



Title	オブジェクト指向によるソフトウェアプロセスの記述モデル
Author(s)	萩原, 剛志; 飯田, 元; 井上, 克郎 他
Citation	ソフトウェア・シンポジウム' 90論文集. 1990, p. 37-44
Version Type	VoR
URL	https://hdl.handle.net/11094/51274
rights	
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

<https://ir.library.osaka-u.ac.jp/>

The University of Osaka

オブジェクト指向によるソフトウェアプロセスの記述モデル

A Process Description Model Based on Objects

荻原剛志[†] 飯田元^{††} 井上克郎^{††} 鳥居宏次^{††} 奥井順^{‡‡}

Takeshi Ogihara[†], Hajimu Iida^{††}, Katsuro Inoue^{††}, Koji Torii^{††}, and Jun Okiu^{‡‡}

[†]大阪大学情報処理教育センター ^{††}大阪大学基礎工学部情報工学科 ^{‡‡}名古屋工業大学工学部情報工学科
Education Center for Information Processing, Faculty of Engineer Science, Faculty of Engineer,
Osaka University Osaka University Nagoya Institute of Technology

ソフトウェア開発プロセスを形式的に記述し、その記述に基づいて開発作業を支援をする試みがなされている。この方法による開発支援をさまざまな開発作業に対して適用するためには、できあいのプロセス記述を利用するよりも、あらかじめ用意されたプロセス記述を部品として組み合わせて使う方法が有効である。

本稿では、記述の部品化と再利用という点からオブジェクト指向による記述に注目し、さらに、オブジェクトを用いてプロセス記述を行うための記述モデル Trine を提案する。Trine はプロセス記述を作業の順序の制御、プロダクトの動的な変化やその属性、プロダクトの実現の3つの部分に分け、それぞれの部分を独立に記述する。このモデルはプロセスを記述する上での共通した枠組みを与えるものであり、記述の部品化、再利用性の向上に有効であると考えられる。

1. はじめに

ソフトウェア開発プロセス（以下ではプロセスと記述する）を形式的に記述し、その記述に基づいた開発支援環境を作成してソフトウェア開発の作業効率と信頼性を高めようとする研究が進められている。

開発作業においては、対象となるソフトウェアによって作業の手順が異なり、開発者や計算機環境によって利用できるツールなどが異なることもある。また、ソフトウェアの構成要素、プロダクト（文書やソースプログラムなどの生成物）の内容や構成は各開発ごとに異なり、用いられるツールもさまざまである。ソフトウェア開発作業の支援環境として、できあいのひとつのプロセス記述を提供するだけでは、このような実際のさまざまなソフトウェア開発作業を支援するためには不十分である。

そこで、あらかじめ用意され、蓄えられたプロセス記述を部品として組み合わせて使えるような環境を作成しておけば、作業手順やツールなどへの色々な要求に応えることが可能となり、開発対象となるソフトウェアにとって適切な開発環境を部品の再利用によって構築できるようになると考えられる。

プロセスの記述には、これまでにさまざまな方法が提案されているが、これまでの記述方法では、各作業単位、各プロダクトに対する記述を部品としてモジュール化し、再利用するための方法が示されていなかった。

本稿ではまず、記述の部品化と再利用という点からオブジェクト指向による記述に注目し、開発作業の進行に伴って生成、使用されるさまざまなプロダクトをオブジェクトと見なすことによって、プロセス記述が行えることを示す。

次に、オブジェクトに基づいてプロセスを記述する上での構造を与えるモデルとして、プロセス記述モデル Trine を提案する。Trine はプロセス記述を作業の順序の制御、プロダクトの動的な変化やその属性、プロダクトの実現の3つの部分に分け、それぞれの部分を独立に記述することができる。このモデルは、開発対象となるソフトウェアの種類を問わず、プロセスを記述する上で共通した枠組みを与えるものであり、記述の部品化、再利用性の向上に有効であると考えられる。

2. プロセス記述法と記述の部品化

ここではこれまでに提案された主なプロセス記述法を示し、プロセス記述の部品化と、プロダクトの記述について検討を加える。

2.1 手続き型言語による記述

Osterweil ら [7] は Ada の拡張言語 Appl/A を用いてプロセスを記述しようと試みている。Appl/A は Ada を下敷にすることによって、プロセスを記述する上でのさまざまな要求に応える能力を備え、さらにプロダクトや資

源の間の関係を記述できるような拡張が加えられている。この言語によって、作業の手順とプロダクトの構造を手続き的に記述する。

Osterweil は各プロダクトをオブジェクトとして記述することが重要であると述べており、Appl/A にもそのような記述のために必要な機能（Ada のパッケージの機能など）が備えられている。しかし、プロセスを実際にどう記述するための方法やモデルは示されていない。

図 1 は手続き的な記述方法によるプロセス記述の一部

```
Function All_Fn_Perf_OK(executable,tests);
  declare executable executable_code,
    tests testset,
    case,numcases integer,
    result derived_result;

  All_Fn_Perf_OK := True;
  For case := 1 to numcases
    derive(executable, tests[case].input_data, result);
    if ~resultOK(result, tests[case].req_output)
      then All_Fn_Perf_OK := False; exit;
    endif;
  end loop;
end All_Fn_Perf_OK;

declare testset array of testcase[1..numcases];

declare testcase record of
  (input_data real,
   req_output record of
     (fcn_tol, time_tol predicate,
      fcn, time real));

declare derived_result record of
  (fcn_output, observ_timing real);

Function resultOK(result,needed);
  declare result derived_result,
    needed req_output;
  if ~fcnOK(result.fcn_output,needed)
    OR (result.observ_timing > needed.time)
  then resultOK := False;
  else resultOK := True;
  endif;
end resultOK;

Function derive(pgm,test,result);
  declare pgm,executable_code,
    test testcase,
    result derived_result;
  reset_clock;
  result.fcn_output := execute(pgm,test.input_data);
  result.observ_timing := read_clock;
end derive;

Function fcnOK(result,needed);
  declare result derived_result,
    needed req_output;
  fcnOK := needed.fcn_tol(needed.fcn,result.fcn_output);
end fcnOK;
```

図 1 手続き型言語によるプロセス記述
(文献[7]による)

である（文献 [7] より）。この例はアプリケーションソフトウェアのテストを行うプロセスを記述したものであるが、主に機能の詳細化によって記述を行っていることがわかる。レコード型 testcase, derived_result のフィールドはいろいろな関数内部で参照、変更されているため、これらの型の定義を変更した場合、どこまでその影響が及ぶのかが明らかでない。このように、作業手順の記述とプロダクトの属性や管理についての記述が混在してしまい、部品化、再利用性を損なうおそれがある。

2.2 ルールによる記述

Kaiser らは、あらかじめ記述したルールに従って自動的にツールを起動したり、プロダクトに対する操作を行うことができる開発環境 Marvel を設計した [3]。Marvel はプロダクトやツールを管理するためのオブジェクト群と、作業の手順を表現する拡張可能なルール群から構成されている。ルールは前提条件、完了条件、および開発作業からなり、あるルールの条件が満たされた時、そこに記述された開発作業を表すツールが起動される。

Marvel は使いやすいツール環境を構築できる可能性を持っているが、ルールの構築、デバッグが行ないにくく、また、複数人で利用できる開発環境への拡張が困難であるなどの問題点も報告されている [4]。

図 2 はプログラムの編集、コンパイル作業の Marvel による簡単な記述である（文献 [3] より）。Class の部分はオブジェクトを定義している。ただし、PROGRAM は PROCEDURE の集まりから構成されるものとする。Tools の部分はツールについての記述、Rules の部分はルールの記述である。ルール中で ? の付いた名前は変数であり、同じ型のオブジェクトとマッチする。compile というルールでは、?prog を構成する PROCEDURE のうち、更新時刻が ?prog のものより新しいオブジェクト ?p があればコンパイルをやり直し、その結果を status に保存するという動作を表している。

この例も図 1 と同様に、オブジェクト内部の情報がルール定義で使われるなどの問題点がある。Marvel ではクラスの継承機構を利用し、このような問題が生じないような記述を行うこともできるが、そのための方法は示されていない。

2.3 動作的モデルによる記述

Williams はプロセスを、前提条件、完了条件、開発者あるいはツールがすべき作業、および各作業間の通信の 4 つの要素から成るものとして SPM(Software Process Model) というモデルを提案し、これに基づいてプロセスの記述を行っている [10]。SPM のように、いつ、どのような作業を行うのかということを中心に、繰り返し、前提条件、完了条件などによってプロセスを記述しようとするモデルを動作的 (behavioral) モデルと呼ぶ。

動作的モデルは作業の手順を中心にプロセスを記述するため、プロダクトの管理方法についての情報は作業手順の記述の一部に含まれてしまう。従って、プロダクトに関する記述を作業手順の記述と独立に記述することが難しいといった問題点がある。

図3は動作的モデルに基づいたプロセス記述を行うために我々が開発した言語 SPDL[5] による、簡単なCプログラム開発の記述の一部である。SPDLでは各作業単位を前提条件、行う作業、完了条件の組で記述する。例えば、compile というプロセスでは、ソースプログラムの存在が前提条件であり、存在しない場合は edit プロセスが実行される。前提条件が満たされた場合は実際にコンパイルが行われる。完了条件はコンパイルが正常に終了したことであり、そうでない場合は edit プロセスが実行される。

SPDLではファイル名、ツール名は一種のマクロとして記述されている。プロセス compile でも、ツール cc を適用することやその終了状態の検出方法など、プロダク

トに対する具体的な操作内容が作業手順の定義に含まれている。

2.4 問題点

プロセス記述の部品化、再利用を進めるためには、部品となる記述の単位のモジュール化が必要である。それぞれの部品の機能を明確に定義し、不必要な内部情報を隠蔽することによって、部品の汎用性を高めることができる。

上で取り上げたプロセス記述の方法でも、プロセスをモジュール化して記述し、再利用することは可能である。しかしこれらの方法では、どのようにすれば部品化や再利用といった目的に適した記述を行うことができるかという点が示されていない。

さまざまなプロセス記述で、あるモジュールが共通の部品として使用されるためには、それらのプロセス記述がそのモジュールを同じ枠組みの中で使用できなければならない。個々のプロセス記述がそれぞれ違った方法でモジュール分割をしているのでは、部品化や再利用を進めることは困難である。

そこで、プロセス記述をあらかじめ決められた構造に

Strategy Programming Environment

Imports: Programming Language;
Exports: EDITOR, COMPILER;

Classes

```
PROGRAM ::=
  timestamp: integer
  status: (compiled, notcompiled, errors, linked)
  executable: binary
PROCEDURE ::=
  timestamp: integer
  executable: binary
  status: (compiled, errors, notcompiled)
```

Tool

```
EDITOR
  edit[?n: TEXT] -> <mapping for "/usr/bin/emacs">
COMPILER
  compile[?p: PROGRAM] -> ...
  compile[?p: PROCEDURE] -> ...
```

Rules

```
edit[?p: PROCEDURE]:
  /* the editor can only be invoked by user */
  [ EDITOR edit ?p.code ]
  ?p.timestamp = Now();

compile[?prog: PROGRAM]:
  exists PROCEDURE ?p
  suchthat member(?prog.procs ?p):
    ?p.timestamp > ?prog.timestamp
  [ COMPILER compile ?prog ]
  ?prog.status = compiled and ?prog.timestamp = Now();
  ?prog.status = errors and ?prog.timestamp = Now();
```

図2 ルールによるプロセス記述
(文献[3]による)

```
product
  SPEC = "Spec.txt";
  SRC = "foo.c";
...
tool
  Editor = "vi";
  Viewer = "less";
...

process main [ SPEC -> SRC, EXE ] ==
  child edit, compile, link, test;
  ( !exist(SPEC, $) : exit("Spec doesn't exist") )
  [ edit, compile, link, test ]
  ()
endproc

process edit [ SPEC -> SRC ] ==
  ()
  [ (wopen, Editor(SRC), wclose)
    @ (wopen, Viewer(SPEC), wclose) ]
  ()
endproc

process compile [ SRC -> OBJ ] ==
  ( !exist(SRC, $) : edit )
  [ wopen, Compiler(SRC), wclose ]
  ( status($) <> 0 : edit )
endproc

process link [ OBJ -> EXE ] == ...

process test [ EXE -> ] == ...
```

図3 動作的モデルによるプロセス記述
(文献[5]による)

従って記述しておくことが考えられる。プロセス記述がどのような働きを持つモジュールからどのように構成されるかを、ひとつの記述モデルとして定めておくのである。一般のプログラムの場合には、記述対象が非常に広範であるため、あらかじめ記述モデルを決定しておくことはできないが、プロセス記述の場合は記述対象が比較的限定されているため、ある程度まで記述の構造を決定しておくことが可能である。

この記述モデルはまた、作業手順の記述とプロダクト管理の記述が混在することなど、上の例で示した問題点を解決できるような構造を持つことが望ましい。

3. オブジェクトによるプロセス記述

3.1 オブジェクトとクラスの導入

ここでは、開発作業に伴って生成、使用されるさまざまなプロダクトとそれらに対する操作を記述するため、オブジェクトの概念を導入する。

プロダクトはその形式、用途、内容などによって、仕様書、ソースプログラム、実行プログラムなど、いくつかの種類に分けられる。プロダクトはこれらの種類ごとに、共通に持つ属性、および操作についての記述をクラスとして定義する。例えばソースプログラムに対する操作としては、内容の表示や編集、また、属性としては使用言語や更新時刻を考慮することができる。実際に開発で使われるプロダクトは、このクラスのインスタンスオブジェクトとして動的に生成されるものとする。

また、クラスの定義に継承機能を導入することによって、さまざまなクラスをより簡単に分かりやすく記述することができ、記述の再利用が容易になる。

本稿ではクラス定義の詳細については論じないこととするが、例プログラム中では、整数、文字列などのデータ型、および集合などのデータ構造はすでに定義されているものとして自由に用いる。

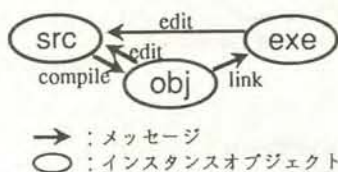


図4 簡単なC開発プロセスのオブジェクトとメッセージ

図5 簡単なC開発プロセスのクラスによる記述

```

class SRC is
  var OBJ next;
  proc edit(STRING name) {
    exec("vi " + name + ".c");
    next.compile(name);
  }
end;

class OBJ is
  var SRC back;
  var EXE next;
  proc compile(STRING name) {
    exec("cc -c " + name + ".c");
    if (status() == 0) next.link(name);
    else back.edit(name);
  }
end;
  
```

```

class EXE is
  var SRC back;
  proc link(STRING name) {
    exec("cc -o " + name + " " + name + ".o");
    if(status() == 0) execute(name);
    else back.edit( );
  }
  proc execute(STRING name) {
    exec(name);
    select {
      "ok" : exit(0);
      "no" : back.edit( );
    }
  }
end;
  
```

3.2 プロセスの記述

開発作業に伴って作成されるプロダクトをオブジェクトと見なし、それぞれに対する操作と属性の定義をクラスとして記述することによってプロセス記述を得ることができる。

例として、C言語による簡単な開発作業をクラス定義によって記述してみる。初めにソースプログラムを作成し、次にコンパイルを行なうが、失敗したらソースプログラムの編集に戻る。成功したらプログラムを実行させるが、結果が仕様と異なる場合はソースプログラムの編集に戻る。

オブジェクトと考えられるものは、ソースプログラム、オブジェクトプログラム、実行プログラムである。簡単のため、ソースプログラムはひとつのファイルからなるとする。それぞれのクラスのインスタンスの間では図4のようにメッセージが送られる。これからクラス定義を作成すると図5のようになる。組込手続き exec は与えられた文字列からツールを起動でき、関数 status はツールの実行が成功した場合に 0 を返す。

3.3 オブジェクトの並列実行

ソフトウェア開発では、いくつかの作業を並列に行いたいことがある。特にワークステーション上のウィンドウ環境では、関連するいくつかのソースプログラムを別々のウィンドウで並行して編集するなどの作業が頻繁に行われる。

並列に行われる複数の作業が、同時にあるひとつのプロダクトの内容を変更するということは通常は生じない。そこで、並列に行われる作業はそれぞれ別のオブジェクトに対する操作であると見なすことができ、並列作業を複数のオブジェクトの並列実行として自然な形で記述することができる。このことは、オブジェクトによるプロセス記の利点のひとつであると考えられる。

オブジェクトの並列実行の方法についてはさまざまな方法が提案されているが、本稿では実現の方法については立ち入らないことにする。

4. プロセス記述のモデル

ここでは開発プロセスを記述するための、オブジェクトに基づいた記述モデルである Trineモデルを提案する。この記述モデルは、プロセスの流れの定義、プロダクトの相互関係の定義、プロダクトへの操作の定義の3つの層からなる。記述されたプロセスは図6のようなオブジェクトの構成を持つ。各層の間の情報交換は、操作の起動を通してのみ行うものとする。

以下では、各層の記述方法と例を示し、次にこのモデルを用いて開発プロセスを記述する手順について述べる。

4.1 プロセスの流れの記述

一番上の層のオブジェクトは開発作業によって作成されるプロダクトのうち、「仕様」、「ソースプログラム」といった同じ種類のプロダクトをそれぞれ集合的に表している。この層ではこれらのオブジェクトがどのような操作を持ち、どのように開発作業を進めてゆくかを記述する。プロダクトに対する操作は中段の層のオブジェクトの操作を起動することで行い、この層では記述しない。

例えば、Cプログラムを開発するプロセスは図7の上段のようなオブジェクト間の関係として表すことができる。まず仕様が用意できたらソースプログラムに対してプログラム生成の操作が起動され、プログラムが作成できたらオブジェクトプログラムに対してコンパイル操作が起動される。コンパイルが成功すれば、実行プログラムに対してリンク操作が起動されるが、失敗すればソースプログラムに対して編集の操作が起動される。

図7のプロセスの流れをクラスとして記述したものを図8に示す。クラス P_SPEC, P_SRC, P_OBJ, P_EXE はクラス PROCESS を継承しており、変数 back, next はひとつ前の作業を行うオブジェクト、次の作業を行うオブジェクトを示す。変数 rel は中段の層のオブジェクトを指す。

4.2 プロダクトの相互関係の記述

中段の層は、上段の層の抽象的なプロダクトが実際にどのような相互関係を持つかを表すオブジェクトである。

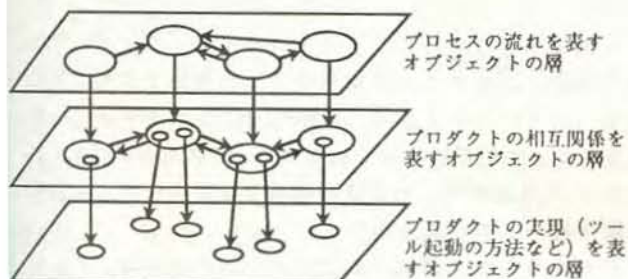


図6 Trineモデルにおけるオブジェクトの層構造

上段のオブジェクトがいくつかのプロダクトの集合を表している場合には、その集合に対する操作および個々の要素に対する操作を記述する。

開発作業を効率的に支援するためには、さまざまなプロダクト間の関係についての情報を利用することが必要である。プロダクト間の相互関係としては、一方から他方が生成される関係、参照・被参照の関係、同じプロダクトで版が違ふという関係など、さまざまなものを考えることができる。この層では、上段の層からの操作の記述に加え、さまざまな関係を持ったプロダクト間の情報交換、操作の起動についての記述を行う。プロダクトの個数やそれらの間の関係は開発作業中に動的に変化することも多いが、これらの管理方法も記述する。

図9にオブジェクトプログラムの相互関係を定義するクラス R_OBJ と OBJCELL を示す。オブジェクトプログラムは複数個存在するため、個々のオブジェクトプログラムを表すクラス OBJCELL と、その集合を管理するクラス R_OBJ を用いている。R_OBJ のインスタンスの compile という操作が起動されると、すべての OBJCELL のインスタンスの compile が起動される。OBJCELL では、対応するソースプログラムの更新時刻を調べ、変更されているものについてのみコンパイルを行う。変数 ent は一番下の層のオブジェクトを指す。

4.3 プロダクトの実現の記述

一番下の層は、中段の層のプロダクトの具体的な実現として、実際に用いられるツールや対応するファイル名など、使用するシステムや開発環境に依存する部分を定義したオブジェクトからなる。

図10はそれぞれ文書とオブジェクトプログラムの実現を定義したクラス TEXT と E_OBJ の記述である。wopen は表示のためのウィンドウを開く組込手続きである。

このプロセス記述の全体を、付録として本稿の最後に掲げる。

4.4 プロセス記述の手順

Trineモデルによるプロセスの記述は、次のような手順で行うことができる。

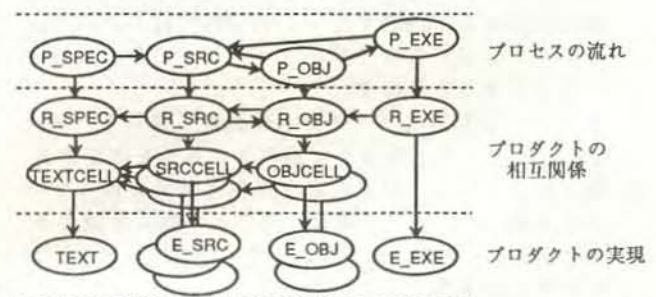


図7 C開発プロセスにおけるオブジェクトの構成

```

class PROCESS is
  var PROCESS back,next;
  var RELATION rel;
end;

class P_SPEC : PROCESS is
  var STRING name;
  proc start() {
    rel.initialize(name+".spec");
    next.edit(); /* P_SRC */
  }
end;

class P_SRC : PROCESS is
  proc edit() {
    if (rel.edit())
      next.compile(); /* P_OBJ */
    else exit(0);
  }
end;

class P_OBJ : PROCESS is
  proc compile() {
    if (rel.compile())
      next.link(); /* P_EXE */
    else back.edit(); /* P_SRC */
  }
end;

class P_EXE : PROCESS is
  var PROCESS src;
  proc link() {
    if (rel.link(n)) execute();
    else src.edit();
  }
  proc execute() {
    if (rel.execute()) exit(0);
    else src.edit();
  }
end;

```

図8 プロセスの流れの記述

```

class RELATION is
  var set(CELL) cel;
  var RELATION ref;
end;

class CELL is
  var ENTITY ent;
  var TIME tm;
  proc remove() {
    ent.remove(); ent = NULL;
  }
end;

class OBJCELL : CELL is
  var SRCCELL src;
  var STRING name;
  proc OBJCELL(STRING n) {
    ent = ENTITY(n); name = n;
    tm = clock();
  }
  func compile() {
    var BOOL ok;
    if (src.ent==NULL) return true;
    if (src.latest() < tm) return true;
    ok = ent.compile();
    if (ok) tm = clock();
    return ok;
  }
end;

```

```

class R_OBJ : RELATION is
  proc R_OBJ() {
    cel = null_set(OBJCELL);
  }
  func BOOL compile() {
    var BOOL ok=true;
    if (is_null(cel)) return false;
    foreach(OBJCELL c = cel)
      ok = ok & c.compile();
    return ok;
  }
  proc addsrc(CELL r, STRING n) {
    var OBJCELL c;
    c = OBJCELL(basename(n)+".o");
    c.src = r;
    cel = add_set(cel,c);
  }
  func STRING namelist() {
    var STRING n="";
    foreach(OBJCELL c = cel)
      if (c.src.ent!=NULL)
        n = n + " " + c.name;
    return n;
  }
end;

```

図9 プロダクトの相互関係の記述

```

class ENTITY is
  var STRING name;
  proc ENTITY(STRING n) {
    name = n;
  }
  proc remove() {
    exec("rm -f " + name);
    name = NULL;
  }
end;

class TEXT : ENTITY is
  proc view() {
    wopen();
    exec("less " + name);
  }
end;

```

```

proc edit() {
  wopen();
  exec("vi " + name);
}
end;

class E_OBJ : ENTITY is
  func BOOL compile() {
    var STRING s;
    wopen();
    s = basename(name)+".c";
    writeln("compile: " + s);
    exec("cc -c " + s);
    return status() == 0;
  }
end;

```

図10 プロダクトの実現の記述

- 1) 開発作業の中で用いられるプロダクトを列举し、どのプロダクトからどのプロダクトが作成されるかを調べる。それによって、プロセスの流れを操作の起動の関係として決定する。この段階では、ソースプログラムなど、いくつかのプロダクトの集合はひとつのまとまりと見なす。
- 2) 上で得た抽象的なプロダクトに対する操作が、個々のプロダクトにどのように適用されるかを決定する。
- 3) 個々のプロダクトがどのような相互関係を持っているかを調べる。また、属性、操作との関係も調べる。
- 4) プロダクトの属性、操作を、どのようなファイル、ツールなどで実現するか決定する。
- 5) プロセスの流れ、プロダクトの相互関係、プロダクトへの操作について、クラスの定義を行う。

5. 議論

5.1 モデルの有効性

Trineモデルに従ってプロセス記述を行うことによって、上の例で示したように、プロセスの流れ、プロダクトの相互関係、プロダクトの実現の3つの部分を各層ごとに独立に行うことができる。Trineはオブジェクト指向の考え方によって記述を行っているため、これらの層ごと、およびオブジェクトごとの情報隠蔽やモジュール化を自然な形で行うことができる。

このようにして構成されたプロセス記述では、それぞれの層および各層内部のオブジェクトの役割が明確にされている。これは2.で論じたモジュール分割の枠組みと

しての記述モデルの条件を満たしており、異なるプロセス記述の間でも部品化、再利用を容易に行うことができる。

5.2 関連する研究

篠田 [8][9] は、属性文法によるオブジェクトと、オブジェクトベースに基づく開発環境を提案している。プロセスを記述するための枠組みとして、属性値を保持できるように拡張された属性文法の部分木をオブジェクトと見なし、オブジェクトの実行によって開発作業が進められると考える。

オブジェクトベースとは、開発環境の中でプロダクトの属性や操作についての情報を専門に管理する仕組みである。オブジェクトベースはプロダクトに対する基本的な操作を提供し、プロセス記述ではこれらの機能を組み合わせてプロダクトの操作を定義する。プロダクトの実現方法に依存せずに、プロセス記述を容易に行うことができるようになると期待される。

開発環境は、開発者の定義した手順に従って作業を行うことのできる Authoring 層とオブジェクトベース層、および中間層からなる。Authoring 層から起動されたツールは中間層の提供するインタフェースを利用してオブジェクトベースにアクセスする。

ここで述べられている層の構造はプロセスの実行のための仕組みであり、プロセスをどのように記述するかという点については触れられていない。これに対し、Trine はオブジェクトの記述の枠組みを与えることによってプロセス記述の部品化、再利用をすすめることを主眼としている。オブジェクトベースのような仕組みを導入した場合にも、Trine の考え方にに基づくプロセス記述を行うことは有益である。

Fritsch[1] は M/C(Manager/Controller) という記述モデルを提案している。このモデルはプロセス記述層、プロダクト定義層、制御層の3つの層からなる。プロセスの実行に関する記述はプロセス記述層で行われ、その実行に伴ってプロダクト定義層への読み書きが行われる。制御層はプロセスの実行自体の管理や実行中の資源の管理などを行う。

このモデルも実行のための仕組みである。プロセスの実行を管理するための制御層を設けた点が特徴的であるが、制御層の機能をどのように記述するのか、プロセス記述層との切り分けはどのようにするのが明確ではない。

5.3 チームによる開発への拡張

本稿ではプロセスを、一人のソフトウェア開発者が行う作業に限定して考えている。しかし、各オブジェクトの並列動作を、協調して働くチームの個々の開発者の作業であると見なせば、Trineモデルをチームによる開発にも適用することができる。この場合には、各開発者の間のメッセージのやりとりや、分散した資源やプロダクトの

管理をどのように行うかということが新たに問題となる。

さらに実際の開発では、各開発者をどの作業に配置するのか、全体の作業の管理や運営をどのように進めるのか、などといった人的な側面が開発作業全体により大きく影響すると考えられる。これらをどのようにプロセス記述に取り入れるかについては今後の検討が必要である。

6. おわりに

本稿では、オブジェクト指向に基づいたプロセス記述モデル Trine を提案し、その有効性を論じた。また、Trine に基づくプロセスの記述方法について述べ、記述例を示した。

現在、Trineモデルに基づくプロセス記述が可能な言語、およびその実行処理系を設計中である。この処理系では複数のオブジェクトを並列に動作させることを目指している。また、複数の計算機による開発を支援する機能や作業しやすいユーザインタフェースについても検討中である。

参考文献

- [1] G. A. Fritsch and D. L. Perry: "A Manager/Controller for the Software Development Process", Proc. of 5th International Software Process Workshop, (to appear) (1989).
- [2] K. Inoue, T. Ogiwara, T. Kikuno and K. Torii: "A Formal Adaptation Method for Process Descriptions", Proc. of 11th ICSE, pp. 145-153 (1989).
- [3] N. S. Barghouti and G. E. Kaiser: "Implementation of a Knowledge-Based Programming Environment", Proc. of 21st HICSS, pp. 54-63 (1988).
- [4] G. E. Kaiser: "Experience with Marvel", Proc. of 5th International Software Process Workshop, (to appear) (1989).
- [5] 荻原, 飯田, 井上, 鳥居: "ソフトウェア開発環境定義用スクリプトの生成法について", 情報処理学会「CASE環境」シンポジウム論文集, pp. 7-14 (平 1-3).
- [6] 荻原, 井上, 鳥居: "オブジェクト指向によるソフトウェアプロセスの記述と実行について", 情報処理学会研究報告 SE-70-4 (平 1-12).
- [7] L. Osterweil: "Software Processes are Software Too", Proc. of 9th ICSE, pp. 2-13 (1987).
- [8] Y. Shinoda: "Object based Integration of Software Environments", Proc. of International Symposium on Future Software Environments, pp. 117-123 (1989).
- [9] Y. Shinoda and T. Katayama: "Object Oriented Extension of Attribute Grammars and Its Implementation Using Distributed Attribute Evaluation Algorithm", Proc. of the International Workshop on Attribute Grammars and their Application, (to appear) (1990).
- [10] L. G. Williams: "Software Process Modeling: A Behavioral Approach", Proc. of 10th ICSE, pp. 174-186 (1988).

付録

Trineモデルに基づくプロセス記述の例を示す。

```
class EXAMPLE is /* main */
  var PROCESS spec, src, obj, exe;
  var RELATION r_spec, r_src, r_obj, r_exe;
  proc EXAMPLE(STRING name) {
    init_process();
    init_relation();
    spec.name = r_exe.name = name;
    start();
  }
  proc start() { spec.start(); }
  proc init_process() {
    spec = P_SPEC(); src = P_SRC();
    obj = P_OBJ(); exe = P_EXE();
    spec.next = obj.back = exe.src = src;
    src.next = exe.back = obj;
    obj.next = exe;
  }
  proc init_relation() {
    spec.rel = r_spec = R_SPEC();
    src.rel = r_src = R_SRC();
    obj.rel = r_obj = R_OBJ();
    exe.rel = r_exe = R_EXE();
    r_src.ref = r_spec; r_src.obj = r_obj;
    r_obj.ref = r_src; r_exe.ref = r_obj;
  }
end;
```

```
/* Process ... See Fig.8 */
class PROCESS is ... end;
class P_SPEC : PROCESS is ... end;
class P_SRC : PROCESS is ... end;
class P_OBJ : PROCESS is ... end;
class P_EXE : PROCESS is ... end;
```

/* Relation */

```
class RELATION is
  var set(CELL) cel;
  var RELATION ref;
end;
```

```
class CELL is
  var ENTITY ent;
  var TIME tm;
  proc remove() {
    ent.remove(); ent = NULL;
  }
end;
```

```
class TEXTCELL : CELL is
  proc TEXTCELL(STRING name) {
    ent = TEXT(name); tm = clock();
  }
  proc view() { ent.view(); }
  proc edit() {
    ent.edit(); tm = clock();
  }
end;
```

```
class SRCCELL : TEXTCELL is
  var set(CELL) othr;
  proc SRCCELL(STRING name) {
    othr = null_set(SRCCELL);
    ent = TEXT(name);
    tm = clock();
  }
  proc relation(set(SRCCELL) cset) {
    CELL x;
    BOOL cnt;
    for(cnt=true; cnt;) {
      select {
        "add" : {
          x = select_set(cset);
          othr = add_set(othr,x);
        }
        "delete" : ...
        "end" : cnt=false;
      }
    }
  }
  func TIME latest() {
    var TIME t = tm;
    foreach(CELL c = othr)
      if(c.tm > t) t = c.tm;
    return t;
  }
end;
```

```
class OBJCELL : CELL is /* See Fig.9 */
  var SRCCELL src;
  var STRING name;
  proc OBJCELL(STRING n) { ... }
  func compile() { ... }
end;
```

```
class R_SPEC : RELATION is
  var TEXTCELL doc;
  proc initialize(STRING n) {
    doc = TEXTCELL(n);
  }
  proc view() { doc.view(); }
end;
```

```
class R_SRC : RELATION is
  var RELATION obj;
  proc R_SRC() {
    cel = null_set(SRCCELL);
  }
  func BOOL edit() {
    while(true)
      select {
        "new file" : add_file();
        "edit" : edit_file();
        "show spec" : show_spec();
        "remove" : rm_file();
        "relation" : relation();
        "end" : return true;
        "stop" : return false;
      }
  }
end;
```

```
proc add_file() { ... }
proc edit_file() {
  var SRCCELL c;
  c = select_set(cel); c.edit();
}
proc show_spec() { ref.view(); }
proc rm_file() { ... }
proc relation() {
  var SRCCELL c;
  c = select_one(cel); c.relation(cel);
}
end;
```

```
class R_OBJ : RELATION is /* See Fig.9 */
  proc R_OBJ() { ... }
  func BOOL compile() { ... }
  proc addsrc(CELL r, STRING n) { ... }
  func STRING namelist() { ... }
end;
```

```
class R_EXE : RELATION is
  var E_EXE exe;
  var STRING name;
  func BOOL link() {
    var STRING list;
    list = ref.namelist();
    exe = E_EXE(name);
    return exe.link(list);
  }
  func BOOL execute() {
    return exe.execute();
  }
end;
```

/* Entity */

```
class ENTITY is ... /* See Fig.10 */ ... end;
class TEXT : ENTITY is ... end;
```

```
class E_SPEC : TEXT is end;
class E_SRC : TEXT is end;
```

```
class E_OBJ : ENTITY is
  func BOOL compile() ... /* See Fig.10 */ ...
end;
```

```
class E_EXE : ENTITY is
  func BOOL link(STRING n) {
    wopen();
    exec("cc -o " + name + " " + n);
    return status() == 0;
  }
  func BOOL execute() {
    wopen();
    exec(name);
    select {
      "ok" : return true;
      "no" : return false;
    }
  }
end;
```