ソフトウェア工学 117－8
（1997．11．21）

コンテクスト概念に基づいたモジュール化機構

鵜林 尚靖　　玉井 哲雄
東京大学 総合文化研究科

現在，サブルーチン，手続き，オブジェクト，クラスなどいくつかのモジュール化方式が提案されている．機能中心型モジュールであるサブルーチンや手続きは「上位－下位」「全体－部分」という特性をもつトップダウン型階層構造のシステムを，オブジェクトやクラスはメッセージ通信により相互結合するネットワーク型構造のシステムを構成する．ところが，複数のモジュールをクロスして初めて意味を持つような機能の記述は，トップダウン型階層構造にも単純なネットワーク構造にも適していない．お互いに関連し合った機能が複数のモジュールに分散してしまい，一つの大域的モジュールとして表現することが困難だからである．このような処理の例として，リアルタイムシステムにおける例外処理やビジネスシステムにおけるユースケースなどがある．

本稿では，このような問題を解決するため，コンテクストという概念をもつ新しいモジュール化方式 COMM(Context Oriented Module Method) を提案する．コンテクストとは，ユースケースやシステムが動作する局面であり，これを一つのモジュールとみなす．COMM では，コンテクスト単位にシステムを分割しプログラミングするという方式をとる．すなわち，システムはコンテキスト記述の複合体として構築される．

A Modularization Mechanism Based on Contexts

Naoyasu UBAYASHI　　Tetsuo TAMAI
Graduate School of Arts and Sciences, University of Tokyo

A number of ideas on modularization mechanisms have been proposed based on such module units as subroutines, procedures, objects and classes. With subroutines and procedures, which are function oriented modules, top down hierarchies can be constructed that have "higher-lower" and "whole-part" relations. With objects and classes, which are object oriented modules, network structures can be constructed that connects components by message passing mechanism. Using these modularization mechanisms, it is difficult to build system functions that range over several modules. Such system functions fit to neither top down hierarchies nor simple network structures, because inter-related sub-functions are distributed among some different modules and an aspect of the system functions cannot be described within a global module. Examples of this kind of system functions include failure handling in real time systems and use-cases in business systems.

To solve this problem, we propose *COMM(Context Oriented Module Method)*, a new modularization mechanism based on a concept of contexts. Contexts correspond to use-cases or aspects of system functions. A context is regarded as a global module. In COMM, a system is divided into a set of contexts and implemented per context, that is, a system is composed of context descriptions.

1 はじめに

一般的に，大規模なシステムを構築する場合，すべてを一度に考えることはせず，別々に取り扱い可能な複数の単純な課題へと問題を構造化する．構造化手法やオブジェクト指向の概念はこのための方法であり，現在提案されているサブルーチン，手続き，オブジェクト，クラス，抽象データ型，パッケージなどのモジュール化方式のベースになっている．サブルーチンや手続きなど機能中心のモジュールで構築されたシステムは，「上位－下位」「全体－部分」という特徴をもつトップダウン型階層構造を形成する．一方，オブジェクトやクラスなどモノ中心のモジュールで構築されたシステムはネットワーク型構造を形成する．ところが，例外処理など複数のモジュールをクロスして初めて意味を持つような処理の記述は，現状のモジュール化方式の枠組内では難しい．機能中心のトップダウン型階層構造や単純なネットワーク構造では，記述が複数のモジュールに分散されてしまい，処理全体の見通しが困難になるからである．

本稿では，このような問題を解決するため，コンテクストという概念をもつ新しいモジュール化方式COMM(Context Oriented Module Method)を提案する．コンテクストとはシステムが動作する局面であり，複数のモジュールをクロスして初めて意味をもつような処理機能やオブジェクト指向開発方法論におけるユースケースなどがこれに相当する．COMMでは，コンテクストをモジュールと考え，この単位にシステムを分割しプログラミングするという方式をとる．システムは，「全体-部分」型の階層構造ではなく，コンテクストという平面が複数重なり合う並列構造として構成される．

本稿では，まず，第2節で，サブルーチン，手続き，オブジェクト，クラスなどの従来型モジュール化方式に内在する問題点について述べる．これを踏まえた上で，第3節で，COMMを提案すると共にいくつかの例を取り上げる．第4節で，COMMをベースとするプログラミング言語Edenについて述べる．第5節で関連研究について，続く第6節で今後の課題について述べる．最後に第7節でまとめを行う．なお，本稿では，「コンテクスト」と「文脈」を同じ意味で用いる．ただし，厳密にシステム概念を定義する際には前者を用いることにする．

2 従来型モジュール化方式の問題点

モジュール化方式とソフトウェアアーキテクチャ

モジュールは対象システムのソフトウェアアーキテクチャを決定する構成要素であり，モジュール化方式が異なれば，ソフトウェアアーキテクチャの構造も異なってくる．サブルーチンやオブジェクトなど現時点で提案されているモジュール化方式が構成するソフトウェアアーキテクチャには以下のような類形がある．以下，これらの概要と問題点をみていくことにする(表1).

- トップダウン型階層構造
- ネットワーク型構造
- フレームワーク，パターン型構造

表 1: 従来型モジュール化方式

No	方式	ソフトウェアアーキテクチャ	記述困難な問題
1	サブルーチン 手続き	トップダウン型階層構造	イベント駆動 型システム
2	オブジェクト クラス	ネットワーク型構造	システム局面 (コンテクスト)
3	同上	フレームワーク パターン型構造	多重的な システム局面

トップダウン型階層構造

第1番めのトップダウン型階層構造は，サブルーチンや手続きなど構造化手法をベースとするモジュール化方式により構成されるソフトウェアアーキテクチャである．これらは，部品としてシステムの中に組み込まれたり，他のモジュールから手続きとして呼ばれたりするものであり，システムが機能中心の「全体－部分」型の階層構造によって表現し得る場合に有効となる．通常，この場合のソフトウェアアーキテクチャはツリー構造になる．ツリー階層の上位に位置するモジュールは，その下位に位置するモジュール群の機能を統合した複合機能を表現するため，システム全体の機能はツリー構造を上から下に追って行けば理解できる．

ところが，Michael Jackson が[5]の中で指摘しているように，現実のシステムはトップダウン型階層構造のみでは表現が困難な場合が多い．特に，「上位－下位」「全体－部分」の関係で捉えられないシステム，たとえば，オペレーティングシステムやリアルタイム制御などイベント駆動型のシステムの記述は困難である．これらのシステムを，サブルーチ

ンや手続きなど，元来トップダウン型階層構造に向いたモジュールのみで構築すると，実際のイベントの流れと「上位－下位」「全体－部分」型のモジュール階層の間にギャップが発生する．必ずしも，システム全体の機能はツリー構造を上から下に単純に追って行っても理解できないという状況が発生する．

ネットワーク型構造

第2番めのネットワーク型構造は，オブジェクトやクラスなどオブジェクト指向によるモジュール化方式により構成されるソフトウェアアーキテクチャである．通常，実世界に対応する実体を表すものをオブジェクト，オブジェクトの型に相当するものをクラスと呼ぶ．オブジェクトはデータと手続きをカプセル化したものであり，メッセージ通信のみでお互いが結合される．そのため，その構造はネットワーク状になる．

構造がネットワーク状になるため，「上位－下位」「全体－部分」で表現できないシステム，特に，トップダウン型階層構造で問題となったようなイベント駆動型のシステムの記述は行いやすくなる．ところが，オブジェクトのようなモジュールは個々にカプセル化されるため，複数のモジュールをクロスして初めて意味を持つような処理の場合，これを一つの大域的モジュールとして表現することは，現状のモジュール化方式の枠組内では難しい．具体的な例としては，以下のようなものがある [1].

- failure handling
- persistence
- communication
- replication
- coordination
- memory management
- real-time constraints

これらの例のような機能を現状のモジュール化方式の枠組内でプログラミングすると，処理記述が複数のモジュールに分散されてしまい，処理全体の見通しが困難になる．例外ハンドラの記述を例にこの問題を考えてみることにする．この場合，障害発生という処理文脈で必要となるモジュール群や個々のモジュールに求められる処理機能は正常状態のそれとは異なる．どのように異なるかは，一般に，個々の処理内容を詳細に追って行かなければ理解することができない．というのは，図1のように，正常処理と例外処理の記述が不可分に絡み合っており，例外処理のみを別モジュールとして切り出すメカニズムが，現在のモジュール化方式の枠組にないからである．個々のモジュール単独では，従来の例外処理メカニズム － たとえば，C++ 言語や Java 言語における throw ～ catch などのメカニズム － を利用して例外処理を記述することは可能であるが，複数のモジュールが関わる処理文脈全体で例外処理の処理文脈のみを分離して記述することは難しい．

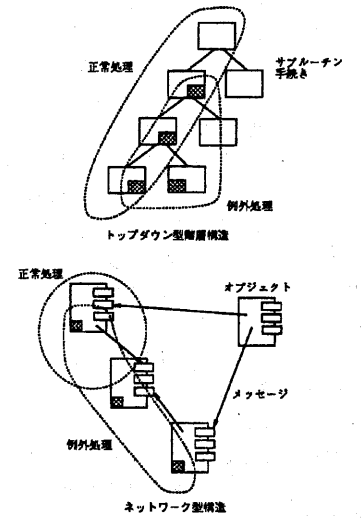

図 1: 従来型モジュール方式による例外処理の記述

フレームワーク，パターン型構造

第3番めのフレームワーク，パターン型構造は，モジュール化方式というよりは，複数モジュールの関係や協調関係を記述する枠組であり，通常，オブジェクト型のモジュールを対象にしている．フレームワークとはオブジェクト間の協調動作を抽象クラスの形で表現したものであり，実オブジェクトのクラスはこの抽象クラスのサブクラスとして定義される．フレームワーク中に現れる抽象クラス群の内容を追って行けば，そのフレームワークが提供する機能の実現方法を大まかに理解することができる．

一方，パターンには分析パターンや設計パターンなどいくつかあるが，モジュールの組織化という観点で特に関係するのは後者の設計パターン [4] である．フレームワークは特定の問題を対象にしたものであるが，設計パターンは汎用性のあるオブジェクト間の協調動作記述を対象にしている．たとえば，Creational Patterns, Structual Patterns, Behavioral Patterns などにカテゴライズされるものがある．

これらのパターンは，通常，フレームワークを記述する際に利用される．ただ，フレームワークの概念を導入したとしても，そのフレームワークが動作する文脈が一つでない場合，個々の文脈におけるオブジェクト間の協調動作の把握は依然として困難なままである．フレームワークにおけるクラス概念は固定的であり，オブジェクトが所属するクラスを動的に変更することはできない．したがって，抽象クラス間の関係で表現したオブジェクト間の協調動作は一つの文脈しか記述できない．通常，オブジェクトは複数の文脈に所属し，各々の文脈において，他のオブジェクト - 相手は文脈ごとに異なる - と協調動作しながら与えられた役割を果たすが，現状のフレームワークでは記述が難しい．文脈単位にフレームワークを記述し，個々のオブジェクトがフレームワーク中の抽象クラスを個々の文脈に応じて多重継承することにより，複数文脈に対応することは可能であるが，洗練された記述とは言いがたい．ある一つのオブジェクトに着目した場合，どのフレームワークのどの抽象クラスを継承しているのか，追跡して行くのは大変である．

以上，現在提案されているモジュール化方式に対応するソフトウェアアーキテクチャとして，トップダウン型階層構造，ネットワーク型構造，フレームワーク，パターン構造を見てきたが，いずれも，文脈という視点が抜けている．一般に，システムが動作する環境には複数の文脈がある．また，柔軟なシステム構造の実現を考えると，文脈自体が計算の進行と共に動的に変化する場合も考えられる．ところが，現在のモジュール化方式の枠組では，この文脈という概念をモジュールとして明示的に記述する方法がなく，システムのソフトウェアアーキテクチャの見通しを悪くしている原因の一つとなっている．

3 COMM

3.1 概念

COMM は文脈という概念を明示的に取り込んだモジュール化方式である．COMM では，計算主体のことをオブジェクト，複数オブジェクトが協調動作する文脈や環境のことをコンテクストと呼ぶ．通常，コンテクストは，システムが動作する局面 (システムを構成する機能あるいはユースケース) に対応して定義される．オブジェクトは以下のような手順を経てコンテクスト上で協調動作を開始し，システムのある局面機能を実現する (図 2).

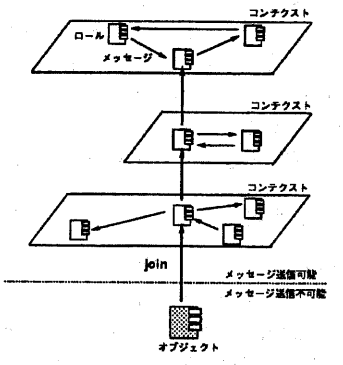

図 2: Context Oriented Module Method の概念図

1. オブジェクトは，どのような立場 (ロールと呼ぶ) で参加するかを指定してコンテクストに所属する．その際，オブジェクトはコンテクスト内でのみ有効なメソッド (ロールメソッドと呼ぶ) と属性 (ロール属性と呼ぶ) が付与される．
2. オブジェクトは，これらのロールメソッドとロール属性を使用して，そのコンテクストに所属する他のオブジェクトと協調動作を開始する．
3. コンテクスト上でオブジェクトが協調動作することによりシステムのある局面機能を実現する．

一つのオブジェクトは複数のコンテクストに所属でき，各々のコンテクストで他のオブジェクトと協調して与えられたシステム機能を実行する．各々のコンテクストは独立しており，オブジェクトは「そのコンテクスト内で付与されたロールメソッドとロール属性」および「オブジェクトが元来持っているメソッドと属性」のみを使用して動作する．他のコンテクストで付与されたロールメソッドとロール属性のことは考える必要がない．

図 2を見て分かるように，システムは複数のコンテクスト平面を並列に重ね合わせた構造になっている．コンテクスト自身は一つのモジュールとして扱うことが可能であり，これによって構築されたシステムは，「上位－下位」「全体－部分」のトップダウン型階層構造でも単純なネットワーク構造でもなく，「個々のシステム局面を示す平面の重ね合わせ」という構造になる．また，各コンテクストは独立しているので，システムに求められる局面を変更したい場合は，その局面に対応するコンテクストを別のものに置き換えれば良い．例外処理の例では，システ

ムを構成する複数の機能と共に，例外ハンドラ自身を一つのコンテクストとして定義することができる．

3.2 例

I / O処理システム

通常，I/O処理はプログラム中でread文やwrite文を使用して記述する．ソースプログラム中では単なるread文やwrite文であるが，実際には，OSレベルでのドライバ記述，ハードウェアレベルのコマンド記述，更には，異常が発生した場合での例外処理など様々な処理局面が存在する．通常，これらは動作レベルが異なるのでレイヤ別に記述されるが，全体としてお互いに関係しあっている．I/O処理は基本的にイベント駆動で動作するため，そのプログラムは，通常，メインプログラムをルートとする「全体－部分」型の階層構造にはならなく，各々の処理局面(イベント処理や例外処理など)単位に密接に関係しあった構造になる．ところが，現状のプログラミング言語には局面単位に処理記述できる枠組がないため，プログラマは無理に「全体－部分」型の階層構造に変換してプログラムを作成せざるを得なくなっている．そのため，出来上がったプログラムは非常に見通しの悪いものになっている．

COMMでは，局面単位に大域的モジュールを定義できる．たとえば，図3に示したようにコンテクスト分割することが可能である．お互いが密接に関係する処理は一つのコンテクストにモジュールとしてカプセル化されるため，処理全体の見通しは従来の「全体－部分」型階層構造と比べて，非常に良くなる．

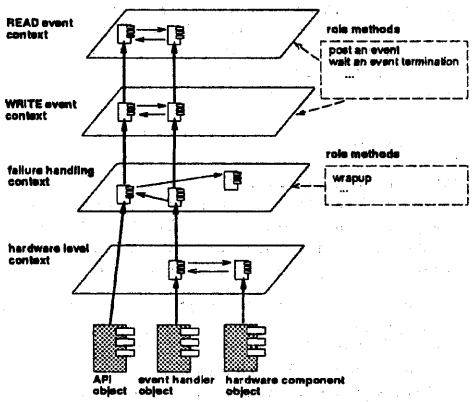

図 3: I/O処理システム

銀行口座のトランザクションシステム

図4に示すように，銀行口座に関するトランザクションには照会，引き出し，預け入れ，振替などのユースケースがあり，この単位にコンテクストが定義される．また，振替時エラーに対するリカバリ処理など複数のオブジェクトにまたがるものもコンテクストとして定義される．ユースケースにコンテクストが1対1に対応し，ユースケースを一種のクラスのように取り扱うことが可能である．

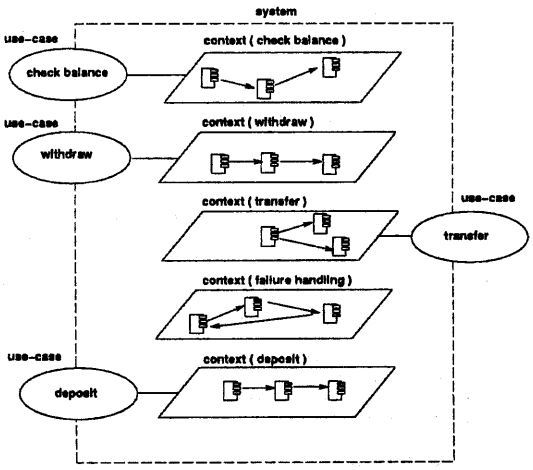

図 4: 銀行口座トランザクションシステム

言語処理システム

言語処理システムは，通常，字句解析，構文解析，コード生成の各フェーズからなる．字句解析フェーズでは，ソースプログラムを解析し記号表を生成する．続く，構文解析フェーズで記号表から構文木を生成し，コード生成フェーズで構文木から実行コードに変換する．従来型のモジュール化方式，たとえば，オブジェクト指向プログラミング方式で言語処理プログラムを記述した場合，構文木というオブジェクトには構文解析とコード生成の両方のメソッドと属性を用意しなければならない．そのため，構文木オブジェクトは巨大なオブジェクトになってしまい，保守性が困難になる．

COMMでは，構文解析フェーズとコード生成フェーズを別コンテクストとして記述できる．更に，コード生成のコンテクストを処理区分に対応させて，複数のサブコンテクストに分割することも可能である．

構文木オブジェクトは，これら複数のコンテクストに所属し，各々の役割を果たす．構文解析コンテクストでは構文木生成に必要なロールメソッドとロール属性のみが付与され，コード生成の各サブコンテクストでは構文木から実行コードへの変換に必要なロールメソッドとロール属性のみが付与される．コンテクストに必要なオブジェクト群のみがそのコンテクストのみで有効なロールメソッドとロール属性を使用して他オブジェクトと協調動作するため，処理の見通しが良くなる．構文解析コンテクストには構文解析処理に必要なものしか現れなし，コード生成コンテクストにはコード生成に必要なものしか現れない．しかも，これらは独立して記述される．

4 プログラミング言語 Eden

4.1 概要

筆者らは，現在，COMM が記述可能なプログラミング言語 Eden を開発している．Eden は Scheme をベースとして，コンテクストやクラス定義などの機能拡張を行った言語である．

```
;;-------------------------
;;  コンテクストの定義
;;-------------------------
(define-context コンテクスト名 (スーパクラス)
  (role ロール ...)
  (attribute 属性 ...))
(define-method コンテクスト名 (メソッド名 引数 ...) 本体 )
;; ロール
(define-role コンテクスト名 ロール名 (動的継承可能なクラス名)
  (attribute 属性 ...))
(define-method ロール名 (メソッド名 引数 ...) 本体 )
;;-------------------------
;;  クラスの定義
;;-------------------------
(define-class クラス名 (スーパクラス)
  (attribute 属性 ...))
(define-method クラス名 (メソッド名 引数 ...) 本体 )
```

コンテクストは define-context で定義される．ロール部にはコンテクストに所属するオブジェクト群のロールが列記され，属性部にはコンテクスト内で共通の属性が記述される．コンテクストのメソッドは define-method で定義され，ここには，オブジェクトのコンテクストへの所属やコンテクストからの離脱などの処理が記述される．

ロールの実際の内容は define-role で定義される．ここには，コンテクスト名とロール名の他に，そのロールを担い得るクラスが指定される．どのようなクラスのオブジェクトでもコンテクストに所属できるわけではなく，あらかじめここに指定されたもののみが所属しうる．ロールは通常のクラスと同様，属性，メソッドをもつことができる．メソッドは define-method で定義される．

クラスは define-class で定義される．属性部にはクラスオリジナルのインスタンス変数が定義される．また，クラスオリジナルのメソッドは define-method で定義される．

4.2 記述例

ここでは，3 節で取り上げた銀行口座のトランザクションシステムを Eden により記述する．なお，ここでは，説明の都合上，振替トランザクション・コンテクストのみを取り上げる．振替トランザクション・コンテクストには，振替処理全体を司る controller，振替元口座の source-account，振替先口座の target-account，トランザクションのログを管理する log の 4 つのロールがある．controller の指示の元で，source-account から指定された金額を引き出し，target-account に振り替える．履歴情報は log に書き込まれる．もし，振り替え時にエラーが発生した場合は，failure-handling コンテクストを生成し，そちらに処理を移行させる．振替トランザクション・コンテクストはあくまでも振替処理に専念し，エラーリカバリ処理自体は failure handling コンテクストに任せるというスタンスである．failure handling コンテクストでは，チェックポイントまでトランザクション処理をロールバックする処理を行う．

Descriptions of context

```
;;
;; context declaration
;;
(define-context Transfer ()
  (role controller:Transfer-transaction
        source-account:Transfer-account
        target-account:Transfer-account
        log:Transfer-log)
  (attribute check-point))

;;
;; context methods
;;
(define-method Transfer (init self)
  (set-role! controller (make Transfer-transaction))
  (set-role! source-account (make Transfer-account))
  (set-role! target-account (make Transfer-account))
  (set-role! log (make Transfer-log)))

(define-method Transfer (add-to-context self x role)
  (case role
  ((controller) (bind controller x))
  ((source-account) (bind source-account x))
  ((target-account) (bind target-account x))
  ((log) (bind log x))))
```

Descriptions of role

```
;;
;; for role controller
;;
(define-role Transfer Transfer-transaction (Transaction))
(define-method Transfer-transaction
            (start-transfer-transaction self amount)
  (set-attribute! check-point (log get-check-point))
  (if ((withdraw source-account amount))
      (if (not (deposit target-account amount))
        (let ((failure-handling (make Failure-handling)))
        (add-to-context failure-handling
                        controller 'controller)
        (add-to-context failure-handling
                        source-account 'source-account)
        (add-to-context failure-handling
                        target-account 'target-account)
        (start-failure-handling
          Failure-handling::controller check-point)))))
        ;;
        ;; if an error occurs, make a context
        ;;  'failure-handling' and move to the context
        ;; in order to roll back the transaction

;;
;; for role source-account and target-account
;;
(define-role Transfer Transfer-account (Account))
(define-method Transfer-account (withdraw self amount)
   ;; rewrite the method 'withdraw' in the class
   ;; Account first, call the method 'withdraw'
   ;;        in the class Account
   ;; next, write an action to the object log
   (write-log log ....))
(define-method Transfer-account (deposit self amount)
   ;; rewrite the method 'deposit' in the class
   ;; Account first, call the method 'deposit'
   ;;        in the class Account
   ;; next, write an action to the object log
   (write-log log ....))

;;
;; for role log
;;
(define-role Transfer Transfer-log (Log))
(define-method Transfer-account (write-log self..)..)
```

Descriptions of original class

```
(define-class account ()
  (attribute owner (balance 0)))
(define-method account (owner self) ...)
(define-method account (check-balance self) ...)
(define-method account (deposit self amount) ...)
(define-method account (withdraw self amount) ...)
```

5 関連研究

AOP

COMMは「コンテクスト単位にシステムを分割しプログラミングする」というモジュール化方式であるが，類似の考え方をもつ研究として，AOP(Aspect Oriented Programming)[1] がある．AOPでは，システムの基本機能(アスペクト)を最も適切なパラダイムで記述し，そのパラダイムにどうしても馴染まない部分(アスペクト)はそれにあった言語で記述する．異なる言語で書かれたプログラム群はweaverと呼ぶ特殊なコンパイラで編み合わされる．たとえば，複雑な分散処理システムを AOP で記述する場合，メインの部分はオブジェクト指向言語でプログラミングし，一方，複写や通信，分散の部分は各々目的に合った固有の言語でプログラミングするという具合になる．これらは自動的に編み合わされて，実行コードが生成される．AOP におけるアスペクト概念は，COMM におけるコンテクスト概念に対応する．AOP の実現方式は，COMM のそれとは異なるが，「アスペクト(コンテクスト)単位にプログラムする」という着想は共通している．特に，着想の原点である現状モジュール化方式に対する問題認識，すなわち，モジュールをクロスして初めて意味をもつ記述が難しいという問題認識は同じと考えて良い．

ネットワーク組織論

本稿で提案したコンテクストという概念は，社会科学の分野でも類似の研究事例が見られる．図5は，今井，金子らのネットワーク組織論 [9] からの抜粋である．ここでは，この図に沿って，彼らの主張を簡単に紹介する．(1) は新しい組織形態としてのネットワークを表すものとして良く解説書などで引用されるが，この図から受ける印象は，自己(ノード)が既にそれ自体独立して存在していて，それぞれの立場から互いの情報を交換するものである．あくまでも，(1) に表された関係はスタティックな情報を介したものであり，そこには関係の中で自己を再解釈していくという情報のダイナミック性はない．一方，(2) は彼らが提唱するネットワーク組織をイメージしたものである．(2) では (1) での自己(ノード)に対応するものは，それが持つすべての関係(文脈)の重なりとして定義される．自己は完成された形で存在するのではなく，自分の持つ関係のうち一つでも変われば自分の形も変化してしまう．関係を持つには自己が必要であるがその自己は関係の中でしか認識されな

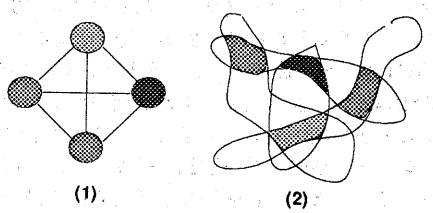

図 5: ネットワーク組織論

いというサイクルが必要である．そのためには，自己はインタラクションの中で自分自身を変化させるという自己言及性が必要となる．

自己をオブジェクトに，関係をコンテクストに置き換えてみると，(1) が従来のオブジェクト指向の考え方に，(2) が COMM の考え方に近いことが分かる．COMM ではオブジェクトの性質はコンテクストの重なりとして捉えることが可能であり，それは計算の進行と共に刻々と変化する．また，オブジェクトはコンテクスト内でのインタラクションの結果として，動的に新たなコンテクストを生成したり削除したりでき，その重なりを変化させることができる．今後，COMM に自己反映計算のメカニズムを付加すれば，より (2) の考え方に近づけることが可能になる．従来のオブジェクト指向の考え方では，オブジェクトは明確に他者と区別され，それ固有の機能を持つものとされてきた．計算の進行と共に役割が変化したり，他者との関係が変化したりすることは通常考慮されていない．柔軟なシステムを構築するには，本当にそれだけでよいか再考する必要があると思われる．コンテクスト概念は，これに対する一つのアプローチだと考えている．

6 今後の課題

現時点での COMM では，まだ，開発方法論における位置付けが明確になっていない．現在，オブジェクト指向開発の世界では，OMT 法 [7]，Booch 法 [2]，OOSE 法 [6]，それらを統一した UML(Unified Modeling Language)[3] などの方法論が提案されている．これらの方法論の中で重要な概念の一つとしてユースケースがある．ユースケースとは I.Jacobson によって提案された概念で，システムに求められる個々の機能を事例として外部アクターからの視点でピックアップしたものである．開発の初期段階にこのユースケースを抽出し，その後，このユースケースを元にメッセージトレース図を作成したり，オブジェクト抽出の補完に役立てたりする．

コンテクストはユースケース概念を形式化するメカニズムと考えることができる．その意味では，COMM は，現在提案されているオブジェクト指向開発方法論との相性が良く，それを補完する概念と捉えることが可能である．以下は，COMM を既存のオブジェクト指向開発方法論に織り込んだ手順のアウトラインである．具体的な手順決めは今後の課題としたい．

1. コンテクストの抽出：システム構築にあたっては，まず，「システムがどのようなコンテクストを持つか？」を分析するところから始める．コンテクストは，システムの局面やユースケースに対応して定義される．

2. ロールの抽出とロール間の協調動作の記述：各コンテクストに必要なロールを抽出する．その後，コンテクストによって実現される局面やユースケースに必要なロール間の協調動作を記述する．協調動作はメッセージ通信で実現する．

3. オブジェクトの抽出とロールとの対応づけ：システム中に現れるオブジェクトを抽出する．その後，抽出したオブジェクトとコンテクスト，ロールとのマッピングを定義する．

4. システムに求められるコンテクスト，ロール，オブジェクトが具体化するまで，上記の 3 つの手順を繰り返す．

7 おわりに

本稿では，コンテクスト概念に基づいた新しいモジュール化方式 COMM を提案した．COMM は，従来のモジュール化方式では記述が困難であった，複数モジュールをクロスして初めて意味を持つような処理記述を可能にする．今後は，課題のところでも述べたように，開発方法論における位置付けを明確にすると共に，COMM の有効性が分かるようなシステム記述例を増やして行きたい．

参考文献

[1] AOP, Home page: http://www.parc.xerox.com/aop, 1995.

[2] G.Booch, *"Object-Oriented Analysis and Design with Applications Second Edition,"* Redwood City, CA:Benjamin/Cummings, 1993.

[3] M.Fowler and K.Scott. *UML Distilled*, Addison-Wesley Publishing Company, Inc., 1997.

[4] E.Gamma, R.Helm, R.Johnson, and J.Vlissides, *"Design Patterns - Elements of Reusable Object-Oriented Software,"* Addison-Wesley Publishing Company, Inc., 1995.

[5] M.Jackson, "Software Requirements & Specifications - a lexicon of practice, principles and prejudices," Addison-Wesley Publishing Company, Inc., 1995.

[6] I.Jacobson, M.Christerson, P.Jonsson, and G.Overgaard, *"Object-Oriented Software Engineering - A Use Case Driven Approach,"* Workingham,England:Addison-Wesley Publishing Company, 1992.

[7] J.Rumbaugh, M.Blaha, W.Premerlani, F.Eddy, and W.Lorensen, *"Object-Oriented Modeling and Design,"* Englewood Cliffs, New Jersey:Prentice-Hall, 1991.

[8] 鵜林尚靖, 玉井哲雄, "オブジェクト間の協調動作を表現する自己反映並行計算モデル," 情報処理学会ソフトウエア工学研究報告 112-4, pp.25-32, 1996.

[9] 今井賢一, 金子郁容, "ネットワーク組織論," 岩波書店, 1988.