オブジェクトの広場

[2006 年 1 月号]

［技術講座］

システム開発にUMLを適用するためのFAQ

日頃の疑問にお答えします

(株)オージス総研
阿左美 勝、綱島 義人、小坂 優、井山 幸次

※本稿は、著者達が1年前に執筆したCQ出版社刊 『Design Wave Magazine 2004 December』 [9] の記事を参考に Web へ公開したものです。

目次

1 ：UMLはオブジェクト指向なしで使えるのでしょうか？

ローカルルールを決めることで使えます。しかし、UMLを効果的に利用できるとも言えず、その用途が限定されてしまいます。そもそも、UMLはオブジェクト指向開発のために考案された表記法であり、オブジェクト指向開発にベストフィットすることを認識していただきたいと思います。

単なる表記法のUMLがこれほどまでに騒がれている理由は、開発対象の大規模複雑化が起因していると考えられます。一昔前の組み込みシステムならば、そのシンプルさからソフトウェアの解決策（ソースコード）へ直結させても、全体を見通すことが可能でした。しかし、近年の組み込みシステムは、安易にソースコード化すると不具合が多発し、手戻り作業の工数が馬鹿にならなくなってきました。そこで、ソースコード化の前に図面（モデル）を作成し、ソフトウェアの品質を早期に評価・検証し、ソフトウェア開発の効率化を目指すプロジェクトが増えてきています。

それでは、オブジェクト指向なしの場合、開発作業の中でどのようにUMLを利用できるのか、シーケンス図とステートチャート図の例を挙げて説明します。両図とも、UMLにおいて振る舞いを表現する図です。今回は、従来型の組み込みシステム開発言語として、一番多く使用されていると思われるC言語を実装言語と想定し、設計する場合の例を示します。C言語等の手続き型言語には、クラスやオブジェクトといった概念がありません。そのため、ローカルルールを決め、UMLの形式で表現できるようにします。

オブジェクト指向なしシーケンス図のローカルルール

シーケンス図は本来、処理の流れを時系列に表現する図です。オブジェクト間の相互作用をメッセージで表現できます。ここで、キーワードとなるオブジェクトとメッセージを以下のように再定義します。

- オブジェクト = サブシステム（関数をまとめた単位）
- メッセージ = サブシステムのインタフェース

図1－1はシステム内のサブシステムが、ある機能を実施する流れを表現しました。

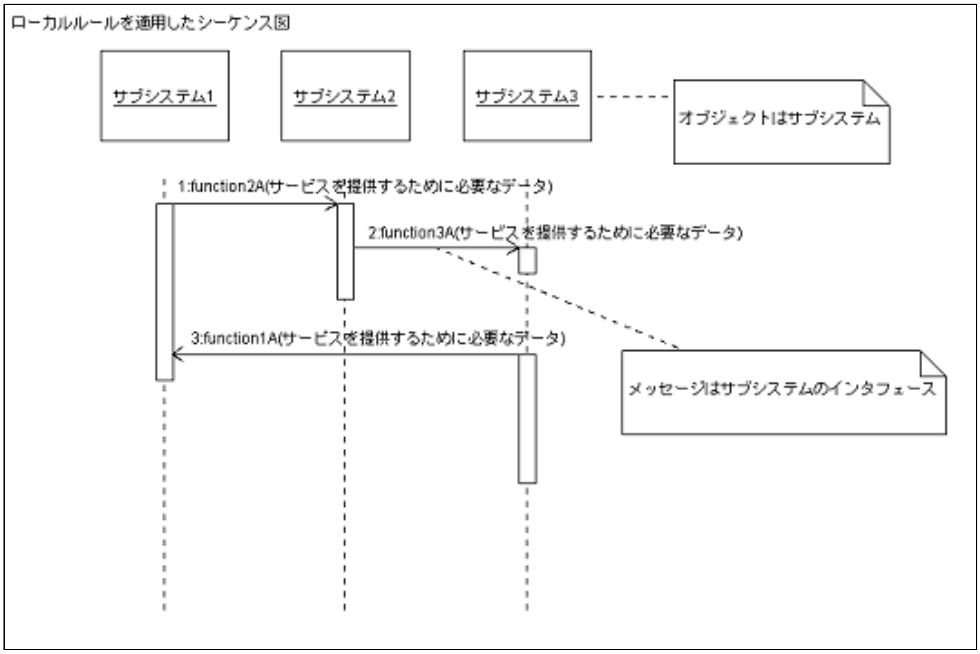

図 1-1 ローカル・ルールを適用したシーケンス図の例

この図を書くことで、まずはシステム内の動作を整理できます。メッセージの方向からサブシステム間の依存関係がわかり、変更に伴う副作用の局所化が可能となります。また、各メッセージを実施するために必要なデータも把握できます。

オブジェクト指向なしステートチャート図のローカルルール

ステートチャート図は本来、オブジェクトのライフサイクルを表現する図です。オブジェクトの期間停滞を状態として切り出し、オブジェクト外部からの刺激をイベントとして受け、状態遷移する様を表現できます。

ステートチャート図は、オブジェクトのライフサイクルに範囲を限定せず、制御対象の移り変わっていく様子をライフサイクルとして表現する図として活用できます。この場合、ソフトウェア内部の世界だけでなく、設計作業における開発者の考えを整理するために使用します。ステートチャート図を利用することで、組み込みシステムの特徴である、イベント入力に応じてアクションを実行すること、つまりイベント駆動の動作を表現できます。ここで注意していただきたいことは、1つのステートチャート図は1つの視点で記述することです。例えば、ユーザー視点でシステムモードを表現、制御対象であるメカ自身の視点で制御モード

を表現、通信プロトコルの視点で通信制御における送・受信状態とその移り変わりを表現といった具合です。

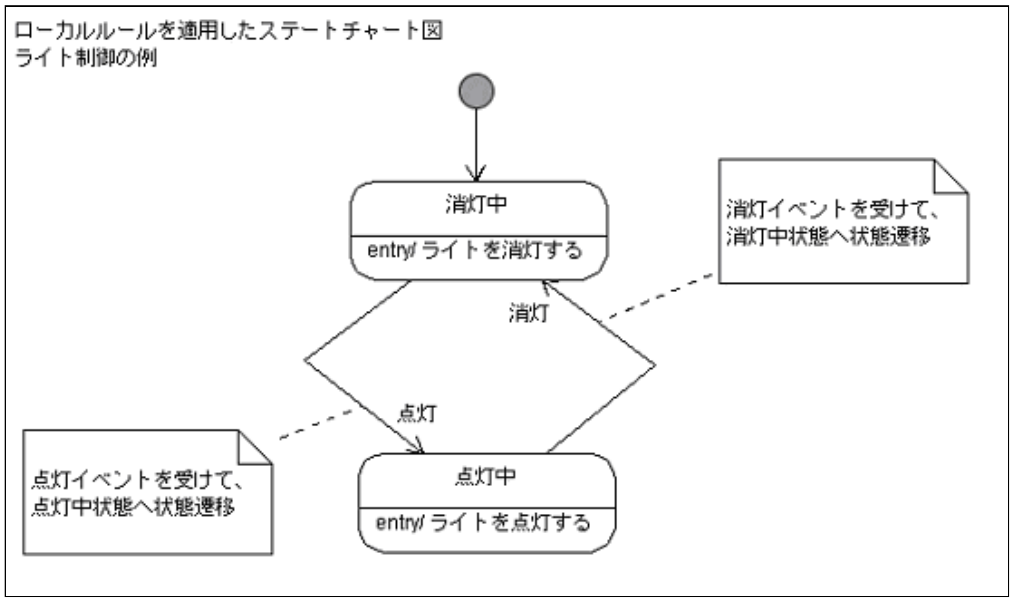

図1-2 ローカル・ルールを適用したステートチャート図の例

この例(図1-2)では、制御対象であるライト自身の視点（図の作成者がライトになったつもり）で、その点灯・消灯のライフサイクルを表現しています。

オブジェクト指向なしでも、 ローカルルールを決めることでUMLを利用可能になります。まずは、ソースコード偏重の開発スタイルから、図を使った開発スタイルへとシフトしてはいかがでしょうか。そうすることで、各サブシステムや関数の呼び出しのみが記述され、複雑度が減り、ソフトウェアの挙動を概観できます。ソースコードでは、詳細すぎるソフトウェアの挙動を図で表現した結果、要求の実現性だけでなく、挙動の複雑度も設計時に見通すことができるでしょう。

オブジェクト指向を適用するならば、当然ローカルルールを決めなくてもUML仕様のまま使用できます。しかし、オブジェクト指向初級者が、自力でオブジェクト指向開発すると、破綻するケースが多いことも事実です。特に製品化プロジェクトならば、オブジェクト指向スペシャリストにコンサルしてもらい、オブジェクト指向開発の敷居を少しでも低くすることをお勧めします。ただし、理想を追求するコンサルタントではなく、適用プロジェクトのスキルレベルに合わせた手引きができるコンサルタントを採用することが大切です。

また、対象プロジェクトメンバーのスキルレベルに応じて、UMLの使用目的を決め、以下のように段階的に活用するのがお勧めです。これは、UML（表記法）の習得スピードが数ヶ月であるのに対し、オブジェクト指向（考え方）の習得スピードが数年と、全く違うためです。

1. 開発ドキュメントにUMLを使用 → ソースコードしか残らない状況を打破
2. 開発作業の成果物として使用（モデル開発） → 設計のみに適用
3. オブジェクト指向開発（分析型開発） → 分析と設計を分離した開発

2 ：UMLには図がたくさんあるが、全部使う必要はあるの？ いつ、どの図を使うべき？

UMLには9種類の図（diagram）が規定されています。しかし、初めからすべての図を使いこなす必要はありません。UMLは、システム（作成しようとしているソフトウェア）をモデリングするための言語です。システムには、機能、構造、ふるまいなど、さまざまな側面があります。こうした側面に応じて適切にモデリングするために、9種類の図があるわけです（表2-1）。

表2-1 UML のそれぞれの図がシステムを表現する

分類	図(diagram)	概要
機能側面	ユースケース図	システムがアクタに提供するサービスを表現する
静的側面	クラス図	概念（クラス）の静的な構造を表現する
	オブジェクト図	オブジェクト（インスタンス）間の関係を表現する クラス図で表現された構造のある時点でのスナップショットを表現する
	コンポーネント図	ソフトウェアを構成するコンポーネント間の依存関係を表現する
	配置図	ソフトウェア・コンポーネントと物理的な資源（CPU、ネットワーク上のノードなど）のマッピングを表現する
動的側面	ステートチャート図	オブジェクトの状態とその遷移（生成から消滅までのライフ・サイクル）を表現する
	アクティビティ図	処理（または業務）の流れを表現する
	シーケンス図	オブジェクト（インスタンス）間の相互作用を、時間軸に沿って表現する
	コラボレーション図	オブジェクト（インスタンス）間の直接的なつながりと相互作用を表現する

開発における図の使いかた

UMLの図のうち、オブジェクト指向開発を行う際に中心となる五つの図を図2-1に示します。これを見ながら、要求分析、分析での各図の使いかたを見ていきましょう。

1. まず、これから作成するシステムが「何をするシステムなのか？」を把握し、機能側面をユースケース図で表現します。
2. そのシステムには「どんな概念で構成されるのか？」を検討し、構造の静的側面をクラス図で表現します。
3. 実際にユースケースが実現される際に、各クラスのインスタンスの間に「どのような相互作用があるのか？」を検討し、相互作用に着目した動的側面をシーケンス図やコラボレーション図で表現します。
4. 各クラスのインスタンスが「どのようにふるまうのか？」を検討し、ふるまいに着目した動的側面をステートチャート図で表現します。

UMLの各図はそれぞれ、システムをある視点から見た一側面を記述している図です（コラム**「UMLの図が語るもの」**を参照）。つまり、それぞれの図の間で整合性が保たれている必要があります。例えば、クラス図で定義されている操作が、シーケンス図ではそのクラスに対応するインスタンスへのメッセージ、ステートチャート図では遷移のきっかけとなるイベントとして表現されます。図 2-1 に示す通り、実際の開発でもクラス図、シーケンス図やコラボレーション図、ステートチャート図を行き来しながら検討されます。こうした作業を通じて、システムが明確にモデル化されていきます。

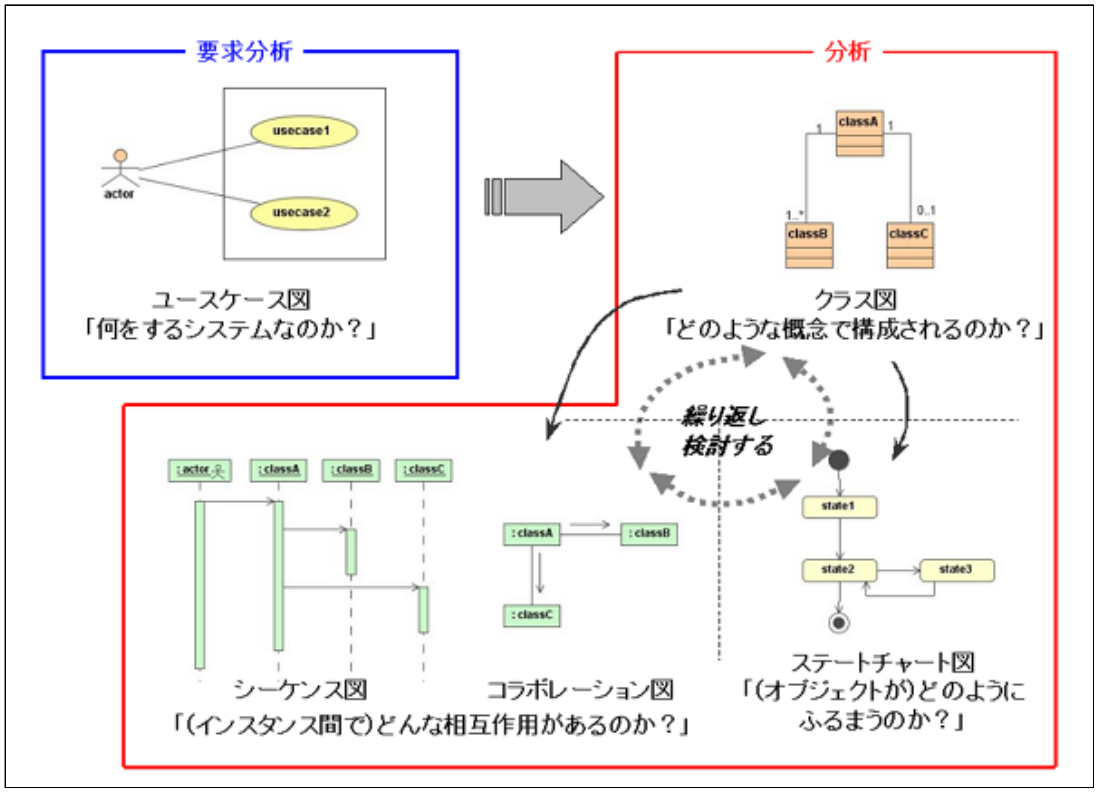

図 2-1 開発の流れと使用する UML の図

UMLの図が語るもの

図2-1 で紹介した図について、もう少し詳しく解説します。それぞれの図が、違う視点からシステムを記述していることを理解していただければと思います。

- ユースケース図
  ユースケース図は、システムが外部の利用者（アクタ）に提供するサービスや機能を表現し、システムの範囲を明確にします。ユースケース図は要求をとらえる際に有効です。ユースケース図を作成する際には、アクタ（役割、外部環境など）を識別し、そのアクタがシステムを利用する場面をユースケースとして抽出します。また、アクタの利用詳細（アクタとシステムの相互作用、アクタにとって価値のあるシステムのふるまい）をユースケース記述として文書化します。ユースケース図においては、アクタ視点からシステムの機能側面が表現されるため、利用者や対象システムの専門家とのコミュニケーションが円滑になります。
- クラス図
  クラス図は、UMLにおいてもっとも中心的で重要な図です。クラス図では、概念を表す「クラス」とクラス間の関係を表す「関連」、「多重度」などを用いて、対象とするシステムの静的構造を表現します。クラス図を描くことで、システムの構成要素とそれらの関係が見て取れるわけです。また、属性や操作を記述することで、クラスがどんな知識やサービスを持っているかという責任範囲も明確にすることができます。
- シーケンス図、コラボレーション図
  シーケンス図は、オブジェクト間でやり取りされるメッセージを時間軸に沿って記述します。これにより、どのような順番で実現されていくかがわかります。ユース・ケースが実現される具体的なシナリオに沿って、インスタンスどうしがどのように協調動作するかを検討することができます。

コラボレーション図も、インスタンスどうしの協調動作を記述するための図です。こちらは、順番というよりも、どのインスタンスからどのインスタンスにメッセージが流れるのかといったインスタンスどうしのつながり（構造的な構成）が見やすい図です。

- ステートチャート図

  組み込みシステムでは、イベント、状態に応じて複雑なふるまいをするものが多いです。このようなときに、ステートチャート図を用いて、イベントによる状態の遷移、状態に応じたアクションを明確にします。図2-1では、一つのオブジェクトに対する状態遷移の定義を示していますが、検討の範囲をシステム全体として、システムの状態遷移を検討することも可能です。

3：忙しくてモデルを書いている時間はない。ほんとうにモデルを書く必要があるの？

結論から言えば、モデルを書くことには大きなメリットがあります。しかし、「ソースコードを書くだけでも精いっぱいなのに、モデルを書く時間なんてない」と感じている読者も多いのではないでしょうか？ここで、ソースコード中心の開発とモデル中心の開発を比較して、ほんとうに「モデルを書く必要があるのか」を検討してみましょう。

ソースコード中心の開発は手戻りが多発

ソースコード中心の開発では、どのような実装をしたらよいかを個々人が頭の中で考えてコーディングを開始します。この場合、以下のような状態に陥りやすくなります（図3-1）。

- それぞれが思うようにソースコードを書くことによって、統一感のないコードになる。
- 考えを整理しないままにコーディングを始め、ソースコードのレベルで詳細に（全体像を把握することなく）検討することによって、実装が複雑になる。
- 不ぐあいを発見するのがコーディング後（開発終盤）に集中するため、手戻りが多発し、修正に多大な時間を要してしまう。これは、上記の考えを整理していない、即ち、複雑さを解き明かしていないことにも起因する。
- 作業中の相談やレビューをソースコードのレベルで行うことになり、コミュニケーションが非効率的になる（他人の書いたソースコードは理解しにくい。ときには、自分が過去に書いたソースコードさえ、どのように考えて書いたのか理解できないことがある）。

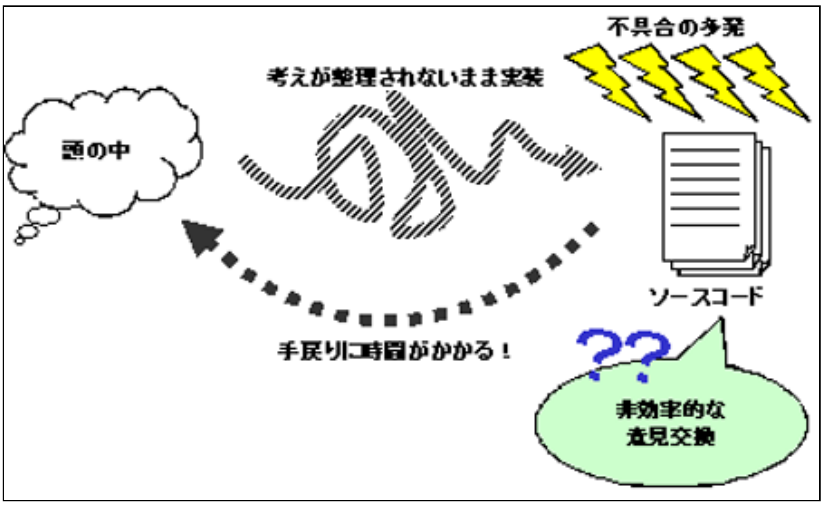

図 3-1 ソース・コード中心の開発

モデル中心の開発で手戻りを抑制

モデル中心の開発では、モデリングを通して（ソースコードを書き始める前に）全体の構造を検討することになります。この場合、以下のような効果が期待できます（図3-2）。

- 図示することで考えを整理できる（複雑さを解き明かし、全体像を理解できる）。考えが整理されれば、実装の作業効率が向上する。
- ソースコードを書く前にモデルでシステムを検証できる。つまり、開発の早期に不ぐあいが発見され、手戻りが減る。
- モデルは人間が理解しやすい表現手段である。作業中の相談やレビューといったコミュニケーションが効率的になる。モデルは共有の資産としても利用しやすい。

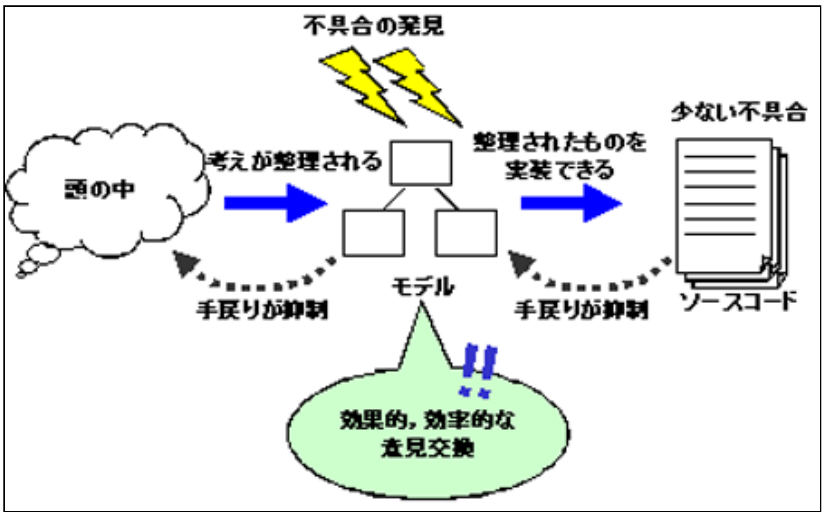

図 3-2 モデル中心の開発

モデルのメリットは導入コストを上回る

このように、モデルを書くことによって、これまで不ぐあい修正にかけてきた時間を減らせる可能性があります。短期間の開発になれば、なおのこと手戻りは許されません。場合によっては、UMLを覚える負荷がかかるのを嫌うかたもおられることでしょう。しかし、長期的な視点で見ると、このコストは十分に取り返せると筆者は考えます。

組み込みソフトウェアが複雑になるほど、その解明が必要です。多人数の開発になるほど（規模が増大するほど）、属人性を排除し、認識を共通化しておく必要があります。このような状況では、モデルの必要性はますます高まっていくと思われます。何らかの対策をしなければ、今日のソフトウェア開発が抱えている問題は解消されません。最近では、UMLモデルからソースコードを自動生成するツールなども開発されています。UMLを導入することにより、旧態依然とした開発から抜け出せる可能性が出てきます。

4 :オブジェクト指向は、私の担当している組み込み製品にとってはオーバースペックなのではないかと思うのですが？

ここで、オーバースペックと思われる理由は、組み込みシステムがプログラム容量と実行速度を重視するシステムであるためと考えられます。まずは、この2点について見ていきましょう。

オブジェクト指向は、従来手法と比べ、実装面で冗長な部分があると思われがちです。これは、ビジネス系システムにおけるライブラリの大きさに起因していると思われます。弊社では、組み込みシステムにオブジェクト指向を適用することにより、冗長なシステムを最適化でき、従来手法と比較してプログラム容量を約2/3に削減できたという事例もあります。今までのソースコード主体の開発では、システム全体の見通し

が悪く、重複したコードが存在する傾向がありました。分析作業により、この重複した冗長部分の削減がプログラム容量の削減につながることが期待できます。プログラム容量に関して、オブジェクト指向が必ずしもオーバースペックになるとは限らないのです。

次に、実行速度に関して制約がある場合は、無理にオブジェクト指向を適用することはせず、その部分には従来手法を適用します。そうです、オブジェクト指向は、システム全体に適用しなくても良いのです。実際、現状のオブジェクト指向を適用した組み込みシステムの多くは、従来手法とのハイブリッドシステムであると言っても過言ではないでしょう。特に、ハードウェアを意識するデバイスドライバ部分は、C言語で実装するケースがほとんどと思われます。

また、組み込みシステムの開発環境によっては、オブジェクト指向言語のサポートがされない場合もあります。このような制約がある場合、C言語での実装となることが一般的なようです。オブジェクト指向言語ではないC言語であっても、設計まではオブジェクト指向で開発可能です。このような開発では、UMLとC言語のマッピングルールを作成し、それに準拠して実装作業を行います。ここでは、そのルールの具体例を一部ご紹介します(図4-1)。

- クラスの扱い
  クラスは、その属性をメンバに持つ構造体として表現し、typedefを使いクラス名を型に定義します。
- 属性
  属性は、クラスをあらわす構造体のメンバ変数として実装します。
- メソッド
  メソッドは、関数として実装します。関数の第一引数に、そのメソッドが属するクラスのポインタを配置します。ただし、モデル上では表現が冗長になるので、引数は第二引数以降のみ書くようにします。

図 4-1 マッピング・ルールに基づいてクラスをC言語に変換する

```
/* ClassA.h */
・・・

/* 属性の宣言 */
typedef struct {
   int attr1;
   char* attr2;
} ClassA;
・・・

/* メソッドの宣言 */
void ClassA_method1(ClassA* this);
int ClassA_method2(ClassA* this, int arg);
```

今回はUMLをCにマッピングする概観をつかんでいただくために、そのルールのほんの一部をご紹介しました。製品化プロジェクトに適用する場合は、これ以外にも、パッケージの扱い、可視性、コンストラクタ・デストラクタ、関連、継承等々に対して規定が必要になります。

ここで、オブジェクト指向・UMLの適用メリットを、製品の性格・特色毎にまとめてみました。

全ての製品に言えること

- 開発上流から下流まで、同じ考え方で作業できる
- 開発者の考えをモデルとして表現することで、システム構造を可視化できる
- システムの仕様をモデルとして表現することで、全体像の把握を容易にする
- 実装する前に設計レビューができ、設計図が手元に残る
- ソースコードの自動生成が可能（モデリングツール使用）
- モデルとソースコードを乖離させないことが可能（モデリングツール使用）

大規模・複雑な製品

オブジェクト指向開発は大規模・複雑なシステムに適用するケースが多いことも事実です。これは、オブジェクト指向の基本的な考え方の1つである分割統治（役割分担）の効果が期待できるためと思われます。役割分担により、考慮すべき範囲が局所化できることで、一極集中になりがちな組み込みシステムを自律分散型のシステムにしやすくなります。

シリーズ展開する製品

製品のロードマップがあるようなシリーズ展開する製品ならば、メリットはかなりあります。今後どこを拡張、メンテナンス、再利用するのか、UMLで見通せるよう、今後につなげる図を残しておくことを、現在の開発作業の中で意識します。当然、初回の開発では余計に工数がかかりますが、今後発生する開発コストまで含めトータルでみれば、コストパフォーマンスは良いと考えられます。

一品物の製品

一品物の単発製品の場合、やっつけ仕事で開発し、今後の開発に役立つ図も不要ですが、開発者間のコミュニケーションに図を使用することで、意思疎通の面でチーム開発の作業効率の向上が期待できます。

数年前まで、オブジェクト指向は、現実解として不透明な手法であったと思います。しかし、近年では、製品開発へ適用し成功を収めているメーカーも多くあり、その認知が広がりつつあると感じられます。このことは、コンサルタントとしてオブジェクト指向開発を支援させて頂いている筆者らにとって、うれしい限りです。

5 ：タスク、割り込み等、組み込みシステム特有の概念をUMLで表現できるのか？

ソフトウェアとして実現することを意識した組み込みシステムの設計では、タスク、割り込み等の組み込み特有の検討が必要になります。ここでは、それらをどのようにUMLで表現するのかを解説していきます。

タスク

UMLでは、タスクの基点（エントリーポイント）となる関数を所有するクラスのことを、アクティブクラスと呼びます。このクラスのオブジェクトがタスクとマッピング（紐付け）され、1つの制御スレッドを形成します。以下のように太枠で囲まれたクラスとして表記します(図5-1)。また、通常のクラスの区画が3つ（クラス名・属性・操作）であるのに対し、アクティブクラスは第4の区画を付加します。この区画には、そのアクティブクラスが受信可能なシグナルを記載します。

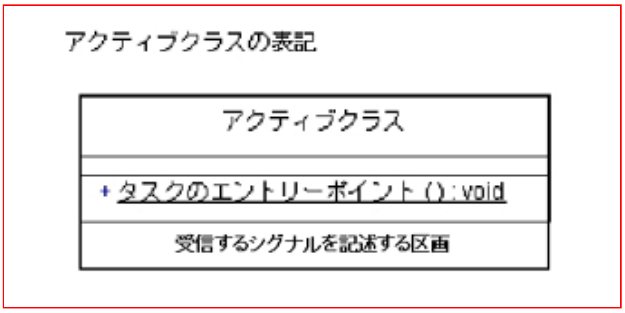

図 5-1 アクティブクラスの表記

アクティブクラスの実体（インスタンス）がアクティブオブジェクトとなります。OS部分からアクティブオブジェクトのエントリーポイント用メソッドを呼び出すまでの動作をシーケンス図で表記しました(図5-2)。ここで、制御スレッドはオブジェクト間のコラボレーションのパス（通り道）となります。複数スレッドの表記は、UMLでサポートされていません。そこで、シーケンス図で同一スレッドを明示したい場合には、線で囲んだり、色分けしたり、メッセージ番号にアルファベットを付加したりして表記上の工夫をします。今回の例では、線で囲むことでそのスレッドを明示しています。この図では、2つのアクティブオブジェクト登録から、アクティブオブジェクト1実行中に、アクティブオブジェクト2へプリエンプトされる際のコラボレーションを表しています。各アクティブオブジェクトから使われるオブジェクトを、パッシブオブジェクトとして記載しました。通常、パッシブオブジェクトは複数ありますが、ここではそれぞれのスレッドに1つだけ抽出しました。また、OSは開発対象範囲外と考えアクターにて表記しました。

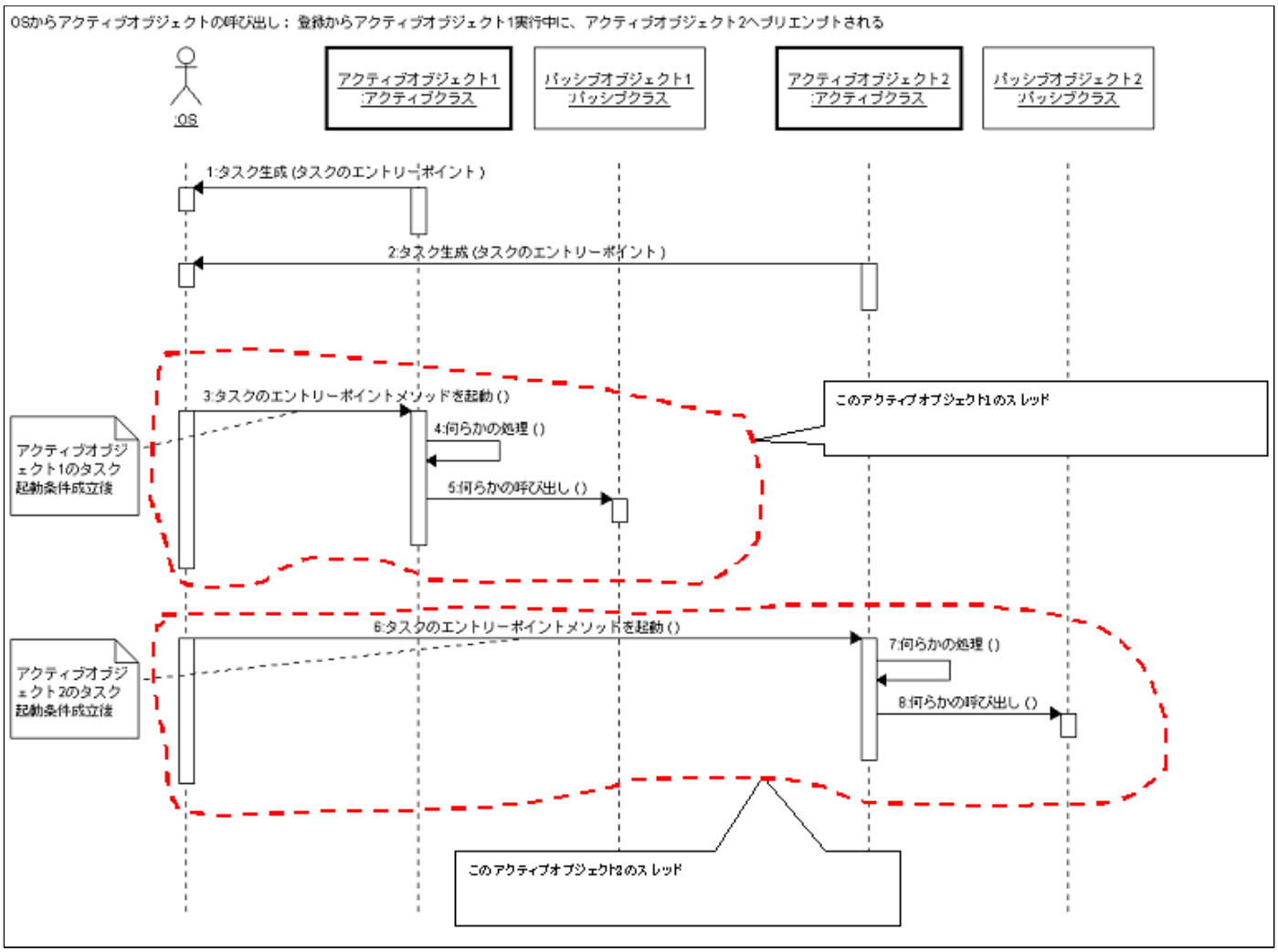

図 5-2 OS からアクティブ・オブジェクトを呼び出す様子（シーケンス図で表現）

割り込み

割り込みハンドラは、システムと同じ生存期間を有するオブジェクトとして扱うことが一般的です。割り込みイベントとして、何らかの刺激（外部入力ポートのエッジ変化、シリアル通信終了、DMA転送終了・・・等々）を受け、割り込みベクターが発生し、該当ハンドラが起動します。割り込み内の処理は、タスクへイベント送信する程度の最低限の処理で済ませることが一般的です。ただし、OSのディスパッチ処理時間さえも、介入させたくないほど緊急度の高い割り込み処理の場合、割り込みハンドラから目的の処理をする関数を呼び出し、割り込み内で該当する処理を実行してしまうこともあります。割り込みハンドラとそこから呼び出される関数を所有するクラスをクラス図(図5-3)で、割り込み発生から、該当処理の起動までの一連の流れをシーケンス図(図5-4)にしました。尚、UMLの仕様では割り込みの表現をサポートしていないため、ローカルルールを決めてクラス図を表記しています。そのため、割り込みハンドラから呼ばれるスタティックメソッドを所有するクラスに、ステレオタイプ<<interrupt>>を付与しています。この図で、割り込みハンドラは、開発対象範囲外と考えアクターにて表記しました。

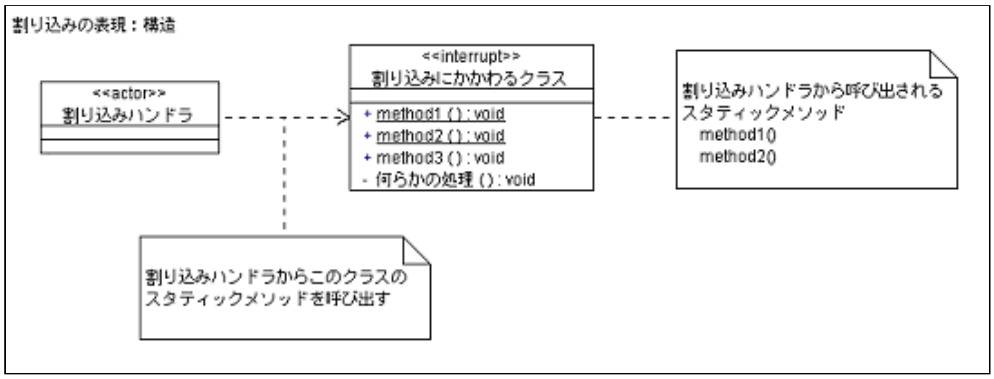

図 5-3 割り込みの表現（構造）

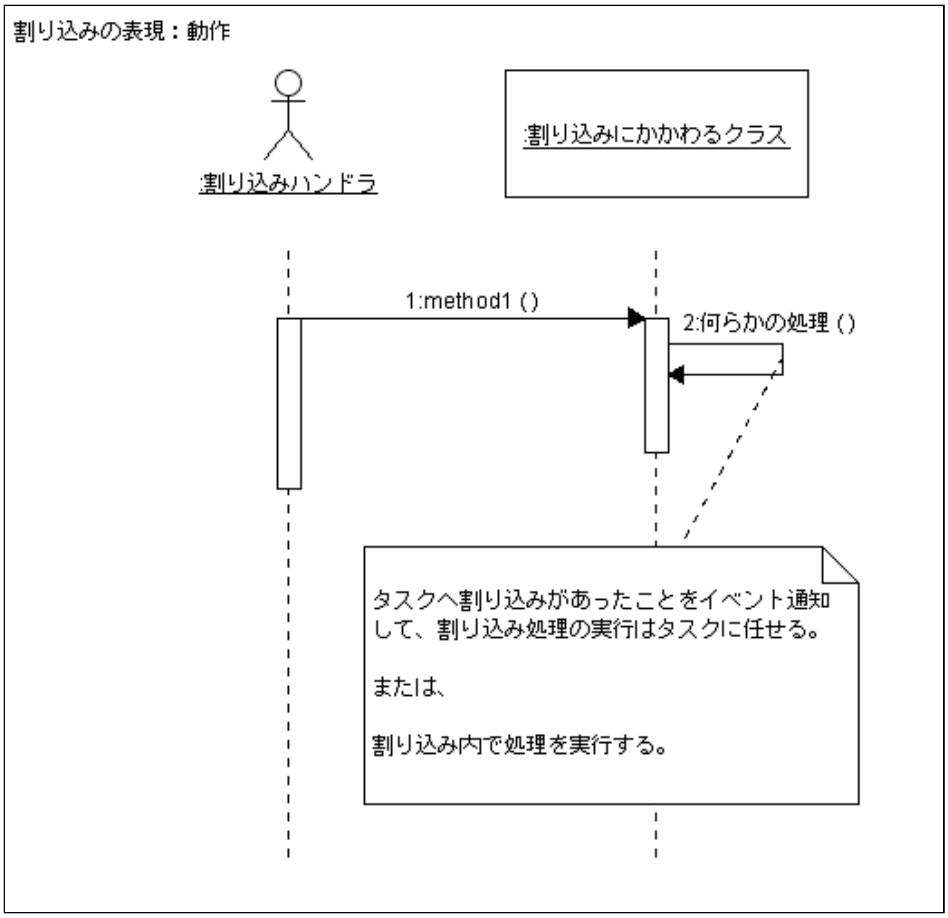

図 5-4 割り込みの表現（動作）

正常処理と異常処理の制御フロー戦略

組み込みシステムでは、正常処理以上に異常処理が多く存在します。オブジェクト指向で、これら処理の考え方について説明します。ここでの異常処理とは、異常を復旧させるためのリカバリ処理があるものと想定しています。一般的に、正常処理と異常処理を混在させてしまうと、処理内容が極めて複雑になります。さらに、異常処理は、正常処理よりも開発最中に変更・追加されるケースが多いのです。そのため、オブジェクト指向では役割分担と変更可能性の観点から、正常処理と異常処理の責務、つまりクラスを分けます(図5-5)。

尚、このような処理を行うクラスの場合、正常処理と異常処理、双方ともステートチャート(図5-6,図5-7)を持つオブジェクトとなるでしょう。2つのオブジェクト（正常処理と異常処理）が協調することで、処理要求を満たします。

ここで、これら2つのステートチャート（正常処理と異常処理）のコラボレーションをシーケンス図(図5-8)で表現します。

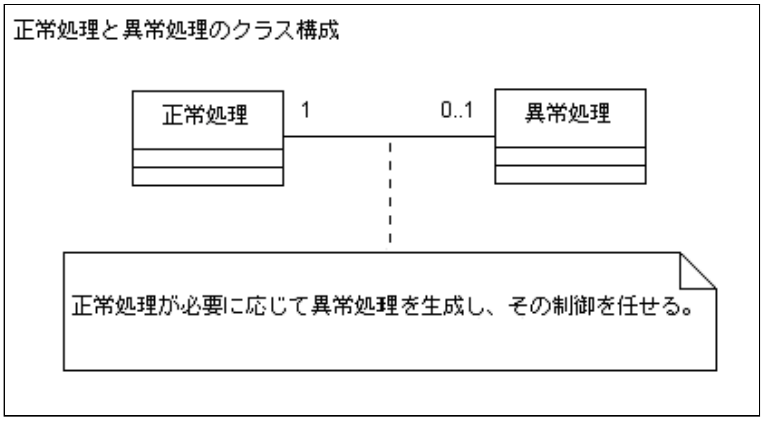

図 5-5 正常処理と異常処理のクラス構成

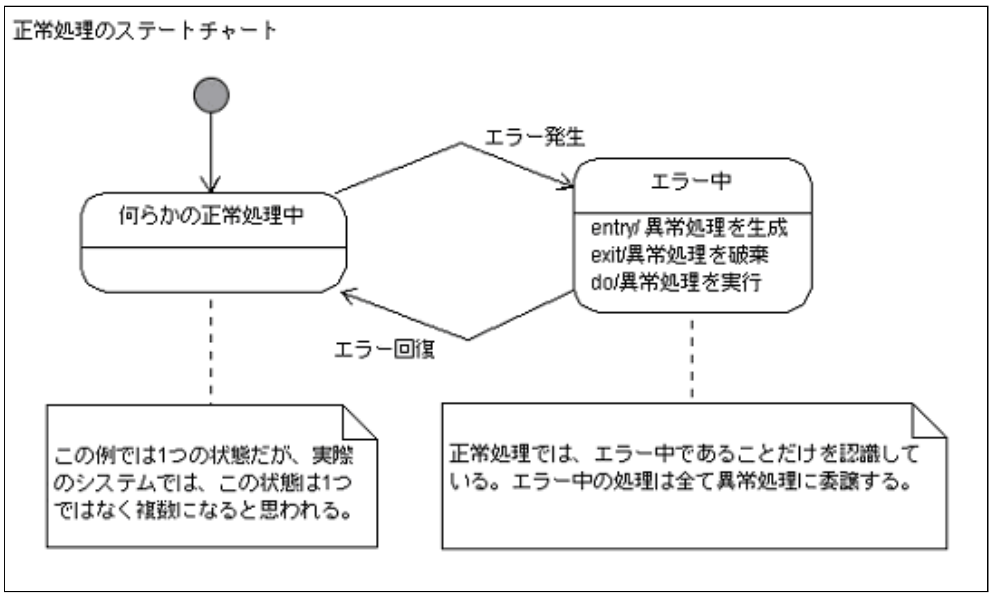

図 5-6 正常処理のステートチャート図

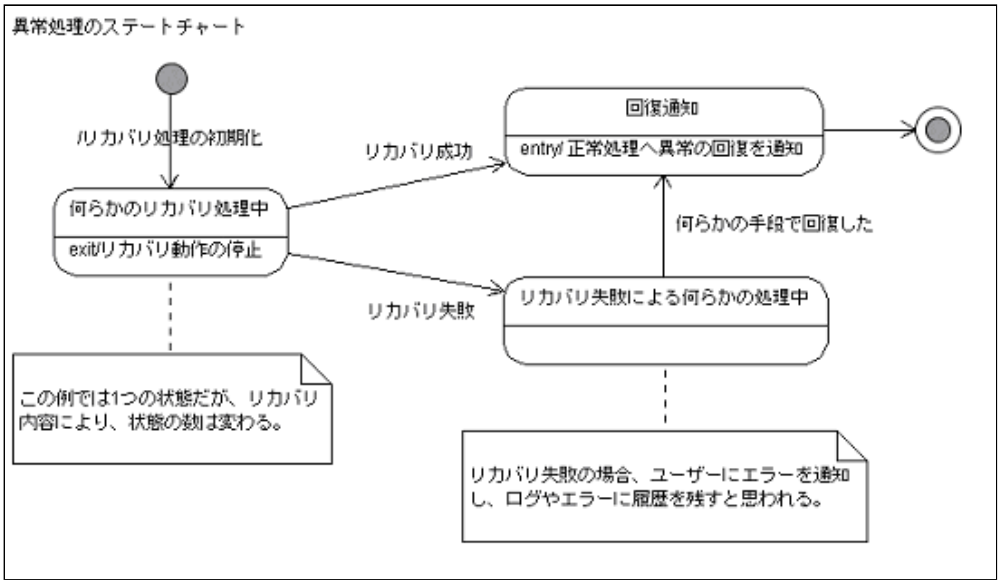

図 5-7 異常処理のステートチャート図

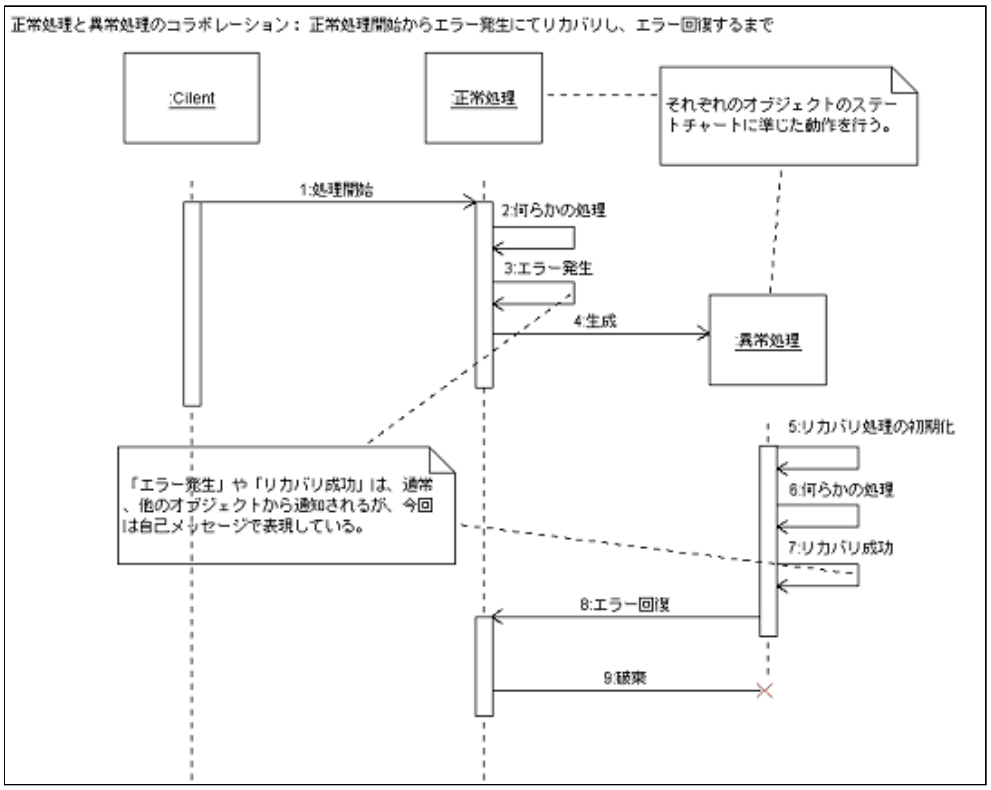

図 5-8 正常処理と異常処理のやりとり

TDDでシンプル設計を体得する

シンプルな設計ってどんな設計？

そんな人にはTDD(Test Driven Development：テスト駆動開発) [5] をおすすめします。

TDDはテストコードを書いてからクラスのコードを書くという特徴的なプログラミング手法です。TDDでは、テストコードを書いてから、クラスのコードをパスするまでをひとつのサイクルとして、その小さなサイクルを繰り返してプログラムを完成させていきます(図A-1)。小さな目標(テストにパスすること)をひとつずつ達成しながらプログラムを完成させていくやり方は、開発にリズムを与えるので、多くの開発者から支持されています。

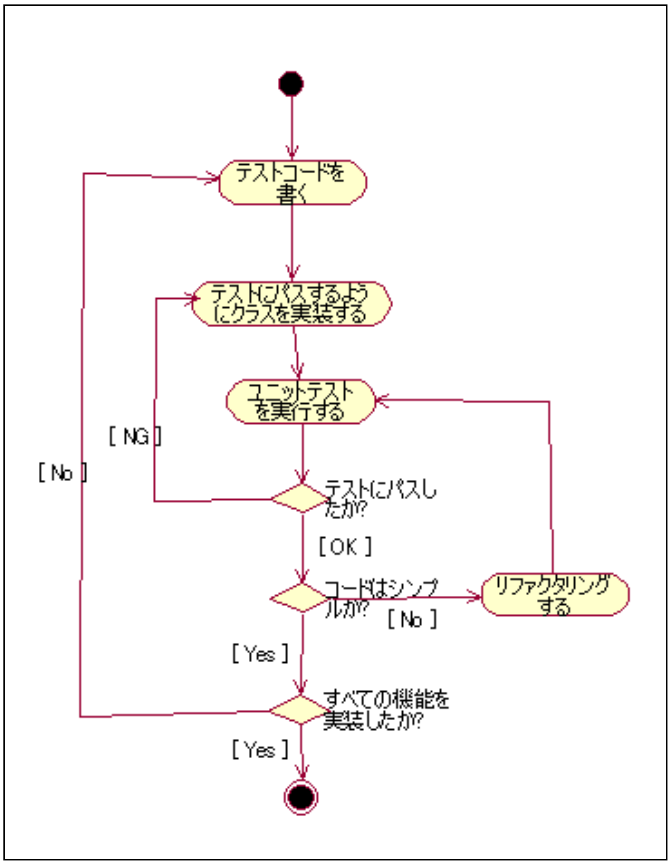

図 A-1TDD の作業の流れ

筆者らは、以前参加していた開発プロジェクトでTDDを取り入れました。目的はクラスの単体テストを徹底させるためです。そのプロジェクトではモデル中心の開発プロセスを採用していたので、設計を完成させてからTDDで実装しました。つまり、設計されたクラスに対するテストコードを書いてから、クラスの実装を行うのです。

そして、TDDを取り入れたイテレーションの前後で、設計モデルの構造が変わったことに気づきました。

- クラスやメソッドの粒度が小さくなった
- クラスやメソッドの独立性が高くなった
- クラスやメソッドの名前がわかりやすくなった

なぜこのような変化があったのでしょうか。それは「視点の変化」です。テストを書くときには、利用者の視点でクラスを見ます。そのときに、自分が設計したクラスの使いやすさを体験することになるのです。

開発者はテストコードを書くときにクラスの役割の適切さを知ります。また、テストするためのオブジェクトの生成や、オブジェクトの操作を呼び出した後の結果の確認を行うときに、他のクラスとの依存関係の強さを知ります。そして、テストするシナリオでオブジェクトを扱うときに、クラス名や操作名がどれくらい自然であるかを知ります。

開発者は自分の設計に対するフィードバックを自ら得ることになるので、他人に指摘されるより効果的なのです(表A-1)。このように、自分の設計を自分で評価できるところも、TDDが支持されるひとつの要因なのでしょう。

表A-1 自分と他人の評価の違い

自分による評価	他人による評価
自分でテストコードを書いて評価するため、フィードバックが早い。	レビューは時間がかかるので、フィードバックが遅い。または、フィードバックが無いこともある。
自分の間違い(不適切な設計)は受け入れやすい。	他人の指摘は受け入れにくい。
学習効果が高い。	同じ間違いを繰り返しがちである。

6 ：今からUMLを始めるのだが、UML 2.0を学ぶべき？

UMLは1997年にバージョン1.0が発表された後、何回かマイナ・バージョンアップが行われてきました（最新バージョンは1.5）。メジャー・バージョンアップとなるUML 2.0は、2003年8月～10月に最終案が公開され、現在、正式版を発表するまでの最終段階を迎えています。

UML 2.0では、より詳細にモデルを記述できるようになりました。ただ、ソフトウェアの基本的な側面（機能や構造、ふるまいなど）を記述するには、UML 1.x でも十分表現できます。UML 2.0は、UML 1.x と比べて覚えることが多く、習得がより難しくなります。また現状では、UML 2.0に対応するツールは多くありません（今後は増えるだろうが…）。

モデルの表記法としてUMLを利用する方は、まずはUML 1.x から始めることをお勧めします。1.x を習得した後、1.x では表現能力が足りない、書きにくいと感じるところから、2.0での表現方法を習得していくとよいのではないでしょうか。また、MDA（model driven architecture）の導入を考えている方は、UML 2.0から始めてもよいでしょう。

UML 2.0で図の表現能力が強化された

UML 2.0では、図の表現能力が強化され、メタモデル（UMLを定義するモデル）がより厳密で柔軟性のあるものに変更されました。メタモデルの変更は、図の変換やコード生成などを行うツールには大きく影響しますが、たんにモデルの表記法としてUMLを使う方は、それほど意識する必要はないでしょう。ここでは、図の表現能力がどのように変更されたかを紹介します。

表6-1にUML 2.0で変更された図の概要を示します。また、図6-1にUML 2.0から表記可能になったシーケンス図の階層化の例を、図6-2にタイミング図の例を示します。詳細に関しては、参考文献 [7] を参照してください。

UML 2.0の背景にMDAあり

UML 2.0においてメタモデルの厳密性や柔軟性が向上した背景には、MDA（model driven architecture ; モデル駆動型アーキテクチャ）への対応があります。MDAは、モデル主導のシステム開発、およびライフ・サイクル管理を実現するための参照アーキテクチャです。

MDAでは、システムの本質的なモデル（ビジネス・ロジックやデータ構造など）と実装技術（OSやプログラミング言語など）に着目したモデルを分離して開発します。比較的短い期間で変更される実装技術のモデルを、システムの本質的なモデルと分離することによって、変更に柔軟に対応できるようになります。

表 6-1 UML 2.0 で変更された図の概要

大きな変更があった図	
シーケンス図	分岐、ループ、ブレークといった例外など、複雑なふるまいが記述できるようになった。また、複数のシーケンス図の関係（階層化）を示せるようになった。
アクティビティ図	スイムレーンの多重化、例外の記述、アクティビティの構造化ができるようになった
コンポーネント図	ポートを用いて、外部とのインタフェースを明確に記述できるようになった
新しく追加された図	
コンポジット構造図	クラスやコンポーネントの内部構造を明確に記述する
相互作用概要図	複数の相互作用間の関係を記述する
タイミング図	時間軸に沿ってオブジェクト、クラスなどの状態の変化を記述する
名称が変更された図	
コミュニケーション図	UML 1.x のコラボレーション図から名まえが変更された
タイミング図	UML 1.x のステートチャート図から名まえが変更された サブステートが記述できるようになった

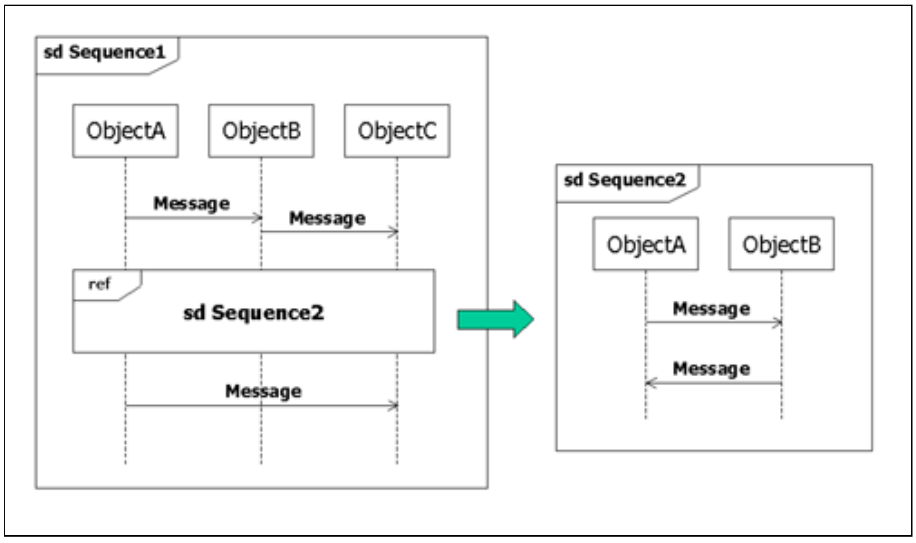

図 6-1 シーケンス図の階層化の例

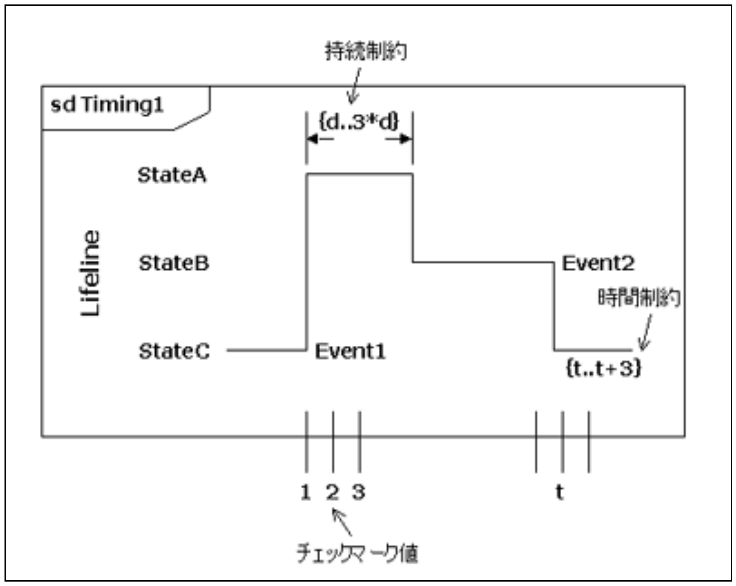

図 6-2 タイミング図の例

7 ：製品知識は分かっているのに、分析する意味はあるの？

分析する意味はあります。それは、分析作業にて対象システムのあるべき姿を見い出すことで、何をするシステムなのか明確化された状態にて設計作業に入ることができ、作業効率も向上することです。
既にソフトウェアとして実現できたという事実（どう作るか）と、製品知識が分かっていることは異なります。そこで、この質問にある「分かっている」に対する確認事項が2つあります。

確認事項1： システム化の範囲と要件について、本当に曖昧なところはないのでしょうか？

特に要件から洗い出す仕様に関しては、曖昧なまま開発が進むケースが多々あります。開発着手時に要件が明らかでなくても良いので、どこが決まっていて、どこが決まっていないのかを明らかにし、開発中その要件の管理をしていくことが大切です。

確認事項2： ある開発者の頭の中だけに存在する知識ではなく、組織（プロジェクトチーム）の知識になっていますか？

対象製品のあるべき姿を正確かつ論理的にドキュメント化できている。そして、そのドキュメントを読んで、第三者が対象製品を正しく理解できる状況になっていることが重要です。自然言語では、曖昧さや受け手の解釈にギャップが発生します。チーム開発するシステムならば、このギャップは開発を進めていく上でのリスクになります。ドキュメントに正確性、論理性のあるUMLを取り入れることで、そのギャップをできるだけ小さくすることが可能になるでしょう。

分析では、これから開発する製品を、どんな部品で構成するかに注力します。ここでは、システム化する上で、対象の重要な概念を抽出し、その論理構造を導き出します。そして、その構成要素がシステムの要求を満たすことができるのか、モデルを使って動作を確認します。言い換えると、システム内に必要な役割を持った部品を用意し、それらの協調動作で機能を実現させます。

設計では、分析で明確になったシステム内部の構成要素を、ソフトウェアで実現することに注力します。ここでは、ソフトウェア化する上で、特定のプラットフォームや実装言語等々の実行環境、メモリや実行時間等々の制約を考慮します。設計は、ソフトウェアとして実現する落としどころを探る作業と言えます。

従来型開発では、分析（何をするか）と設計（どうするか）を同時に作業していましたが、これでは検討する範囲が広すぎ、モレ・ヌケを発生しやすいのです。

分析と設計は上記のような違いがあり、明確に作業を分割できるのです。組み込みシステムの大規模・複雑化により、ソフトウェア化よりも、まずはシステム化を考えることで、検討すべき範囲を局所化してはいかがでしょうか。今まで、見落としていた重要なことが見つかるかも知れません。テストやデバッグ作業にてバグという形になる前に・・・

タスク偏重の弊害

組み込みシステムは、外部からのイベントや割り込みに対して、即時応答し処理を遂行することが求められます。多くのケースでは、システムを複数のタスクで構成し、それらが並列、若しくは協調動作し、決められた時間内に処理を遂行することを実現しています。このようなシステムを開発する上で、限られたコンピュータ資源を無駄なく活用して、求められる速度性能を達成するには、タスクをいかに設計するかが重要であり、組み込みシステムの大きな特徴といえるでしょう。

しかし、速度性能を達成するための重要なタスクではありますが、あまりにタスクを中心に開発を進めると弊害も起きます。典型的な例は、開発の早期にタスクを見出し、タスクという粒度でシステムを分割（タスク分割）し、タスク単位で内部の詳細設計を実施するといった開発スタイルです。

このような開発では、システム全体を見渡す設計が行われず、タスク単位で閉じた設計が行われます。必然的に、タスク間で共通化されるべきデータ構造・モジュール・資源が見出されず、それらが各々のタスクにて個別に実装され、開発効率は勿論ですが、実行速度の面でもデメリットを生むことになります。また、各タスクは完全に独立しているのは稀で、たいていは競合する資源を排他制御して利用したり、タイミングを待ち合わせたりしています。タスク分割のスタイルでは、このような部分に対しても十分な検討がなされず、システムテスト時にあるタイミングでの予期せぬ動作や排他制御の不具合によるデッドロックも発生します。

確かにタスクはシステムにおいて重要な要素でありますが、あまりに偏重するとこのような事態を招きかねません。そもそもタスクとは、OSから見た一連の処理を並行に実行することのできる資源であり、本質的な機能の実現ではありません。タスクは、システムに求められる時間制約に対する実現手段であることを心に留めるべきです。つまり、タスクとは、システムの動的な側面であり、同時に使用するコンピュータ資源（CPUの演算速度）やその構成といった制約によって影響を受けやすい、変動しやすい要素といえます。このような変動しやすい要素を基盤にシステムを開発することは、特に技術革新が目覚しいハードウェアの進化による環境変化にシステムを晒すことになりかねないということです。

ある組み込みシステムの開発において、度重なる機能追加の果て、「タスク数が160個を超えてしまった」という実例もあるようです。この例では、機能や開発者への作業割り当て毎にタスク抽出・分割をしたために、機能の増加と正比例してタスクが増加し、結果としてタスク数の爆発を招いてしまったようです。

ではこのような状況を回避するために、どのようなタイミングでタスクを意識して、開発を進めたらよいのでしょうか？

一般的ではありますが、以下のように考えられます。第一に、システム全体を静的な側面から分析を行い、システムの機能を実現するための要素とその関係、共通性を洗い出し、論理的に機能が実現されることを検証します。次に設計にて、物理的な要

因から発生する制約を加味します。今回の例では速度性能を考慮して分析で検証された成果に、平行性を盛り込み、どのモジュールをどのタスクで動かすのか（タスク割り当て）を検討することが妥当ではないかと思われます。このような進め方であれば、設計の成果が、対象とする物理的な資源や制約によって都度の見直しが発生したとしても、システムの静的な側面を表す分析の成果は、タスクという変動要素の影響を受けず、長いライフサイクルでの利用が可能となります。

組み込みシステムにおいて、タスクは非常に重要な役割を果たします。しかし、性急なタスク分割は避けるべきといえるでしょう。

８：機能が多すぎてユースケースが爆発してしまいます。ユースケースの管理しやすい方法を教えてください。

ユースケースの数が多くなる理由として、そもそも規模が大きなシステムで機能が多い場合と、ユースケースの抽出がうまくできていない場合があります。ここでは、まずユースケースが正しく抽出されているかについて今一度確認し、その後に抽出されたユースケースの管理しやすい方法について説明します。

まず正しくユースケースが抽出されているかについて確認しましょう。ユースケース抽出に不慣れな場合、特にシステムの設計仕様に詳しい開発者がユースケースを抽出する場合、システムの内部機能が膨大な数のユースケースとして抽出されている現象が見られます。例えば、自動販売機のシステムにおいて、「投入されたお金の真贋を判定する」「金額を表示する」「指定された商品を排出する」「お釣りを返却する」といったユースケースが抽出されているような場合です。もちろん、これらは自動販売機に求められる要件ではありますが、果たしてユースケースとなりえるでしょうか？

そもそもユースケースは、システム外部の利用者であるアクタの視点に立ち、そのアクタがシステムにどのようなサービスや価値を期待するか（あるいはシステムがどのようなサービスを提供するか）といった単位が原則です。この原則に照らし合わせると、先のユースケースがそれだけでアクタ（この例では商品を購入しようとする顧客）に対して価値を提供しているとは言えません。投入されたお金の真贋を判定や、その金額が表示されることだけに満足感を覚える顧客はいないでしょう。つまり、お金を入れてから商品を選択、商品・お釣りの受け取りといった流れを踏んで「商品を購入する」ことが、顧客の視点で自動販売機に求めるサービス、つまりユースケースとなります。
開発者の方は、システムの内部を詳しく知っているため、ついつい内部機能や処理手順をユースケースに表現してしまい、ユースケースが爆発することが多々見られます。少々乱暴な言い方になりますが、システム内部をまったく知らない素人になったつもりでユースケースを抽出すると良いかもしれません。

さて、ここまでで正しくユースケースが抽出されたことを前提として、ユースケースの管理しやすい方法について説明します。

正しくユースケースが抽出されても、実際のシステム開発では規模に応じたユースケースが抽出されます。10～20個ほどのユースケースであれば何ら工夫も要りませんが、30個を超えるとなると何らか軸に沿ってユースケースを分類し、パッケージなどにまとめることが必要になってきます。

ここでは分類別けの２つの例を紹介します。

関連するアクタごとに分類する

システムには、それを利用するアクタが複数存在することが一般的です。システムはそれらアクタに対して快適な操作性を通して、サービスを提供することが求められます。例えば、システムの詳細な仕様を知ら

ない・操作に不慣れなユーザと、よく知っている保守要員とでは、システムが提供するユーザインタフェースも変わってくるでしょう。また、アクタには人以外に他システムもあります。このような場合、ユーザインタフェースの代わりに何らかの通信プロトコルが必要になります。アクタごとに分類することで、そのアクタとシステムの相互作用の実現に違いを識別することができます。

また、一般のユーザと保守要員をアクタとして識別することは、システムの利用シーンを識別するとも言えるでしょう。通常の運用時に利用されるユースケースと、ある特殊な状況にて利用されるユースケースを分類することにも役立ちます。アクタが一つしかないような場合は、利用シーンごとに分類すると良いでしょう。

機能の種類ごとに分類する

システムには、過去のバージョンから備わっている必須の機能と、ある仕向けや客先ごとに付加されたり・取り外されたりするオプション的な機能があります。また、システムには正常系の機能以外に、異常が発生した後のリカバリやシステムを解析するためのログなどもあります。このような機能の種類や、関連性の強いものごとに分類するのも良いでしょう。

開発者にはこの分類が、親和性があると思われます。

ここでは 2 つの例を示しましたが、これが正解というものはありません。ユースケースはシステムに求められる要求を表現する以外に、システムの開発やリリースの単位にもなります。つまり、プロジェクトをどのように計画・運営するかといったプロジェクト管理者としての狙いによっても分類の軸は異なってきます。ユースケースを分類する目的・狙いに十分留意して、そのプロジェクトにあったユースケース分類を心がけてください。

9：「問題領域の概念を抽出する」って何でしょう？

今でこそ少なくなりましたが、UMLやオブジェクト指向関連の書籍や記事には、オブジェクト指向特有の難しい表現（専門用語）が数多く使われていました。オブジェクト指向初心者は、まずここで挫折する方々が多くいらっしゃったと思います。筆者らも例外ではなく、オブジェクト指向に興味を持ち始めた頃は、これら用語に翻弄され、いろいろな書籍を読みあさった苦い経験があります。それでは、オブジェクト指向にありがちな専門用語のいくつかを見ていきましょう。

問題領域

問題領域と言われる理由は、システム化により何らかの問題を解決するからです。問題領域とは、対象をある視点でグルーピングした範囲のことをいい、ドメインとも呼ばれます。
例えば、システム全体ならば、システムの実現手段と言う視点で切り出すと、ハードウェアとソフトウェアも問題領域になりえます。ここでは、クラス図にて問題領域を表現するために、UMLのパッケージとそのステレオタイプ<<domain>>を使用します(図9-1)。

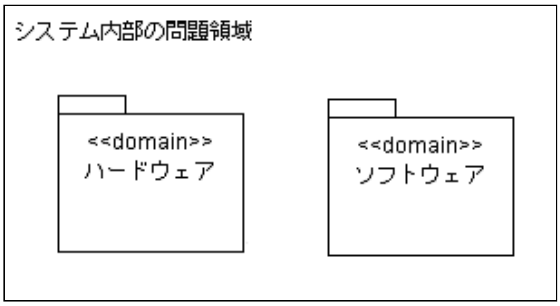

図 9-1 システム内部の問題領域

ソフトウェア内部の構造は、目的と手段といった視点でグルーピングすることが一般的です(図9-2)。

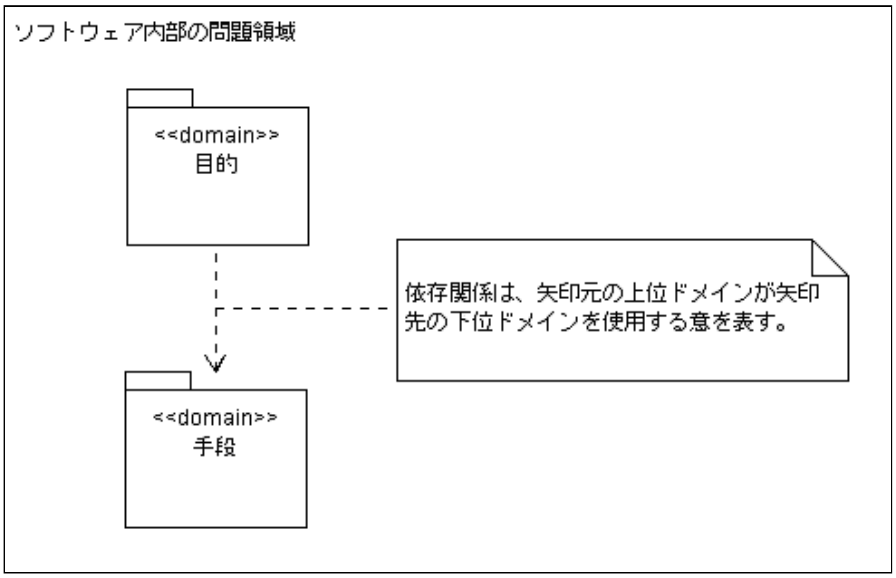

図 9-2 ソフトウェア内部の問題領域（一般例）

実際の組み込みシステムには、どのような問題領域が存在するのかイメージしていただくために、扇風機のソフトウェア内部の問題領域を具体例として掲載します(図9-3)。

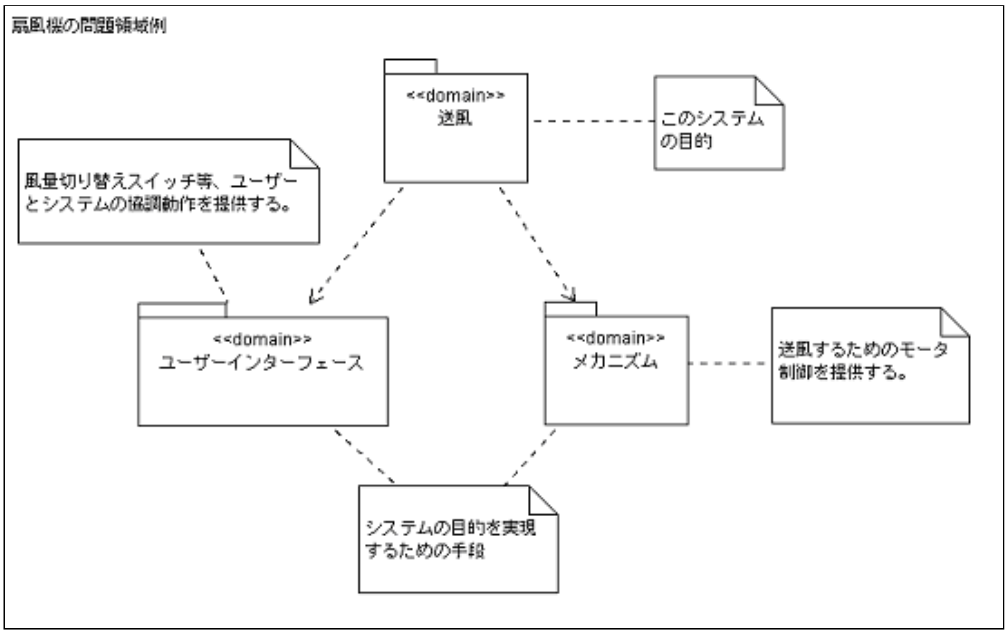

図 9-3 扇風機の問題領域の例

概念

概念とは、物に共通した特徴をある基準でまとめた定義のことです。言い換えると、概念は、物を識別する上で、その物の重要な側面（興味がある特徴）だけをまとめた単位といえます。ここで注意していただきたいのが、特徴を単に複合させるだけでなく、ある基準・視点を軸に仕様化する点です。

実は、私たち人間は日々の生活の中で、この概念を通して物を認識しています。例えば、インスタントコーヒーを作るために、電気ポットに入っている水（お湯）を注ごうとするときに、皆さんはどこまで正しく水を認識していますか？水を正しく認識する、つまり水の特徴をすべて表現するとなると、透明度や匂い、ペーハー、突き詰めれば分子構造まで表現することが必要になります。しかし、このような局面において、

注がれる水の分子構造を意識する人は皆無でしょう。ほとんどの人が、ポットに入っている水の温度や量に関心があり、それで十分なはずです。

また、水を取り扱うシステムであっても、水についての関心事が異なります。システムにとって、水に関する全ての性質は必要ないのです。例えば、電気ポットならば温度と量、洗濯機ならば量、ペーハー判定システムならば水素イオン濃度指数が、それぞれのシステムにおける「水」という概念の特徴となるでしょう。これら概念とその特徴は、UMLにおいてクラスと属性として表記します。例にあげた各システムの水クラスは以下のようになります(図9-4)。

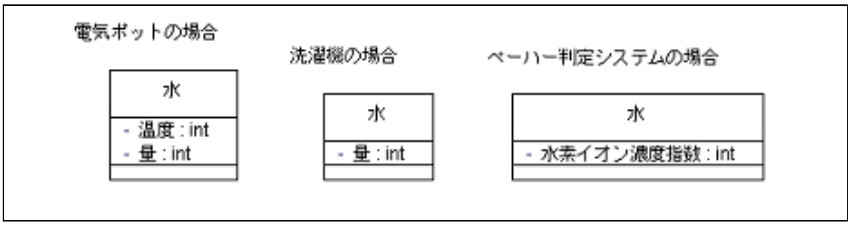

図 9-4 例にあげた各システムの水クラス

つまり、その特徴だけで事物・事象のすべてを表現することはできないが、ある局面において十分な表現、それが概念なのです。

抽象化

抽象化とは、対象を語る上で重要と思われる特徴を残し、それ以外を切り捨てて考えることです。ここでの「重要と思われる特徴」とは、それなしでは対象を説明できない本質的な事柄となります。既に、説明した「概念」や概念と概念の関係を抽出することが抽象化となります。

新聞に掲載されている首相の似顔絵などが、その人を抽象化した結果といえます。たいていの人は、その絵を見れば○○国の首相だと識別できると思います。

また、ボードゲームで遊ぼうとしたとき、あいにくルーレットが壊れていたら、あなたはどうしますか？もしそこにサイコロが転がっていたらどうしますか？おそらくサイコロを代用してゲームをしますよね。つまり、この局面でルーレットやサイコロは、その外観に関する特徴は切り捨てられ、どちらも"ランダムに数を出す"という特徴が共通の概念として抽出されたわけです。結果、ルーレットと同じ概念でくくられるサイコロを代用することが可能となったわけです。仮にルーレットの外観を特徴ととらえ、それに固執したらどうなったでしょうか？サイコロではその代用はできません。

このように抽象化は、実生活の融通を利かせている場面において、無意識ではありますが必ず働いています。融通が利く人は抽象化する能力が高いとも言えるでしょう。

１０：クラス図を書くのが難しいのですが？

クラス図の目的は、物や事柄の関係を導き出すことです。ここでは、初期段階のクラス図を作成することに的を絞って説明します。そのため、ここで出てくるクラスには、操作がありません。実際の開発では、その後、相互作用図を用いて振る舞いを考えます。

クラス図は、UMLにおける中心的な図です。クラス図を書くためには、システム化対象の動きではなく、その構造をとらえることが求められます。時系列の動作説明ではなく、対象を説明する上で欠かせないキーワードとキーワードの構造的なつながりを説明する能力が必要です。

特に長年、組み込みシステムを開発されてきた方は、システムの挙動を主体に考えがちなことが、クラス図を苦手とする理由の一つでしょう。

また、クラス図作成時の状況にもその原因があると思います。クラス図の作成前には、要求分析（要件定義）にて、システムの機能とその動きを洗い出しているはずです。ここでの作業は、システムの外部仕様として、システム外部から見た動きを明らかにしているだけです。このシステム外部の挙動しかない状況から、システム内部の構造を抽出することが難しいのです。

さて、どうしたらクラス図を記述する敷居が下がるでしょうか。クラス図を作成するためには、当然、クラスを抽出することが必要です。まずは、構造化手法で抽出する機能と、オブジェクト指向で抽出するクラスの違いを明確にしておきます。ここで、機能とは、単なるサービスを意味します。クラスとは、サービスと知識の集合を意味します。サービスが操作、知識が属性に相当します。例えば、筆者らは、UMLとオブジェクト指向の「知識」があるから、本記事の執筆を「サービス」していると言えでしょう。あなたが、ある仕事を誰かに任せる場合、その仕事の「知識」を持った専門家に依頼するでしょう。その専門家がオブジェクトとなります。この人間の思考に近い考え方が、オブジェクト指向の考え方なのです。

逆に、構造化手法でも機能を実現するために知識（データ）を使いますが、機能と知識は、分離しています。C言語のデータと関数を分ける考え方は、コンピュータの処理に合わせたものであると考えられます。この手法では、データと関数の2つの軸で構成されるため、どちらかの変更による影響のトレースがしにくくなるといえます。

それでは、クラス図の作成方法をご紹介します。クラス図の構成要素は、クラスと関連が主たるものとなります。ここで、関連（クラス間の関係）は、構造的なつながりを意味するものです。この大原則を意識し、自然言語からクラスと関連を抽出します。今回は、次の文章をクラス図で表現してみます。これが、システムのユースケース（機能）になります。

- 顧客が商品を注文する

この文章は、主語の「顧客」と目的語の「商品」が、「注文する」という関係でつながります。この文章から以下のクラス図が作成できます(図10-1)。

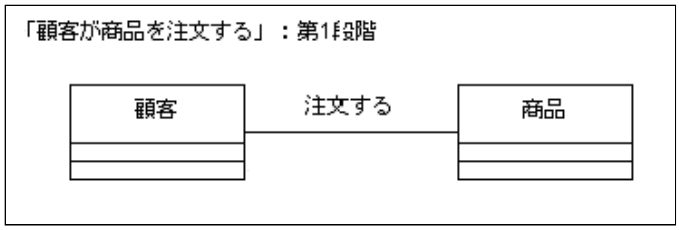

図10-1 「顧客が商品を注文する」ことを表現したクラス図（第1段階）

続いて、以下の要件を付加します。

- 顧客は顧客番号、氏名、住所をもつ
- 商品は商品コード、価格をもつ
- 顧客は一度に複数の商品を注文できる

「顧客」の特徴として、顧客番号、氏名、住所があると分かりましたので、「顧客」クラスの属性にします。「商品」の特徴として、商品コード、価格があると分かりましたので、「商品」クラスの属性にします。顧客は一度に複数の商品を注文できることから、顧客から見た商品の多重度は”多”としました。また、商品が複数の顧客に注文されると考え、商品から見た顧客の多重度は”多”としました。この分析内容から以下のクラス図が作成できます(図10-2)。

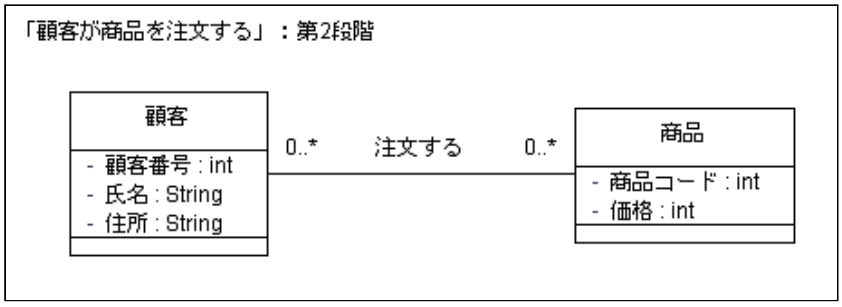

図 10-2 顧客や商品に属性を加えたクラス図（第2段階)

最後に、以下の要件を付加します。

- 顧客が商品を注文した日付を残す
- 顧客は1つの商品を複数個注文できる

ここで追加された、日付と個数は、顧客や商品の固有の特徴ではなく、注文にまつわる特徴と考えました。そこで、関連「注文する」をクラス化し、「注文」クラスを追加します。日付と個数は、「注文」クラスの属性にします。この分析内容から以下のクラス図が作成できます(図10-3)。

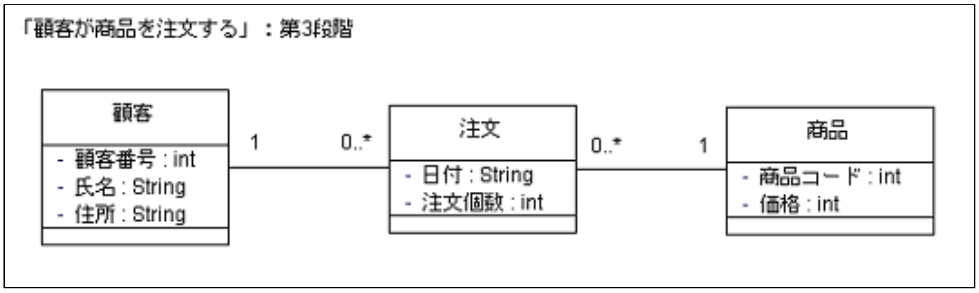

図 10-3 注文クラスを加えたクラス図（第3段階）

いかがでしたでしょうか？クラス図は、人間の考えた内容をそのまま表現できるのです。それだけではなく、クラス図を書くことで、文章では曖昧であった状況を整理できます。考えることと、図に書くことを相互に行き来することで、対象の構造を正確にとらえていくこともできます。皆さんもクラス図を、自分自身の思考を整理するツールとして有効利用してください。

１１：クラス図、ステートチャート図が複雑になり手に負えません。

ソースコードと同じように、モデルは開発が進むにつれて複雑さが増していきます。複雑だと感じたら、原因を特定して、改善し、常にシンプルなモデルを保つように心掛けることが大切です。

ここでは、クラス図とステートチャート図の複雑さを軽減するためのチェックポイントについて説明します。ここで説明する内容は、分析モデルや設計モデルに共通する一般的な内容です。

クラス図

1. **クラスの大きさは適切か**

属性や操作(特にPublic操作)が多いクラスは多くの役割を持っている傾向があります。変更の影響を最小限におさえるためには、クラスの役割をシンプルに保っておく必要があります。大きいクラスは、属性や操作を意味的に分類して、クラスを分割できないかを検討する必要があります。筆者らは、クラスの属性やPublic操作が10以上あればクラス分割の検討を行います。

2. **関連が多すぎないか**

クラスの関連の数が多いと、そのクラスが変更される可能性が高くなります。そのため、クラスの関連は必要最小限に保っておく必要があります。クラスの関連が多いようなら、クラス同士の依存度を弱めるようにクラスの役割分担を再検討する必要があります(図11-1)。筆者らは、関連の数が5以上あれば、これらの検討を行います。

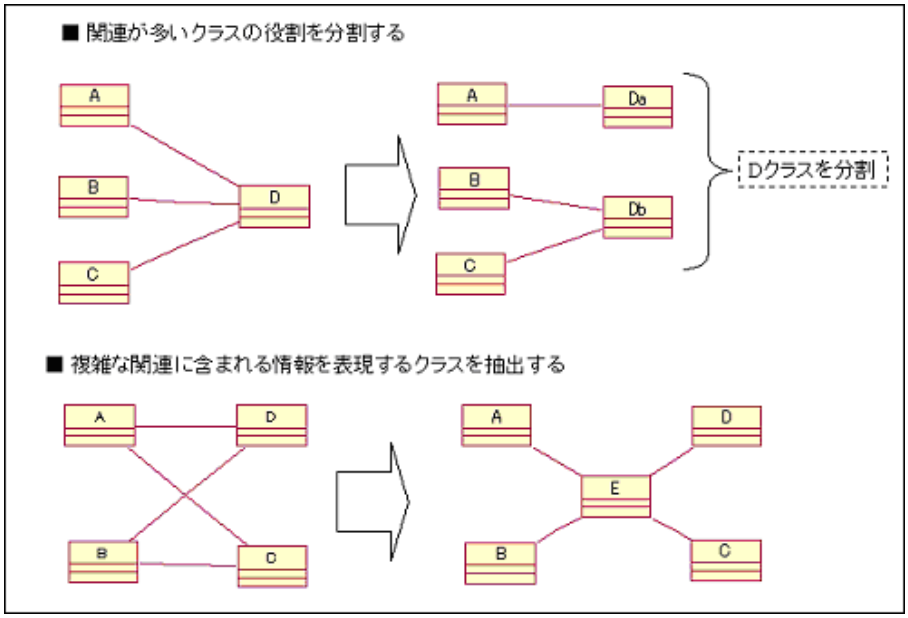

図 11-1 関連を基準にしたクラス抽出の例

関連の方向が決まっているなら、関連の向きに着目します。問題になるのは外部への関連が多いクラスです。外部への関連が多いということは、それだけ多くのクラスに対する変更の影響を受けます。それに対して、外部からの関連が多いクラスは必ずしも悪い設計ではありません。外部からの関連が多いということは、それだけ多くのクラスから利用されているということなので、クラスの役割が凝集している傾向にあります。このようなクラスに変更があると、利用している多くのクラスに変更の影響を与えることになるので、安定性についてレビューしておく必要があります(図11-2)。

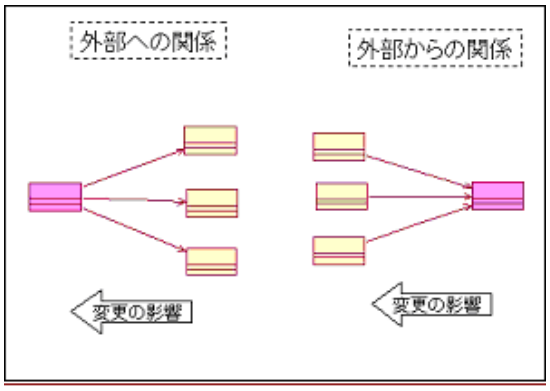

図 11-2 変更の影響

3. わかりやすい名前か

良いクラスや操作には「名は体を表す」適切な名前がつけられています。名前からクラスの役割や操作の内容が理解できないようであれば、より適切な名前を検討する必要があります。適切な名前は利用するクラスの視点でつけられています。

また、適切な名前がつけられない場合は、クラスや操作に問題がある兆候です。クラスや操作の分割が適切か見直す必要があります。

4. パッケージ分割が行われているか

クラスの数が多くなってきたら、パッケージ分割によってクラス群を大きなかたまりに整理する必要があります。パッケージを使用して整理することで、モデル全体が扱いやすくなります(図11-3)。

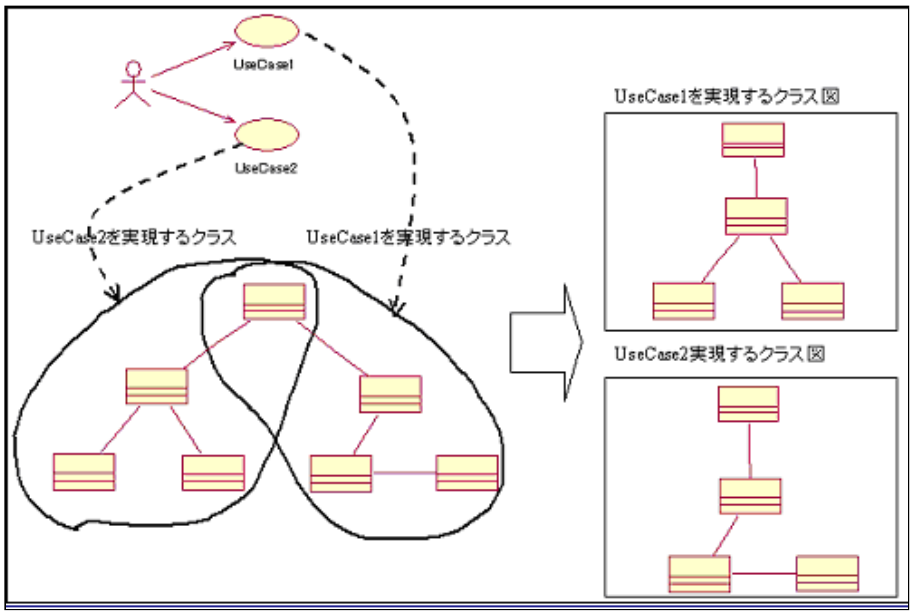

図 11-3 パッケージ分割

以下は、パッケージ分割の指針です。

- 関連する役割のクラス群をまとめる(役割を基準にする)
- 関係の強いクラス群をまとめる(関連を基準にする)
- 再利用する粒度でまとめる(再利用要求を基準にする)
- 変更可能性の高いものと低いものを分ける(変更可能性を基準にする)
- 並行作業の粒度でまとめる(開発作業を基準にする)

パッケージ分割の指針は **オブジェクト倶楽部のメールマガジン (2004-18号)** で詳しく説明されています。そちらも参考にしてみてください。

5. 図が適切に分割されているか

ひとつの図ですべてが表現されるモデルは理解しにくいものです。良いモデルは適切な観点にもとづいて作成された、複数の図で表現されています。また、それぞれの図には表現内容を端的に表した名前が付けられています。このようなモデルは、図の一覧を見るだけで、何をモデル化しているかを把握することができます。

以下は、筆者らが図を分割するときに使用している指針です。

- 概要
  問題領域の中心となるクラスとそれらクラス同士の関係を説明します。ここでは、クラスの属性や操作を表示しないことが多いです。クラス名とクラス同士の関係から概要が理解できないモデルは、名前付けや役割分担に問題がある兆候です。
- ユースケース
  ユースケースを基準に図を分割して、ユースケースを実現するクラス群を明確にします(図11-4)。
- 抽象度
  全体を表現するとボリュームが大きくなるような場合は、全体・詳細、抽象・具象といった抽象度の違いによって図を分割するとわかりやすくなります。さきほど説明した「概要」は全体を説明する図で

す。また、複数の継承構造が関係しているときは、抽象クラス同士の関係と、それぞれの継承構造の詳細は図を分割するとわかりやすくなります(図11-5)。

- 大きさ
  ひとつの図で表現するモデル要素(クラスや関連など)は、PCの画面や印刷結果として読みやすい量におさえます。筆者らは「A4一枚に印刷して文字が読める」ことを目安にしています。

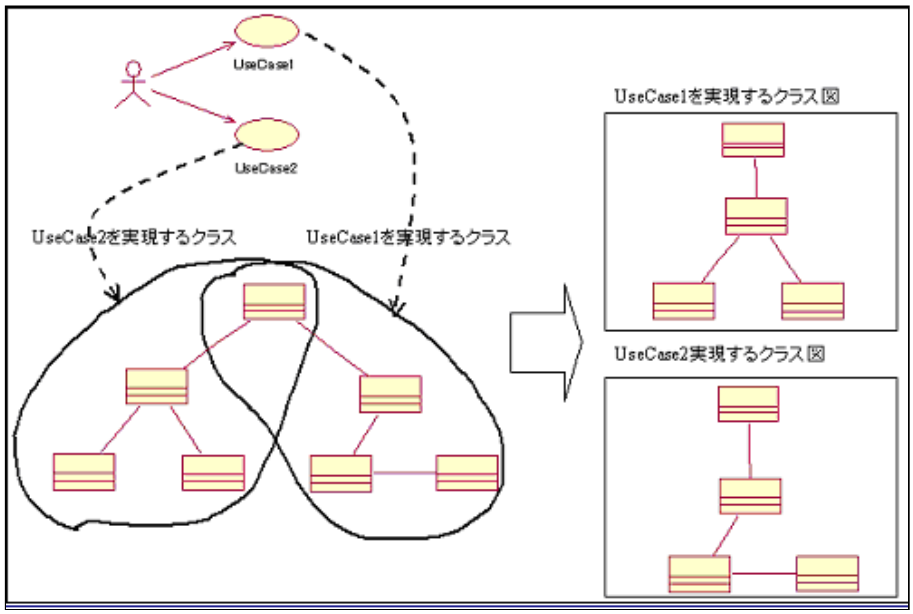

図 11-4 ユースケースを基準にした図の分割

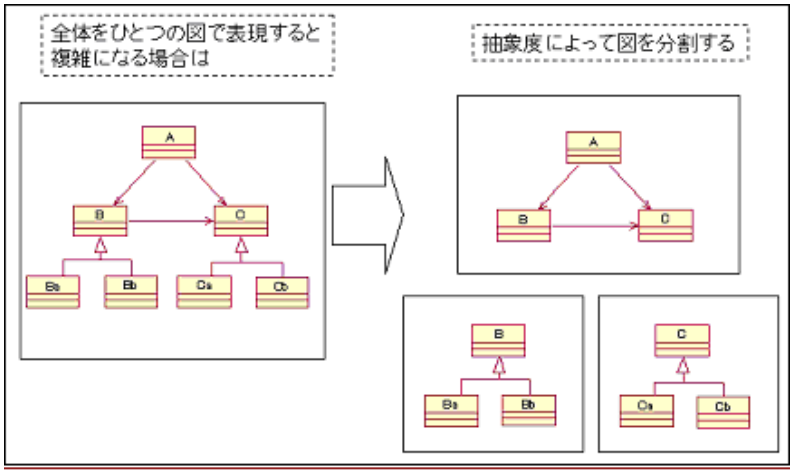

図 11-5 抽象度を基準にした図の分割

6. モデル要素のレイアウトがわかりやすいか

良い図は、読みやすいようにモデル要素がレイアウトされています。以下は、レイアウトについての方針です。これらの方針に従ってモデル要素を配置することができないこともありますが、このように心掛けることが読みやすい(理解しやすい)モデルにつながります。

- クラスのレイアウト
  継承関係はスーパークラスを上に、サブクラスを下に配置します。集約関係は全体を左に、部分を右に、あるいは全体を上に、部分を下に配置します。また、関連の方向は上から下、あるいは左から右になるようにクラスを配置します。
- 関連のレイアウト
  関連の線が交差しないようにクラスを配置します。関連が交差しているクラス図は読みにくいからです。関連の交差が起きないように配置できない場合は、ひとつの図で表現するクラスの数が多すぎるか、クラスの役割が多すぎる兆候です。

ステートチャート図

1. **状態の抽象度が揃っているか**

わかりやすい状態図は、状態の抽象度が揃っています。状態の抽象度にばらつきがある場合は、コンポジット状態を使用して階層化できないかを検討します。

2. **状態の数が多すぎないか**

状態の数が多すぎる場合は、状態の妥当性を確認して不要な状態は削除します。適切な状態は、動的な特徴の視点で抽出されています。完了遷移が多いフローチャートのような状態マシンは適切ではありません。

それでも状態の数が多い場合は、状態の抽象度にばらつきがあるか、状態マシンのコンテキストが大きいことを示す兆候です。状態マシンのコンテキストはシステム、ユースケース、クラスなどに設定されます[6]。クラスの状態マシンが大きい場合は、クラスの役割を分割して、複数の状態マシンで実現することを検討します。

3. **遷移の数が多すぎないか**

遷移先の状態が多い場合は、遷移先の状態の抽象度にばらつきがある兆候です。ガード条件によって遷移の数が多くなっている場合は、分岐などの擬似状態を使用して、遷移の共通する特徴を共通化するのも効果的です。

また、異常系シナリオを実現するために遷移の数が多くなる場合があります。このような場合は、ステートチャート図ですべて表現しようとせず、状態遷移表と組み合わせて説明するのも効果的です。

4. **モデル要素のレイアウトはわかりやすいか**

- 初期状態、終了状態のレイアウト
  初期状態は図の左上に、終了状態は右下に配置します。
- 遷移の線のレイアウト
  状態遷移の線が交差しないように状態を配置します。

参考文献

1. 渡辺博之，芳村美紀，桑本茂樹，敷山喜与彦； 思考系UMLモデリングエクササイズ， 翔泳社.
2. 渡辺博之，渡辺政彦，堀松和人，渡守武和記； 組み込みUML， 翔泳社.
3. 児玉公信； UMLモデリングの本質， 日経BP社.
4. James Rumbaugh, Ivar Jacobson, Grady Booch，UMLリファレンスマニュアル， ピアソン・エデュケーション.
5. ケントベック，(株)テクノロジックアート 訳, テスト駆動開発入門，ピアソン.
6. グラディ・ブーチ，オージス総研 訳, UMLユーザーガイド，ピアソン.
7. テクノロジックアート，UML2ハンドブック，翔泳社.
8. 組み込みネット，https://www.kumikomi.net/index.html.
9. 阿左美 勝，綱島 義人，小坂 優，井山 幸次，システム開発にUMLを適用するためのFAQ，Design Wave Magazine 2004 December，CQ出版.
