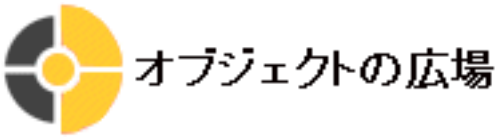


［UMLとオブジェクト指向分析・設計が開発リスクを軽減する］
(DDJ 2月号抜粋)

システムが進化し大規模化するにしたがって
組み込みシステム開発ではあまりなじみのなかった
分析作業の重要性が確実に増大しています
ここでは要求分析やアーキテクチャ設計
システム設計などの上流工程に焦点を当て
オブジェクト指向による組み込みシステム開発の
ポイントについて解説します

ＩＮＤＥＸ

株式会社オージス総研
オブジェクト第一事業部 開発技術コンサルティング室
羽生田栄一／渡辺博之
HANYUDA, Eiichi / WATANABE, Hiroyuki

HOME TOP

オブジェクトの広場

［UMLとオブジェクト指向分析・設計が開発リスクを軽減する］

はじめに

組み込みシステムとは家電製品や工業用機械，自動車などに組み込まれるコンピュータ制御システムのことを指します．これらは従来，機器の付属的な存在でしたが，最近では組み込み機器自体の高機能化やデジタル化が進み，製品に占めるその割合は徐々に高まってきています．
これら組み込みシステムは他の分野のソフトウェアに比べて以下のような特徴があります．

・ハードウェアの制御が中心であり，割り込みやデバイスへのI/Oなど低レベルでの処理が多い．
・時間的制約が厳しく，処理時間が決められているものが多い．
・外部環境から発生する多くの事象に対応するために，複数の処理を並行して行わなければならない．
・プログラムのサイズやメモリなどのリソースが制限されている．

組み込みシステムの世界ではこれらに対処するために，リアルタイムOS（以降，RTOSと呼びます）を使用し，並行性や実行効率を重視したソフトウェア開発が行われてきました．しかし最近ではそれらに加え，製品のライフサイクルの短縮化，ハードウェアの頻繁な変更，製品バリエーションの増加，などに迅速に対応できるだけの能力が求められるようになってきました．これらの要求に応えるためには，拡張や変更に強く，開発リスクを軽減できるようなソフトウェア開発手法が必要になります．また，GUIやネットワーク機能など従来にはなかったような機能が追加されることにより，今まであまり必要とされなかった分析作業の重要性も増してきています．

一方，オブジェクト指向（OO）技術はソフトウェアの柔軟性，再利用性などを高める方法として，現時点で最も進んだアプローチのひとつといえます．しかし，組み込みシステムに対する実績はというと，実行効率やリソース制限などの問題から一部の大規模なシステムを除いてはその適用があまり進みませんでした．現在ではCPUの高速化，コンパイラやオブジェクト指向言語の急激な進歩，メモリなどリソースの低価格化などにより，それらの制限は徐々になくなりつつありますが，まだまだ組み込みシステムへの適用事例は少なく具体的な方法等も十分に提供されているとはいえない状況です．

本稿ではそれらの事実を踏まえ，組み込みシステムをオブジェクト指向で開発するためのポイントについて考えていきます．特に今回は，組み込みシステム開発ではあまりなじみのない要求分析やアーキテクチャ設計，システム設計までの上流工程に焦点を当てます．小さなシステムではこれらの工程が欠けてもさほど問題になりませんが，システム規模が大きくなるにしたがってその重要性が増してきます．一般の入門書にはビジネス系システムを例題にした上流工程の作業内容などが載っていますがシステムの性質の違いから組み込みシステムへの適用はそう簡単ではあ

りません.
　本稿では，組み込みシステムに代表的なエレベータシステムの例を使いながら，開発工程自体の説明と特に組み込みシステムで留意したい以下の内容を中心に考えていくことにします.

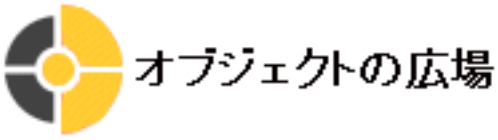
・ユースケースの見つけ方
・クラスの見つけ方
・アーキテクチャ設計
・オブジェクトとスレッドのマッピング方法
・ハードウェアとソフトウェアの切り分け

　なお，用語については以下の基準で使用しています.

・「スレッド」と「タスク」は同じ意味で使用しています．通常は「スレッド」で統一していますが，RTOS関連の話題に限り「タスク」を使用しています.
・「オブジェクト」という語はクラスのインスタンスという意味で使用しています.

　また，紙面の都合上，オブジェクト指向一般の知識や各開発フェーズの詳細な説明，UMLに関する説明などについては触れていません．これらについては他の書籍を参照するようにしてください.

Index Next



［UMLとオブジェクト指向分析・設計が開発リスクを軽減する］

オブジェクト指向開発による利点

まず始めに，組み込みシステムをオブジェクト指向で開発する利点について考えてみましょう．
前述したように，現在の組み込みシステムの多くはRTOS上で動作する複数のタスクにより処理を行っています．このタスクという概念自体は大きなオブジェクトと捉えることもできます．現在のシステムでもタスク間の依存を少なくした設計を行うことで，システム全体の柔軟性やタスク単位での再利用性はある程度実現することができるでしょう．しかし，タスク自体の開発を考えた場合は，粒度があまりにも大きく内部の構造も機能的な視点で設計されているため拡張性や柔軟性という点では大きな問題を持っているといえます．

オブジェクト指向では，システムをタスクではなく，さらに細かなオブジェクトの集合として捉えます．粒度をオブジェクトの単位にすることで開発性や再利用性を向上させることができます．また，オブジェクトを作り出すものをクラスとして定義することで同じオブジェクトを簡単に作り出すことができます．これは，同一機器やプロトコルなどを複数制御することが多い組み込みシステムでは非常に便利です．
さらに，オブジェクト指向開発では抽象的なインターフェイスで設計・実装を行える点も大きな特徴です．オブジェクト指向のポリモフィズムという機能を使うことにより，実装に左右されない汎用的な設計を行うことが可能になります．GUI，ハードウェア，内部のアルゴリズムなど頻繁に変更されることが予想される部分は，この機能を使うことで変更に強い拡張性のある作りにすることができます．また，同じような機種に対してはフレームワークと呼ばれる「基本機能の実装と拡張用のインターフェイスから構成されるシステムの枠組み」を作成することにより効率的な差分開発が可能になります．
開発プロセス面から見た場合も，オブジェクト指向開発の大きな特徴である繰り返し型の開発はさまざまな開発リスクの低減に大きな威力を発揮します．

このように，オブジェクト指向は組み込みシステムの開発においても非常に大きな利点を持っていることがわかります．しかし，その一方では，先に述べたような制約が存在する事も事実です．たとえば，
・リアルタイム性を求められる時間的制約の厳しい部分
・ハードウェアを直接操作しなければいけない部分
などについては，拡張性や再利用性などよりも効率やサイズ等を重視した設計が必要になってくるでしょう．こういった部分は，無理矢理オブジェクト指向で開発するよりも，むしろ従来のリアルタイム制御に特化した設計技法やアセンブラ，Cなどによる実装が適しているかもしれません．

結論を言うと，組み込みシステムの開発では目的に合わせて一番有効な開発手法を用いることが開発のポイントになります．すべてに対してオブジェクト指向を適

用するというよりは，従来手法と組み合わせ，より有効な部分に対して適用しそのメリットを引き出していく方が望ましいと思います．

組み込みのためのOO開発手法はUMLで！

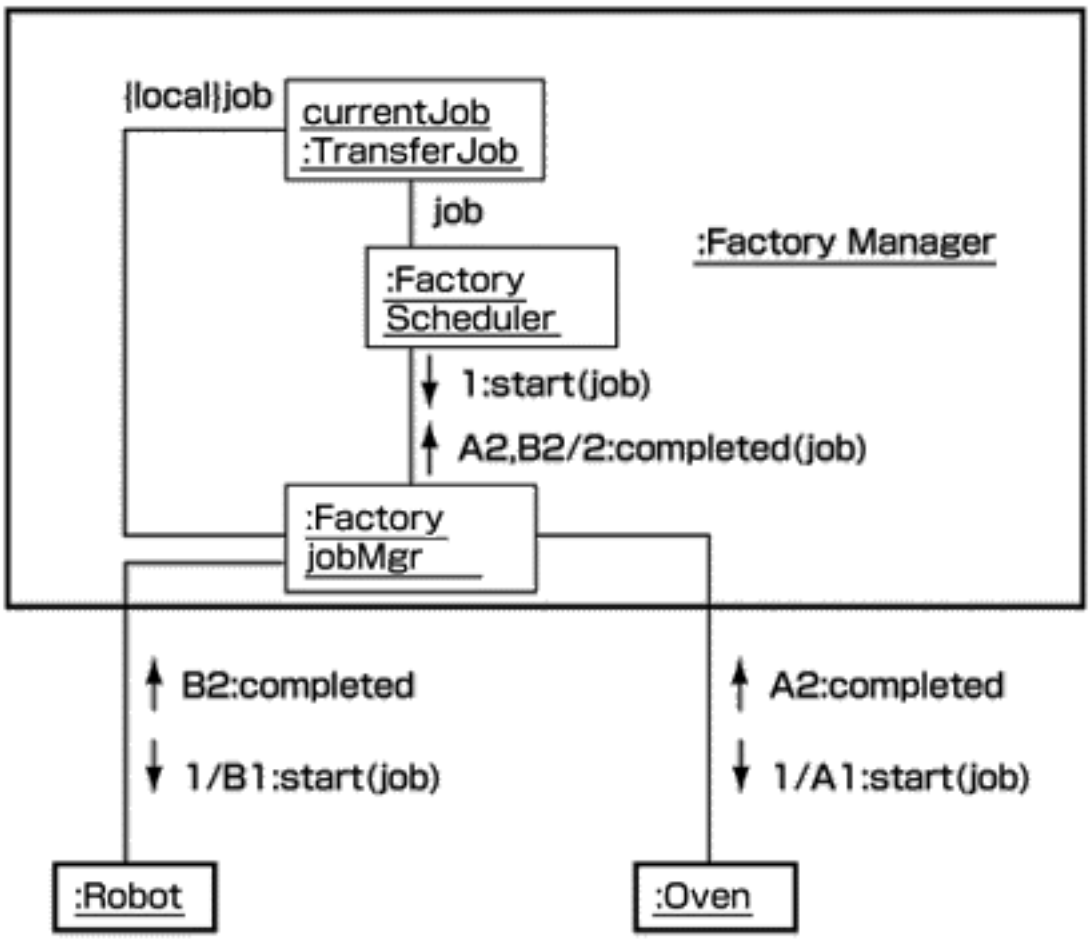
ここで簡単に今回の記事のベースになっているリアルタイム向け開発方法論および，そこで使われるオブジェクトモデリング記法について触れておきます．
今回私たちが開発技法として紹介したものは，フィンランドのテレコム関連の会社であるNokiaのOOチームが提案しているOCTOPUS手法を参考に，オージス総研オブジェクト第1事業部での組み込み分野の開発経験を踏まえて独自に拡張したものです．そして，UMLを使った分析設計モデリングは組み込み系リアルタイム系の分野にもかなりフィットするという感触を得ています．特に基本的なシステム要件を定義するユースケース図とシーケンス図，問題領域やシステム内の構造を表現するクラス図やパッケージ図，オブジェクトの集団の振る舞いを表現するコラボレーション図，各コンポーネントの状態マシンとしての定義はステートチャート図で行い，各コンポーネントのハードウェアへの割り付けを表現する配置図といった組み合わせでほぼ必要十分なリアルタイム組み込み向けのモデリングを行えると経験上感じています．
実際，UML提案者の1人であるBooch氏は，現状のUMLはリアルタイムシステムの設計に十分だと述べていますし，彼自身多くのリアルタイム・リアクティブシステム（その多くは防衛関係）の設計にUMLを利用しています．特に，現在のUMLには，アクティブオブジェクト（図1で太線で示したボックス）の概念があるので，関連するオブジェクト集団間の一連の相互作用を表現するコラボレーション図をうまく使うと並行に動く複数オブジェクト（すなわちスレッドやタスクを割当てられたアクティブオブジェクト）間のイベントシーケンスを明確に表現できます．

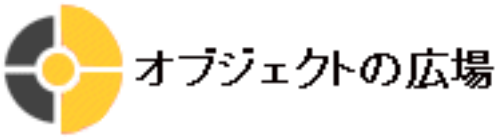
たとえば，図1にあるように，3つのオブジェクトFactoryManager, Robot, Ovenが各々タスクを割り当てられたアクティブオブジェクトでかつFactory Managerは3個の内部オブジェクトを保持し，それらの間でのイベントのやり取りや待ち合わせ（2: completed(job)はイベントA2, B2を待ち合わせている）が明確に表現されています．
ただし元来のコラボレーション図はオブジェクトを基本要素としていて，アーキテクチャ設計には要素が細かすぎます．OO技法の基本単位である小粒度のオブジェクトとリアルタイムシステムの基本単位である中粒度のタスクの間の折り合いをつけるノウハウが組み込み系にオブジェクトを適用する際のひとつのポイントだとすると，大局的な観点から比較的大きな粒度のコンポーネントどうしの相互作用をモデル化するための手段が必要です．そこで本技法では，システムアーキテクチャの設計時に高レベルのコラボレーション図（この目的のために開発されたユースケースマップという記法を利用するとさらに効果的です）を作成して各ユースケースに対応したスレッドの最適性やリスポンシビリティのコンポーネントへの割り当てのトレードオフを検討しています．

本技法は標準のUMLを最大限利用して組み込み系分野のモデリングに対処するという現実的なスタンスですが，リアルタイム系専用のモデリング要素を現在のUMLに追加するというアプローチも存在します．Bran Selic氏らの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ます．オブジェクトが演じる役割をカプセルと呼び，その役割が必要と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します．これらの概念は，UMLのステレオタイプ機能を用いて次の版のUMLにリアルタイム拡張として取り入れられる可能性があります．ただし，どの概念も工夫次第で現状のUMLでも表現できますので，組み込みシステムの開発に現在のUMLを適用する上でこれらの概念の欠如はそれほど問題にはなりません．待つよりは早く適用して経験を積む方を選択することをお勧めします．

図1：コラボレーション図





［UMLとオブジェクト指向分析・設計が開発リスクを軽減する］

開発サンプル

本稿では開発サンプルとして，組み込みシステムの例題でよく使われるエレベータシステムを取り上げました．とはいっても，この手の例題によくある1台限りのエレベータではなく，複数が同時に動作する，より現実に近いものを考えることにします．
システムの主な仕様は以下のとおりです．

・エレベータは合計4台あり，8階建てのビルに設置されている．
・エレベータやフロアの構成，呼び出し時の動作などについては一般のエレベータとだいたい同じと考える．
・今回は非常時の対応や安全性，正確性等については配慮しない．

また，本来であれば各エレベータは別々のCPUで制御されると思いますが，今回は例題ということですべて同一のCPUで操作できることを前提にします．
これ以降は，一般的な説明の後に随時このエレベータシステムでの検討結果を示しながら進めていくことにします．
それでは，さっそく要求分析からはじめてみましょう．

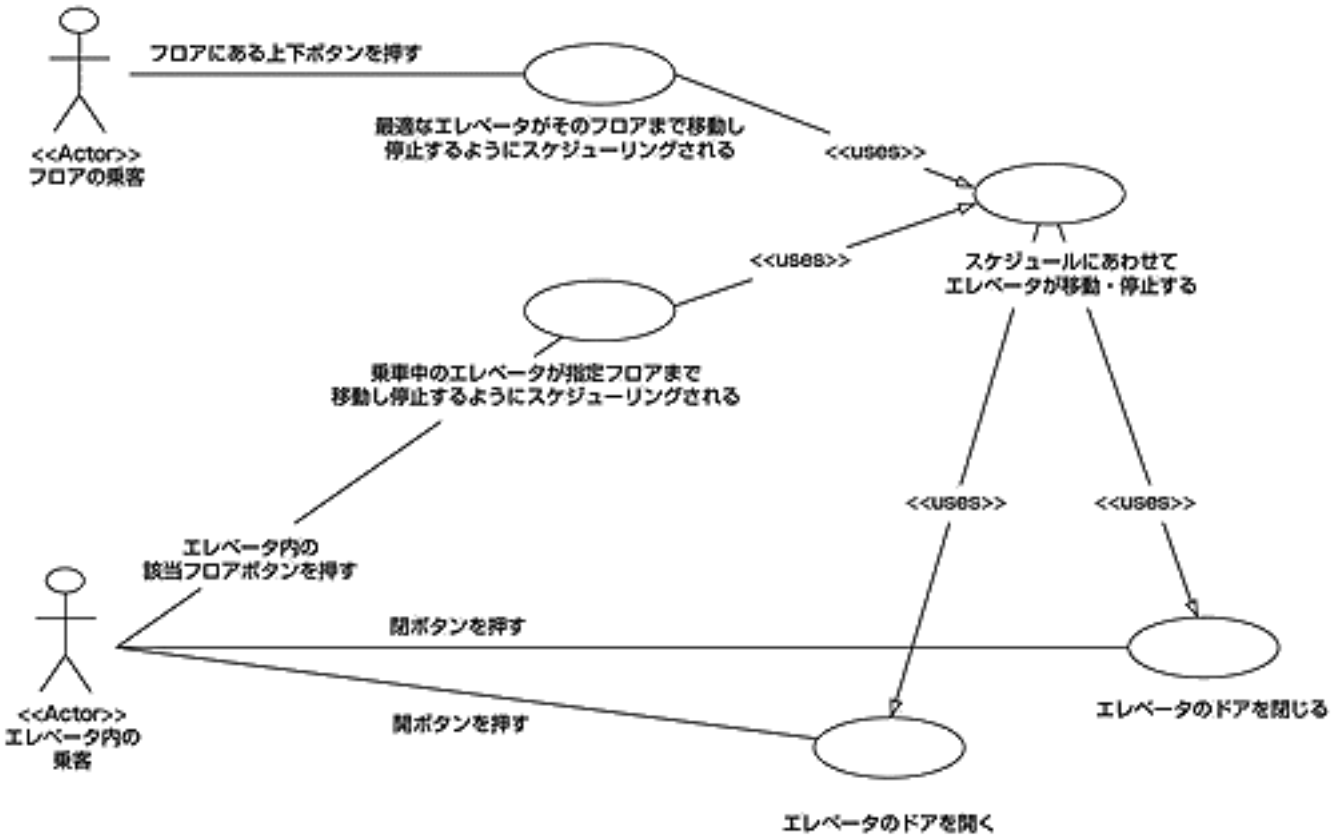
オブジェクトの広場

［UMLとオブジェクト指向分析・設計が開発リスクを軽減する］

要求分析

　オブジェクト指向開発での要求分析にはユースケースが使用されます．ユースケースを組み込みシステムで使用する場合には，アクターのとらえ方とユースケースの粒度がポイントになります．
　今回の開発サンプルでは，アクターとしてフロアの乗客とエレベータ内の乗客が考えられます．これらをもとに作成したユースケース図を図2に示します．

図2：ユースケース図



　エレベータシステムではアクターからのアクションで適度な大きさのユースケースを切り出せました．しかし通常の組み込みシステムでは，外部からの指示を受けた後はシステムが自動的に判断し処理を継続することが多く，アクターに合わせたユースケースでは粒度が大きくなる傾向があります．このような場合には，ユースケースを適度な粒度まで機能単位に分割します．
　また，ユーザーからの入力をほとんど持たないロボットやクルーズコントロール

などの制御システムなどではアクターからの指示に該当する部分を，外部の状況を機能的に判断することで代行しているため，システムからサービスを受けるアクターに注目しただけではユースケースを見つけることができません．このような場合はシステムが監視したり操作したりするハードウェアや他のシステムなどの外部環境をアクターとして扱います．そして，これらとのやりとりの単位をユースケースとしてとらえるようにします．

なお，後者のような外部環境からの検討を行う際には，あらかじめコンテキスト図を作成します．コンテキスト図とはシステムと外部の間でやり取りされる情報を表わしたものです．ユースケースの契機となるような外部イベントは，コンテキスト図を元に検討すると容易に見つけることができます．

Prev. Index Next

オブジェクトの広場

［UMLとオブジェクト指向分析・設計が開発リスクを軽減する］

分析

分析の目的はシステムで扱う領域の構造を理解することです．この分析作業から得られるモデルを開発のベースにすると，システムの機能や実現方法などに左右されない安定した構造を得ることができます．
分析では具体的に次のような作業を行います．

・システムを構成するクラスを見つけだす．
・サブシステムに分割する．
・ユースケースをサブシステム単位に分割し，それを使ってクラスを検証する．

(1)システムを構成するクラスを見つけだす
まず，クラス候補の洗い出しを行います．組み込みシステムの場合，初期のクラス候補は大きく以下の3種類に分類できます．

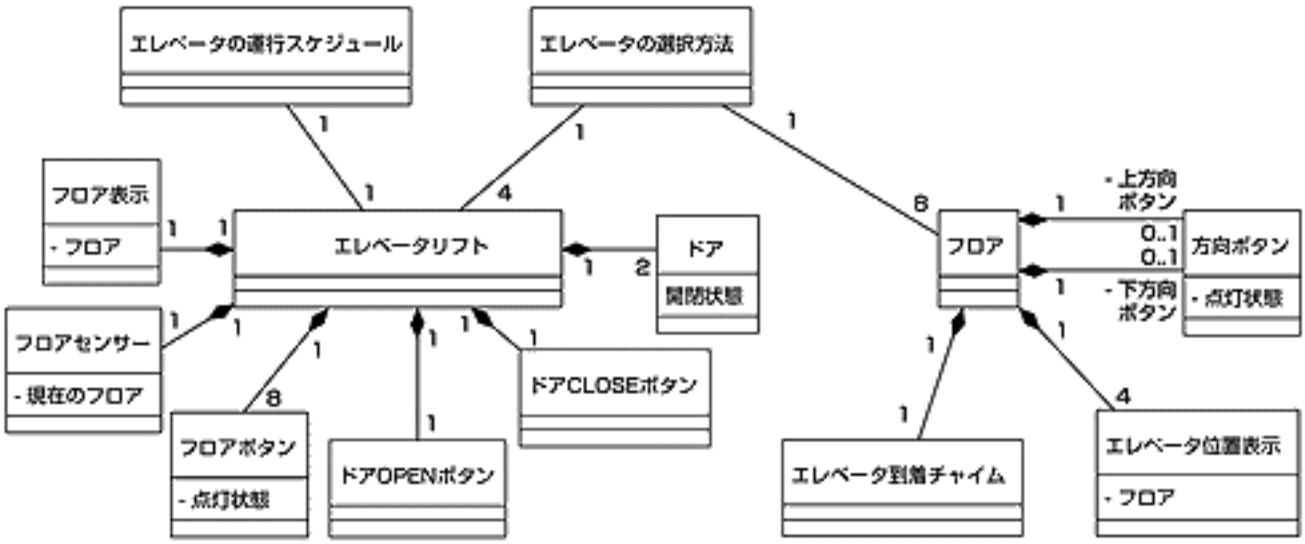
●ハードウェアラッパー
制御する必要のある物理的なハードウェアとのインターフェイスとなります．エレベータシステムでは「エレベータリフト」や「フロアボタン」「方向ボタン」「ドアOPENボタン」「フロアセンサー」などが該当します．但し，「エレベータシャフト」のようにシステムとして直接制御する必要がないものはクラス候補にする必要はありません．

●制御の対象となるもの
制御の対象となる実世界の物やシステムで扱う情報を表わします．エレベータシステムの場合，実際に運ぶ「人」や「荷物」が該当します．他システムの例だと，自動販売機システムの「商品」や「硬貨」，クルーズコントロールシステムの「路面状況」や「速度」などが考えられます．

●システムのコンセプト
システムのコンセプトもクラスの候補となります．エレベータシステムでは「エレベータの運行スケジュール」やフロアから呼ばれた際の「エレベータの選択方法」などが候補として見つかります．組み込みシステムの中心であるシーケンス制御や計測制御なども，ここに分類されるクラス候補となります．

以上のような分類を参考にしてクラス候補を抽出したら，次にグループ分けを行います．物理的，機能的に近いクラス同士をグループに分け，各グループ単位にその中のクラス候補がクラスとして適当かどうかを検討し洗練していきます．エレベータシステムの例だと，制御対象として候補に挙げた「人」や「荷物」は直接システムが意識する必要がないのでクラス候補からはずします．
次に，各クラスの属性やクラス間の関連などを検討しながら，最終的にクラスとして特定していきます．この時点で作成されたモデルは概念モデルと呼ばれます．
エレベータシステムの概念モデルの一部を図3に示します．

図3：エレベータシステムの概念モデル



(2)サブシステムに分割する

システムがある程度大きな場合は，クラスを物理的・機能的な面からサブシステムと呼ばれるグループに分割します．サブシステムへ分割することによって得られるメリットは次のとおりです．

・システムの全体の見通しが良くなる．
・開発しやすい大きさになる．
・各サブシステム間の依存関係を少なくすることでシステムの保守性を高め，並行開発を可能にする．

サブシステムは，それ単体で特定の機能を持つもの，物理的に隣接するもの，他のサブシステムにサービスを提供するもの，などいろいろなものが考えられます．この時点では，先に行ったグループ分けを参考にして，あまり小さくならない程度のサブシステムに分割します．これらのサブシステムはUMLのパッケージとして表現されます．

エレベータシステムでは物理的，機能的な面からエレベータサブシステム，フロアサブシステムをそれぞれ導出します．なお，各サブシステムのインスタンス数もコメントなどを使って明記しておくと良いでしょう．今回の場合はそれぞれ4，8になります．

(3)ユースケースをサブシステム単位に分割し，それを使ってクラスを検証する

分析の最後はユースケースを使ってサブシステムごとにクラスの構成を検証します．

まずは，各ユースケースを実現するために必要なサブシステムを考えます．たとえば，外部とのインターフェイスやシステム全体の制御などを行うサブシステムなど概念モデルに含まれていないものはこの時点で追加する必要があります．

次に複数のサブシステムにより実現されるユースケースを各サブシステム単位のユースケースに分割します．

最後に，サブシステムごとに分割されたユースケースを使ってシナリオ図を作成し，サブシステム内部のクラス構成を検討します．この段階で，各クラスに操作が

割り当てられたり，「処理」や「操作」を表わすクラスが追加されます．また，複数のクラスを制御する必要があれば，コントローラと呼ばれるクラスを導入します．

具体的にエレベータシステムの例で考えてみましょう．まずフロアの乗客とエレベータ内の乗客からのリクエストにはエレベータおよびフロアの各サブシステムで対応します．最適なエレベータの選択についてはエレベータ選択サブシステムを新たに追加します．また，エレベータとフロアのやりとりなどシステム全体の制御を行うシステム制御サブシステムも追加します．乗車中のエレベータのスケジューリングについては各エレベータサブシステム内で対応します．以上の検討の結果をパッケージで表現したのが図4になります．

そしてエレベータサブシステムのユースケース図と内部のクラス構成を検討した結果を図5，図6に示します．ここでは複数のハードウェアラッパーを制御するエレベータコントローラクラスを新たに追加し，主要なメソッドも合わせて記述しています．

図4：更新されたサブシステム

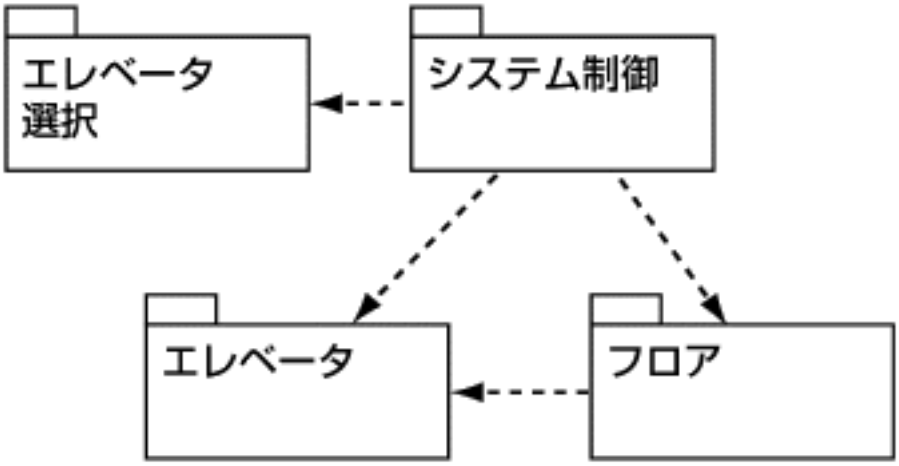


図5：エレベータサブシステムのユースケース図

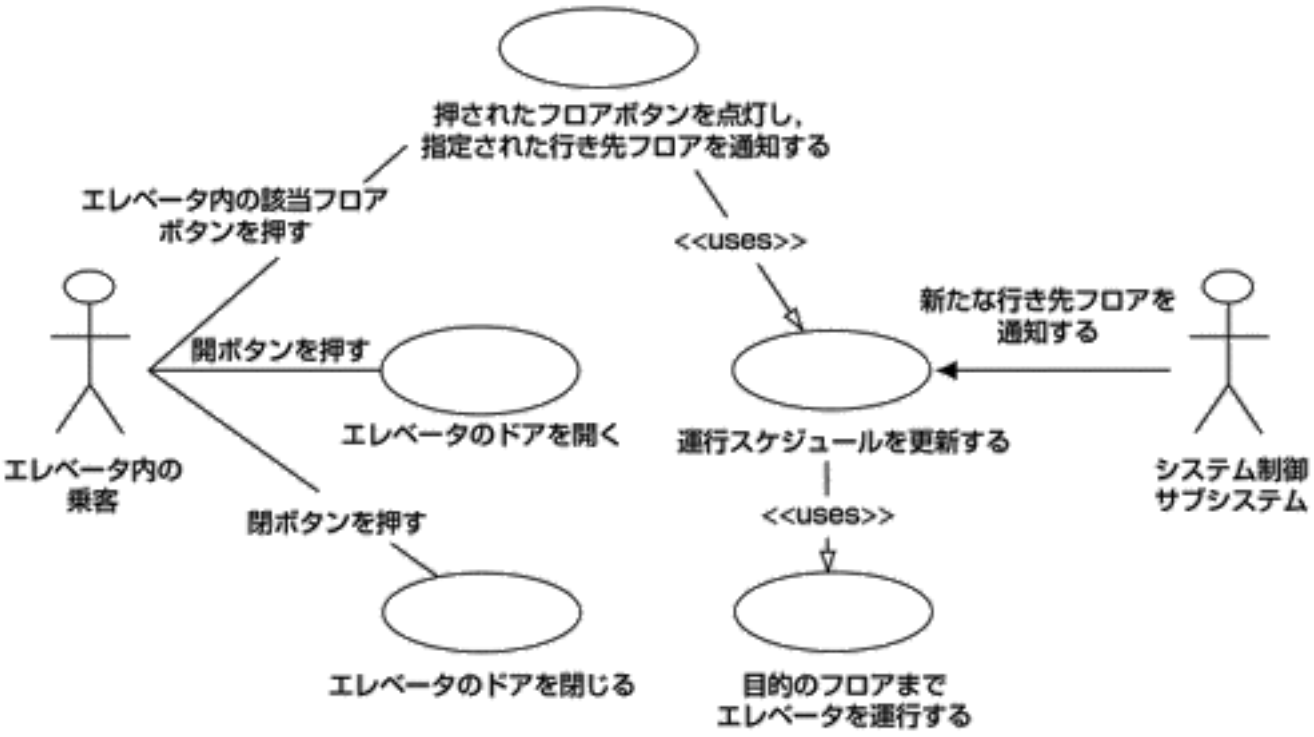
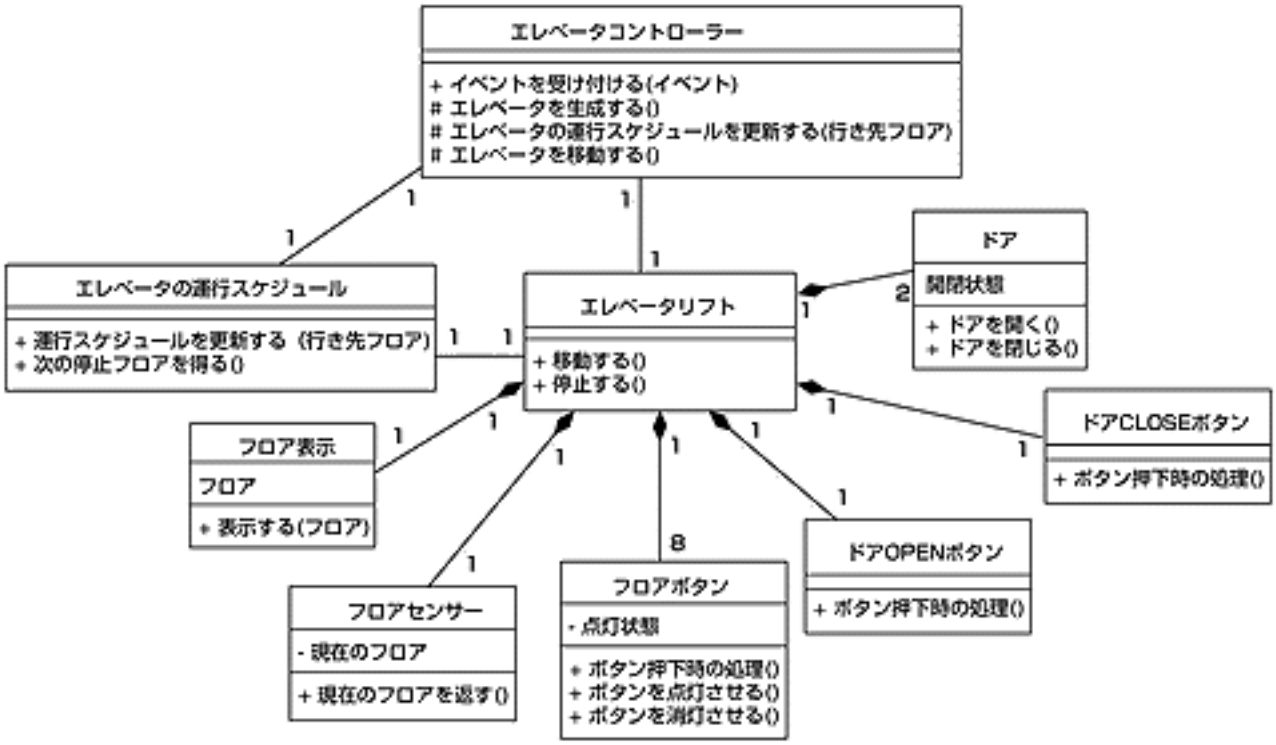


図6：エレベータサブシステムのクラス図



オブジェクトの広場

［UMLとオブジェクト指向分析・設計が開発リスクを軽減する］

アーキテクチャ設計

設計作業に入る前に，まずシステム全体のアーキテクチャを決めます．アーキテクチャとはシステムの基本的な構造や制御の流れを指します．言い換えれば，システム全体のマクロな設計方針ともいえます．組み込みシステムの場合は，ハードウェアの変更などに柔軟に対応できるようなクラス構成やシステムの制御方法を決めるスレッド設計などが中心となります．

(1)システムの階層化
オブジェクト指向ではクラスを使用することで変更に対する柔軟な構造を実現していますが，クラスの数が多くなるとその管理が大変になり，それだけでシステムの大きな変更に対応することは難しくなります．そのため，クラスより大きなレベルで，あらかじめ変更に耐え得るようなシステムの構造を設計しておくことが必要です．ここではそういったシステムの構造として最も一般的な階層化について考えてみましょう．
まず，システムをその役割に応じて階層的に分割します．上位の階層ほどシステムに固有の制御や表現などを受け持ち，通常の変更はこの階層内で吸収します．下位の階層はシステムに影響されないドメイン固有の機能を提供します．分析で見つけたサブシステムやクラスは主にこの階層に含まれます．
複数の階層で共通に使用するサービスなどについては，それらを提供するグループを作ります．これらのグループやそこに含まれるクラスはシステムを構築する際に必要となるもので分析モデルには含まれていません．アーキテクチャ設計を行うことにより新たに見つけ出されることになります．
そして最後にこれらの階層やグループを適切な依存関係で結びます．
このようにして作成されたシステムは，変更範囲を特定の階層に局所化することができるため，変更への対応が容易になります．また，汎用的な階層のクラスは再利用性を増します．
組み込みシステムの場合には，さらにハードウェアの変更から受ける影響を最小限にするためにハードウェアラッパーと呼ばれる階層が追加されます．
主な各階層やグループとしては次のようなものが考えられます．

●ルート
システムの起動，終了に関わるグループで，スタートアップルーチンから呼び出されるクラス，初期化を行うクラス，システム全体の生成を行うクラスなどから構成されます．

●システム制御
システム全体にわたる振る舞いや，各サブシステム間の制御を行います．

●アプリケーション
システム固有のドメインに特化しない処理を行う階層です．ユースケースによる分析で導出したコントローラの一部や，どのサブシステムにも属さない処理を表わ

すクラスなどはここに含まれます.

●ドメイン
　システムで扱う問題領域の中心を表わします．分析で導出したシステムの対象となるクラス，システムのコンセプトを表わすクラスなどが含まれます.

●ハードウェアラッパー
　システムで制御するハードウェアを表わします．分析で導出したハードウェアラッパーから構成されます.

●共通ファシリティ
　システム全体で使用するユーティリティ的なサービスを提供します．タイマーやログ出力，エラー処理に関するクラスなどから構成されます.

●分散
　複数ノードに分散しているシステム等では，分散処理に必要となるクラスをまとめます．CORBAやDCOMなどを用いた場合にIDLから作成されるクラスなどがここに含まれます.

●オペレーティングシステム
　OSの提供するサービス群（API）を抽象的なインターフェイスで提供します．プロセスやスレッド，メッセージキュー，セマフォなどのクラスから構成されます．ただし，このグループをどの程度まで充実させるかは汎用性と効率とのトレードオフになります.
　それでは実際にエレベータシステムを使って階層化の作業を行ってみましょう．各階層はサブシステムと同様にパッケージとして表わします.
　まず，システム全体のルートパッケージを追加します．エレベータ選択サブシステムはアプリケーション層に該当します．また，エレベータおよびフロアの各サブシステムはその内部をアプリケーション，ドメイン，ハードウェアラッパーに階層化できます．たとえば，エレベータコントローラはアプリケーション，エレベータの運行スケジュールはドメイン，各ハードウェアはハードウェアラッパーにそれぞれ該当します．今回のシステムでは各階層に数個のクラス程度しか存在していないので特にパッケージとしては分割していませんが，今後各レイヤーのクラスが増えていく可能性が高ければ各レイヤーごとにパッケージとして分割しておくほうがよいかもしれません．また，エレベータパッケージの各オブジェクトを生成するエレベータファクトリークラスを追加し，ハードウェアラッパークラスにはステレオタイプを付与します．図7と図8にこれらの検討を行った最上位パッケージ，エレベータパッケージをそれぞれ示します.
　なお，特定の用途に合わせたフレームワークを作る場合には，階層をさらに汎用部分とカスタマイズ部分に分けて考える必要があります.

図7：最上位パッケージ

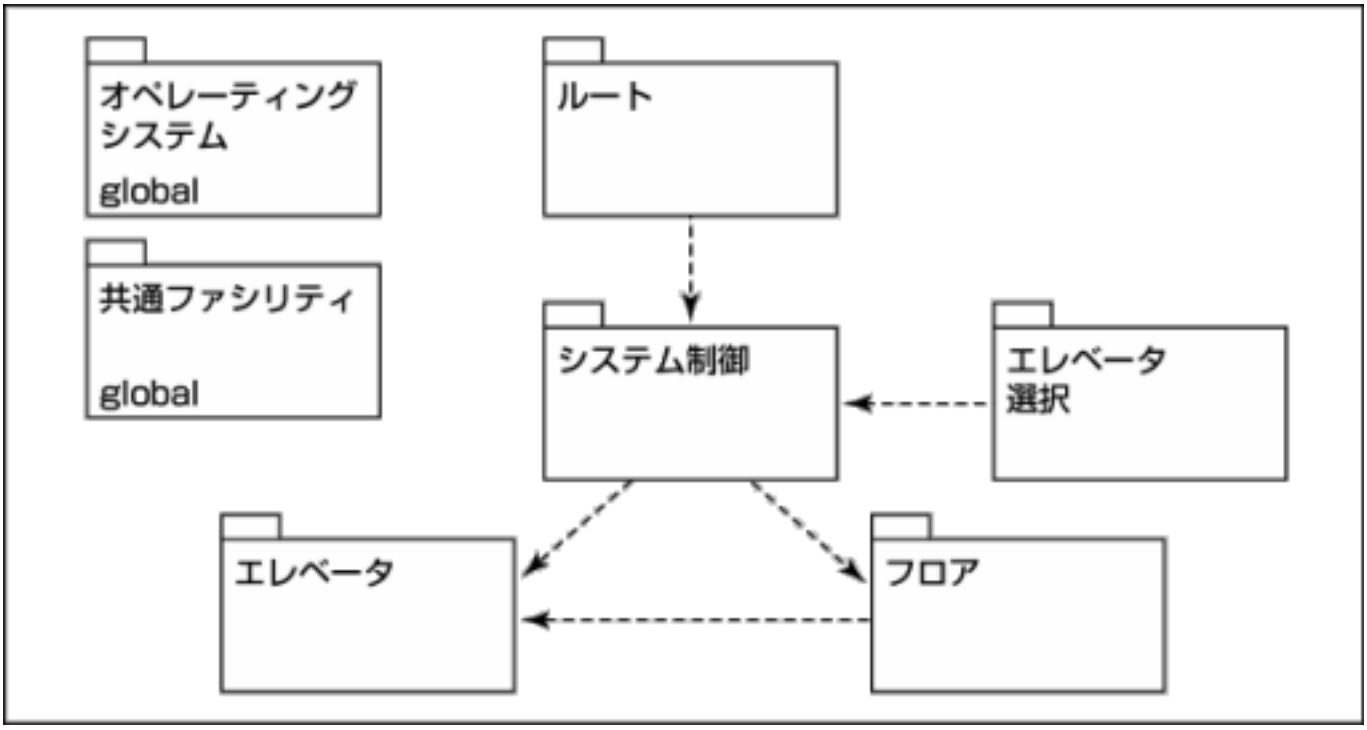
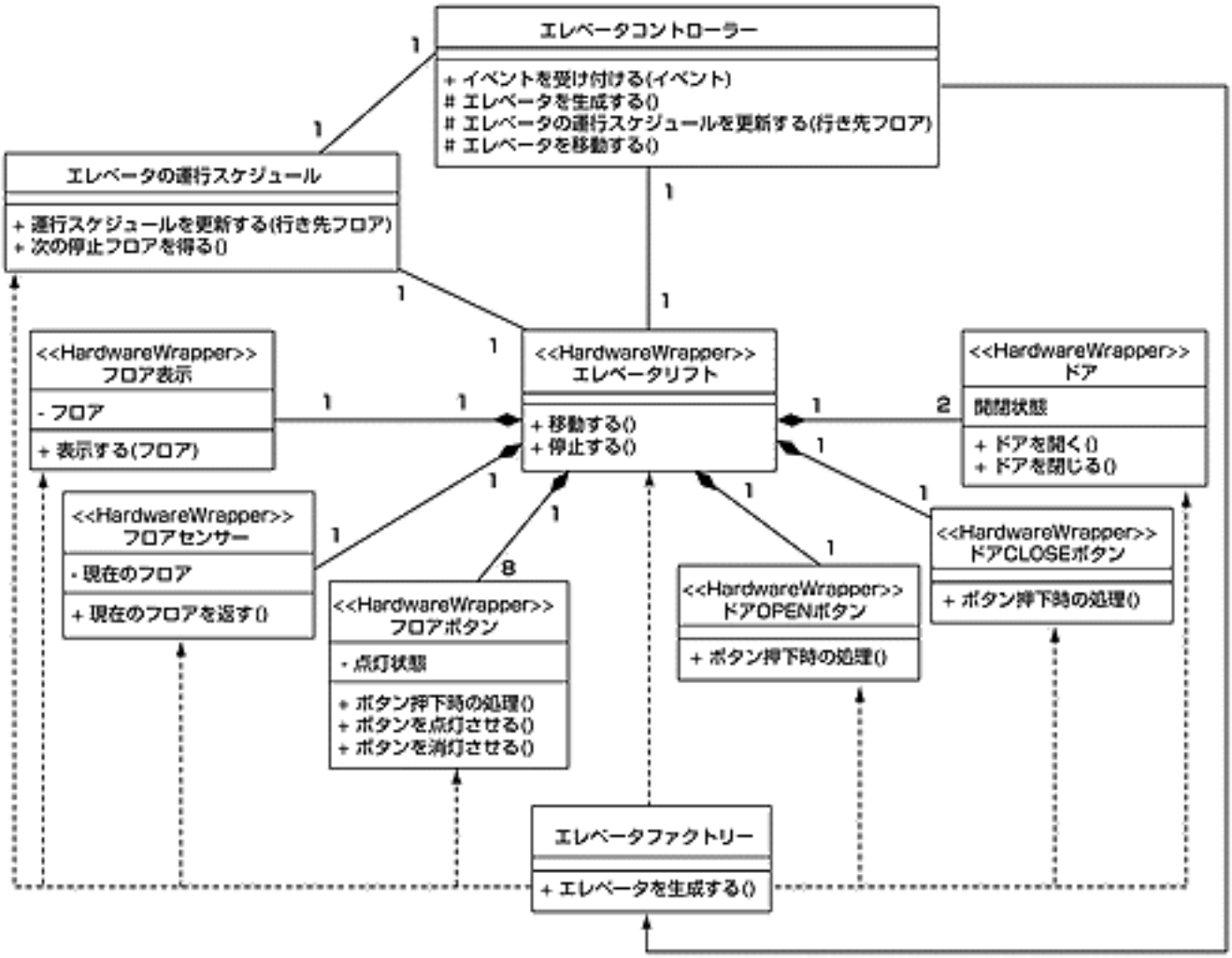


図8：エレベータパッケージ



(2)スレッドの設計

ユースケースの項で述べたように，ビジネス系のソフトウェアでは通常ユーザーからの入力が一連の処理の契機になるのに対し，組み込みシステムでは，外部環境からのいろいろなイベントを契機に処理が実行されます．ですからシステムの制御構造には，同時に発生するこれらのイベントに迅速に対応できる能力が必要とされます．これらに一番大きな影響を与えるのはスレッドの設計です．以下では組み込みシステムのスレッド設計方法のいくつかについて具体的に考えてみます．

●シングルスレッドで処理する

外部からの割り込みをいったんすべてイベントに変換し，メッセージキューに保持します．main処理の中では永久ループでメッセージキューからイベントを取りだし，該当するオブジェクトのメソッドを呼び出します．この方法のメリットとしては，簡単なこと，制御のためのOSが不要なためCPUパワーをすべて使用できること，などがあげられます．デメリットは，単一スレッドのため並行処理が行えないことです．いずれにしても，この方法はあまり大きくない，リソースの制限されたシステムで採用されることが多いようです．

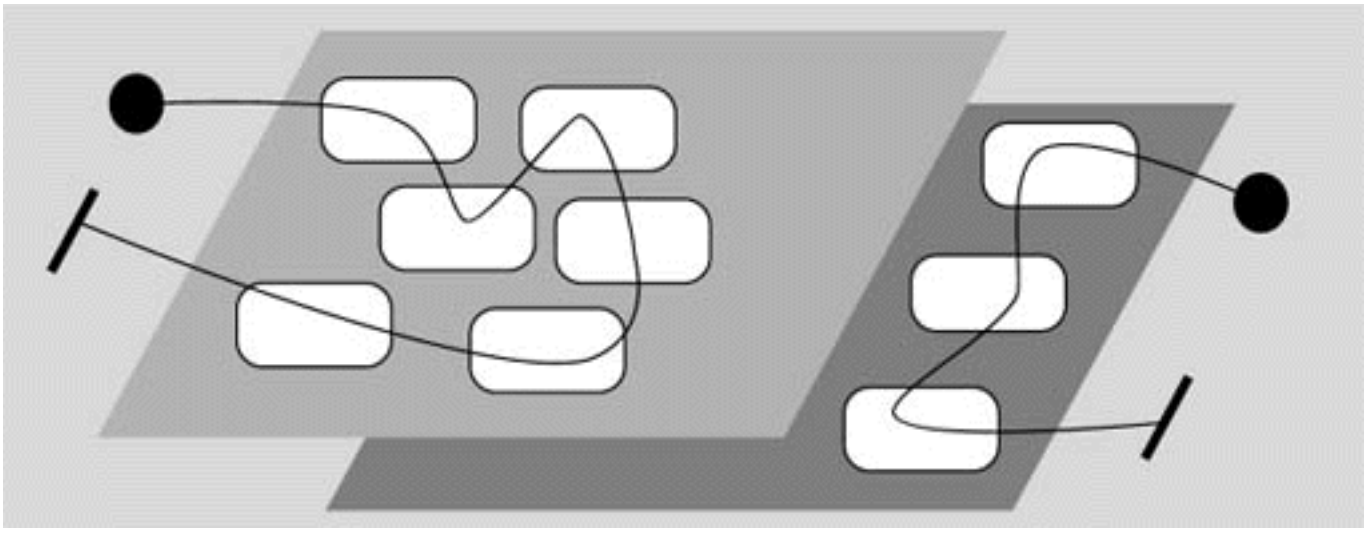
●ひとつのイベントをひとつのスレッドで処理する

外部からのイベントを並列に処理するため，イベントを処理する一連のパスごとにスレッドをひとつ割り当てます．つまり，ひとつのイベント処理の実行パスのすべてをひとつのスレッドに割り付けます．これを実現するためには，あらかじめイベントに割り当てるスレッドをまとめて生成しプールしておき，イベント発生に応じその処理を行うパスにスレッドを割り当てる等の処理が必要になります．

この方法のメリットとしては，プリエンプティブな並行処理が可能なこと，オブジェクト呼び出しが関数コールとなるため呼び出しにかかるコストが小さいことなどがあげられます．一方，並行に行われるイベント処理間で共有されるオブジェクトが多くなる場合には，リエントラントな呼び出しが多発することになります．クラスがリエントラントでない場合には，これを防ぐためにクラスの各メソッドをクリティカルセクションやセマフォなどを使用して排他制御する必要があります．また，それによる効率の低下も問題になってきます．

この方法のイメージを図9に示します．ひし形がスレッドを，角の丸い四角がオブジェクトを表わします．黒丸はパスの開始点をバーはパスの終了を示します．

図9：ひとつのイベントをひとつのスレッドで処理する



●ひとつのイベントを複数のスレッドで処理する

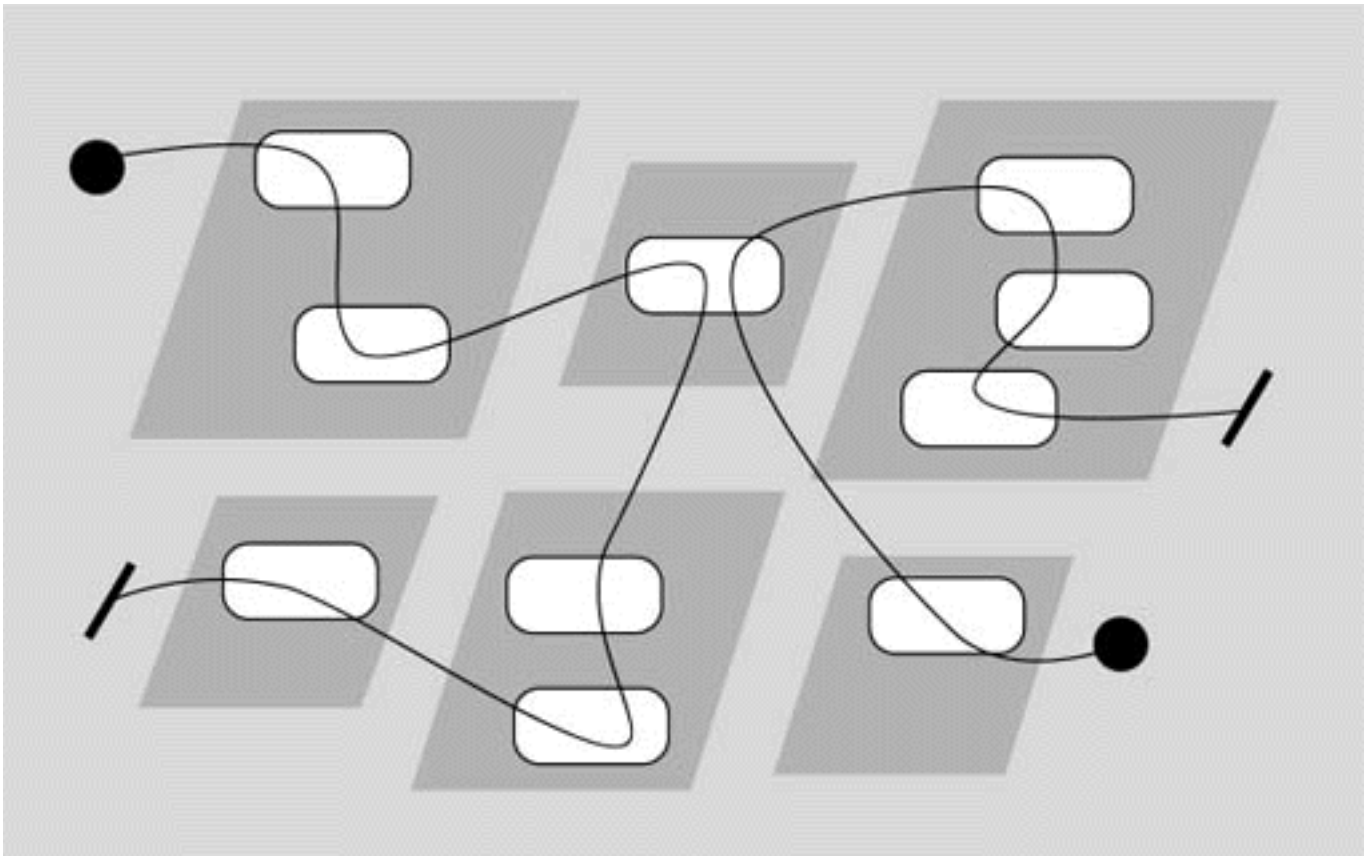
この方式では特定の責務単位にスレッドを割り当てます．ですからイベントを処理する一連のパスは複数のスレッドの協調処理で実現されることになります．この方法では，原則としてスレッド内では同期呼び出しを使用し，スレッド間ではメッセージキューなどを介した非同期呼び出しを使用します．他のスレッドへの要求はメッセージキューなどによりいったん直列化されるため，先の方式とは異なりリエントラントな呼び出しへの配慮は不要になります（ただし共有リソースを保護するための排他制御は必要です）．

この方法の場合，さらに責務の分け方によっていくつかのスレッドパターンが存在します．たとえば「マスタースレーブ」パターンではひとつの作業をいくつかのスレーブに分割して並行に作業させます．「パイプライン」パターンはひとつの作業に対し，各スレッドが決められた順番どおりに流れ作業的に処理を行う方法です．

一般的なマルチスレッド処理の場合は通常この方法がとられます（スレッドのパターンは最適なものが選択されます）が，スレッド間でのオブジェクト呼び出しを非同期通信に変換しなければならないこと，および非同期通信自体のコスト等を考慮しておく必要があります．この方法のイメージを図10に示します．

エレベータシステムでは，スレッド設計についてはこの方法を採用することにします．なお，これ以降の具体的な検討はこの後の設計フェーズで行います．

図10：ひとつのイベントを複数のスレッドで処理する



Prev. Index Next

オブジェクトの広場

［UMLとオブジェクト指向分析・設計が開発リスクを軽減する］

設計

　設計では，構造を中心としてとらえていた分析モデルを実装を考慮した制御的なモデルに洗練していきます．設計作業では，クラスではなく実際に生成されたオブジェクトどうしの活動に焦点を当てるため，モデル検討の際にも，静的な構造を表わすクラス図ではなく，オブジェクト単位の表現が可能なコラボレーション図やシーケンス図を多用します．
　設計作業はさらに，システム全体に対するトップダウン的視点からのシステム設計と，クラスやサブシステムを洗練していくオブジェクト設計のふたつのフェーズに分けられます．本稿では紙面の都合からシステム設計に絞って話を進めていきたいと思います．
　システム設計における具体的な作業および順序は以下のようになります．

・イベントを整理する
・イベントに対するコラボレーションを設計する
・全体のスレッド構成を決める
・アクティブオブジェクトを追加する
・共有されるオブジェクトについて検討する
・ハードウェアクラスの構造を検討する
・デザインパターンを適用する

(1)イベントを整理する
　組み込みシステムでは外部環境から非常に多くのイベントを受け取ります．ここでは，コンテキスト図やユースケース図の作成時に見つけ出されたイベントを整理し，クラス構造として表わします．このようにすることで，イベントの構造をわかりやすくするとともに，すべてのイベントをポリモフィックに扱える汎用的なイベント処理が可能になります．
　まず，すべてのイベントのルートとなる「イベント」クラスを定義し，そこから新たに「物理イベント」と「論理イベント」を派生させます．「イベント」クラスは属性としてすべてのイベントに必要な発生時刻や発生元の情報などを持ちます．「物理イベント」はハードウェアなどの割り込み処理やOSからの呼び出しの際に使用される外部環境からのイベントを表わします．「論理イベント」はサブシステム間での呼び出しなどのようなシステム内部で使用されるイベントを表わします．
　次に，外部イベントと内部イベントを上記にしたがって「物理イベント」と「論理イベント」に分類し，さらにそれぞれから派生した独立のクラスとして定義します．
　エレベータシステムの場合は，さらにイベントを目的に応じてパッケージ単位に分けます．エレベータパッケージでは，パッケージ外に公開するイベントを別のパッケージにまとめています．このような構成をとると，ハードウェアからの物理イベントのようなパッケージ内でのみ使用されるイベントは他のパッケージからは参照されないため，ハードウェア変更に伴うイベントの変更などがあった場合でも

変更の影響をエレベータパッケージ内に局所化することができます．エレベータシステムのイベントのクラス図を図11，図12に示します．

図11：エレベータ内部イベント

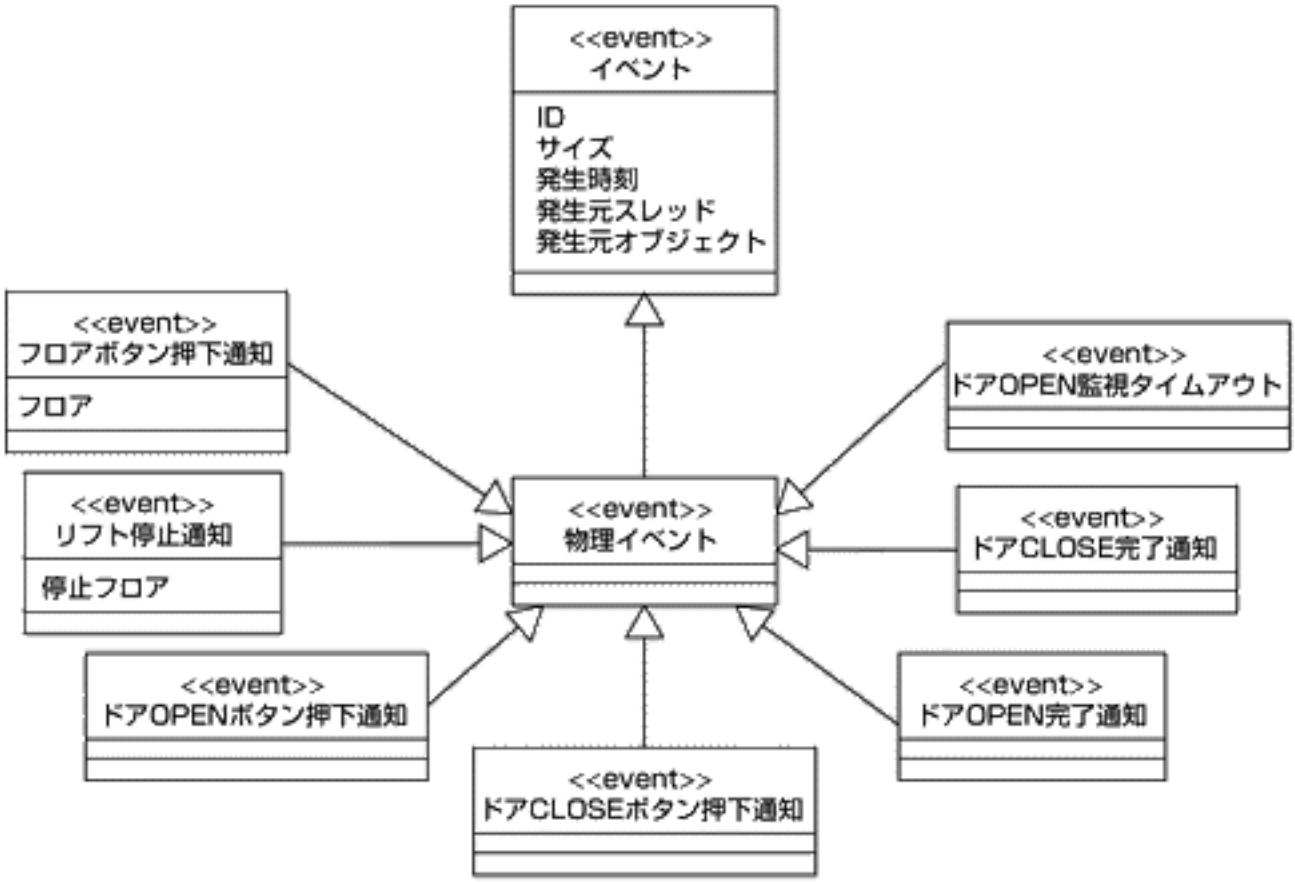
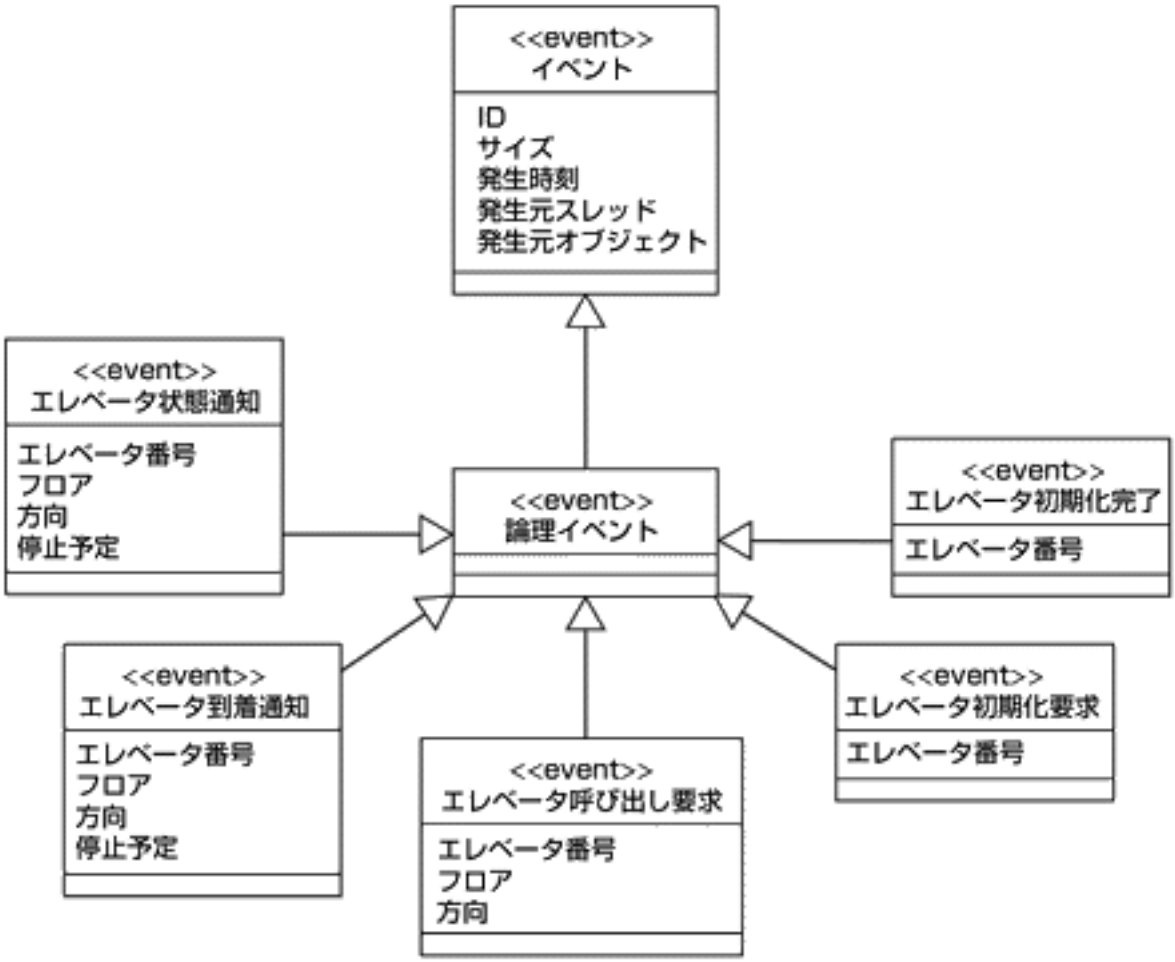


図12：エレベータ外部イベント



(2)イベントに対するコラボレーションを設計する
　イベントを受け取った後，サブシステム内部のオブジェクトがどのように相互作用しあって処理を行うのかをコラボレーション図で表わします．複数のサブシステムにまたがって処理が行われる場合は，先に分類した論理イベントを経由して他のサブシステムと繋がることになります．この時点では，どのオブジェクトがどのような順番で相互作用するのかに重点を置き，具体的なイベントの送受信方法までは検討しません．

(3)全体のスレッド構成を決める
　ここでは，スレッドの構成とそれに合わせたオブジェクトどうしのコラボレーションの洗練を行っていきます．

●ひとつのイベントをひとつのスレッドで処理した場合
　この方法をとる場合，外部イベントに対する一連の処理ごとにスレッドを割り当てることになります．検討にはこの前の作業で作成したコラボレーション図を使用します．
　まず，外部とのインターフェイスとなるオブジェクトに注目します．GUI，ネットワーク，ハードウェアラッパー，他のシステムとのインターフェイスなどがこれに該当します．外部イベント処理はこれらのオブジェクトを起点として処理されますので，最低でもこのインスタンス数分のスレッドを用意しなくてはなりません．
　次に，各イベントごとのコラボレーション図を使って，さらに細かなレベルでの検証を行います．その前に，今まで論理イベントとして表現していたサブシステム

間の呼び出しについては，すべてオブジェクトのメソッド呼び出しに変更しておきます．コラボレーション図の中に，以下のようなオブジェクトが含まれているかどうかを調べます．

・同時並行に動作させたいもの
・実行時間の長い処理を行うもの

もしこれらに該当するオブジェクトがあれば，それらは別スレッドを割り当てる候補になります．別スレッドにする場合は，それらのオブジェクトへの呼び出し部分をイベントを使用した非同期通信に変更し，そのイベントに対しては新しい論理イベントを割り当てます．

●ひとつのイベントを複数のスレッドで処理した場合

この方法の場合は，ある程度機能ごとに分割されているサブシステムを使用することができます．

まず，各サブシステムのインスタンスに対してスレッドをひとつずつ割り当て，これをシステム全体のスレッド構成のベースとします．

次に，これらのスレッドの必要性について検討します．各インスタンスが同時並行に動作するもの，実行時間の長いものなどは別々のスレッドでかまいませんが，順番に処理が行われても問題のないもの，実行時間の短いものなどは他のインスタンスとスレッドを共有できるかどうか検討を行います．スレッドの数が多いとリソースや効率面からマイナスとなるため，共有が可能であればそれらに割り当てたスレッドをマージしていきます．

反対に，前の方法と同じくサブシステム内で別スレッドを作る必要性についても検討を行います．アーキテクチャ設計の項で述べたようなスレッド構成のパターンを適用する場合などは，さらに詳細なスレッド構成を検討する必要があります．

今回のエレベータシステムでは，エレベータ，フロア，システム制御の各サブシステムはすべて同時並行に動作する必要があるためスレッド数はそれらのインスタンスの総計となります．さらに，エレベータサブシステムの場合はリクエストの受付とリフトの操作を別々に行う必要があるので，リフト部分には別スレッドを割り当てることになります．

(4)アクティブオブジェクトを追加する

導出した各スレッドごとに，スレッドクラスのオブジェクトをひとつとメッセージキュークラスのオブジェクトを必要な分だけ，それぞれ割り当てます．

スレッドクラスは自ら起動を行うアクティブクラスであり，OSごとに異なるスレッドの生成や削除に関する処理を行います．各スレッドはそれを所有する特定のメソッドを呼び出します．メッセージキュークラスは，スレッドと同様にOSに固有の仕組みを使ってメッセージキュー機能を実現します．

(5)共有されるオブジェクトについて検討する

(3)の作業で，いくつかのスレッドにまたがるオブジェクトやリソースが見つかることがあります．これらの共有されるオブジェクトに対しては，どのスレッドに割り当てるか，アクセス方法や排他制御はどのように行うか，などを検討しなければなりません．

共有されるオブジェクトへの呼び出しが同期通信の場合には，呼び出されるメソッドの前後で排他制御を行う必要があります．また，特に同じインスタンスを共有しなくても良いのであれば，別々のオブジェクトにしてそれぞれのスレッドに持たせてしまうことも可能です．ただし，その際はリソースとのトレードオフとなり

ます.

ここで，エレベータシステムに対し(2)から(5)までの作業を行った結果のクラス図およびコラボレーション図の一部を図13，図14に示します.

図13：エレベータパッケージ内クラス図

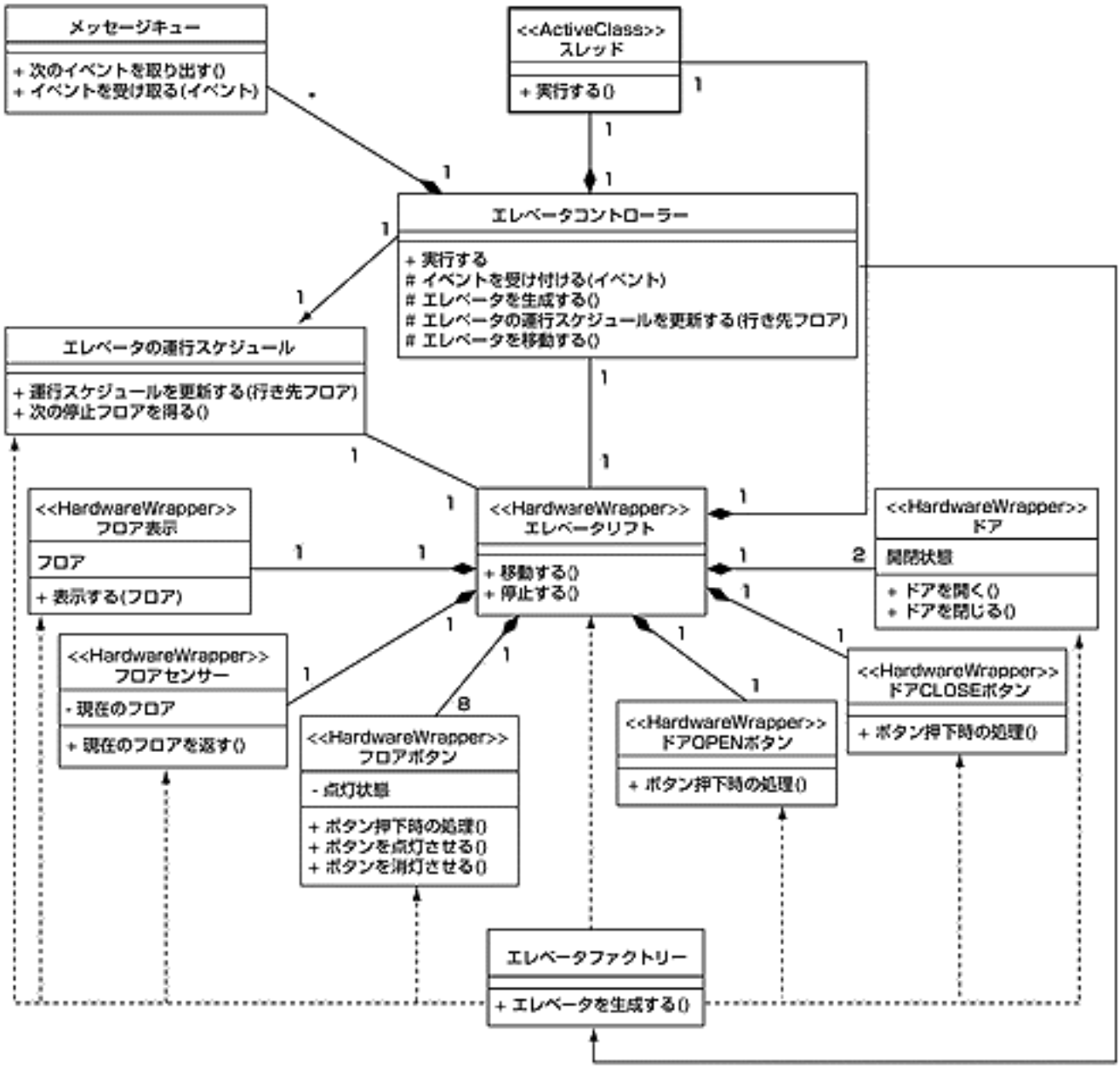
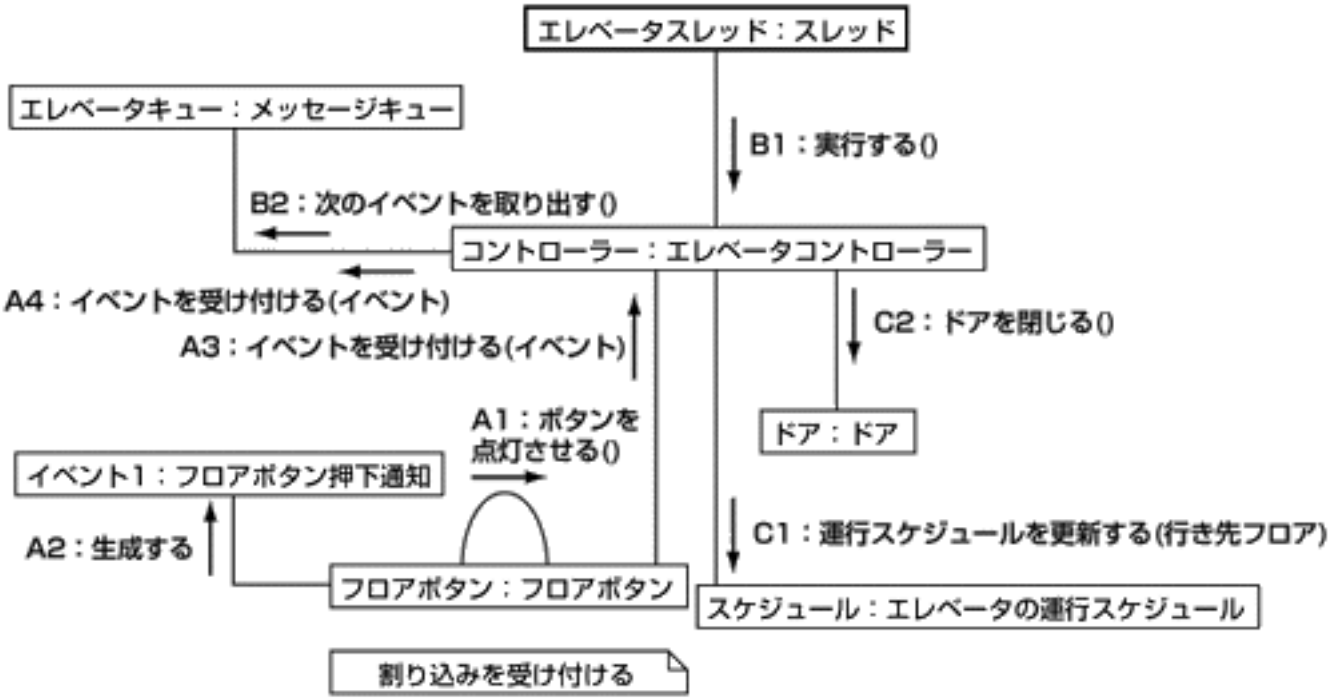


図14：フロア押下時のコラボレーション図



(6)ハードウェアラッパーの構造を検討する

ここでは，ハードウェアラッパーの実装方法について考えてみます．ハードウェアラッパーは外部に対し，ハードウェアの操作プロトコルや物理的な違いなどを隠蔽した抽象的なインターフェイスを提供する一方，内部ではそれを具体的なハードウェア操作に変換し実際の制御を行います．たとえばエレベータリフトクラスは，外部に対し目的のフロアを引数とし「移動する」という簡易な操作を提供しますが，内部ではフロアセンサーを使って現在のフロア位置を調べながら目的のフロアに到着するまでのモーター制御を行います．

このようにハードウェアのインターフェイスと実装を分離するとハードウェアの内部仕様が固まらないうちからソフトウェアの設計を進めることが可能になります．またスイッチやボタン，モーターなど汎用的に使用されるハードウェア部品についてはあらかじめそれらの制御操作を提供するハードウェアラッパークラスとして作成しておくことも可能です．

ハードウェアラッパーを実装する際には，物理的なハードウェアとの接点に注意する必要があります．実際のハードウェアの操作部分については効率や拡張性の面からもハードウェアラッパー内で実装するよりCやアセンブラで記述されたドライバなどを利用する方が望ましいでしょうし，イベントを発生するハードウェアラッパーの場合にはアセンブラなどで記述された割り込みハンドラからハードウェアラッパーオブジェクトを呼び出すための仕組みを考えておかなければなりません．

なお，分析では実際のハードウェア構成にあわせてクラスを導出しましたが，将来のハードウェアの変更や物理的な制約などに影響されないためには，ある程度抽象的なハードウェアとしてクラスを作成することも重要になります．

(7)デザインパターンを適用する

オブジェクト指向では特定の問題に対して有効なクラスの組み合わせを，パターンとして分類しています．パターンを使うことで，より確実で有効な設計を行うこ

とが可能になります．また，設計の意図もより伝わりやすくなります．パターンは非常に多くのものが紹介されています．ここでは，特に組み込みシステムで使われることの多いパターンについて簡単に紹介したいと思います．
　最初の7つは有名なGammaらによる『オブジェクト指向における再利用のためのデザインパターン』（★注1）から，最後の3つは『REAL-TIME　UML』（★注2）から紹介します．なお，これらのパターンの具体例やクラス図などについては今後弊社webなどで公開していく予定です．

●Abstract Factory
　システムのハードウェア構成がシリーズやバージョンごとに少しずつ異なるような場合，Abstract Factoryを使用することでハードウェアクラスのインスタンス生成に関わる制御を隠蔽します．

●Composite
　おもに，ハードウェアクラス同士の構造を表わすのに使用されます．

●Command
　スレッド間で受け渡されるイベントなどに使用されます．

●Facade
　複数のハードウェアクラスをまとめて，ひとつのハードウェアのように見せる場合などに使用されます．

●Strategy
　ハードウェアクラスの提供するサービスを複数の方法で実現したい場合などに使用されます．たとえば，センサーのもつ補正アルゴリズムを複数種類サポートしたい場合などは，アルゴリズムに該当する部分だけをStrategyクラスとして独立させます．

●State
　多くの状態遷移を持つクラスを，複数の状態クラスの集合で置き換えます．ただし，状態が極端に多い場合などは，その分クラス数が増えてしまうので管理が大変になります．

●Mediator
　ハードウェアクラス同士が直接イベントを送り合うと，ハードウェアの変更などによる仕様変更に対応できなかったり双方に多くの依存関係が発生することになります．これを解決するために，Mediatorとして振る舞うコントローラクラスを導入します．

●Transaction
　このパターンは，プロトコル処理から送達確認を行う部分だけTransactionクラスとして独立させることで，プロトコルを階層的に設計する事を可能にします．

●Rendezvous
　スレッド間で何らかの同期を行いたい場合に使用します．このパターンを使用すると，各OSごとに異なる同期の判別処理を局所化する事ができます．競合するリソースの管理などにも，このパターンを使用することができるでしょう．

●State Table
　これは，先に紹介したStateパターンに，遷移処理を行うためのTransitionクラスを

イベント単位で追加することにより，より細かなクラスの協調で状態遷移の設計を行えるようにしたパターンです．状態遷移の設計でよく使用される「状態」と「イベント」からなる状態遷移表の各ます目の処理を行う部分がTransactionオブジェクトになります．


Prev. Index Next

オブジェクトの広場

［UMLとオブジェクト指向分析・設計が開発リスクを軽減する］

開発環境について

最後に今まで述べてきたようなことを実現するための開発環境について少しだけ触れたいと思います．

まず，実装時の言語の問題ですが，最近は組み込み系のチップ向けのC++コンパイラ等も増えてきており効率やサイズ等の問題が解決されれば十分使用可能なレベルに向かっているといえます．

次にRTOSの問題があります．一般のRTOSを使用する場合，オブジェクトの生成や削除などで発生するメモリのアロケーション管理などがサポートされている必要があります．これについても，VxWorksや一部のμITRON環境などではすでにサポートされており，実際の開発現場でも使われ始めています．今後，環境の標準化とクラスライブラリの充実などが進めば，さらに使いやすい環境になっていくでしょう．

最後に

冒頭でも触れたように，組み込みシステムに対するオブジェクト指向開発はまだまだ進んでいないのが現状です．本稿を読むことで，組み込みシステムに携わるエンジニア方々のオブジェクト指向に対する疑問や敷居の高さを少しでも取り除くことができたなら幸いです．

■羽生田栄一
オージス総研オブジェクト第一事業部開発技術コンサルティング室室長．オブジェクト指向開発技法，モデリング技術の研究開発とそれらをベースにしたコンサルティングを行っている．近年は，UML，デザインパターン，ソフトウェアアーキテクチャ等の普及に努めている．

■渡辺博之
主に通信機器のファームウェアや通信アプリケーションの開発に携わっていたが，Windows 3.0の頃からオブジェクト指向に傾倒し始め，96年オージス総研に入社．最近は，開発コンサルティングの傍ら「オブジェクト」と「組み込み」の2語にどっぷりはまった毎日を送っている．

★注1…『オブジェクト指向における再利用のためのデザインパターン』（Erich Gamma，Richard Helm，Ralph Johnson，John Vlissides 著／本位田真一，吉田和樹 監訳／ソフトバンク）

★注2…『REAL-TIME UML』（Bruce Powel Douglass 著／Addison-Wesley）