ソフトウェア工学 77−20
（1991. 2. 8）

設計上の決定 = プログラム − 仕様

福田 剛志, 深澤 良彰, 門倉 敏夫
早稲田大学 理工学部

ソフトウェア開発過程における設計段階では、与えられた仕様記述に対して、アルゴリズムやデータ構造など種々の決定が行なわれ、その結果が続いて行なわれるプログラミングに反映される。このように考えると、「仕様＋設計上の決定＝プログラム」と言える。本研究では、この設計上の決定を仕様記述とプログラムから抽出し、プログラム理解やプログラムの自動合成に役立てることを目的とする。すなわち「設計上の決定＝プログラム − 仕様」と捉えて、知識ベースを用いた慣用句の認識を利用することにより、プログラム中に含まれるこの設計上の決定を抽出する方法を提案する。

Design decisions = Program − Specification

Takeshi Fukuda, Yoshiaki Fukazawa, Toshio Kadokura
School of Science and Engineering, WASEDA University

In the design process of the software development, various kinds of design decisions, such as algorithms, data structures and so on, are made. These results have influence on the following programming process. In this sense, we can say "*specification + design decision = program*". In our research, these design decisions are extracted from both its specification and program. These results are utilized in the program understanding and automatic program modification. In this paper, we propose a method to abstract the design decisions in programs using knowledge-based recognition of the "*cliché*". This is what we call "*design decisions = program − specification*".

1 始めに

通常の人手によるプログラム開発において、仕様記述者によるソフトウェアの定義である「仕様」に対して、設計者やプログラマが「設計上の決定」として、アルゴリズムやデータ構造を与える。すなわち、プログラムの設計とは、ソフトウェアの仕様に対し設計上の決定を加えていく作業である。

このことから、概念的に「仕様 + 設計上の決定 = プログラム」と言える。さらに、この設計上の決定に注目すると、「設計上の決定 = プログラム − 仕様」と捉えることができる。この設計上の決定とは、抽象的な仕様を、プログラムとして具体化していく時に採用された方針である [3]。

以上のような考えに立つと、仕様とプログラムから設計上の決定を分離することができれば、次の様な目的のために利用可能である。

1. **プログラム理解**
   要求の変更などによるプログラムの保守を行なう際には、必ずプログラムを十分理解しなければならない。デバッグが困難である原因の一つは、プログラム理解が難しく、熟練を要するためである。そこで、プログラム理解に関する研究がなされてきたが、次のようなプログラム実現の多様性により十分な成果を上げられなかった。

   (a) 表現の多様性
   (b) アルゴリズムの多様性
   (c) データ構造の多様性
   (d) 実現の重複と不連続性
   (e) 既知の慣用句以外での実現

   設計上の決定は文字通りプログラムと仕様のギャップである。すなわち、設計上の決定は設計者の行なった仕様を実現する過程そのものであり、これを人間に提示することにより、プログラム理解を容易にできる。

2. **プログラムの自動変更**
   既に存在する仕様とプログラムに対して、仕様には現れない (実行時間・規模・記憶容量などに関する) 変更を行なう際、設計上の決定を直接変更することにより、新しい決定を持つプログラムを合成できる。仕様に現れない変更とは、設計上の決定の変更に他ならないので、プログラムテキストを扱うよりも設計上の決定を変更する方が有利である。設計上の決定は構造的であるから、変更の影響範囲を捉えることも容易である。

3. **文書の自動合成**
   設計上の決定を行なった設計者やプログラマは、保守等のためにそれらを文書として記録しなければならない。しかし、自動的に設計上の決定を抽出できれば、この作業を自動化することができる。文書作成の自動化により、仕様やプログラムだけが変更され、文書が更新されない「古い文書の害」から解放することができる。

そこで本研究では、以上の様に有用な設計上の決定を、仕様とプログラムから抽出することを目的とする。

2 本手法の概要

人間がプログラムを記述するとき、慣用句的な記述を多用していることが、心理学的、統計的研究で支持されている。従って、従来のプログラム理解の研究で用いられてきた、慣用句 (*cliché*[2]) をプログラム中から発見する「慣用句探索」によるプログラムの抽象化には可能性があると考える。

しかしこの方法は、プログラムの持つ実現の多様性から、部分グラフの同形性を探索するための膨大な計算量を必要とするため、規模の大きい問題を解くことができない [1]。

そこで本手法では、プログラムと仕様の対応関係を次の様に用いて、慣用句の探索空間を小さく絞り、慣用句認識の問題点を克服する。

- **検索する慣用句の限定**
  仕様とプログラムの対応に基づく知識を用いて、プログラム中に存在する可能性の高い慣用句を限定し、探索を効率化することができる。

- **プログラムの分割**
  プログラムは、サブルーチンや関数などとして分割されていることが多い。しかし、プログラム理解の観点からみて、これらの分割が必ずしも適切であるとは限らない。すなわち、プログラム単独ではプログラム理解のために適切に分割することはできない。一方、仕様中の意味単位 (操作) はプログラム理解のために適切な分割を教えている。本手法では、仕様を入力として得ているので、この情報を用いてプログラムを小さな部分に分割し、探索問題自体を小さく絞ることができる。

本手法は、入力されたプログラム中に存在する設計上の決定を出力する。本手法では設計上の決定を、次の 3 つにより表現する。

1. **設計上の決定木**
   仕様中の操作を実現していく詳細化の過程を表現する木。「何をどうやって」実現しているかと言う大まかなアルゴリズムの決定を理解することができる。

2. **慣用句の接続**
   プログラム中に存在している慣用句の接続関係。設計上の決定木に現れない細かいデータの流れ等を捉えることができる。

3. **データ構造の決定**
   プログラム中のデータと仕様中の抽象的データの関係を表す対応関係。データ構造の決定を理解することができる。

これらの出力を得るために、我々が開発した仕様記述言語 "Solaris" により記述された仕様と、それを実現している C 言語により記述されたプログラムを本手法の入力とする。Solaris は、ソフトウェアの内部状態を考慮し、その遷移を述語論理を用いて記述する形式的仕様記述言語である。

```
OBJECT USER {
  Name = [firstName, familyName:String];
  UserID = Natural;
  User = [userID:UserID, name:Name];
  UserList:POWER(User);

  PREDICATE
    identify(user1, user2:User) ==
    (  user1.userID = user2.userID );

  CONDITION
    FORALL user:UserList HOLD
    (  NOT EXIST yet_another_user:UserList HOLD
       (  NOT user = yet_another_user
       &  identify(user, yet_another_user)
       )
    );

  OPERATION
    AddNewUser(user: User) ==
    (  NOT EXIST old_user:UserList HOLD
       (  identify(old_user, user)
       &  UserList' = UserList + { user }
       )
    );
}
```

図 1: Solaris による書庫管理システムの仕様

3 本手法の適用例

図 1, 2 に書庫管理システムの Solaris による仕様と、それを実現した C プログラムを示し、本手法の働きを説明する。

この例のように、仕様とプログラムは同名の操作と関数 (AddNewUser) あるいは同名の状態変数と変数 (UserList) によって対応付けることができると仮定する。即ち、プログラム中の関数は、仕様中の同名の操作を実現している、また、プログラム中の変数は、仕様中の同名の状態変数を実現しているものと仮定する。プログラムは仕様から作られることを考えると、この仮定は自然である。

この二つの入力から、本手法で得られる設計上の決定を図 3, 4, 5 に示す。

図 3 (設計上の決定木) は、仕様中の操作の、プログラム中での実現の詳細化過程を表現している。この図から、木の根の表す操作がどのような決定に従って、木の末端 (葉) が表す具体的な慣用句にまで詳細化されるかを理解することができる。

図 4 (慣用句の接続関係) は、操作を実現する際に用いられた具体的慣用句の接続関係を表現している。この図から、慣用句相互の関係を理解することができる。

図 5 (データ表現の決定) は、仕様中の抽象的なデータ (状態) のプログラム中での実現方法を表現している。この図から、仕様とプログラムの間のデータの対応と、プログラム中のデータの実現方法を理解することができる。

4 本システムの構成と動作

本手法を実現するために構築したシステム内部の構成を図 6に示す。

```
typedef struct {
    char *firstName;
    char *familyName;
} NameType;
typedef int UserIDType;
typedef struct {
    UserIDType userID;
    NameType   name;
} UserType;
typedef struct _UserListType {
    struct _UserListType *suc;
    UserType User;
} UserListType;

UserListType *User List = NULL;

void AddNewUser(UserType user)
{
    UserListType *p;

    for (p = UserList; p; p = p->suc)
        if (p->User.userID == user.userID)
            return;
    p = malloc(sizeof(UserListType));
    p->User = user;
    p->suc = UserList;
    UserList = p;
}
```

図 2: C による書庫管理システムのプログラム

4.1 プログラムの前処理

4.1.1 データ表現解析部とデータ慣用句ライブラリ

データ表現解析部は、抽象データの標準的な実現方法をデータ慣用句として集めたデータ慣用句ライブラリを利用して、プログラム中の具体的なデータ表現が実現している抽象データを発見する。

データ慣用句ライブラリ中の各データ慣用句は、データ構造を表すグラフで表現されている。現在、データ慣用句は数十程度用意している。個々のデータ表現のグラフは通常小さいので、データ慣用句は比較的容易に発見できる。

4.1.2 データフロー解析部

同一のアルゴリズムを実現したプログラムであっても、表面的には様々な表現が可能である。即ち、プログラムは元来、構文上の多様性を持っている。このため、プログラムの表面的な構造からアルゴリズムを調べることは難しい。そこで、データフロー解析部でフロー解析することにより、この多様性を吸収する。図 2の解析結果を図 7に示す。

4.2 仕様の前処理

仕様解析部は、仕様を構文解析して状態変数の構造と状態遷移操作の記述を分離、抽出し、後の推論のための中間コードに変換する。

4.3 データ表現対応部

仕様中のデータ表現は、仕様解析部によりグラフ化されている。プログラム中のデータ表現は、データ表現解析部により抽象データの構造を表すグラフ化されている。この二つのグラフの同じ名前のノードは対応していると考えられる。データ表現対応部は、この名前による対応関係を鍵に仕様中のデー

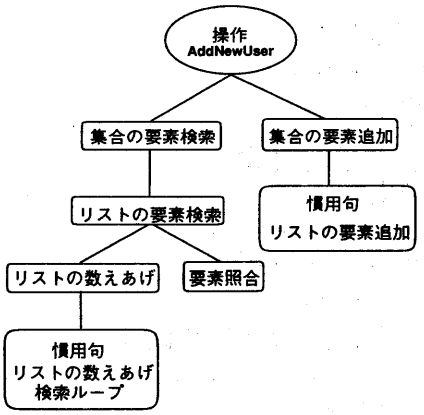


図 3: 設計上の決定 1: 設計上の決定木

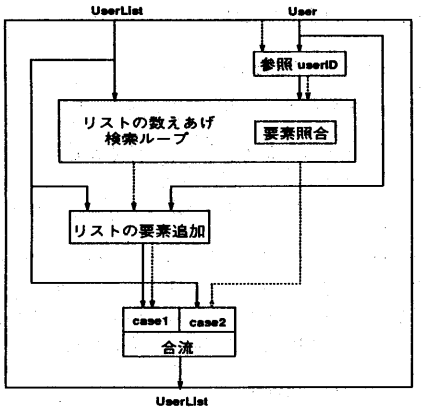


図 4: 設計上の決定 2: 慣用句の接続関係

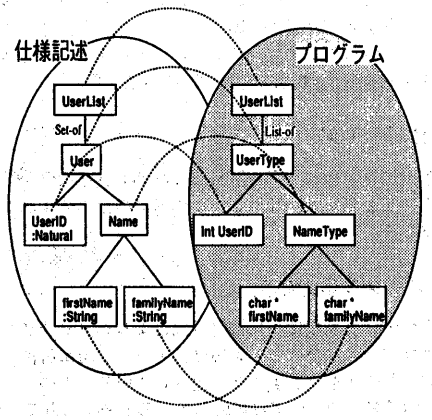


図 5: 設計上の決定 3: データ表現の決定

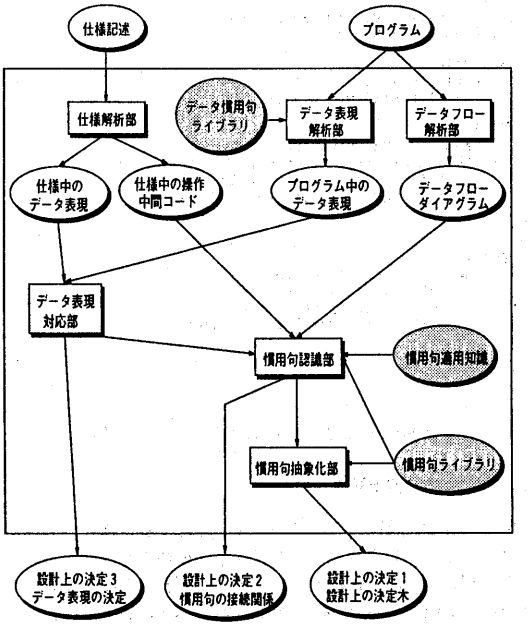


図 6: システムの内部構成図

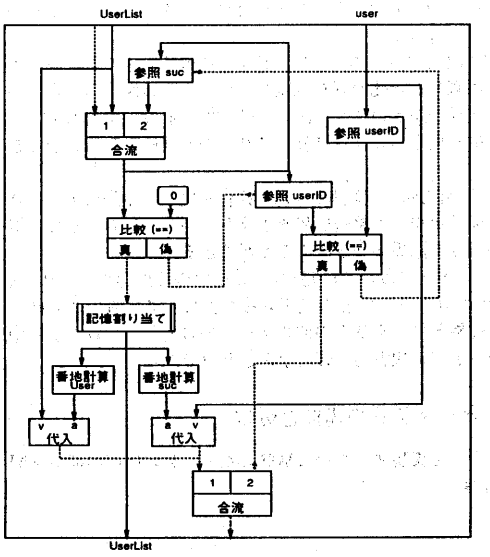
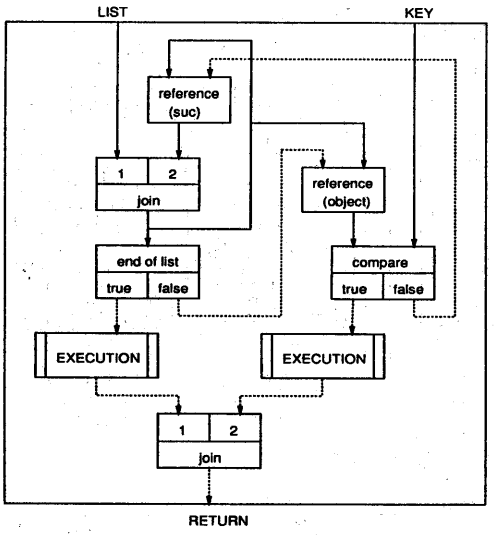


図 7: AddNewUser 関数の中間コード



目標: リストの加算

図 8: 慣用句の例: リストの数えあげ検索ループ

タ表現とプログラム中のデータ表現の対応をとり、データ表現の決定 (設計上の決定 3) として出力する。

4.4 慣用句認識部

慣用句認識部は、仕様の操作毎に分割して、データフロー解析部によって作られたプログラムのデータフローダイアグラム中から慣用句を発見し、慣用句の接続関係 (設計上の決定 2) を生成する。このとき次の慣用句ライブラリと慣用句適用知識を利用する。

4.4.1 慣用句ライブラリ

慣用句ライブラリは、プログラム中に存在する標準的に用いられる問題解決手法を慣用句として集めたデータベースである。

ライブラリ中の各慣用句は、プログラムの中間コードと同様のデータフローダイアグラムで表現されている (図 8)。また、慣用句が実現する目標を合わせて持ち、その慣用句を探索する際の条件として利用する。

各慣用句はライブラリ中に図 9の様に AND/OR 木として構造的に格納される。慣用句の抽象度で階層をなし、決定の種類で OR、決定中の構成で AND の枝分かれをしている。慣用句ライブラリは、この構造により慣用句の抽象化、決定の変更、慣用句へのアクセスの効率化などに利用できる。

4.4.2 慣用句適用知識

慣用句適用知識は、仕様の内容と、仕様とプログラムの対応から、プログラム中に存在する慣用句を推定する知識を集めた知識ベースである (図 10)。

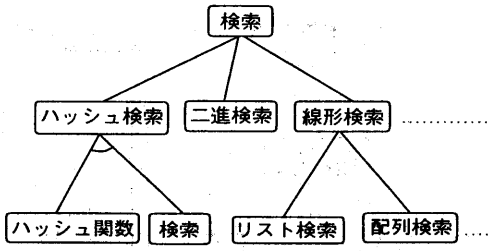


図 9: 慣用句ライブラリの構造例

仕様中	状態変数 x に関する演算 y が存在
対応関係	状態変数 x のプログラム中の表現は z である
	⇓
プログラム	状態変数 x に関する演算として z の y が存在

例

仕様中	状態変数 $UserList$ に関する演算加算が存在
対応関係	状態変数 $UserList$ のプログラム中での表現はリストである
	⇓
プログラム	状態変数 $UserList$ に関する演算としてリストの加算が存在

図 10: 慣用句適用知識の例

この例のように、慣用句適用知識は直接具体的な慣用句の存在を導くのではなく、慣用句が実現する目標を導く。慣用句認識部は、慣用句ライブラリからこの目標を実現する慣用句の適用を試みる。

4.5 慣用句抽象化部

慣用句抽象化部は、慣用句認識部が生成した慣用句の接続関係を元に、慣用句ライブラリの構造を利用して、具体的な慣用句を抽象化し、その過程を設計上の決定木 (設計上の決定 1) として出力する。

5 評価

ソフトウェアの仕様と、それを様々な方法で実現したプログラムに本手法を適用し、出力を比較することにより、本手法が出力する設計上の決定のプログラム理解における有効性を評価した。

顧客管理システムに関する仕様 (図 13) の二通りの実現を本手法により処理して得た設計上の決定を図 11、図 12に示す。これら二つの出力から、プログラムテキストでは大きくことなるソフトウェアの相違点と類似点がよく理解できる。

6 終りに

本手法は、プログラムと仕様の対応を用いて、従来困難であった慣用句認識を実現し、プログラムと仕様からの設計上

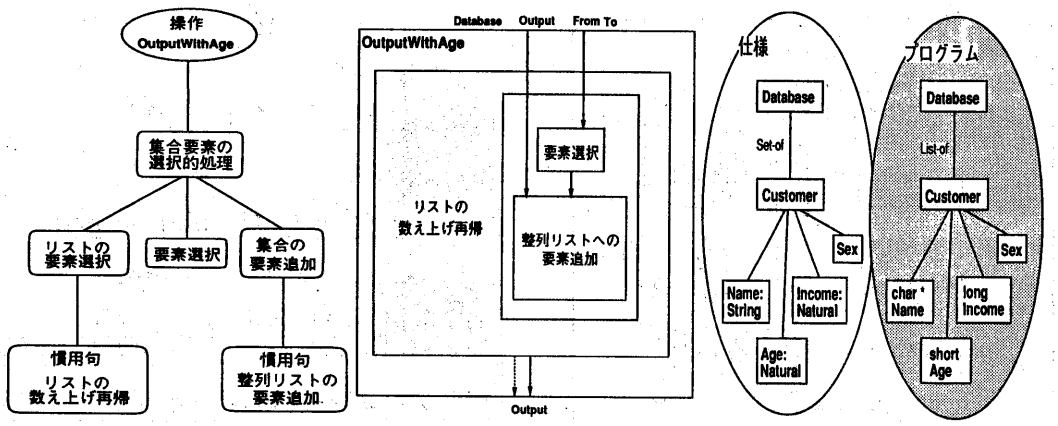


図 11: 顧客管理システム: 実現 1 の設計上の決定

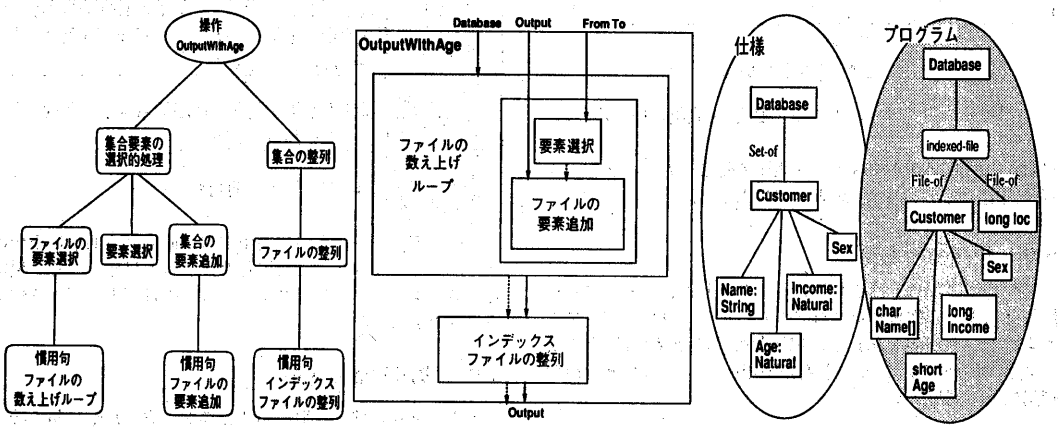


図 12: 顧客管理システム: 実現 2 の設計上の決定

```
Customer = [ name : STRING, age : NATURAL,
             income : NATURAL, sex : SEX ];
Database : POWER(Customer);

....

OutputWithAge(from : NATURAL, to : NATURAL) ==
    (   FORALL p : Database HOLD
        (   (  from  <= p.age
            &  p.age <= to
            )
        ->  p : Output'
        )
    &  SortedWithAge(Output')
    );

....
```

図 13: 顧客管理システムの仕様

の決定の抽出を可能にした。この設計上の決定はプログラム理解に有効であった。

現在、本手法のインプリメントと慣用句及びその適用知識を収集中である。今後、実用的な規模のソフトウェアに対する有効性と対象領域といった、性能評価を行なっていく予定である。

参考文献

[1] Charles Rich and Richard C. Waters. *The Programmer's Apprentice*. ACM Press, 1990.

[2] Richard C. Waters. Program translation via abstraction and reimplementation. *IEEE Transaction on Software Engineering*, Vol. 14, No. 8,, August 1988.

[3] 福田剛志, 深澤良彰, 門倉敏夫. プログラムと仕様からの設計上の決定の抽出. 情報処理学会全国大会講演論文集, volume 5, pp. 141–142, 1990.