プログラムのソースコードにおけるコーディングパターンのモジュール化

Creating Modules from Coding Patterns in Program Source Code

深瀬道晴*
Masaharu Fukase

Email: fukase@dokkyo.ac.jp

オブジェクト指向プログラミングなどのプログラミング技法を用いることで、ソフトウェアの同一の機能を実現する要素をモジュール化することができる。しかし、ソフトウェア開発においては、様々な理由でモジュール化できない要素が存在し、このような要素はコーディングパターンを構成する。プログラムのソースコードにおけるコーディングパターンを除去することは、ソフトウェア保守の観点から非常に重要である。アルゴリズムの高速実装が要求されるソフトウェアにおいても、ソースコードの保守性が高いことが望まれる場合があるが、そのために実行速度が犠牲にされるべきではない。本論文では、LLL アルゴリズムなどを高速実装している NTL ライブラリのソースコードと参考文献(9)において NTL ライブラリを用いて作成されたプログラムのソースコードにおけるコーディングパターンをモジュール化できるかについて検討する。

Programming paradigms such as object-oriented programming make it possible to create a module from software elements which achieve a particular feature. However, it is known that in software development activities there are some elements from which a module can not be created for some reason. Such elements form coding patterns. From the point of view of software maintenance, it is important to remove coding patters in program source code. Although high maintainability is needed also in some situations concerning a software that is supposed to provide high quality implementations of some algorithms, it must be avoided to sacrifice runtime performance for maintainability. In this paper, we consider possibilities of creating modules from coding patters in program source codes of NTL that provides high quality implementations of some algorithms such as LLL algorithm and those written using NTL in (9).

*: 獨協大学経済学部

1. はじめに

主要なプログラミング技法として、構造化プログラミングとオブジェクト指向プログラミングが知られている。また、最近では、オブジェクト指向プログラミングの限界を補完する技法[1]として、アスペクト指向プログラミングが知られている。これらの技法は全て、プログラマが「関心事の分離（separation of concerns)」を実現することを手助けするものである。ソフトウェア開発において、「関心事の分離」が適切になされていれば、ソフトウェアの保守性が向上する。また、これらの技法、特にオブジェクト指向プログラミングの特徴を活かし、ソフトウェアの保守性を向上させるための指針として、リファクタリング(1)やデザインパターン(2)などの概念が知られている。

上で述べたプログラミング技法のそれぞれにおいて、「関心事の分離」のレベルは異なるが、ソフトウェアにおける同一の機能を実現する要素をできるだけモジュール化するという方針は共通している。しかし、ソフトウェア開発においては、様々な理由でモジュール化できない要素が存在し、このような要素はコーディングパターンを構成する。そして、コーディングパターンは、複数のモジュールの中に分散し、ソフトウェアの保守性を低下させる要因となっている(6)。

ソースコード中に散在するコーディングパターンを見つけ出し、モジュール化できれば、ソフトウェアの保守性や拡張性が向上する。また、見つかったコーディングパターンに対して、既存のモジュール化手法のいずれも適用できない場合は、解決策を検討することによって、新しい手法の構築につながる可能性がある(7)。

アルゴリズムを高速実装するソフトウェアにおいても、ソースコードの保守性が高いことが望まれる場合がある。例えば、アルゴリズムのサブルーチンの実行速度を計測したい場合、サブルーチンを別のサブルーチンに置き換えたい場合などである。そのため、そのようなソフトウェアにおいても、実行速度が低下しないのであれば、コーディングパターンをモジュール化するなどして、可能な限り保守性を高めることが望ましい。

本論文では、LLL アルゴリズム(3)などを高速実装しているC++で書かれたNTLライブラリ(5)、及び、参考文献(9)においてC++とNTLライブラリを用いて作成されたプログラムのソースコードのコーディングパターンをモジュール化できるかについて検討する。プログラムのコーディングパターンに着目し、サブルーチン化のような手法でモジュール化するということは、プログラムにおける不適切な部分の発見と改善を可能にし、プログラムの保守性を高めるという重要な意味がある。

以降、2 章では、本章で述べた概念のうち主要なものについて説明をする。3 章では、参考文献(9)において NTL ライブラリを用いて作成されたプログラムのソースコードにおけるコーディングパターンを抽出し、モジュール化した結果を示す。4 章では、NTL ライブラリに実装されている BKZ アルゴリズム(4)のサブルーチンのモジュール化に伴う実行速度の変化を計測した実験結果を示す。5 章において、本論文の結論を述べる。

2. 関連する概念

本章では、1 章で述べた概念のうち主要なものについて説明をする。

2.1 関心事の分離

1 章で述べたように、プログラミング技法毎に、「関心事の分離」のレベルは異なる。ここで、「関心事」とは、同一の機能を実現する、あるいは、同一の観点を表現する、ひとまとまりの要素である。構造化プログラミングにおける関心事は、「処理（手続き)」であり、サブルーチンという形でメインとなる処理フローから分離されている。一方、オブジェクト指向プログラミングにおける関心事は、「処理」とそれに関連する「データ」であり、クラスという形で分離されている[2]。このように、オブジェクト指向プログラミングにおいては、構造化プログラミングにおけるよりも、高度な「関心事」の分離がなされている。しかし、オブジェクト指向プログラミングにおいても、「複数のモジュールに横断的に散在する単一の処理（横断的関心事)」はモジュール化できないことが知られている。アスペクト指向プログラミングでは、本質的な関心事を従来のオブジェクト等のモジュールによって分離し、横断的関心事をアスペクトと呼ばれるモジュールによって分離する。

2.2 コーディングパターン

コーディングパターンとは、ソースコード中に頻出する構造の類似した記述である。コーディン

[1] ただし、アスペクト指向プログラミングは必ずしもオブジェクト指向を前提としない.

[2] オブジェクト指向プログラミングでは、「処理（手続き)」に対して、特定のクラスに属するサブルーチンという意味でメソッドという用語を用いる.

グパターンのうち、コピーアンドペーストなどによって生じるほとんど同じ記述であるコード片は、モジュール化が比較的容易である。しかし、記述の一部が抽象化されているコーディングパターンをモジュール化することは、一般的に、容易ではない[8]。

2.3 リファクタリング

リファクタリングとは、ソフトウェアの外部に対する振る舞いを保存したまま、ソースコードの可読性を高め、また、その修正が容易になるように、ソースコード内部の構造を改善することである。例えば、リファクタリングの基本方針の一つは、プログラムのソースコードにおけるコーディングパターンを取り除くということである。上で述べたプログラミング技法の違いに関わらず、一般的に、ソースコードにおけるコーディングパターンは、ソフトウェアの保守性を低下させる。それは、あるコード片において変更の必要が生じたとき、それと同じ、または、類似した全てのコード片を把握した上で、それらに対しても変更を加えることを検討しなければならないからである。

2.4 デザインパターン

デザインパターンとは、ソフトウェア開発時に発生する様々な問題とその解決法を文書化したものであり、再利用可能な設計パターンである。リファクタリングの際に、問題がソフトウェア開発において発生する典型的なものであれば、デザインパターンを適用することが考えられる。

2.5 C++

プログラミング言語 C++は、代表的な構造化プログラミング言語 C を拡張し、オブジェクト指向プログラミングを可能にしたものである。C++は、C と同様、メモリや実行速度に関する効率が高いことが知られている。

2.6 LLL アルゴリズム

LLL アルゴリズムは、ベクトル空間である格子（lattice）における短ベクトルを求めるアルゴリズムである。LLL アルゴリズムの改良型アルゴリズムとして、BKZ アルゴリズムが知られており、共に、NTL ライブラリに高速実装されている。LLL アルゴリズムの本質的なアイディアは、Gram-Schmidt 直行化の演算を整数演算で近似することである。LLL アルゴリズムは、多項式素因数分解、整数計画法、暗号解読など、幅広く応用されている。

3. プログラムにおけるコーディングパターンの除去

本章では、参考文献(9)において C++と NTL ライブラリを用いて作成されたプログラムのソースコードのコーディングパターンをモジュール化できるかについて検討する。モジュール化を行う際、高速実装する必要のあるコードにおいてコーディングパターンが見つかった場合、モジュール化によって実行速度が低下する場合は、モジュール化できないものとみなす。

3.1 プログラムの概要

本章で用いるプログラムは、参考文献(9)において提案された手法を実装したプログラムである。プログラムのソースコードの合計の行数は、約 1 万行である。実装した手法は、ベクトル空間である格子（lattice）における短ベクトルを探索によって求めるものである。しかし、プログラムにおいては、探索に関わるコードよりも、データの前処理や後処理などに関わるコードが大部分を占める。また、プログラム全体において、NTL ライブラリの機能を用いており、特に、データの前処理において、NTL ライブラリに実装されている BKZ アルゴリズムを用いている。

3.2 コーディングパターンの抽出とモジュール化

上記プログラムにおいて、抽出されたコーディングパターンは、以下の 3 つである。

（1） 文字列型引数を長整数型に変換するパターン
（コーディングパターン 1）

（2） ファイル入力の際に、ファイル名とエラーの処理を行うパターン
（コーディングパターン 2）

（3） 2 つの変数の値を入れ替えるパターン
（コーディングパターン 3）

以下、それぞれのコーディングパターンの詳細を説明し、手動でモジュール化を行った結果を示す。以下に図示されるプログラムコードは、紙面の都合上、空行が取り除かれている。

図 1 の左側は、コーディングパターン 1 を示している。コーディングパターン 1 は、データの前

処理をするプログラムファイルの 1 つに出現した。パターンの出現数は、5 である。パターンの内容は、プログラムの main 関数が受け取る文字列型引数の値を、長整数型変数に格納するものである。コーディングパターン 1 が出現するコードは、高速実装する必要のあるコードとは無関係である。また、パターンを構成するコード片はほとんど同じ記述であり、変数名が異なるのみである。そのため、コーディングパターン 1 をモジュール化することは容易である。図 1 の右側は、コーディングパターン 1 を、argv_to_long という関数によってモジュール化した結果と、パターンを構成するコード片を argv_to_long を呼び出すことでまとめた結果を示している。モジュール化を行った結果、プログラムファイルのソースコード量は、215 行から 215 行になり、増減が見られなかった。これは、パターンを構成するコード片をまとめることで減少したコード量が、関数 argv_to_long の宣言によって増加したコード量によって相殺されたためである。このことから、パターン出現数とパターンを構成するコード片のコード量が少ないとき、モジュール化の効果は目に見えて現れないことが分かる。

```
long s;
stringstream
ss1;
ss1 << argv[1];
ss1 >> s;
....
long beta;
stringstream
ss2;
ss2 << argv[2];
ss2 >> beta;
....
```

→

```
argv_to_long(long& k, char *ch)
{
 stringstream ss;
 ss << ch;
 ss >> k;
}
....
long s;
argv_to_long(s, argv[1]);
....
long beta;
argv_to_long(beta, argv[2]);
....
```

図 1　コーディングパターン 1

図 2 の左側は、コーディングパターン 2 を示している。コーディングパターン 2 は、コーディングパターン 1 と同一のデータの前処理をするプログラムファイルの 1 つに出現した。パターンの出現数は、3 である。パターンの内容は、ファイル入力に際して、ファイル名の処理と、ファイルが見つからなかったときのエラーの処理を行うものである。コーディングパターン 2 が出現するコードは、高速実装する必要のあるコードとは無関係である。また、パターンを構成するコード片は、変数名と sprintf の受け取る文字列が異なるのみである。そのため、コーディングパターン 2 をモジュール化することは容易である。図 2 の右側は、コーディングパターン 2 を、filein_basis という関数によってモジュール化した結果と、パターンを構成するコード片を filein_basis を呼び出すことでまとめた結果を示している。モジュール化を行った結果、プログラムファイルのソースコード量は、215 行から 191 行になり、減少が見られた。これは、パターンの出現数は少ないが、パターンを構成するコード片のコード量が比較的多いためである。このことから、パターンを構成するコード片のコード量が多いとき、パターン出現数が十分に多ければ、モジュール化の効果は目に見えて現れうることが分かる。

```
char chr_private[30];
sprintf(chr_private, "ggh-private-
%ld-%ld-%ld", s, beta, z);
string filename_private
(chr_private);
ifstream filein_private
(filename_private.c_str());
if (!filein_private) {
 cerr << "no file" << "\n";
 return 1;
}
filein_private >> R;
filein_private.close();
....
以下、同じ構造が何度か繰り返し
```

→

```
long filein_basis(mat_ZZ& B, char *ch,
long s, long beta, long z){
 char chr[30];
 sprintf(chr, "%s%ld-%ld-%ld", ch, s,
beta, z);
 string filename(chr);
 ifstream filein(filename.c_str());
 if (!filein) {
  cerr << "no file" << "\n";
  return 1;
 }
 filein >> B;
 filein.close();
}
....
filein_basis(R, "ggh-private-", s, beta, z);
....
```

図 2　コーディングパターン 2

図 3 の左側は、コーディングパターン 3 を示している。コーディングパターン 3 は、データの後処理をするプログラムファイルの 1 つに出現した。パターンの出現数は、7 である。パターンの内容は、2 つの変数の値を入れ替えるものである。コーディングパターン 3 が出現するコードは、高速実装する必要のあるコードとは無関係である。しかし、パターンを構成するコード片において、入れ替えを行う変数の対象が long 型であったり、double 型であったりと、変数の型が異なる。そのため、コーディングパターン 1 とコーディングパターン 2 をモジュール化したような自明な方法では、コーディングパターン 3 をモジュール化できない。コーディングパターン 3 をモジュール化するには、型の違いを吸収するようなモジュール化が必要であるが、それは、C++ではテンプレートを用いることで実現できる。図 3 の右側は、コーディングパターン 3 を、パターンを構成するコード片を swap を呼び出すことでまとめた結果を示している。swap は、C++の標準ライブラリにおいてテンプレートを用いて実装されている。モジュール化を行った結果、プログラムファイルのソ

ースコード量は、675 行から 647 行になり、減少が見られた。パターン出現数とパターンを構成するコード片の量は少ないが、コーディングパターン 1 とコーディングパターン 2 のモジュール化においてよりも、コード量がより減少した。これは、コーディングパターンのモジュール化に用いた関数 swap が、C++の標準ライブラリにおいて既に実装されており、新たに関数宣言を行う必要がなかったためである。このように、モジュールとして既に用意されている関数 swap を用いることは一般的にモジュール化とは言えないが、ここでは、コーディングパターンをモジュールによって除去するという意味でモジュール化したと考えることとする。

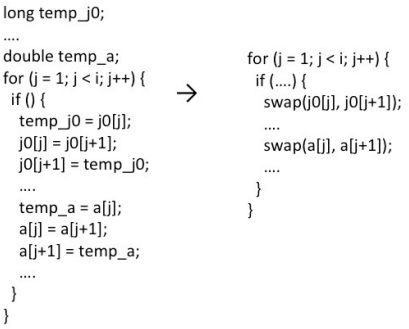

図 3 コーディングパターン 3

本章でモジュール化を行ったコーディングパターンは、いずれもデータの前処理と後処理の高速実装に無関係なコードに出現するものであり、比較的容易にモジュール化を行うことができた。そのため、著者が作成したプログラムのソースコードからは、高速実装に関わるコードにおけるモジュール化を検討することはできなかった。4 章において、NTL ライブラリに実装されているアルゴリズムの高速実装に関わるコードにおけるモジュール化を検討する。

4. モジュール化に伴う実行速度の変化について

本章では、NTL ライブラリのソースコードのコーディングパターンをモジュール化できるかについて検討する。ここでは、NTL ライブラリのプログラムファイルのうち、LLL_FP.c、LLL_QP.c、LLL_XD.c、LLL_RR.c、G_LLL_FP.c、G_LLL_QP.c、G_LLL_XD.c、G_LLL_RR.c の 8 つのプログラムファイルを対象にして、コーディングパターンのモジュール化を検討する。具体的には、モジュール化する場合とモジュール化しない場合の実行速度を比較する。

一般に、関数呼び出しの削減によって実行速度の向上につながることが多いため、モジュール化しない場合の方が高速であることが予測される。しかし、モジュールの使用回数等の条件によっては実行速度がほとんど変化しない場合もある。また、モジュール化による実行速度の変化の程度を具体的に見るために、実際に実行速度を比較することが重要である。

4.1 プログラムの概要

上記 8 つのプログラムファイルは全て、BKZ アルゴリズムを、異なる精度の浮動小数演算を用いて実装している。そのため、ソースコード全体の構造自体が基本的に同一であり、記述も非常に類似している。また、これらのプログラムファイルのコード片はすべて、BKZ アルゴリズムの高速実装に関わるものと考えられる。ソースコード量は、それぞれのプログラムファイルにおいて、1000 行～2500 行である。

4.2 サブルーチン ENUM のモジュール化

上で述べたように、8 つのプログラムのソースコードは非常に類似しているが、浮動小数演算の精度に関わるコード片において違いが生じている。また、エラー処理における出力メッセージが、各プログラム毎に異なるため、コーディングパターンが見つかったとしても、モジュール化は困難である。

8 つのプログラムのソースコードのそれぞれにおいて、1 箇所だけ出現する BKZ アルゴリズムのサブルーチン ENUM の実装に相当する 100 行前後のコード片は、これらのプログラムにおける出現数 8 のコーディングパターンと考えられる。しかし、上で述べた理由によって、このコーディングパターンをモジュール化することは困難である。

そこで、ここでは、8 つのプログラムにおける ENUM の実装に相当するコード片を 1 つのモジュールにするのではなく、それぞれのプログラムにおいて ENUM をモジュール化することを検討する。具体的には、void 型関数 ENUM を宣言し、その中に ENUM の実装に相当するコードを記述する。そして、元々のプログラムにおいて ENUM の実装に相当するコードが存在した箇所において、void 型関数 ENUM を呼び出す。それぞれのプロ

グラムにおいて、1 箇所にだけ出現する ENUM をモジュール化する利点として、ソースコードの可読性が向上するということが挙げられる。現状のソースコードでは、ENUM の実装に相当するコードの範囲を把握することが困難である。また、その範囲において用いられている変数の種類を把握することも困難である。これらのことは、ENUM の実行速度を計測したい場合や ENUM を別のサブルーチンに置き換えたい場合の妨げとなる。

例えば、LLL_FP.c における ENUM の実装に相当するコード片のコード量は、96 行である。このコード片を関数として手動でモジュール化したところ、関数の受け取る引数の数が 27 となった。そこで、ENUM の実装に相当するコードをモジュール化しない場合とモジュール化した場合のそれぞれにおいて、BKZ アルゴリズムの実行速度を計測した。その結果を、図 4 に示す。それぞれの場合について、ベクトル空間の次元が 120、140、160、180 の格子において、BKZ アルゴリズムを実行した。各次元において、5 つの格子を用いた。したがって、それぞれの場合について、各次元において 5 回 BKZ アルゴリズムを実行し、5 回の実行時間の平均値を計算した。プログラムの実行は、CPU：2.7GHz Intel Core i7、メモリ：8GB、OS：Mac OS X の環境において行った。図 4 の横軸は格子の次元を示しており、縦軸が各次元における実行時間（秒）の平均値を示している。下の実線が ENUM をモジュール化しない場合の結果を示しており、上の破線が ENUM をモジュール化した場合の結果を示している。

図 4 から、ENUM をモジュール化した結果、各次元のそれぞれにおいて、実行速度が低下していることが読み取れる。上で述べたように、ENUM の関数の引数の数は 27 であり、このように多数の引数がスタックに格納されるために、実行速度が低下しているものと考えられる。ENUM は、次元 120 のある例では、7221 回呼び出されていた。このため、実行速度の低下が無視できない程度で引き起こされたと考えられる。

4.3 モジュールの展開に伴う実行速度の変化

ここでは、あるコード片をモジュール化するのではなく、逆に、モジュール化されているコード片を、その使用箇所全てにおいて展開することを検討する。

上の ENUM をモジュール化した場合の結果は、関数呼び出しを避けることで、実行速度が向上する可能性を示唆していると考えられる。そこで、BKZ アルゴリズムの別のサブルーチンである LLL アルゴリズムの実装に相当する関数を、その使用箇所全てにおいて手動で展開した。LLL_FP.c における当該関数は、ll_LLL_FP である。ll_LLL_FP は、LLL_FP.c において実装されている BKZ アルゴリズムの中で、5 箇所に出現している。ただし、ll_LLL_FP が呼び出される回数と、ENUM が呼び出される回数はほとんど同じである。ll_LLL_FP が受け取る引数の数は 12 であり、コード量は 339 行である。

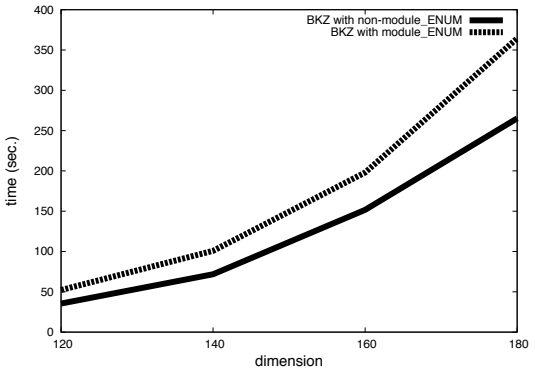

図 4 ENUM をモジュール化しない場合とモジュール化した場合の実行速度の比較

ll_LLL_FP を 5 箇所において展開し、ENUM のモジュール化の場合と全く同じ条件で、実行速度を計測した。その結果を、図 5 に示す。図 5 の破線が ll_LLL_FP を展開した場合の結果を示しており、実線が ll_LLL_FP を展開しない場合の結果を示している。図 5 の実線が示す結果は、図 4 の実線が示す結果に相当する。図 5 の破線と実線がほとんど重なっていることから分かるように、実行速度の違いがほとんど見られなかった。これは、ll_LLL_FP を展開することによって実行速度が向上するという予測に反しているが、その理由として以下のものが挙げられる。ll_LLL_FP の中では、多数のローカル変数が宣言されている。それらのローカル変数は、ll_LLL_FP の呼び出しが終了したときにメモリから解放されるが、ll_LLL_FP を展開したことによって、ll_LLL_FP の呼び出しと共にメモリから解放されなくなった。このことによる実行速度の低下と関数呼び出しをしないことによる実行速度の向上の効果が相殺し合って、実行速度の違いが見られなかったと考えられる。

上で述べたように、ll_LLL_FP の関数の引数の数は 12 であり、ll_LLL_FP の中で宣言されているローカル変数は多数であった。それに対して、本

章 2 節においてモジュール化した ENUM について、ENUM の関数の引数の数は 27 であり、ENUM の中で宣言されているローカル変数は僅かな数であった。このため、本章 2 節における実験では、関数呼び出しの影響とローカル変数の影響が相殺し合うことはなく、実行速度が明らかに低下したと考えられる。

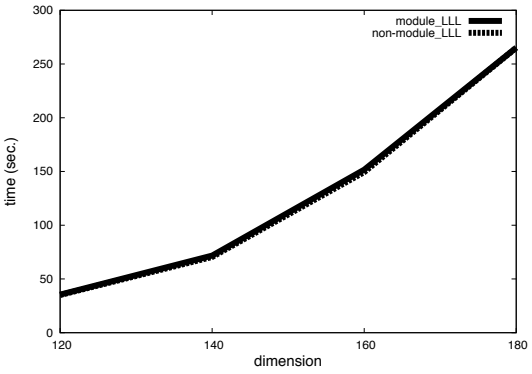

図 5 ll_LLL_FP を展開した場合と展開しない場合の実行速度の比較

5. 結論

本論文では、NTL ライブラリのソースコードと著者が NTL ライブラリを用いて作成したプログラムのソースコードにおけるコーディングパターンをモジュール化できるかについて検討した。その結果、アルゴリズムの高速実装に無関係な部分では、モジュール化可能であることが分かった。しかし、高速実装に関わる部分においては、モジュール化によって実行速度の著しい低下が招かれる可能性があることが分かった。

6. 考察

今回の実験において、ENUM をモジュール化する作業よりも、ll_LLL_FP を展開する作業の方が困難であった。これは、例えば、ll_LLL_FP の呼び出し元スコープにおいて宣言されている変数と ll_LLL_FP の中で宣言されている変数との間で、多数の名前の衝突が起こるためである。また、関数内部で宣言されているローカル変数が関数の受け取る引数によって初期化されていることが多く、展開した際に、改めて変数の初期化を書き直さなければならない場合がある。

ソフトウェア開発において、将来変更される可能性の低いコードについてデザインパターンを適用することは、保守性を高めるどころか、むしろソースコードを複雑にしてしまうことが指摘されている。デザインパターンの適用は、関数としてのモジュール化よりも複雑であり、一度デザインパターンを適用した箇所を元に戻すことは困難な作業を伴うことが予想される。

以上の考察から、コーディングパターンのモジュール化やデザインパターンの適用に際しては、実行性能と可読性の兼ね合い、また、将来のコード変更の可能性などを十分に考慮に入れる必要があると考えられる。

謝辞

本研究の一部は、情報学研究所研究助成によるものである。

参考文献

(1) Fowler, M.:”Refactoring; Improving the Design of Existing Code”, Addison Wesley (1999)
(2) Gamma, E., Helm, Johnson, R., Vlissides, J.:”Design Patterns Elements of Reusable Object-Oriented Software.”, Addison Wesley (1999)
(3) Lenstra, A.K., Lenstra, H.W., Lovasz, L.:”Factoring Polynomials with Rational Coefficients.”, Mathematische Ann., Vol. 261, pp.513-534 (1982)
(4) Schnorr, C.P., Euchner, M.:”Lattice Basis Reduction: Improved Practical Algorithms and Soving Subset Sum Problems.”, Math. Programming, Vol. 66, pp.181-199 (1994)
(5) Shoup, V.:”NTL – A Library for Doing Number Theory.” Available at http://www.shoup.net/ntl/index.html
(6) 石尾隆、伊達浩典、三宅達也、井上克郎：“シーケンシャルパターンマイニングを用いたコーディングパターン抽出”、情報処理学会論文誌、Vol.50、No.2、pp.860-871（2009）
(7) 田中哲：“データマイニングを利用したプログラムの改善”、Proceedings of SPA 2004、pp.1-8（2004）
(8) 渥美紀寿、山本晋一郎、結縁祥治、阿草清滋：“FCDG に基づいたコーディングパターン”、日本ソフトウェア科学会コンピュータソフトウェア、Vol. 21、No.4、pp.27-36 (2007)
(9) 深瀬道晴：“Finding a Very Short Lattice Vector in the Extended Search Space”、博士論文、東京大学 (2011)

(2011 年 9 月 30 日受付)
(2011 年 12 月 21 日採録)