Information Processing Society of Japan

ソフトウェア工学 136−16
(2002. 3. 7)

オブジェクト指向プログラムのリバースエンジニアリング

齊藤 寿和　海尻 賢二

信州大学大学院 工学系研究科
情報工学専攻

あらまし　近年、ソフトウェア開発において、オブジェクト指向開発が注目されている。しかし、オブジェクト指向開発の普及につれて適切なドキュメントを持たないソフトウェアも増えてきている。また、オブジェクト指向言語におけるリバースエンジニアリングでは、実装段階でのクラス構造とリバースエンジニアリングによって実現されるべき分析・設計段階におけるクラス図との抽象度の違いについて考慮する必要がある。本研究では、オブジェクト指向言語としてC++、ドキュメントとしてUMLのクラス図を対象とした。リバースエンジニアリングを行う事によって得られる情報に抽象的な情報を付加することで、より設計に近いクラス図を復元するための手法を提案する。

キーワード　オブジェクト指向, リバースエンジニアリング, C++, クラス図

Reverse Engineering of Object Oriented Program

Toshikazu Saito　Kenji Kaijiri

Department of Information Engineering,
Graduate School of Science and Technology
Shinshu University

Abstract *In recent years, in software development, object-oriented development attracts attention. However, along with the spread of object-oriented development the software not having a suitable document is also increasing. Moreover, in reverse engineering for object-oriented language it is necessary to consider the differences between the degree of abstraction in each stages. The target of this research is C++ as an object-oriented language and UML's class diagram as a document. This paper proposes the technique for restoring more abstract class diagram, by adding abstract information to the information acquired by carrying out reverse engineering.*

Key words　*Object Oriented, Reverse Engineering, C++, Class Diagram*

NII-Electronic Library Service

Information Processing Society of Japan

1 背景と目的

近年、ソフトウェア開発において、オブジェクト指向開発が注目されている。これは、オブジェクト指向開発が、保守性や再利用性の面で、従来の開発手法より優れていると考えられているためである。しかし、オブジェクト指向開発の普及につれて適切なドキュメントを持たないソフトウェアも増えてきている。

そういったソフトウェアに対してリバースエンジニアリングを適用することによって、ドキュメントを生成することが可能である。オブジェクト指向言語におけるリバースエンジニアリングは従来の非オブジェクト指向言語に加えて、オブジェクト指向独自の抽象概念を考慮することが必要である。しかし、オブジェクト指向プログラミングで構築されているクラス構造とリバースエンジニアリングによって実現されるべき分析・設計段階におけるクラス図とは抽象化のレベルが異なるという問題点がある。

そこで、本研究では、オブジェクト指向言語で実装されたソースコードから適切なドキュメントを復元することを目的とする。対象は、オブジェクト指向言語として C++、ドキュメントとしてクラス図を採用することとする。[1] では、Java のソースコードから設計情報を復元するためにパターンベースのアプローチを採用している。本論文では、C++のソースコードに対してリバースエンジニアリングを適用する事によって得られる解析結果に抽象的な情報を付加することで、より設計に近いクラス図を復元する手法を提案する。

2 クラス図

クラス図は、システムの静的な側面を表したモデルであり、UML(Unified Modeling Language)([2],[3]) で定義されているモデルの 1 つである。クラス図は、その基本構成として、クラスとその特性 (属性、操作)、クラス間の関係を保持する。ここで、属性、操作についてはソースコードとほぼ一意に対応付けが可能である。しかし、クラス間の関係については抽象化処理が必要となる概念が存在する。

2.1 クラス図の復元

ここでは、クラス図を復元する上で最も基本となる情報、すなわちソースコードに記述されている構文とクラス図が一意に対応付け可能な情報を示す。

2.1.1 クラス

クラスは、C++では次のように表される。ここでは、"className"をクラスの名前として認識する。

```
class className { ... };
```

クラスはその性質から単純なクラス以外に、以下に示すような UML の仕様で定義されているステレオタイプによる分類が可能である。しかし、C++ではインターフェースや抽象クラスの明示的な構文がないため、これらの明確な分類を行う必要がある。

《abstract》	抽象クラス
《interface》	インターフェース
《struct》	構造体
《union》	共用体
《utility》	ユーティリティクラス

ここで、ユーティリティクラスとは生成されるインスタンスの数が 0 となるクラスであり、クラススコープの属性と操作のみを保持するクラスである。

2.1.2 属性

属性はクラスの構造的な性質を示し、クラス内部で宣言された変数を属性として認識する。ただし、型がクラスである場合、対象となる変数はクラスの構造的性質を示すものではなくクラス間の関係を示すものとして、属性として認識しないものとする。

```
class foo {
private:
    int data;              // 属性
    AssociatedClass ac;    // 関連
};
```

2.1.3 操作

操作はクラスの振る舞いを示し、クラス内部で宣言された関数を操作として認識する。操作の宣言には、クラス内に操作の定義を含む場合と、クラスの外で行う場合の 2 つがある。ここで、C++では宣言時に引数の名前を省略することが可能であるので、省略された名前はクラス外での定義から取得することとする。

```
class foo {
public:
    void func1(int);
    void func2(int a) { ... }
};
void foo::func1(int b) { ... }
```


NII-Electronic Library Service

2.1.4 関係

本研究で認識する基本的な関係は依存、汎化、関連、実現の4つである。関連についてはその特性に従いいくつかの拡張が存在するが、ここでは、一意に認識可能な関係についてのみ記述し、より複雑な関係は次章で詳細な定義を記述する。また、実現は汎化の特殊形であるため、これについても次章で記述する。

2.1.5 依存

依存は、2つのクラスの間に存在する使用関係を示す。これは一般に、あるクラスが他のクラスを操作のパラメータとして使用している場合に存在する。ただし、依存するクラスは属性として保持していないものとする。依存の例は、次に示す通りである。

```
// クラス foo のクラス A に対する依存
class foo {
public:
    void UseA(A a);
};
```

また、依存について UML の仕様で定義されているステレオタイプは、次のように認識することとする。

- **《bind》**
  テンプレートクラスとその具象クラスとの間に存在する依存関係。
- **《friend》**
  C++における friend 関数を利用しているクラスとその friend 関数を保持しているクラスとの間に存在する依存関係。
- **《instantiate》**
  あるクラスとそのクラスで定義されている操作内で生成したインスタンスのクラスとの間に存在する依存関係。
- **《use》**
  あるクラスで定義されている操作の内部で、依存関係にあるクラスの公開部分へのアクセスが存在する場合に2つのクラスの間に存在する依存関係。

2.1.6 汎化

汎化は、クラス間の継承構造を示す。C++では、継承として言語レベルでサポートしているため、汎化は一意に認識することが可能である。汎化で定義されているステレオタイプは次に示す通りである。

《implementation》 private 継承

ステレオタイプ《implementation》による継承の例を次に示す。

```
// 汎化: 《implementation》
class derive : private foo { ... };
```

2.1.7 関連

関連は、クラス間の構造的な関係を示す。従って、属性の型がクラスであるとき、そのクラスと属性を保持しているクラスとの間に存在する関係を関連とする。また、typedef によって定義された型の元の型がクラスである場合、そのクラスとの間にも関連が存在するものとして認識を行う。関連の例は次に示す通りである。

```
typedef Relation RelationClass;
class foo {
private:
    AssociateClass ac;  // 関連
    RelationClass  rc;  // typedef による
};
```

2.2 抽象度の違い

分析・設計段階でのクラス図と実装段階でのクラス構造は、それぞれその抽象度が異なる。これは、分析・設計段階でのクラス図は意味的な表現であるが、実装言語でのクラス構造は構文的な表現であるためである。従って、分析・設計段階でのクラス図に実装言語でサポートされていない抽象概念を含むとき、実装時に情報の欠落が生じる。このような情報の欠落が抽象度の違いとなって現れる。

例1: 関連と集約 例えば、関連と集約は混同しやすい。これは、関連、集約の実装方法にプログラミング言語としての区別が存在しないためである。C++においては、集約は実インスタンスであるメンバ変数を定義することによって実装できる。しかし、インスタンスへのポインタあるいは参照としてそれらを定義する方がより一般的である。関連も同様に、ポインタや参照として実装できる。すなわち、関連と集約の違いは言語機能より、実装時の意志決定によって決定されると言うことができる。

例2: クラスの分類 クラスは、その特性からインターフェースや抽象クラスとして認識することができる。Java では、これらの概念が言語で実現されているため抽象度に違いはないが、C++では言語での明示的な定義が存在しない。

NII-Electronic Library Service

Information Processing Society of Japan

3 クラス図の抽象化

クラス図の抽象化については、図1に示すように、前章で行ったクラス図の復元に対して行うものとする。

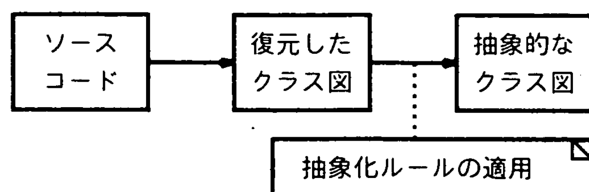

図 1: 抽象化の流れ

ここでは、抽象化に必要となる情報と抽象化の手法について述べる。

3.1 抽象化に利用可能な情報

抽象化を行う上で利用可能な情報は次の2つに分類することができる。

- ソースコード内部の情報
- ドメイン知識による情報

以下では、これらの情報について詳しく述べる。

ソースコード内部の情報 ソースコード内部の情報は次のように分類することができる。

- 構文情報
- コメント情報

構文情報は、クラス図の復元に必要となる情報と以下で記述するクラス図の抽象化に必要となる情報を表す。コメント情報は、ソースコードに記述されているコメントを表し、特にクラス図におけるノートやタグ付き値、クラスの責務など自然言語で記述されるべき記述を導出するために有用である。また、実装時の意志決定を含むことも考えられるため、この情報から抽象化を行うことも可能である。コメント情報を取得するためには、何らかの規約を用いてソースコード内のコメントを制御する必要がある。例えば、Javaではjavadocがこれに相当する。

ドメイン知識による情報 ドメイン知識による情報は、次のように分類することができる。

- シソーラスを利用した名前辞書情報
- 人間による認識情報

シソーラスを利用した名前辞書情報は、クラス図で用いられている名前の間に存在する情報を示す。これは、特に名前の間に存在する「全体-部分」の関係を検出することを目的とする。名前は、実装時における最も単純な意志決定であると考えることができるので、抽象化情報として有用である。人間による認識情報は、クラス図を人間の目で確認することによって得られる修正情報を示す。

3.2 抽象化手法

ここでは、前章で復元したクラス図に対して抽象的な情報を付加することで、より抽象度の高いクラス図を生成するための手法を提案する。

3.2.1 クラスの抽象化

クラスはその特性から、インターフェースや抽象クラスに分類することが可能である。これらは、インスタンスの生成が不可能な抽象的なクラスを示す。インターフェース・抽象クラスは、クラスについて次の条件をすべて満たすものとする。

- インスタンスの生成が不可能。
- 属性についてそれぞれ次の条件を満たす。
  - インターフェースは属性を保持しない。
  - 抽象クラスは属性を保持しても良い。

C++では、インスタンスの生成が不可能なクラスは、対象となるクラスの操作として純粋仮想関数を含むこと、または対象となるクラスの基底クラスで定義されている純粋仮想関数を対象となるクラスで実装していないことを示す。

3.2.2 委譲、伝搬の認識

操作に対してその操作が委譲、伝搬を行っているかどうかを明確にすることは、クラス間の関係に意味づけを行うときに有用な情報となる。

操作の委譲 操作の委譲は、ある処理を自クラスでは処理せずに、関係するクラスにその処理を依頼する事を示す。従って、操作の委譲は、あるクラスの操作内でそのクラスに関係のあるクラスの操作を呼び出すときに考慮する。次のような条件をすべて満たすとき、対象となる操作に委譲が存在するものと認識する。

- 呼び出し元の操作と呼び出し先の操作の名前が同じ。

NII-Electronic Library Service

Information Processing Society of Japan

- 委譲呼び出しで使用される変数以外の値の変更を行わない。
- 委譲呼び出し以外の操作の呼び出しを行わない。ただし、上の条件を満たすための操作呼び出しは行ってもよいこととする。

同様の名前を使用した操作の呼び出しは、実装時の意志決定を示すものであると考えられる。また、委譲する際に行う処理は委譲に必要となる最小限の処理であるものとし、残り2つの条件を追加した。委譲の例は次に示すとおりである。

```
// foo => Associate の委譲が存在
class foo {
private:
    Associate a;                     // 関連
public:
    void print() { a.print(); }  // 委譲
};
```

属性の伝搬　あるクラスで利用したいデータをそのクラスの属性として保持せずに、関係するクラスの属性として保持したいことがある。これは、概念的にそういった属性を保持するべきでないときに起こりうる。ここで、対象となる2クラス間でのその属性値の移動が属性の伝搬として考えられる。これは、a),b) いずれかの条件をすべて満たすものとする。

a)
- 関係先で対象となる属性を変更してない。
- 関係元クラスから関係先クラスの属性に値を格納している。
- 関係元クラスから上で格納した関係先クラスの属性を取得している。
- 関連先クラスで対象となる属性についての変更を行っていない。

b)
- 関係先で対象となる属性を変更している。
- 関係先クラスでの変更が関連元での算出に影響を与える。

3.2.3　汎化の抽象化

汎化は、その構成要素である基底クラスの性質によって実現となり得る。ここでは、汎化の抽象化として実現を扱う。

実現の認識　実現は、インターフェースに対して実装を与える関係を示す。すなわち、C++では、インターフェースとして認識されたクラスを継承することを示す。ただし、継承したクラスはインターフェースや抽象クラスではないものとする。

3.2.4　関連の抽象化

ここでは、前章で認識した関連を元にしてより抽象的な関連を導き出す。関連は、その拡張機能からいくつかの種類に分類することが可能である。ただし、それらの分類処理のいずれにも当てはまらなかった場合、それは単純に関連であるとする。

3.2.5　集約の認識

集約は、「全体-部分」と表現される関係であり、ある特別な意味付けを与えられて強固に結びついた関連の特殊形態であると定義されている。ここでは、以下のルールを関連に対して適用することで、集約であるかどうかを決定する。

ルール 1: 構文情報による認識　構文情報を元に、次の条件をすべて満たすとき集約であると認識する。

- 関連するクラス間に「全体-部分」の関係が成り立つ。
- 関連するクラス間に操作の委譲、属性の伝搬のいずれかが存在する。

■

ここで、「全体-部分」の関係の成立は、対象となる2クラス間に存在する誘導 (関連の存在する向き) が「全体」から「部分」への方向にのみ存在するものであることを示す。加えて、その誘導に従った方向に、操作の委譲もしくは属性の伝搬が存在するものとする。このとき、誘導の元となるクラスを全体側のクラスとして認識し、誘導先のクラスを部分側のクラスとして認識する。

ルール 2: 内部クラスの定義　構文情報を元に、次の条件を満たすとき集約であると認識する。

- クラスに内部クラス定義が存在する。

■

C++では、あるクラスの内部でクラス定義をすることが可能である。このような内部クラスの定義は、実装時に2つのクラスの間に主従、すなわち「全体-部分」の関係が存在するといった意志決定であると考えられる。従って、内部クラスを部分側のクラスとして認識し、内部クラス定義を保持するクラスを全体側のクラスとして認識する。内部クラス定義の例は次に示すとおりである。

```
class foo {
private:
    class innerClass { ... };
};
```


NII-Electronic Library Service

Information Processing Society of Japan

ルール 3: シソーラスを利用した認識　次の条件を満たすとき集約であると認識する。

- 関連する2つのクラスの名前を対象として、その名前間に「has-a」または「part-of」の関係が存在する。

■

「全体-部分」という定義は概念的なものであり、自動的に認識することは難しい。ここで、名前の間に存在する関係を認識するためにシソーラスを利用し、2つのクラスの名前の間にどのような関係が存在するかを調べる。

ルール 4: コメント情報による認識　コメント情報について、次の条件をすべて満たすとき集約であると認識する。

- 集約であるという明示的な記述が存在する。
- 集約関係にある2つのクラスがどのクラスであるかといった明示的な記述が存在する。

■

3.2.6 コンポジション

コンポジションは集約より強固に結びついた関係である。これは、次の条件をすべて満たすものとする。

- 対象となる関連が集約を示す。
- 全体側のクラスが部分側のクラスの生成・破棄を管理する。

ここで、「全体側のクラスが部分側のクラスの生成・破棄を管理する」ということは、全体側のクラスの操作で部分側のクラスを作成し、全体側のクラスが破棄されるまでの間に、部分側のクラスもその全体側のクラスによって破棄されていることを示す。

3.2.7 限定子付き関連

限定子付き関連は、関連する2つのクラスの間の多重度を減少するために利用される。すなわち、ある限定子固有に関係付けられたオブジェクトを認識することによって限定子付き関連を認識することが可能となる。限定子付き関連は次の条件をすべて満たすものとする。

- 1対多または多対多の関連である。
- 限定子が存在する。

ここで、限定子とは、関係するオブジェクトを識別するために必要となる特性を示す。従って、限定子は関連の属性であり、関係するオブジェクトの属性を示すものではない　ここで、限定子は関係元となるクラスの属性として実装されており、そのクラス内部で次のような処理を行っているとき限定子であると認識する。

- 関連するクラスの生成時に、ある属性をキーとして、生成するクラスを分類している。
- 関連するクラスをある属性をキーとして生成し、利用時にそのキーに従って検索を行っている。

3.2.8 関連クラス

限定子付き関連で記述したように、限定子は関連についての属性である。従って、関連クラスは限定子付き関連の特殊形であると考えられる。ここで、関連クラスは次の条件をすべて満たすこととする。

- 対象となる関連が限定子付き関連を示す。
- 限定子がクラスである。

3.3 人間による認識情報を利用した抽象化

人間による認識を行うためには、クラス図をグラフィカルに提示する必要がある。そこで、本研究では、生成した設計情報をCASEツール([5], [6], [7], [8])にインポートすることで、そのクラス図の閲覧を可能にすることを考慮し、出力対象としてXMI [4]を採用する。

ここでは、CASEツールによってクラス図を認識し、その認識による編集を行うことでより抽象度の高いクラス図の生成を考慮する。特に、IIOSS [8]はダイアグラムの診断機能を持ち、インポートしたクラス図に対して適正であるか、矛盾がないかといったチェックをすることが可能である。

この抽象化は、すべての抽象化処理の終了後に生成されたクラス図に対して適用するものとする。

3.4 抽象化の流れ

クラス図の抽象化は、クラスの抽象化を行った上で関係(汎化と関連)の抽象化を行うものとする。これは、汎化の抽象化にクラスの抽象化が必要となるためである。

関連の抽象化の流れ　関連の抽象化は、まず集約か限定子付き関連であるかを認識する。ここで、集約を認識するために、操作の委譲と属性の伝搬について調査する。集約であれば、さらにその関連がコンポジションであるかどうかを認識し、同様に、限定子付き関連であれば、さらに関連クラスであるかどうかを認識す

NII-Electronic Library Service

Information Processing Society of Japan

る。以上のいずれにも当てはまらなかった場合、単純に関連であるとする。

4 システム

本研究で開発したシステムは、入力とするオブジェクト指向言語として C++を、出力するドキュメントとしてクラス図を対象とした。本システムの構成とその流れは図 2 に示した通りである。

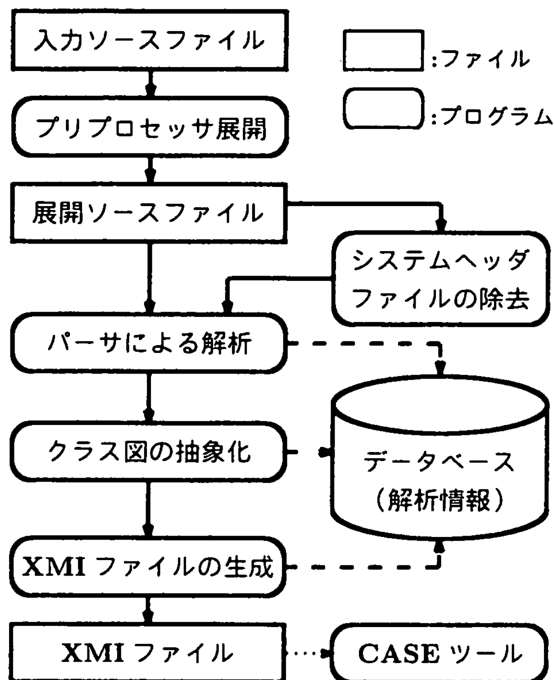

図 2: システムの流れ

ここで、システムに入力するソースファイルは ANSI C++準拠であるものとする。

4.1 プリプロセッサ命令の展開

本システムにおけるパーサで利用している文法は、プリプロセッサ命令に対する対応を行っていない。このため、パーサによる解析を実行する前に、プリプロセッサ命令の展開を行う必要がある。ここで、プリプロセッサ命令の展開手法には、次の 2 つの手法が考えられる。

- 既存のツールを利用する。
- 独自の展開ツールを開発する。

本研究では、既存のツールを利用したプリプロセッサ命令の展開を行うものする。ここでは、既存のツールとして g++を利用する。g++ではオプション"-E"を与えることで、プリプロセッサ命令の展開のみを行い、標準出力にその展開結果を出力する。

展開による問題点　プリプロセッサ命令の展開による問題点は、主に define 文に現れる。define 文は、文字列の置換等に利用されるため、変数名や関数名などが実装者の意図と異なるものに変更されてしまう可能性がある。しかし、本研究で対象としているクラス図ではそのような影響は少ないと考え、事前に展開を行ってしまうものとする。

4.2 システムヘッダファイルの除去

システムによるインクルードファイル (stdio.h など) には ANSI C++準拠でない構文が含まれていることがある。ここでは、そのような構文を含むヘッダファイルに対して解析を行わないために、対象となるヘッダファイルの除去を行う。

4.3 パーサ

パーサでは、クラス図の抽象化を行うための前段階として、クラス図を構成する要素とそのクラス図を抽象化するために必要となる情報をデータベースに格納する。パーサは PCCTS を用いて実装を行い、データベースには PostgreSQL を使用した。

4.4 クラス図の抽象化

クラス図の抽象化では、パーサによって抽出された情報に対して抽象化処理を行うことによってより抽象的なクラス図へと変換を行う。クラス図の抽象化処理は、3.2 "抽象化手法"の記述に従う。ただし、現段階では、コメント情報とシソーラスを利用した名前辞書情報を利用した抽象化については実装していない。

4.5 XMI ファイルの生成

XMI ファイルの生成は、パーサとクラス図の抽象化によって生成されたデータベースから必要な情報を読みとり、入力ソースファイルに対応した XMI ファイルを生成する。

XMI ファイルの利用　前述したように、生成した XMI ファイルは他の CASE ツールにインポートすることでそのクラス図をグラフィカルに表示することを考慮する。本研究では IIOSS をその対象としている。

NII-Electronic Library Service

4.6 システムの適用例

ここでは、本システムでの解析の流れを具体例を用いて示す。次のようなソースコードを入力とした例を考える。

```
// ClassB,ClassC はクラスとして定義済み
class ClassA {
private:
    ClassB b;
    ClassC c;
public:
    void print() { b.print(); }
    void func();
};
```

これを解析することで得られるクラス図 (抽象化を行う前のクラス図) は図3に示す通りである。

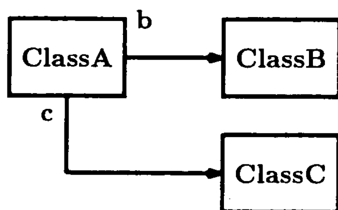

図 3: 抽象化前のクラス図

この段階では、ClassB, ClassC は属性として保持された他のクラスであるので、ClassA とその2つのクラスとの間には関連が存在する。この段階では抽象化を行っていないので、関連以上の表現を見いだすことは出来ない。このクラス図に対して抽象化を行うことで、図4のようなクラス図が生成される。

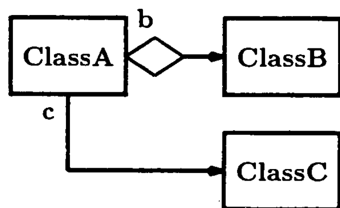

図 4: 抽象化後のクラス図

この例では、ClassA と ClassB の間に存在する関連に対して集約の抽象化ルール1を適用することで、対象となる関連を集約として認識している。ClassA と ClassC の間に存在する関連に対しては、どの抽象化ルールに対しても不適合であるため、対象となる関連は単純に関連であると認識している。

5 考察

本研究では、ソースコードから一意に認識可能なクラス図を復元し、そのクラス図に対して構文から得られた情報について抽象化ルールを適用することによって抽象化を行った。また、この機能を実現したシステムを開発した。ここで、生成したクラス図を XMI ファイルとして出力することで対象となるクラス図の参照が容易となり、より一般的な使用が可能となる。これにより、開発のライフサイクルにリバースエンジニアリングを組み込むことで、設計、実装の繰り返し開発が容易となると考えられる。

ここで、本研究で提案した抽象化についての妥当性が保証できないという点が問題となる。しかし、これは CASE ツールでの参照や、IIOSS の診断機能を利用することによってある程度間接的に保証することが可能であると考えられる。

今後は、抽象化の妥当性を保証するために、シソーラスやコメント情報、デザインパターン [9] などの利用を考慮する事を考える。これらの情報は、対象となるソフトウェアの分析、設計、実装時に行われた意志決定を含むものと考えることができる。従って、これらの情報を利用することで、抽象化に対する妥当性を保証することが可能であると考えられる。

参考文献

[1] Jochen Seemann and Jurgen Wolf von Gudenberg, "Pattern-Based Design Recovery of Java Software", SIGSOFT'98

[2] "OMG Unified Modeling Language Specification version1.3", March 2000

[3] G. ブーチ著, "UML ユーザーガイド", ピアソン・エデュケーション

[4] "OMG XML Metadata Interchange (XMI) Specification version1.1", November 2000

[5] "Rational Rose", http://www.rational.com

[6] "Together/E", http://togethersoft.com

[7] "ArgoUML", http://argouml.tigris.org

[8] "IIOSS Project", http://www.iioss.org

[9] E. ガンマ他著, 本位田真一監訳, "オブジェクト指向における再利用のための デザインパターン", SOFTBANK

NII-Electronic Library Service