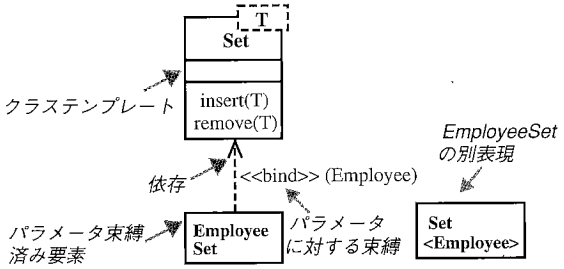
解説

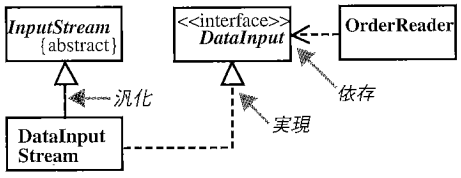
オブジェクト指向モデリング言語UML(2)

羽生田栄一
(株) オージス総研オブジェクト第1事業部技術コンサルティング室

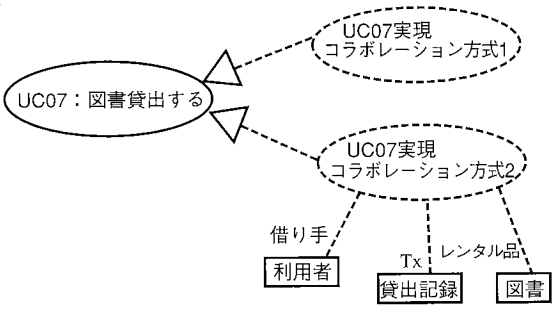
前稿（1）では，統一モデリング言語UML（Unified Modeling Language）の概要とそれを用いてモデルを表現するダイアグラム表記法を紹介した．本稿では，現在OMG内部で検討されているUML1.1を改良したUML1.3の仕様，UMLを使う上での注意点，課題を紹介する．最後に，UMLを前提としたオブジェクト指向開発プロセスとしてRational Unified Processを例に説明する．UML1.3は早ければ1999年4月に正式公開される予定である．



(a) <<bind>>依存関係



(b) インタフェースと実装クラスの間の実現関係



(c) ユースケースにおける実現関係

図-1 実現関係

UML1.3の変更点

UML1.3として現在改訂が予定されている部分のうち，UML1.1と比較して次の2点が重要である．

1) 関係の整理：従来，正確な解釈の難しかったモデル要素間の依存関係や洗練関係，およびユースケース間の「汎化」関係が整理されてすっきりした．
2) パラメータ化コラボレーション：パターン，フレームワークを前提にしたモデルを自然に表現できるように「コラボレーション」概念がパラメータ化された．

以下，これらの変更点を中心に紹介する[1), 2), 3)]．

■関係の整理

UML1.1では，そのメタモデルにおいて，関係は，関連（association），汎化（generalization），依存（dependency），洗練（refinement）の4種類に分類されていた．UML1.3では，関連，汎化，依存と実現（realization）の4種類に変更された．

UML1.1の洗練関係は，分析段階での「人」クラスとそれに対応して設計段階で登場するより詳細化されたPersonクラスのように，同じ対象の異なる抽象レベル表現の間の関係である．洗練関係は，次のような場合に使用されていた．

a) 分析クラスと設計クラスや設計クラスと実装クラス間の関係
b) クラステンプレートArray [T] とインスタンス化されたテンプレートクラスIntArrayの関係
c) インタフェースと実装クラスの間の関係

UML1.3では，a），b）の2つの洗練関係は，**図-1**（a）のようにステレオタイプ<<refine>>や<<bind>>の付いた依存関係として表現される．c）は，UML1.3では，仕様と対応する実装との間の実現関係として理解される．実現関係は，UML1.3では，インタフェースと実装クラスの関係（**図-1**（b）），およびユースケースとそれを実現するコラボレーションとの関係（**図-1**（c））にのみ利用される．この実現関係の導入によって，モデルの仕様と実現とを明確に区別できるようになった．

■ユースケース間の関係

UML1.1では，ユースケース間の関係は，**表-1**のよう

		使用汎化関係 <<uses>>汎化	拡張汎化関係 <<extends>>汎化
現行	定義		
	意味	ユースケースA, B, Cとそれらから共通に呼び出されて使用されるサブユースケースXとの関係	ユースケースPとその振る舞いを拡張するための追加ユースケースQとの関係
改訂	定義	<<include>>依存関係	<<extend>>依存関係
	意味	ユースケース記述からそれらに含まれる共通サブユースケース記述への依存（共通サブユースケース呼出し）	元となるユースケース記述とそれに依存してオプションや変更を補うための拡張部分の追加記述

表-1　ユースケース間の関係表現の変更

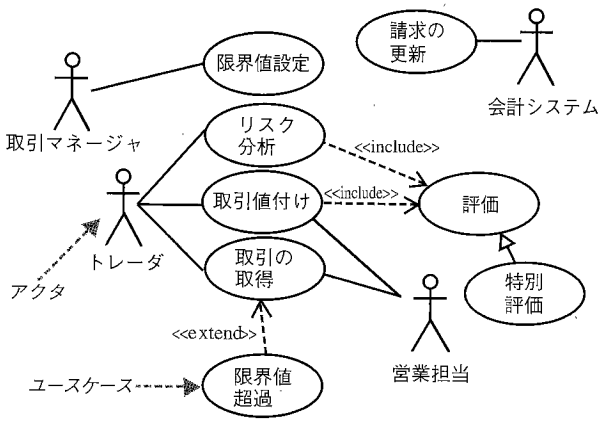
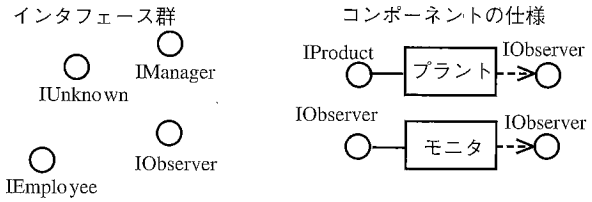
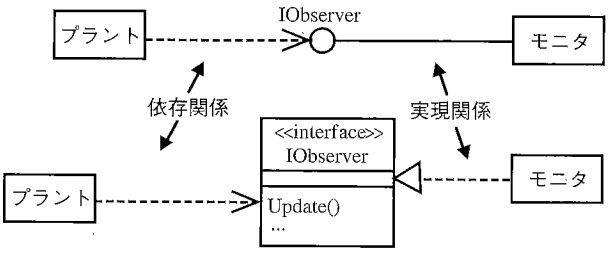


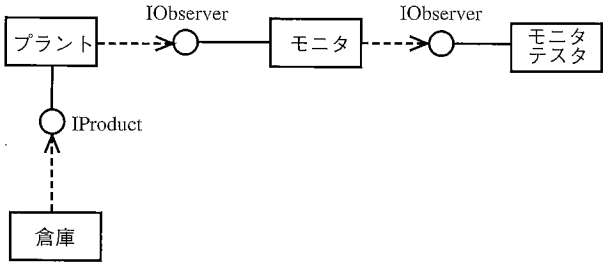
図-2　ユースケース間の関係



(a) インタフェースのロリポップ表現



(b) インタフェースとそのクラス表現との対応



(c) 複数のオブジェクト間のワイヤリング接続例

図-3　2種類のインタフェース表現

に，なぜか「汎化」関係として定義され，使用汎化関係（<<uses>>汎化）と拡張汎化関係（<<extends>>汎化）の2種類が用意されていた．

しかし，それらになぜ継承を意味する汎化関係と同じ記号が使われるのか，なぜ矢印の向きが逆ではないのか，皆いぶかしく思っていた．今回，これらの関係はいずれも「依存」関係として整理され，ユースケース記述どうしを簡潔に管理する仕掛けとして再解釈された．**図-2**に示すように使用汎化関係は<<include>>依存関係，拡張汎化関係は<<extend>>依存関係と定義され，素直に理解できるようになった．<<include>>依存は一種の共通ルーチン呼び出しである．一方，<<extend>>依存では，元のユースケースのどの個所が拡張可能かを「拡張点」として明示できる．拡張点では，元のユースケースにおいてあるステップ（拡張点）が特定の状態になったことをトリガとして拡張ユースケースに制御が遷移し実行（したがってオプショナルな実行である）後，元のユースケースに戻る．

さらに，これらとは別にユースケース間の汎化関係が新たに導入された．これによって，クラス間の汎化と同様に親ユースケースの振る舞いと意味を子ユースケースが継承する場合の表現ができる．

■インタフェースのロリポップ表現

従来，インタフェースは実装クラスから1個以上のロリポップ（棒とその先の玉）を出して表現していた．しかし，コンポーネントウェアが普及し，オブジェクトの提供するインタフェースだけでなく，コンポーネントをワイアリングして接続するプロトコルを表現したいという要求が出てきた．そのため，**図-3**（a）のような，ロリポップにおいて棒を取り除いた玉だけ（IUnknownやIManager等）の独立表現が許されるようになった．これにより，オブジェクト（たとえば「プラント」）が外部に公開するインタフェース（プラントの生成物情報IProduct），そのオブジェクトが機能実現にあたって想定している必要なインタフェース（プラントの状態変化通知IObserver）をそれぞれ表現できるようになった．これにより，(c) のようにコンポーネントどうしがどのようなプロトコルを介して接続可能かを示せるようになった．

■ロールとインタフェース指定子

従来，関連のロール名はクラスがその関連に参加する役割を示唆するに過ぎなかった．UML1.3ではそのような関連ロールも使用できるが，より明確に，あるインタフェースの使用を指定できるようになった．これは**図-4**に示すように，インタフェース指定子（interface specifier）を用いて「ロール名：インタフェース名」で表す．これによって，その関連ロールがどのようなインタフェースすなわち操作のセットによって実現されるべきかが指定できる．Personという同一クラスであっても，参加する関連におけるロールの違いを，異なる振る舞いを持つインタフェースによって演じられる役割の違いとして明示できる効用がある．

■オブジェクトの状態表示

あるインスタンスオブジェクトがある時点において特定の状態にあるということを明示したい場合がある．UML1.3では，図-5に示すようにインスタンス名の下に[状態名]として表現できる．

■生成消滅メッセージ<<create>><<destroy>>

従来，人によってさまざまなメッセージ名が使われていたインスタンスの生成と消滅に対して，共通のステレオタイプ付きメッセージ<<create>>，<<destroy>>が与えられ統一された．

■パラメータ化コラボレーション

オブジェクト間の協調的振る舞いを表現するのがコラボレーションである．UML1.3では，コラボレーションに参加する各オブジェクトの役割を抽出しテンプレート化した「パラメータ化コラボレーション」が導入された．これによって，たとえば，デザインパターンObserverとそれを実装している具体的な複数のオブジェクトのコラボレーションとを区別して表現できる．これを利用すれば，フレームワークをパラメータ化コラボレーションの集合で表し，それらの適切なホットスポット（パラメータで表される）に具体的なオブジェクトやモデル要素のインスタンスを束縛して実際のアプリケーションが構築される，という関係をUMLで表現できる．

今後，アナリシスパターン，デザインパターン，アーキテクチャパターン等から構成される各ドメインのフレームワークを前提にアプリケーションを構築していく開発スタイルが普及することを考えると非常に重要な改良点といえる．ただし，パターンの仕様として扱うには適切な制約記述が必要なことはいうまでもない．

UMLの課題

現在のUMLは，既存のオブジェクト指向技法で採用しているモデリング概念はほとんど取り込み，そのセマンティクスもメタモデルを使って整備されてきたが，いくつかの不備や問題点も存在する．ここでは文献6）～8），12）および著者自身の経験を参考にして，重要な問題点をいくつか指摘する．

■ユースケースに関して

ユースケースはUMLの第1級のモデル要素である以上，ユースケースの意味について明確な指針が必要であるが，今のところ人により解釈が分かれる．特に，構造化分析や機能分割法におけるプロセスや機能との違いや適切な粒度が曖昧である．

また，ユースケースの内容記述に最低限どれだけの情報が必要なのか，その記述スタイルはどうあるべきかの定義もない．現在，アクタに対して意味のあるサービスを提供するシステムの振る舞い，システムの基本操作，ドメインプロセスといった説明があるが，UMLが開発プロセスと切り離されたことも一因となってきちんと定式化できていない[12]．

また，アクタの扱いが一様でない．当該システムに関与する外部システムをアクタにする場合に，付随する相互作用をユースケースとすべきか否かで合意がなくユースケースを識別する上で恣意性を生じている[8]．さらに，ユースケースの具体的な事例をシナリオと呼ぶが，シナリオはどのような意味でユースケースの事例なのかといった，両者の関係の正確な解釈も曖昧である．なぜなら，ユースケース記述は，抽象的アルゴリズム表現というよりは，当事者をパラメータ化したものではあるが振る舞いの典型的なシーケンスであることが多いからである．一方，シナリオも完全に具体的な人物やインスタンスであるよりは「少年A」や「図書X」といったプロトタイプインスタンスが登場することの方が一般的であるため，ユースケースとの抽象度の違いが曖昧である．現在，ユースケース記述を，通常の典型例である主ユースケース，主ユースケースのフローの中途で分岐する2次的ユースケース，正常に終了しない例外ユースケースなどと分類す

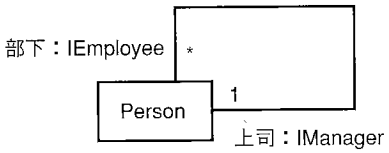


図-4　ロールとインタフェース指定子

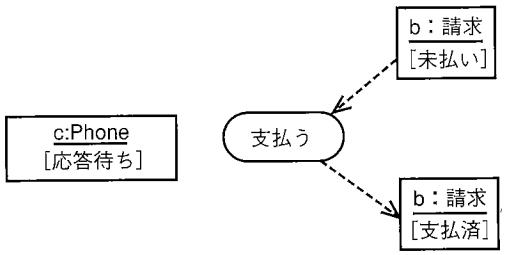


図-5　オブジェクトの状態とアクティビティ図

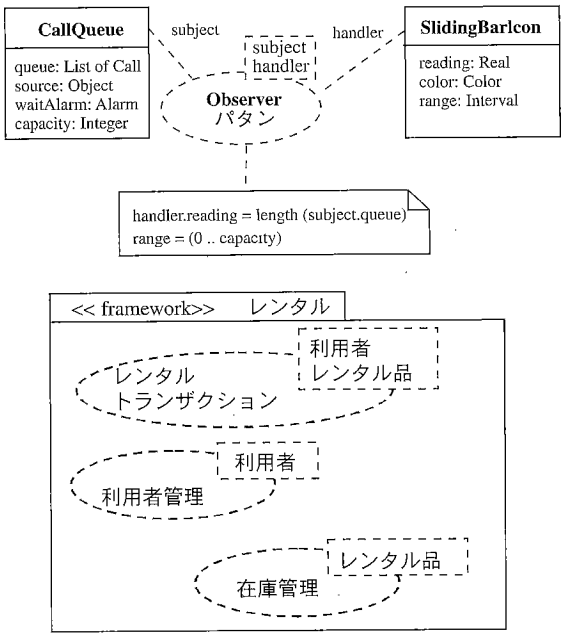


図-6　パラメータ化コラボレーションの応用

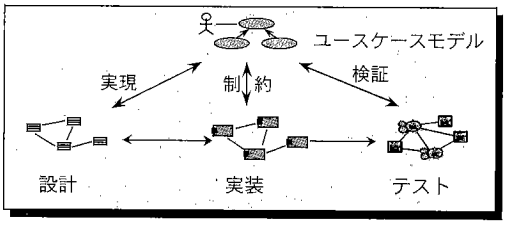


図-7　ユースケース駆動

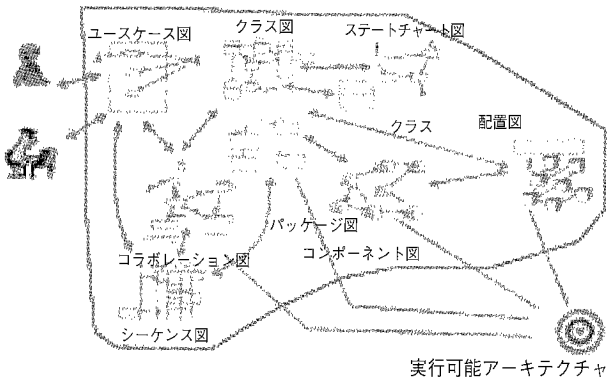


図-8　UML各モデル図とアーキテクチャ・コード

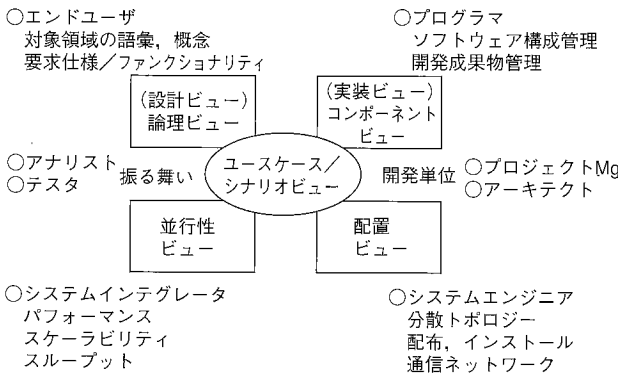


図-9　4+1ビューとアーキテクチャ

ることがある．こうやって区別された各フローがそれぞれシナリオであると考えるのが妥当だろう．

ユースケース間の汎化関係もその呼称が要らぬ誤解を招いてUMLの汚点であったが，先に述べたようにUML1.3では依存関係として解決した．

■関連クラスの存在意義

関連クラスはオブジェクト間の関連自体を実体化したオブジェクトであるが，その定義が一部曖昧であることからセマンティクスがきちんと理解されていない．UMLでは，関連クラスのインスタンスの存在は，対応する関連の両端にある2つのオブジェクトの存在に依存するはず（そうでないと普通のクラスとは別にわざわざ関連クラスを導入する意味がない）である．一般に，その制約は認識されているとはいい難く，通常クラスと同じ意味解釈が適用されやすく，注意を要する．

■シーケンス図の活性区間

シーケンス図はそこに参加するオブジェクトが活性であることをオブジェクト生存線上の白抜きの柱（制御フォーカスとも呼ばれる）を示すことで表現できる．しかし，あるオブジェクトが活性であるという状態の意味が定義されていないためその解釈が曖昧である．それは，(a) スレッドに対応，(b) メソッド起動状態（言語処理系における関数・手続きの実現方式としてのスタックフレーム）に対応，の両方の解釈が入り混じっているため，正確な記法がどうあるべきなのか（戻り値の矢線を書くべきか，非同期メッセージを2度目に受信するとき活性区間は重なるべきかなど）判断できないケースが出てくる．

■アクティビティ図の位置付け

アクティビティ図はステートチャート図の拡張であるが，その目的や用途は大いに異なる．ステートチャート図が各オブジェクト単位のライフサイクルの表現に限定されているのに対し，アクティビティ図はある機能や処理を複数のステップ（各々をアクティビティといい，それぞれが並行処理であってもよい）のフローとして実行する様子を表現できる．したがって，ユースケースの内容記述にはもってこいの図だが，UMLではユースケース図とアクティビティ図はモデル構成の中で積極的には結び付けられていない．

UMLと開発プロセス

開発プロセスとは，開発の手順であり，オブジェクト指向でシステムを開発する際の開発局面や手順，各開発局面での成果物の構成，および各開発局面や手順を構成する作業内容に関するガイドラインやノウハウといったものの体系から成り立っている．しかし，UMLに基づく開発プロセスは，UMLの仕様には含まれない．したがって，中間成果物も含めて開発成果物をUMLに基づいて記述すると決めても，さまざまな開発プロセスがありうる．当然，開発組織の規模や体制や文化，開発対象分野や製品の特性に応じて適切な開発プロセスは異なる．現在，モデルを文書化する言語としてUMLを採用するある程度汎用の開発プロセスとして多くのものが名乗りを挙げている．有名なものだけでも，Rational Unified Process[4]，HPFusion法，Shlaer/Mellor法，組み込み向けのOCTOPUS法[9]などがある．

ここでは，初期からUMLを意識し，体系としても最も整備され広く使われつつあるRational SoftwareのRUP（Rational Unified Process）を紹介する．これは，もともとObjectory法として知られていた手法を進化させたものである．汎用的なオブジェクト指向システム開発技法になっており，一種の開発プロセスの枠組みである．既存の一般的なオブジェクト指向開発手順を取り込んで標準化するとともに，分野や組織に合わせてカスタマイズできる点で注目できる．また，UMLの特徴として，モデル中でユースケース概念がオブジェクト概念と同等に重視される点が挙げられる．その結果，開発プロセスを進めていく1つの駆動力としてユースケースを取り込んでいるのも重要な点である．

RUPの特徴は，既存のオブジェクト指向開発のみなら

ずソフトウェア工学的に有効な経験，知見をうまく取り込んで，UMLによるビジュアルモデリングを前提に1つの体系的な開発手順の枠組みを提示している点である．その枠組みは，(1) ユースケース駆動，(2) アーキテクチャ中心，(3) 管理された繰り返し，(4) プロセス自体のカスタマイズ可能性，の4点で特徴付けられる．

■ユースケース駆動

RUPではシステムに対する機能要求を基本的にユースケースで表現管理するため，ユーザとのコミュニケーションもとりやすい．また，ユースケースがシステムを構成するクラスやその振る舞いを識別し分析モデルを作成する際の基本情報であるのみならず，設計・実装の検証やテストケース作成，ユーザマニュアル作成の情報源としても縦横に活用される．さらに，繰り返し型の開発の各サイクルを駆動する基本単位にもユースケースが採用され，「第n回目のサイクルでは，ユースケースのuc1，uc5およびuc9の基本フローの実現が目標」というように繰り返しのマイルストーンとして使われる．

■アーキテクチャ中心

RUPはシステムに対するモデルとそれを実現する実行可能なソフトウェアシステムそのものとを並行して開発を進めていく．つまり単なる机上のモデルではなく実際に動作させ，機能性・信頼性・安定性等のソフトウェア特性を確認しながらシステムの「アーキテクチャ」を進化させていくことを提唱している．システムに対する要求をサポートし，ミドルウェア／プラットフォームの制約や非機能的要求を反映した実行可能フレームワークをここではアーキテクチャと呼んでいる．そのアーキテクチャの上に段階的に個々のアプリケーションを実現するクラスを載せていく．

したがって，アーキテクチャは複数のビューで相補的にモデル化されるが，基本的なビューは図-9に示す4+1個考えられる．1) 機能や論理的な構造を表現する論理ビュー，2) それを実現するサブシステムやコンポーネントの構成管理を表現する実装ビュー，3) 効率やスループット，スケーラビリティを検討する並行性ビュー，4) システムのトポロジーや物理的な分散・配布を表現する配置ビュー，5) システムの要求や振る舞いを管理するユースケースビューの5つのビューでシステムのアーキテクチャの管理項目を多次元的に表現する．

このアーキテクチャが以後の繰り返し開発のベースラインとなり，この上に個別のユースケースや詳細な設計情報が実現されていく．その意味で，本格的な繰り返し開発に入る前に安定させておく必要があり，各ユースケースに相当する機能を実現するアプリケーション開発のメンバとは別の「アーキテクト」が中心になって，構築フェーズに入る前に作業を進めておく．

■管理された繰り返し

よくオブジェクト指向はスパイラルに開発を進めるから予定通り開発を行えないという意見を聞くが，それは各繰り返しサイクルの目標がきちんと定義されていないから

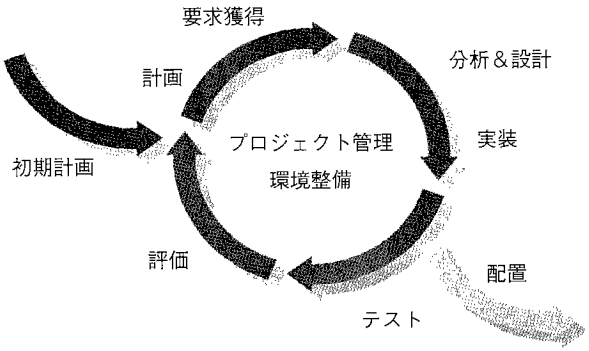


図-10　1サイクルの構造

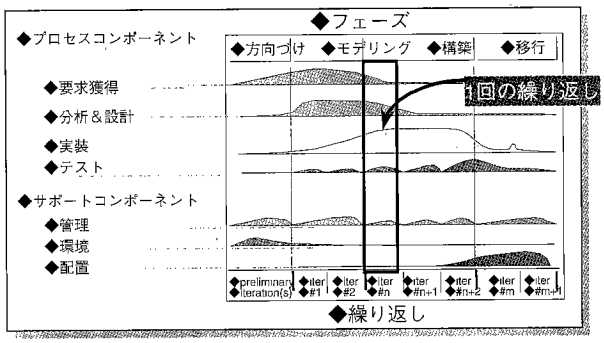


図-11　プロセスコンポーネントの構造

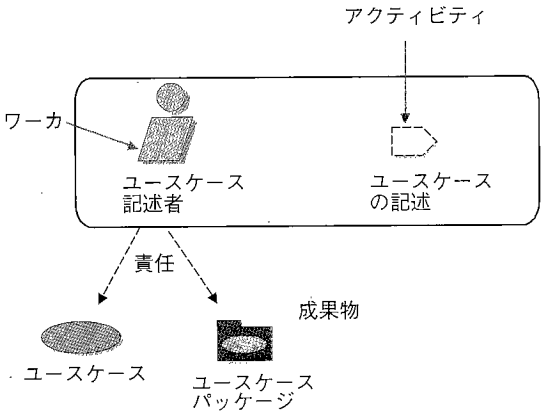


図-12　プロセスワークフローの記法

である．RUPでは，各繰り返しサイクルごとにリスクを減らしたい項目，実現したいユースケース，ミドルウェア等の技術的課題の優先順位を設定しマイルストーンにする．図-10に示すように各サイクルの中で分析・設計・実装・テストがそれらの目標に対して行われるのはもちろん，サイクルの最後でその成果物のユーザと開発者による評価と次のサイクル計画立案を行い適切なフィードバックがなされる．

この管理された繰り返しによって，変化する要求に対処でき，重大リスク，重要ユースケースから片づけていくことで有効なプロジェクト運営が行える．さらに，ベースアーキテクチャが繰り返し検証されて安定し，再利用可能コンポーネントも生成されていく．

■カスタマイズ可能

RUPの全体は図-11に示すように，フェーズとサイクル，プロセスコンポーネントとサポートコンポーネントか

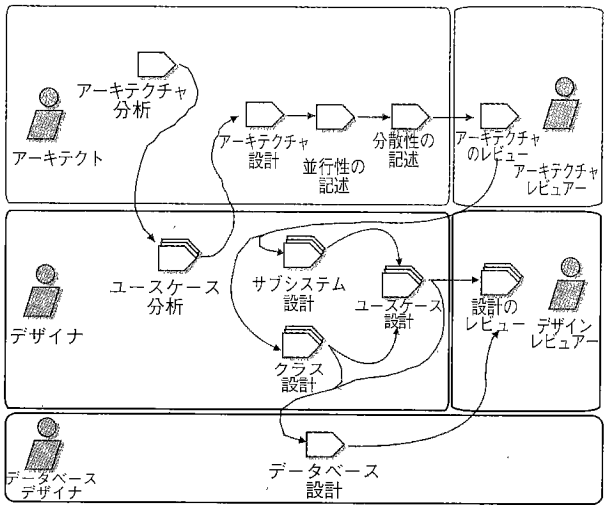


図-13　プロセスワークフローの例

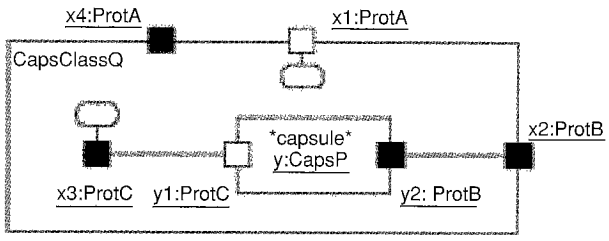


図-14　UMLのROOM仕様リアルタイム拡張例

らなる．さらに，各プロセスコンポーネントはワークフローとして構造化されている．

ワークフローは図-12に示す表記法で表す．これによって，各アクティビティのワーカの役割と担当範囲，各作業内容，その成果物およびそれに依存する次のアクティビティを指定できる．

たとえば，分析／設計ワークフローは図-13のように表現される．こうしたワークフロー図はアクティビティ図のステレオタイプ拡張である．アクティビティは必要に応じてより詳細なワークフロー図に分割される．

抽象的な作業者役割であるワーカを何人，誰に割り当てるか，各アクティビティの作業順序をどう決めるか，成果物の具体的定義，省略するアクティビティ，より詳細化するアクティビティ等を定義することで，開発プロセスをカスタマイズできる．

図-11から分かるように，RUPでは，方向付け，モデリングとアーキテクチャの推敲，構築，移行といったマクロなプロセスである「フェーズ」と，要求定義，分析・設計，実装，テスト，配置・インストール等の各「プロセスワークフロー」と，構成・変更管理，プロジェクト管理，環境・ツール整備といった「サポートワークフロー」とを直交するものとした．この結果，繰り返し開発プロセスに対して上記のようなカスタマイズが可能になったといえる．

以上簡単に見てきたことからも，RUPでは，システムを開発していく上でのWho，What，When，Howが構成的に定義され，よく管理された繰り返し開発によるリスク管理，プロジェクト管理を実現する上で不可欠なプロセスのテンプレートを提示している．

UMLの拡張と今後

UMLは，Extension for Business ModelingとExtensionfor Objectory Process for Software Engineeringの2つの仕様により，ビジネスモデリングとオブジェクト指向開発プロセスにおいてUMLを使う際に有用な拡張をステレオタイプ機構を用いて行っている．それらに加え，リアルタイムシステムをモデリングする要素を現在のUMLに追加する提案も行われている[11]．たとえば，図-14に示すROOM（Real-Time Object-Oriented Modeling Language）手法では，並行なコンポーネント間のコラボレーションをより明確に表現するためにカプセル，プロトコル，ポート，コネクタと呼ばれるモデル要素を追加している．オブジェクトが演じるロールをカプセルと呼びそのロールを起動するプロトコルを実現したものがポート，カプセル間の通信をコネクタが実現する．これらの概念はUMLのステレオタイプ機能を用いて実現可能であり，UML1.3のさらに次の版にリアルタイム拡張として取り入れられる可能性がある．

このほか，Realtime UMLと呼ばれる試み[10]やNokiaのOCTOPUSというリアルタイム組込み向け手法がUMLベースで公開されている[9]．

また，現在，UMLのメタモデルのインタフェースはCORBAのIDLを用いて定義されているが，モデル交換フォーマットのXMLによる定義の策定作業が始まっている．そのほかにも，ステートチャート図における各アクション記述でオブジェクト制約言語OCL[5]で宣言的に書けない部分を実装言語独立に表現する疑似コード言語SMAL（State Machine Action Language）もShlaer/Mellorらから提案され，実行可能モデルの意味論としてUML 1.3以降での採用が検討されている[13]．

UMLに関する3冊の書籍がBooch, Rumbaough,Jacobsonの共著でUML1.3をベースにして出版されつつある（既刊は文献2），3））．

参考文献
1）UML: http://www.rational.com/uml/resources/
2）Booch, G. et al.: The Unified Modeling Language User Guide, Addison Wesley (1998).
3）Rumbaugh, J. et al.: The Unified Modeling Language Reference-Manual, Addison Wesley (1999).
4）Kruchten, P.: The Rational Unified Process－ An Introduction, Addison Wesley (1998).
5）Warmer, J. et al.: The Object Constraint Language－Precise Modeling with UML, Addison Wesley (1998).
6）Odell, J. J.: Advanced Object-Oriented Analysis and Design Using UML, SIGS Book (1998).
7）Simons, A. J. H. and Graham, I.: 37 Things that Don't Work in Object-Oriented Modelling with UML (1998).
8）Fowler, M. et al.: UML Distilled, Addison Wesley (1997). [羽生田栄一（訳）：UMLモデリングのエッセンス，アジソンウェスレイ(1998).]
9）Awad, M. et al.: Object-Oriented Technology for Real-Time Systems,Prentice Hall (1996).
10）Douglass, B.: Real-Time UML, Addison Wesley (1998).
11）Selic, B. and Rumbaugh, J.: Using UML for Modeling Complex Real-Time Systems (1998).（文献1）から参照可能）
12）Schneider, G. et al.: Applying Use Cases－ A Practical Guide, Addison Wesley (1998).
13）http://www.projtech.com/pubs/uml98.html

（平成11年2月8日受付）