い方になる傾向がある．例えば，**cbwShell** や **cbwMessageBox** は，新しいクラスが付け加わったなどの理由で，逆に抽象化度の平均が小さくなっている．このようにソースコードの拡張による変更は必ずしも質を高めることにならないことに注意する必要がある．

コードの改良による質の向上が観測されたことから，われわれのツールがインヘリタンスの判別や質の評価を適切に行っていることがわかる．よって，このツールを用いることにより，プログラマは改良の可能性がある部分を知ることができ，そのような部分を検討し修正することによりプログラムの質を高めることができる．

4.3 クラスライブラリの解析例

C++ のクラスライブラリとして種々のものがあるが，これらの質を客観的に評価する方法はまだない．この節では，われわれが解析を行ったクラスライブラリの中から，C++ のライブラリとして実績のある InterViews 2.6[11]をとり上げる．

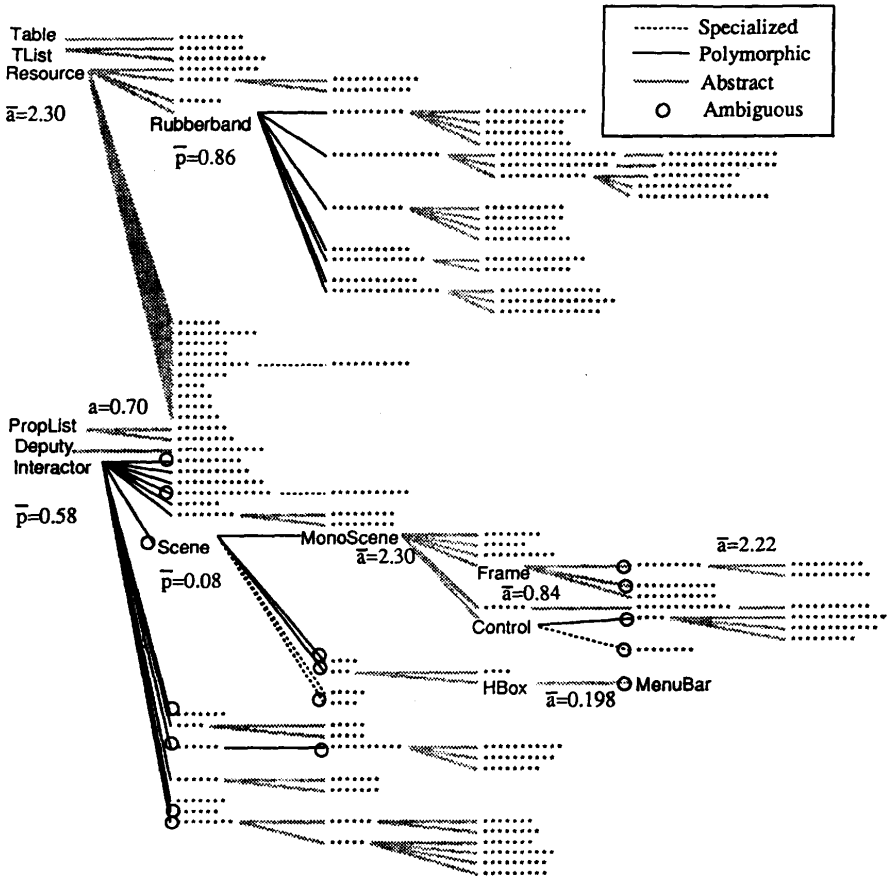
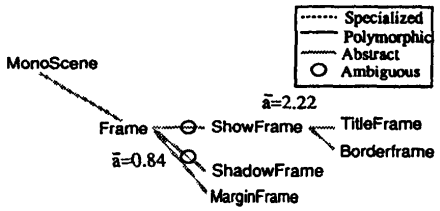


図 4 InterViews 2.6 のインヘリタンス解析
Fig. 4 Inheritance analysis of InterViews 2.6.

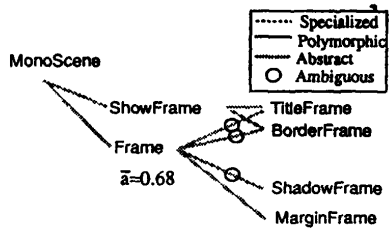
図4は InterViews のクラス階層構造で，図3と同様な表示がしてある．全体として，多相化と抽象化のインヘリタンスが使いわけられ，整った構造をしていることがわかる．例えば，同じベースクラスでも，**Interactor** はウィンドウ部品を表す多相化のクラスであるのに対し，**Resource** はいろいろな表示上のリソースの抽象クラスである．本論文の手法では，この性格の違いを明確に区別できる．

しかし，**Interactor**, **Control**, **Scene** の下のインヘリタンスのように，あいまいであったり（図では丸印を付加），種類分けが不適切なところがいくつかある．さらに，抽象化インヘリタンスの多くは抽象化度が2以上と高いが，**Frame** と **PropList**, **HBox** クラスの下のインヘリタンスは抽象化度の平均が，それぞれ，0.84，0.70，0.20 と小さい値である．一般的に，このような部分は低品質で改良すべき可能性が高いと考えられる．そこで，改良の可能性がある例として，**Frame** クラス以下のインヘリタンスの階層構造を考察する．

図5（a）にもとの **Frame** クラスまわりの構造を示す．**Frame** クラスは，枠がついたウィンドウを実現しているが，ソースコードを解読すると，このクラスは抽象クラスではなく，このクラス自体のインスタンスを生成して使用することができることがわかる．そこで，**Frame** クラスの下の3つのインヘリタンスについて考える．



（a） 変更前
（a） Before change.



（a） 変更後
（b） After change.

図5 "Frame" クラスの階層の再構成
Fig. 5 Restructuring of class hierarchy around "Frame" classes.

ShadowFrame クラスは枠に影がついたもので，**Frame** クラスの機能を特殊化したものであるので，抽象化に分類されているのは誤判別と思われる．実際に抽象化度も低い．**MarginFrame** クラスは，枠内のウィンドウを glue と呼ばれる概念で配置したもので，同様に詳細化インヘリタンスであるのが正しい．

3つめの **ShowFrame** クラスは，マウスの動きに合わせたハイライト機能を実現するもので，**Frame** クラスの機能とは関係しない．**ShowFrame** のサブクラスである **TitleFrame** と **BorderFrame** は，**Frame** クラスにそれぞれタイトルを表す文字表示と枠のハイライト機能を付加したもので，明らかに **Frame** クラスの詳細化になるべきクラスである．

以上の考察から，**ShowFrame** は **Frame** クラスのサブクラスとして不適当であることがわかる．1つの改良例として，多重継承を用いて図5（b）のようにすることが考えられる．変更後の評価値を再計算すると，**Frame** 以下のクラスはなお抽象化と誤判別されるものの，抽象化率の平均は 0.84 から 0.68 と小さくなり，より本来の詳細化に近付いたと言える．

これらのインヘリタンスが抽象化と誤判別される理由としては，これらのサブクラスが型として使われることがほとんどなく，**Scene** クラスを実現する多相化の使われ方をするためである（図4参照）．これは，本論文の手法が局所的にしかインヘリタンスを解析しない，いいかえると，大局的には多相化であってもそのような情報は使わない，という欠点を持っていることを示している．

5. おわりに

本論文では，型を持つコンパイラ方式のオブジェクト指向言語を対象にして，異なるインヘリタンスの使い方があることをソースコードから実証的に示し，それぞれのインヘリタンスの種類がプログラム中で異なった効果が期待されていることを明らかにした．そして，このような使い分けがインヘリタンス，しいては，ソフトウェアの質を決定する要因の1つであると考え，自動化可能なインヘリタンス評価方法を導き出した．また，この手法をもとにして，ソフトウェア開発に役立つようなツールを作成した．

このツールが実際のソフトウェア開発に効果的であることを示すために，開発中のプログラムや既存のクラスライブラリに対してツールの適用を行った．その

結果，開発過程に従ってインヘリタンスの質が向上する様子がツールを用いることによって観察された．また，既存のクラスライブラリに適用することにより，その中の問題点を考察した．これらの適用例から，本論文で提唱した評価方法がソフトウェアの品質を正しく評価し，その向上に十分に役立つことが結論される．一方，この方法が局所的解析に限定したことによる短所も明らかになった．

インヘリタンスの使い方のみがソフトウェア品質や開発効率の向上に貢献するわけではないが，プログラマ個人の経験的技術に依存し客観的に扱うのが困難であったインヘリタンスの使い方を明らかにし，ソフトウェア開発に応用したという点で，本研究はソフトウェアの再利用や生産性向上，保守性向上により役立つものと考える．今後は，評価手法の限界や有効性等の評価をさらにすすめ，かつ，ツールもより利用しやすい形に改良していくことが課題である．

謝辞 本研究を進めるにあたって指導，助言をしていただいた上村務氏，久世和資氏，および，実験の協力や貴重な経験の提供をしていただいたプログラム言語グループのメンバに感謝します．本論文の草稿に対し詳細かつ貴重なコメントをいただいた査読者の方々にも感謝します．

参考文献

1) Meyer, B.: Reusability: The Case for Object-Oriented Design, *IEEE Software*, Vol. 4, No. 2, pp. 50-64 (1987).
2) Black, A., Hutchinson, N., Jul, E., Levy, H. and Carter, L.: Distribution and Abstract Types in Emerald, *IEEE Trans. Softw. Eng.*, Vol. SE-13, No. 1, pp. 65-76 (1987).
3) Blair, G. S., Gallagher, J. J. and Malik, J.: Genericity vs Inheritance vs Delegation vs Conformance vs…, *Journal of Object-Oriented Programming*, Vol. 2, No. 3, pp. 11-17 (1989).
4) Booch, G.: Object-Oriented Development, *IEEE Trans. Softw. Eng.*, Vol. SE-12, No. 2, pp. 211-221 (1986).
5) Cargill, T. A.: Does C++ Really Need Multiple Inheritance? *Proceedings of the Summer 1990 UKUUG Conference*, pp. 53-59 (1990).
6) Coad, P. and Yourdon, E.: *OOA: Object Oriented Analysis*, Yourdon Press (1989).
7) Ellis, M. A. and Stroustrup, B.: *The Annotated C++ Reference Manual*, Addison-Wesley (1990).
8) Gossain, S. and Anderson, B.: An Iterative-Design Model for Reusable Object-Oriented Software, *ECOOP/OOPSLA '90*, pp. 12-27 (1990).
9) Halbert, D. and O'Brien, P.: Using Types and Inheritance in Object-Oriented Programming, *IEEE Software*, Vol. 4, No. 5, pp. 71-79 (1987).
10) Hendler, J.: Enhancement for Multiple-Inheritance, *SIGPLAN Notices*, Vol. 21, No. 10, pp. 98-106 (1986).
11) Linton, M. A., Vlissides, J. M. and Calder, P. R.: Composing User Interfaces with InterViews, *IEEE Computer*, Vol. 22, No. 2, pp. 8-22 (1989).
12) Meyer, B.: *Object-Oriented Software Construction*, Prentice-Hall (1988).
13) Onodera, T. and Hosokawa, K.: Style of Inheritance in COB, *6th Conference Proceedings of JSSST*, pp. 421-424 (1989).
14) Onodera, T. and Kamimura, T.: COB Language Manual, IBM Research (1990).
15) Snyder, A.: Encapsulation and Inheritance in Object-Oriented Programming Languages, *OOPSLA '86 Proceedings*, pp. 38-45, ACM (1986).
16) 久野：多重継承と強い型付けを持つオブジェクト言語 Misty，コンピュータソフトウェア，Vol. 6, No. 3, pp. 9-18 (1989).
17) 北山：オブジェクト指向プログラムにおけるインヘリタンスの使われ方―アプリケーションプログラムをもとにした解析とその分類―，オブジェクト指向ソフトウェア技術シンポジウム論文集，pp. 67-76，情報処理学会 (1991).

(平成4年1月27日受付)
(平成4年6月12日採録)

北山 文彦（正会員）

1964年生．1988年東京大学理学部情報科学科卒業．1990年同大学院理学系研究科修士課程修了．同年より日本アイ・ビー・エム(株)東京基礎研究所に勤務．オブジェクト指向言語，プログラミング環境，ソフトウェア構築法，ユーザ・インタフェース作成法等の研究に従事．