伊藤忠テクノソリューションズ㈱　小麦田 哲也　KOMUGIDA Tetsuya
インダストリアル・エンジニアリング営業部に所属．医療機器，コンシューマー製品の組込み系開発経験を経て，MDDの最先端の状況を把握・習得するために2004年より，Telelogic製品の販売支援・サポート業務・ユーザの開発支援を行っている．

伊藤忠テクノソリューションズ㈱　徳江 愛　TOKUE Ai
インダストリアル・エンジニアリング営業部に所属．1997年伊藤忠テクノサイエンス㈱（現 伊藤忠テクノソリューションズ）入社．2000年より，Telelogic製品の販売・サポートに従事し，国内のユーザに対して開発支援を行っている．

監修：HASHIMOTO SOFTWARE CONSULTING Inc.　橋本 隆成　HASHIMOTO Takanari
SEI認定CMMIインストラクター．護衛艦の艦船搭載用弾道計算プログラム，新型戦闘指揮システム，新型射撃制御システムなどの大規模ハードリアルタイムシステムのソフトウェア開発に10年間従事．以後，日本ヒューレット・パッカード，オージス総研，ソニーにて製品開発業務，開発手法・方法論のコンサルティングおよびCMMIによるプロセス改善業務に従事．

オブジェクト指向によるMDD（Model Driven Development；モデル駆動開発）では、開発および管理プロセスが重要になってきます。近年は開発・管理プロセスが非常に重要視されており、効果的な開発を期待するには、適切な開発・管理プロセスを用いることが重要です。3回シリーズの最終回となる今回は、「コード生成」と「シミュレーション」作業に的を絞って解説を行います。

MDD

3 トリロジー企画

オブジェクト指向技術による モデル駆動開発 実践!

シーズンIII～コード生成編

前回までの内容と今回の記事内容

今回のポイント

前回（Vol.5）、前々回（Vol.7）と進めてきました「エレベータ制御システム」のオブジェクト指向およびMDD（Model Driven Development）による組込みソフトウェア開発のケーススタディも今回で最終回となります。

今回は前回解説した分析作業の結果からアーキテクチャ設計、詳細設計を実施し、その作業手順と成果物を解説していきます。アーキテクチャ設計について具体的な成果物をお見せしながら、特に組込みシステムで重要な非機能的な要件への対応も詳細に解説します。たとえばパフォーマンス要求、保守性・拡張性などへの対応をどのように設計作業を通じて実現するのか、その効果的な方法を紹介します。また、開発環境である「Rhapsody」によるUMLのモデルからMDDアプローチにより実行コードを生成し、動作するまでの一連の作業も今回の解説の大きなポイントとなります。

今回の解説の流れ

最終回の今回は「エレベータ制御システム」のアーキテクチャ設計、詳細設計、MDAによるシュミュレーション、および実際に「エレベータ制御システム」をアニメーションとして動作させた様子を掲載し解説していきます。解説のポイントはRhapsodyによるMDAアプローチの具体的な設計から動作までを紹介している点です。

設計フェーズでは作業も多く、読者のみなさんに設計手法、テクニックなどいろいろなことを説明したいのですが、誌面には限りがあります。本企画は組込みシステムのケーススタディを用いて、RhapsodyとMDDによるオブジェクト指向開発を進め、最後にコード生成、シュミュレーションまでを解説することを第一の目標とします。実際に「エレベータ制御システム」のようなケーススタディを要求定義から自動ソースコード生成と動作検証までを、詳細な作業過程と成果物を紹介しながら、解説付きでお見せすることは雑誌やセミナーでも初めての試みであると思います。

また、みなさんにとっては初めて目にする内容も多いと思いますので、内容が多岐に渡り過ぎ理解が発散しないように、第3回目の今回は、RhapsodyとMDDによるオブジェクト指向開発により、最後にコード生成、シュミュレーションまでを解説することに焦点を置くことにします。

そこで、今回は解説の中で十分に触れられないソフトウェアエンジニアリングポイントをChapter1にまとめることにしました。そして、Chapter2以降ではRhapsodyによる詳細設計、コード生成、シュミュレーションまでを解説していきます。

MDD Chapter 1

ソフトウェアの凝集度と結合度

要件変更を柔軟に受け入れられるアーキテクチャ

はじめに

アーキテクチャの設計を実施するうえで保守性、堅牢性に強く影響を与える「凝集度」と「結合度」について説明しておきます。

凝集度とは

凝集度とは、クラスやパッケージ内の機能要素と情報要素間の関連性の強さを表す指標です。互いに関連する機能や情報があちこちに分散していると、管理が複雑になり、仕様変更が生じた場合の影響範囲が広くなってしまいます。これらの関連の強い機能や情報は局所化されて、なるべく同じモジュール[注1]に収まっていることが望ましいわけです。そのため、基本的には凝集度は高いほど良い設計といえます。

結合度とは

もう1つの結合度とは、クラスやパッケージ間で、呼び出し関係にあるメソッドの結びつきの強さを表す指標です。別の言い方をすれば「依存性」です。結合度が高くなると複数のクラスやパッケージ間で依存度が上がってしまうため、1つのクラスやサブシステムに変更を行う場合に、依存し関連のあるほかのクラスやサブシステムに変更が必要な場合が多くなってきます。つまり、保守やメンテナンス、仕様変更などの対応がしづらくなります。また、設計的な観点からしても、複雑でわかりづらいものになります。結合度を意識することにより、保守性やプログラム構造および設計の理解のしやすさを改善できます。既存のシステムに対しては、結合度が高くなっているクラスやパッケージ間に着目し、結合度を低くするように調整を行えると、結果的に保守性や設計の改善を行うことができるでしょう。

表1と表2はこの「凝集度」と「結合度」についてマイヤーズという人が示した有名な指針が存在します。ここではこの指針について解説していきます。

◇ ◇ ◇

2つの指針の結論を整理すると、凝縮度（モジュール強度）はできるだけ強く、かつ各モジュールの間の相互の結合度はできるだけ弱くすることが基本です。つまるところ、優れたソフトウェアアーキテクチャとは、保守性・拡張性に優れたソフトウェアということになります。

なお、表1と表2のマイヤーズの指針で登場する「モジュール」は、プログラムを意味あるまとまりでまとめる構成単位という意味で用いており、使用するプログラム言語のサポートするメカニズムに適切に置き換えて考える必要があります。たとえば、モジュールは、C言語ではファンクションやファイルスコープを用いたモジュール化、C++な

注1●ここでのモジュールは、クラス、サブシステム、コンポーネントなどコンテキストに応じて読み替えてください。

■表1■ マイヤーズの凝縮度の評価基準

凝集度	名称	特徴
最も強い	機能的結合（機能凝集）	1つのモジュールは1つの機能だけを実行する
強い	情報的結合（連続凝集）	1つのモジュールに複数の機能がまとまっているが、単独に呼び出せる
普通	連絡的結合（連携凝集） 手順的結合（手続凝集） 時間的結合（仮凝集）	データによって複数の機能が順次実行される複数の機能を順次実行するモジュール ある時期にまとめて行うべき処理を一括したモジュール
弱い	論理的結合（論理凝集）	1つのモジュールでいくつもの処理が実行できる
最も弱い	暗合的結合（共存凝集）	1つの機能が単純に分割されたモジュール

■表2■ マイヤーズの結合度の評価基準

結合度	名称	特徴
最も疎結合	データ結合	抽象データ型に代表される、ほかのモジュール用に提供されたインタフェースとなるファンクション プロシージャなどにデータを引渡し（引き数）で処理を委譲する方法。相手の内部のアルゴリズムの 理解は不要であり、また、ブラックボックス化されており、情報の隠蔽（いんぺい）が実現されている
疎結合	スタンプ結合	相互にモジュール内のデータだけを引き数として呼び出す
中間的結合	制御結合	処理用のデータとともに制御用のデータのやりとりを必要とする関係 相互の結合度（つまり相手のモジュールの構造やデータに依存がある）がそれだけ強くなる
密結合	外部結合	お互いのモジュールの外部にとられているデータ（大域変数）あるいは、それぞれのモジュールが共通 にアクセスする変数を介してデータをやり取りする。モジュールとしての独立性は極端に少なくなる
最も密結合	内容結合	ほかのモジュールの内部構造に直接参照して処理する

らクラス、ファンクション、ファイルスコープなどのメカニズムです。Pascal、Ada言語では同様にサブルーチンであるプロシージャとファンクションおよびカプセル化を実現するパッケージ機能などに該当します。モジュールは、プログラムの構成単位ということですので、プログラム言語の持つメカニズムを効果的に用いて階層的な考え方を適用できます。

このような「凝集度」と「結合度」をプログラミングの際に初めて検討することは薦められません。事前に「凝集度」と「結合度」を考慮のうえ、アーキテクチャ設計と詳細設計の段階で十分に凝縮度と結合度を考慮しておくべきです。。

また、このマイヤーズのガイドは「機能に基づくソフトウェアの構造化手法開発」でも、「オブジェクト指向開発」でも適用できるものです。逆に言えば、このガイドの意味を理解すれば、特定のプログラム言語、設計手法に依存していないため、プログラム言語や手法自体はなんであろうと関係なく正しい視点を手に入れられます。

多くの研究者や手法・方法論が述べている効果的なパターンや設計上のアプローチおよび原則やガイドラインも本質はこの「凝集度」と「結合度」をより発展、具体化させたものになります。

変更性の考慮

要求仕様の変更や設計上の都合でアーキテクチャに修正が必要な場合が多々あります。しかし、事前にある程度変更部分を特定し、局所化できます。このように要求仕様の変更や設計上の都合で変更がかかることを「可変性」、そしてその変更部分を特定し局所化することを「可変性の特定」と呼ばれることがあります。実のところ、現在のソフトウェア開発において可変性の局所化と可変性を局所化したアーキテクチャ設計は不可欠です。というよりも戦略的に可変性を「決め打ち」するといってよいでしょう。

これまで発表されている多くのソフトウェアの開発プロセスでは「要件変更は柔軟に受け入れよ」と説いています。もちろんこれは「要件変更を無条件に受け入れる」ということとは異なります。顧客満足度、製品の市場差別化の源泉になる要件を可能な限り、ギリギリまで変更可能にできるような開発体制、プロセスを持つことです。しかし、近年、製造業のシステム開発ではまったく別のアプローチが取られます。特にミッションクリティカルな性質を持ち、グレードの違いや国内や海外

向けに対応して外で微妙に仕様の異なる多彩なバリエーションを開発する企業では「要件変更は柔軟に受け入れよ」を各開発プロジェクトで実施するのはリスクが高すぎるうえに、ソフトウェアの開発戦略、マーケティング戦略、開発コストから見たROI（Return on Investment；投資利益率（投下資本利益率、投下資本収益率、費用対効果））からも得策ではありません。

また、可変性の制御を外部の要因に大きく依存する「受身の変更」という性質があります。

そこで、事前の製品戦略時に市場のニーズや自社の製品ラインナップの検討を実施する際に、事前に可変性を特定し、戦略的にアーキテクチャの開発を行います。これは、予想される製品仕様やユーザのニーズ変化にあらかじめ、仕様策定、製品のマーケティング、組織資産管理など可変性などへの戦略的な対応を、企業間競争を優位にする源泉の1つにしようとするものです。これを「プロダクトライン戦略」と呼びます。

この戦略的に可変性を盛り込んで開発するには、事前に十分な市場調査による消費者のニーズ、今後の技術動向、競合他社の動きなどを十分に検討しておく必要があります。けっして、技術の視点だけで可変性を決定していくことはしません。

さて、このようなことから可変性の特定と、可変性を局所化したアーキテクチャの設計が重要になることが理解できると思いますが、とくにソフトウェアフレームワークやソフトウェアコンポーネントを効果的に開発組織の資産として「再利用」する際に重要になってきます。

組込みシステムの「凝集度」と「結合度」制御と可変性の局所化の基本

ここからは、「凝集度」と「結合度」、および「可変性」についてどのようにアーキテクチャ設計を実施し、優れたソフトウエアアーキテクチャを実現していくかを見ていきましょう。

まず確認しておくことは、アーキテクチャ設計の結果の良し悪しについてです。アーキテクチャ構成と最終的な設計は製品に求められている仕様や開発者の設計意図により異なりますので、画一的に良い設計、悪い設計と決めつけられない部分があります。

ただ、どのような製品の「アーキテクチャ設計」を行う場合でも「凝集度」「結合度」「可変性」について述べた基本的な原則が存在します。これを踏まえて、組込みシステムで基本となるアーキテクチャ設計の基礎を少し具体的に解説していきます。

今回の連載のようなオブジェクト指向開発の場合、アーキテクチャの構成要素の基本はレイヤ、サブシステム、クラスです。アーキテクチャをこのように3つの粒度の視点で検討するのはなぜでしょうか？ これには多くの理由とメリットがあります。アーキテクチャをレイヤやサブシステムで構成することで、まず、システムの可読性が良くなります。

レイヤ間、サブシステム間は互いの内部情報について知らないようにします。つまり、レイヤ間およびサブシステム間には、インタフェースとなる通信のしくみを設け、そのインタフェース以外の詳細な内容はカプセル化することで、レイヤ間、サブシステム内部のクラスとクラス間の関連などはサブシステム内に隠蔽（いんぺい）されるため、利用側はそのインタフェースのみに依存することになります。

設計者（または設計チーム）は、インタフェースを実現する方法を特定する必要はありますが、外部依存性に影響を与えることなく内部のサブシステム設計を自由に変更できます。独立性の高いチームを持つ大規模なシステムでは、大きな効果となるのです。

各レイヤのサブシステムは、原則、下位のレイヤのサブシステムが提供するインタフェースを利用するように注意します。レイヤ間のインタフェース設計は特に保守性、可読性に大きな影響をあたえ、アーキテクチャ全体の堅牢性の基礎となります。

組込みシステムのレイヤ分けの基本は、各階層

は物理的なものから抽象度の高いものへ階層化への分離です（表3）。抽象度に着目した階層化の例としてTCP/IPがあり、4つのレイヤで構成されています。レイヤに分離することで各レイヤの機能が明確化され、保守性、可読性が生まれています。このようにアーキテクチャを検討するときにマイヤーズの指針を利用してレイヤ、サブシステム、クラス間の結合度、凝縮度に注意しながら作業を実施することで、優れたアーキテクチャを実現することが可能になってきます。

時間制約による並行性に関するアーキテクチャ

組込みシステムではさらに実時間を満たすというが非常に重要な課題が存在します。

実時間を満たすということを分析・設計するには、アーキテクチャの動的な構造にも視点を向けなければなりません。これは、具体的に言えば、組込み分散システムであれば、マルチCPU構成や各CPU上で動作させる際のタスク（プロセス、スレッド）構成とタスク間通信を検討する作業です（図1）。

タスク（プロセス、スレッド）の分割は、システムが与えられている機能の処理を、実時間を満たすということの視点からアーキテクチャの構成を検討する作業です。この視点での分割はタスク（スレッド、プロセスなど）であり、「凝集度」と「結合度」や「可変性」の考慮と、抽象度を利用したレイヤ、サブシステム、クラスによる視点のアーキテクチャ設計とはまったく異なる（直交する）視点です。

マイヤーズの示す「凝集度」と「結合度」の評価基準が存在したように、並行性の分析、設計にも開発手法のDARTSで有名なハッサンゴマ博士のタスク分割・統合と優先度付与の指針が存在します。重要な点はタスクの抽出をこの時点で行うのには、分析のところで「イベント分析」をしっかり実施しておかなければならないということです。イベント分析の結果をもとにタスク分割のガイドに従い、タスク分割とタスク間通信およびタスクの優先度を決定する作業を実施します（図2）。

なお、タスクは可能な限り少なくしたほうが、優先度の決定などの作業を複雑にしなくて済みます。優先度逆転などの厄介な問題を避けるためにもシンプルな構成が基本です。また、システムの拡張作業の際もタスク数が多く、タスク間の関係が複雑だとメンテナンス性に大きな問題を残すことになります。組

■表3■ 組込みシステムでよく見られるアーキテクチャの階層（レイヤ）化

レイヤ	説明
GUIレイヤ	ユーザインタフェース
アプリケーションレイヤ	ドメインに特徴的な機能の部分
ビジネスレイヤ（別名ドメインレイヤ）	ドメインに共通な機能
ハードウェアラッパー層	物理的なハード仕様や操作手順を隠蔽する
ハードウェアBIOS層	通常アセンブリ言語、C言語で記述される部分
ハードウェア（物理）層	ハードウェアその物であり、ソフトウェア部分でない。アーキテクチャを表現するために便宜的に表現することが多い

■図1■ 並行性のアーキテクチャ設計の作業内容

①CPUスペック、CPU数、CPU間構成、バス構成
　・各物理ノード上に配置するソフトウェアコンポーネントの決定
②並行性／並列性の基本戦略
　・各CPUで動作させるタスクの決定
　・タスク抽出の戦略決定
　・イベントの送受信方法設計
　・タスク間通信設計
③例外処理設計
④分散性、永続性の実現方法や製品の決定

■図2■ タスク分割の指針

①オペレータと逐次的なやり取りを行う機能／変換は1つのタスクにする
②ほかのタスクが利用しない資源を使う、ほかのタスクの実行結果に依存しない
③具体的にはI/Oの処理はI/O専用のタスクにする。I/O処理以外の処理も同一タスクに含めてしまうと、タスクが長時間の待ちでブロックされる
④周期性がある割り込み（外部イベント）、周期性がない割り込み（外部イベント）でタスク抽出を検討する
⑤長いCPU時間を使う処理は別タスクにする

MDD Chapter 2

アーキテクチャ設計

静的構造と動的構造

静的構造のアーキテクチャ構成

設計段階では、分析モデルをもとに、実装につなげるためにモデルを詳細化していく作業を行います。設計モデルを作成するうえで、ベースとなるものの1つがシーケンス図です。分析段階では、ユースケースシナリオを表現したシーケンス図を作成しましたが、設計段階では、オブジェクト間で実際に処理される操作やその引数などを考慮しながら、詳細なシーケンス図を表現します。このシーケンス図上のオブジェクトは、分析段階で識別したクラスをもとに表記します。

このとき、アーキテクチャ分析で把握したパッケージ間の依存関係がなるべく双方向にならず、依存関係が循環しないように、またレイヤの上下関係を考慮しながらオブジェクト間のメッセージの向きを決め、シーケンス図を記述していきます。なぜなら、パッケージの依存関係は、“パッケージの内容が変更されたときに影響する可能性のある部分”を表しているので、双方向の依存関係がある、あるいは複数のパッケージ間が循環してしまうケースでは、その依存関係に含まれるパッケージへの変更が、どこまでほかのパッケージに影響するかが判断しづらくなってしまうためです。依存関係が双方向にならないようにするために、次のような方法があります。

① シーケンス図から、依存関係を双方向にする原因となるメッセージを見直す（それが状態変化を通知するメッセージであれば、ポーリングで状態を確認させることでメッセージの方向を逆にする）
② アーキテクチャを見直す（複数のパッケージのクラス構成を変更するか、パッケージを分割する）

検討材料

詳細なシーケンス図では、メッセージとして記述されるクラスの操作に対して、引数や戻り値も決めていきます。メッセージの向きは、それが指し示すオブジェクトの操作を呼び出すということです。あるオブジェクトの操作を呼ぶときには、そのオブジェクトが把握しているデータ（そのオブジェクトの属性、もしくは、そのオブジェクトが持つリンクから取得できるデータ）以外は、呼び出し側のオブジェクトが渡さなければならないため、その操作の引数の候補となります。つまり、どのオブジェクトがどのようなデータを把握しているかが、操作の引数とその型を決めるポイントになります。また、呼び出し側のオブジェクトが操作を呼ぶときに、“処理を依頼する”だけなのか、“計算した結果（値）がほしい”のかによって、戻り値の有無と型を検討します。

シーケンス図を作成しながら各クラスが十分に単一の責務だけを受け持っているかを確認します。複数の責務を持つクラスの特徴は、シーケンス図

でほかのオブジェクトからのメッセージが集中するようなクラスのオブジェクトであるという傾向があります。つまり、そのクラスに対する関連が多くなり、クラス間の結合度が高くなってしまう可能性があります。逆に、クラスに適切に1つの責務だけが与えられたケースとは、そのクラスに対する関連が必要最小限であり、クラス間の結合度が低いケースです。シーケンス図で1つのクラスにメッセージが集中してしまうケースでは、あるクラスが単一の責務でなく、本来は別クラスにすべき別の責務も持っているのではないかと疑ってみるとよいでしょう。

単一の責務だけをクラスに持たせるためには、クラスの属性の主キーに該当する属性とほかの属性の間に推移的な従属関係が存在していないかをチェックし（つまり本来は主キーに該当する属性にほかの属性が全従属であるべき）、データベースのER図（Entity-Relationship Diagram）で活用するような第3正規形になっているかを意識することで確認する方法もあります。

もし1つのクラスに異なる責務が混在している場合は望ましくないので、このような場合は、必要に応じてクラスを適切に複数のクラスに分けて責務を分担するようにします。つまり、第3正規形になっていない場合はクラスを第3正規形になるように分割するというものです。

エレベータシステムの例では、分析の段階では、キャビンクラスはドアの開閉制御、キャビン内のボタン上下ボタンの制御、インジケータへの表示を担当しており、これらの責務を実現するために、キャビンクラス自身の操作や関連するクラスの操作を呼び、処理を進める必要がありました。その結果、分析時のシーケンス図では、ほかのクラスに比べてキャビンクラスにメッセージが集中してしまっています。そこで、設計段階では、UpDownButtonクラスの管理はFloorクラスが担当するようにし、インジケータへの表示に関する処理をキャビンの位置を計算できるSpeedControllerクラスが担当するようにしています。

この段階では、何度もシーケンス図とアーキテクチャ図を交互に見直しながら、作業を進めていきます。Rhapsodyのシーケンス図エディタには、分析モードと設計モード[注1]という2つのモードが用意されています。ここでの作業のように、シーケンス図の内容を繰り返し修正するようなケースでは、まず分析モードで図を表記し（この段階では、対応するクラスに操作情報は登録されません）、十分洗練された段階で、登録されていないメッセージ（＝操作）をモデル情報に追加できます。

設計レベルのシーケンス図

図1と図2は、分析段階のシーケンス図（シーズンII Chapter1の図4［Vol.7のp.135］）を設計レベルのシーケンス図に落としたものです。ユースケースシナリオをベースとしているため、前回は1枚のシーケンス図で表現しましたが、ここで表現している“押された上下ボタンを取得し、点灯させ、取得したボタン情報をもとにエレベータの運行をスケジュールする”というシナリオの中には、“配車をスケジュールする”部分と、“キャビンを特定のフロアに移動させる”部分があり、この2つは並行で考える必要があります。上下ボタンやフロアボタンからの配車要求を処理する運行スケジュールの管理と、運行スケジュールに登録されているとおりに各階に停車させるキャビンの移動の管理は独立して処理される必要があるためです。

図1では、Floorオブジェクトが追加されており、このシーケンス図の中では上下ボタンを点灯させ、UpDownButtonDevice（上下ボタン）からのイベントを、DispatchController（配車コントローラ）に渡す処理を行っています。今回の例では、各階層行きのキャビンは1基ずつのケースをシミュレー

注1●設計モードでは、メッセージやインスタンスラインを記述すると、それに対応する要素がモデル情報に登録されます。

図1 配車スケジュール

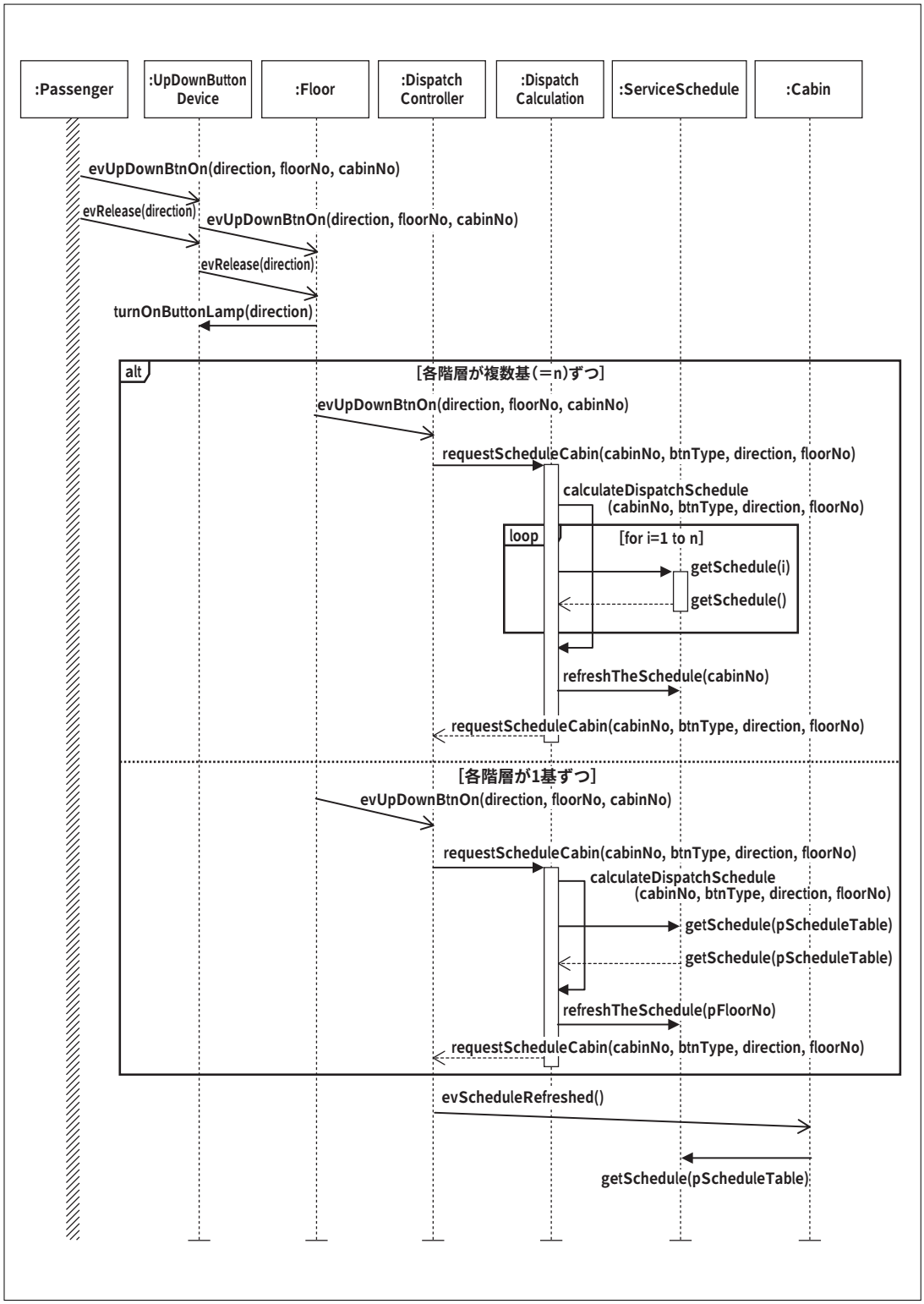
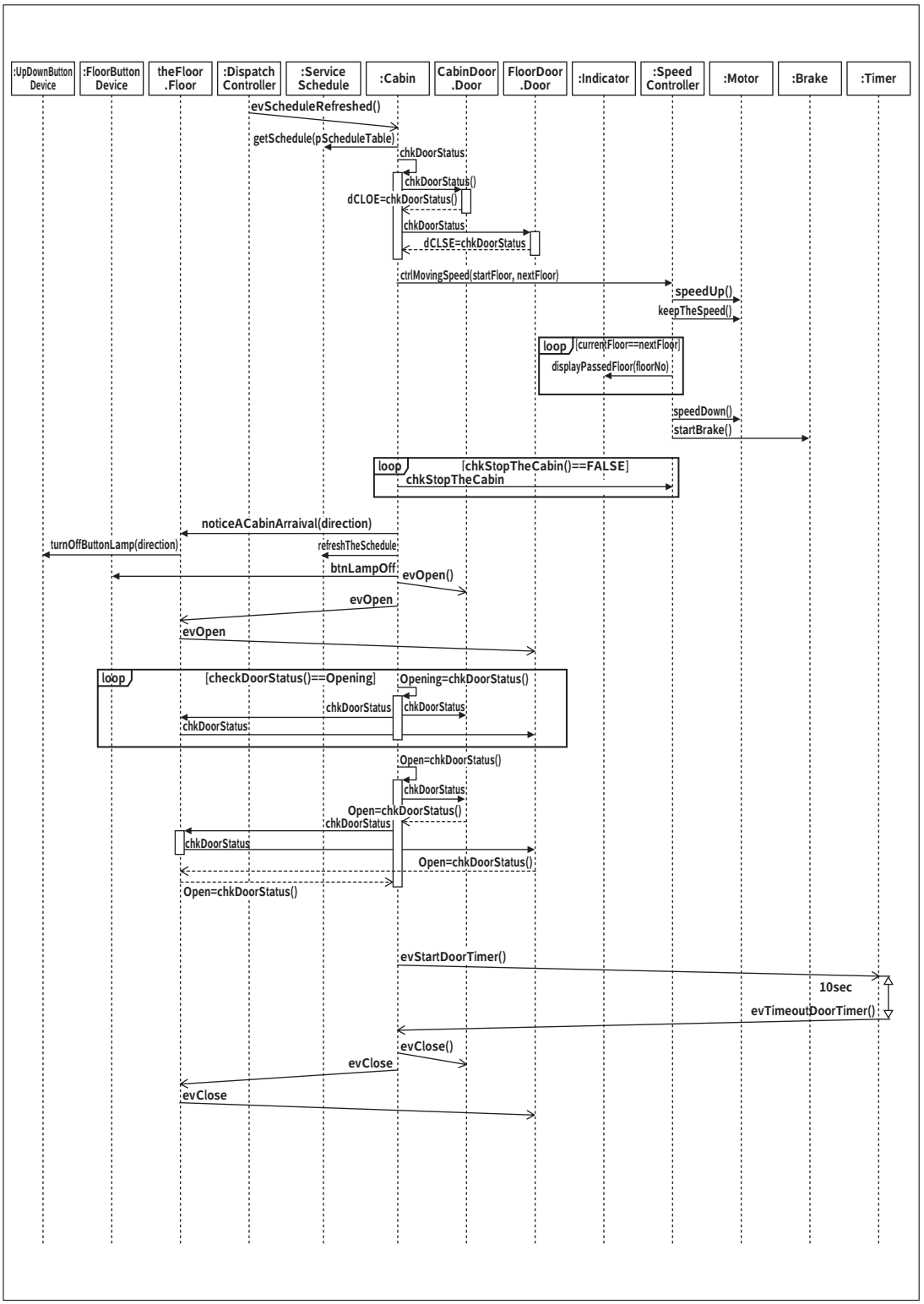

:Passenger
:UpDownButton Device
:Floor
:Dispatch Controller
:Dispatch Calculation
:ServiceSchedule
:Cabin
evUpDownBtnOn(direction, floorNo, cabinNo)
evRelease(direction)
evUpDownBtnOn(direction, floorNo, cabinNo)
evRelease(direction)
turnOnButtonLamp(direction)
alt
[各階層が複数基(＝n)ずつ]
evUpDownBtnOn(direction, floorNo, cabinNo)
requestScheduleCabin(cabinNo, btnType, direction, floorNo)
calculateDispatchSchedule
(cabinNo, btnType, direction, floorNo)
loop
[for i=1 to n]
getSchedule(i)
getSchedule()
refreshTheSchedule(cabinNo)
requestScheduleCabin(cabinNo, btnType, direction, floorNo)
[各階層が1基ずつ]
evUpDownBtnOn(direction, floorNo, cabinNo)
requestScheduleCabin(cabinNo, btnType, direction, floorNo)
calculateDispatchSchedule
(cabinNo, btnType, direction, floorNo)
getSchedule(pScheduleTable)
getSchedule(pScheduleTable)
refreshTheSchedule(pFloorNo)
requestScheduleCabin(cabinNo, btnType, direction, floorNo)
evScheduleRefreshed()
getSchedule(pScheduleTable)

図2 要求されたフロアにキャビンを移動させる



ションしますが、各階層行きのキャビンが複数あるケースを考えると、同じエリアの上下ボタンは連動して点灯するようにしなければなりません。そのため、これを処理させるFloorクラスを用意しています。

図2では、CabinDoorオブジェクトとFloorDoorオブジェクトが追加されています。2つともDoorクラスのオブジェクトですが、CabinDoorはキャビンに付属しているドアで、FloorDoorはフロアに付属しているドアです。Cabinクラスに処理が集中しないように、Cabinオブジェクトが、各フロアのFloorDoorオブジェクトと直接リンクするのではなく、Floorオブジェクトが FloorDoorオブジェクトをコントロールするようにしています。また、

図3 設計レベルのアーキテクチャ図

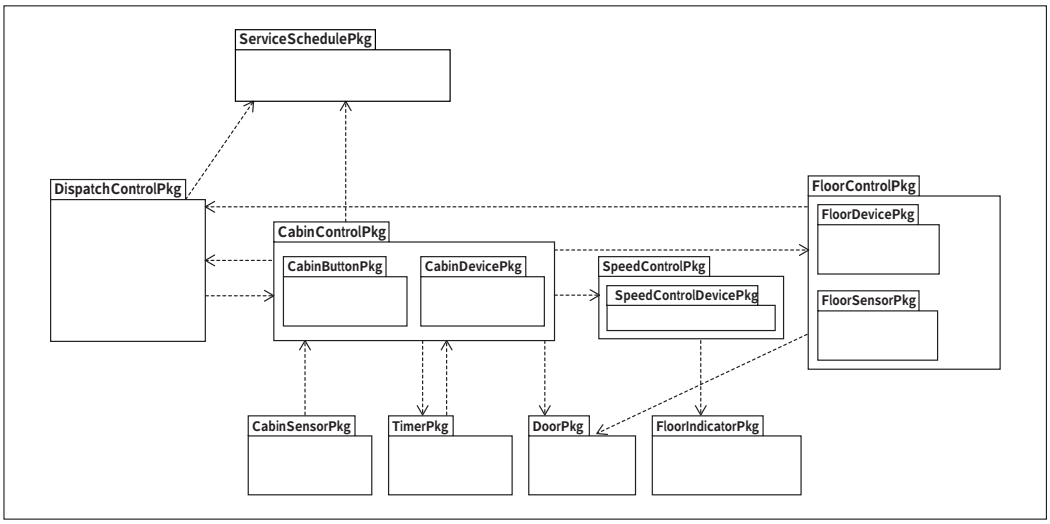
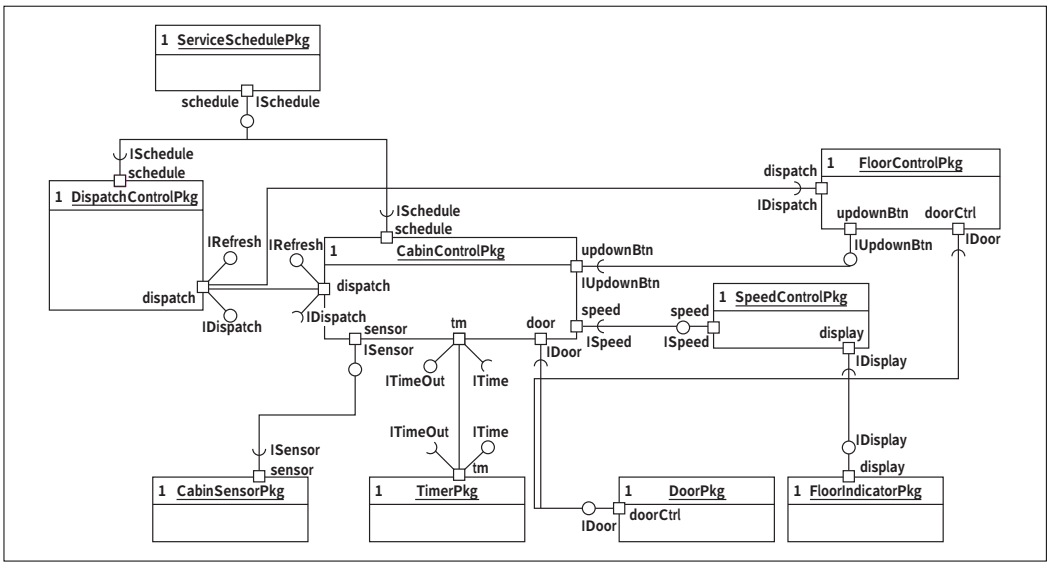


図4 静的なアーキテクチャ図



注2●Rhapsodyでは、同期メッセージを水平アロー、非同期メッセージを斜めのメッセージで表現するようになっています。

インジケータへの表示についても、キャビンの移動を制御しているSpeedController（速度コントローラ）が、移動距離をもとにインジケータに表示する情報（キャビンの現在位置と方向）を計算によって求め、Indicatorに対してフロア番号を渡すようにすることでCabinControlPkgの依存関係を減らしています。

各シーケンス図上で、タスクをまたがるメッセージをイベント（非同期メッセージ）として表現しています[注2]。どのオブジェクトをどのタスクにマッピングするかについては、後述のタスクマッピング図の中で説明します。

静的なアーキテクチャ設計

こうして、複数のシーケンス図を洗練し、オブジェクト間のメッセージの方向によって、パッケージの構成と依存関係を整理します（図3、図4）。

分析段階では、CabinControlPkg（キャビンコントロールパッケージ）が多くの依存関係を持っていましたが、保守性・再利用性を高めるために、これらの依存関係を整理しています。別パッケージとして分類され、依存関係が必要であった、CabinButtonPkg（OpenButton、CloseButton、FloorButtonDeviceクラスを含む）とCabinDevicePkg（Buzzer、Lightクラスを含む）は、CabinControlPkg以外のパッケージと関係を持たないことから、CabinControlPkgに含められます。また、前述のシーケンス図（図1、図2）を作成していく中で、UpDownButtonDeviceオブジェクトとFloorDoorオブジェクトをコントロールするというFloorクラスの責務が明確になっています。Doorクラスを含むDoorPkgは、CabinControlPkgにも関係するためパッケージとして残し、FlooDevicePkg（UpDownButtonDevicefを含む）はFloorPkg（フロアコントロールパッケージ）に含めます。同様に、SpeedControlPkg（速度コントロールパッケージ）に、SpeedControlDevicePkg（Brake、Motorクラスを含む）を含められます。

並行性のアーキテクチャ構成

コミュニケーション図によるオブジェクトの抽出

ここでのコミュニケーション図の作成目的は、シーケンス図で表される詳細なイベント（メッセージ）のやり取りや操作を改めて作成するではありません。この時点でコミュニケーション図を利用する作成目的は、あらかじめイベント分析表で明確にしておいたイベントによって引き起こされる（トリガとなる）システムが行う処理の流れの中で関連するオブジェクトを把握することです。

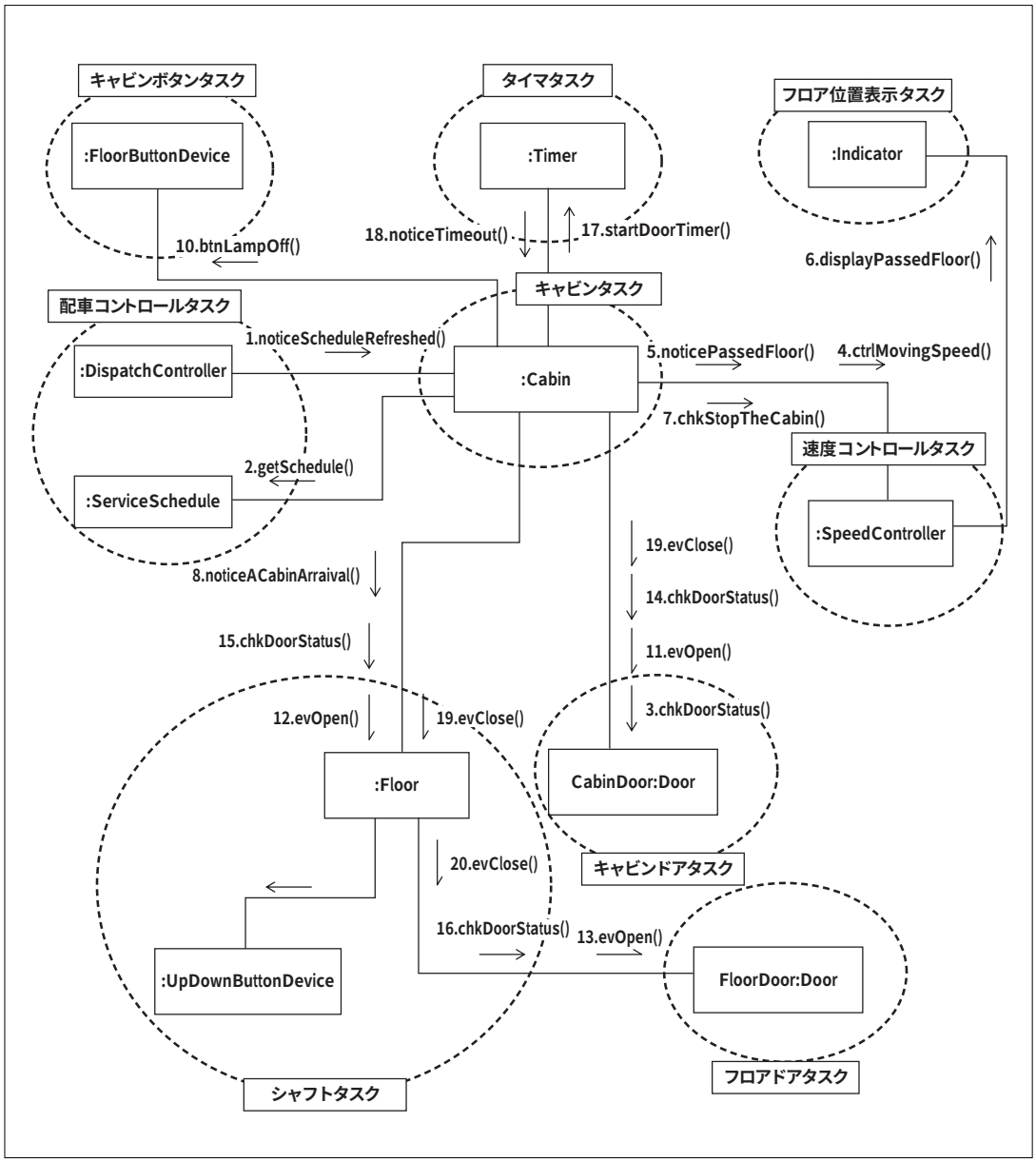
詳細化（デザインレベル）したシーケンス図をもとに、イベントに関連するオブジェクト群を抽出し、明確にします。今回作成したコミュニケーション図（図5～7）には、目安となるタスク境界を引いています。この境界線は、分析段階であらかじめ検討しておいたタスク構成をもとに、タスクとオブジェクトを対応させています。明確にしたオブジェクト群に対し、各タスクとコミュニケーション図を用いてグループ化していく過程で、複数のイベントから利用されるオブジェクトがないか同時に探します。

複数のイベントで利用されるオブジェクトは、資源の競合が発生する可能性があり、実際にタスクマッピングする際になんらかの排他制御を考える必要があります。ただし、タスクとオブジェクトのグループ化を再検討することで、タスク間でオブジェクトの競合を避けることができない場合もあります。そのため、タスクとオブジェクト群のグループ化は、いろいろな候補が存在しますので、グループ化したさまざまな候補に対して検討を重ね、最終的に最もバランスのよいグループ化を選択します。また、相互に通信を行っているオブジェクトがあった場合、想定していなかったタスク分割を行うことになるかもしれません。つま

り、最終的なタスク分割は、この作業を通じて決定されることになります。

コミュニケーション図の図6と図7を比較してみます。図6のキャビン内のフロア指定ボタンを起点として始まるイベントのオブジェクト群（フロア指定ボタン→キャビン→配車コントロール→配車計算→運行スケジュール表）と、図7の各フロアからの上下ボタンを起点として始まるイベント（上下ボタン→フロア→配車コントロール→配車計算→運行スケジュール表）では、それぞれから同じオブジェクト群（配車コントロール→配車計算→運行スケジュール表）を呼び出していることが

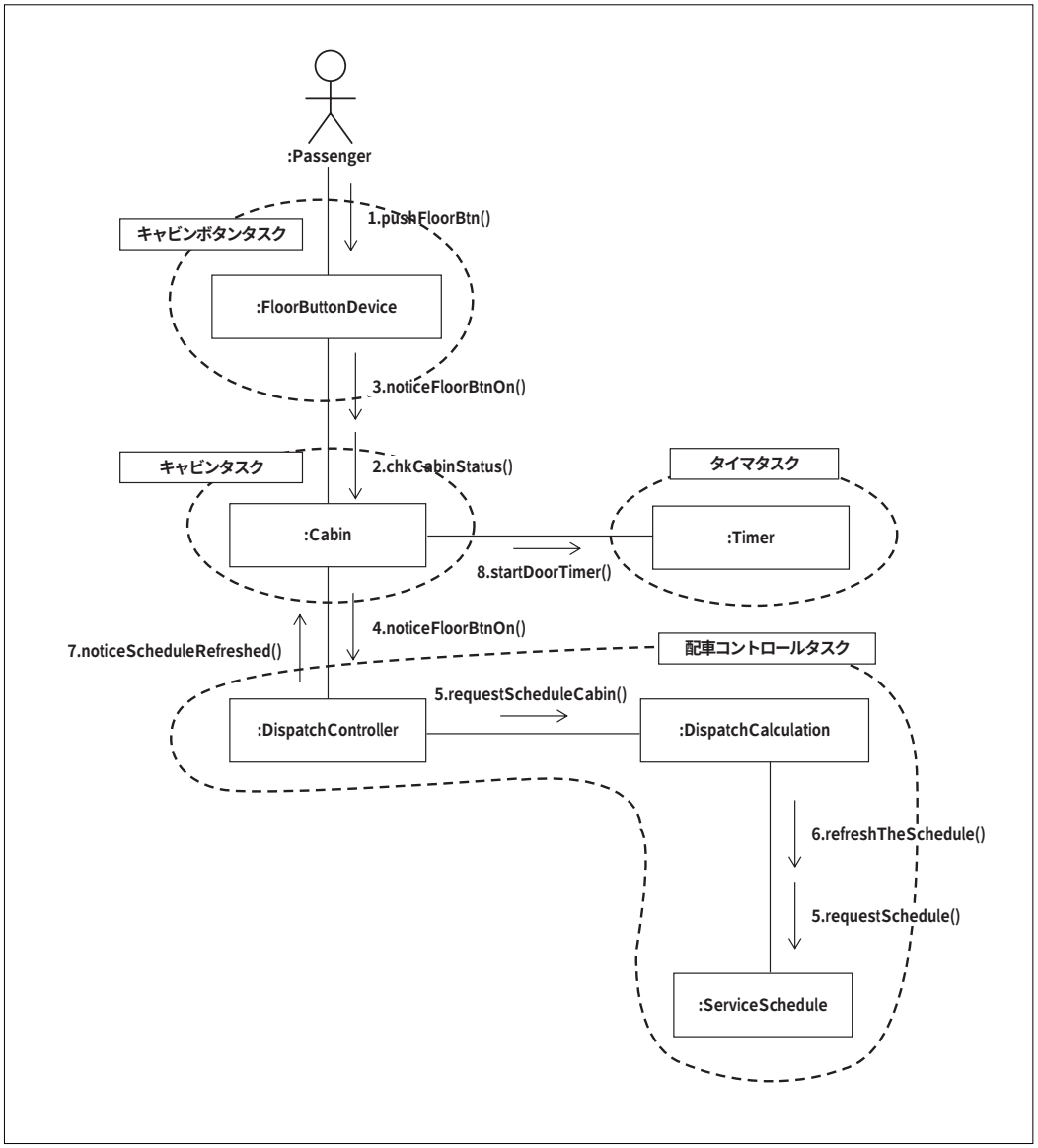
図5 そのフロアにキャビンを呼ぶ



わかります。これら2つのイベントは同時に発生する可能性が十分にあり、この一連のオブジェクト群（配車コントロール→配車計算→運行スケジュール表）に対してなんらかの排他制御を行う必要性があります（図8）。

実際にどのような制御により処理（資源）の競合を回避するかはタスク分割を行った結果、最終的なタスクが決定したあとに決めますが、今回の場合、どういった考え方で競合を回避したか説明します。2つのイベント、キャビン内のフロア指定ボタンを起点とするイベントと、各フロアからの上下ボタンを起点とするイベントから同時に呼ばれているこの一連のオブジェクト群（配車コントロール→配車計算→運行スケジュール表）を配車コントロールタスクとして独立させました。この配車コントロールタスクへの通信は、イベント

図6 キャビンの中からフロア指定ボタンにより目的階を指定する

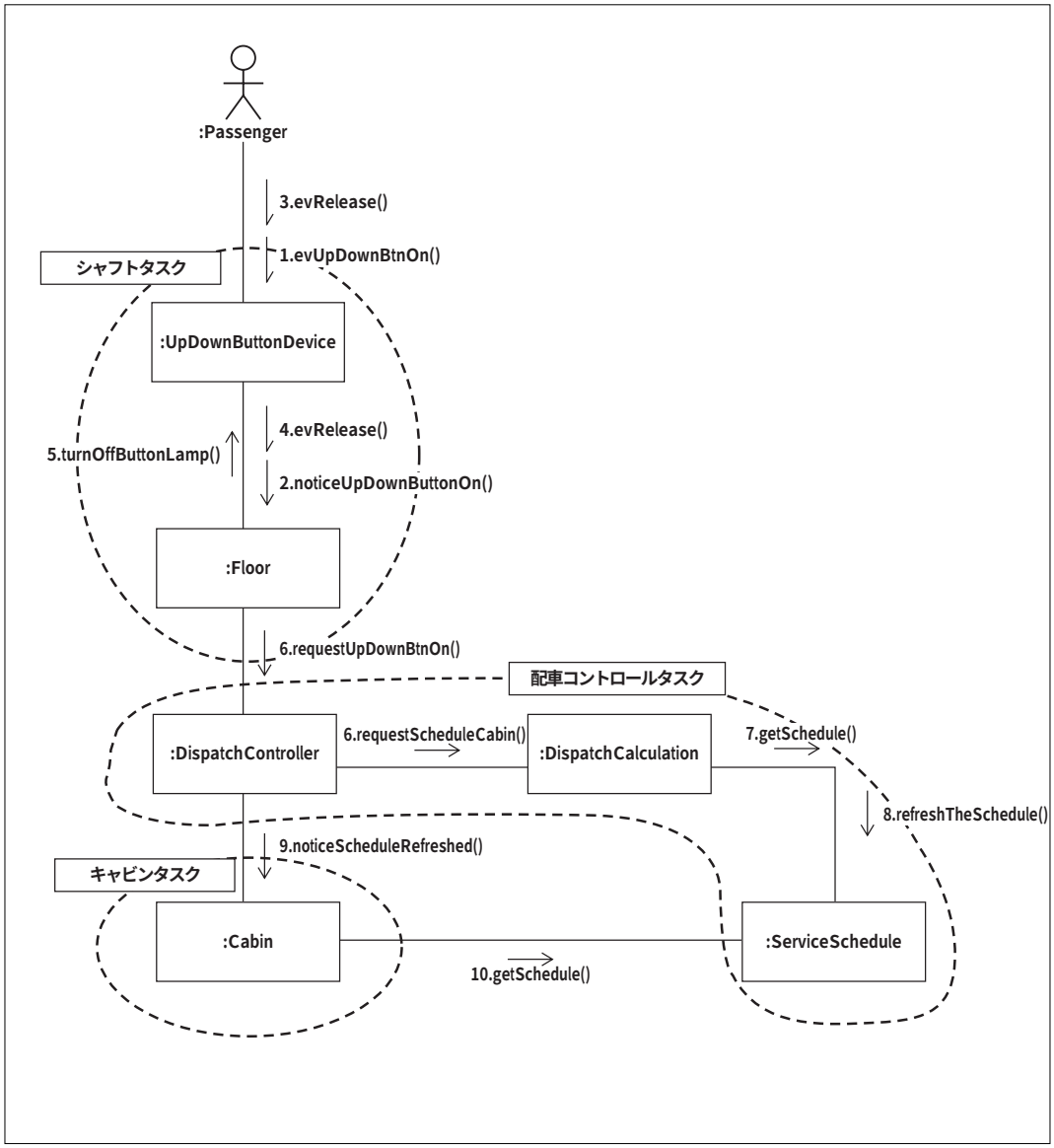


（メッセージ）にしています。配車コントロールタスクに対して複数のイベント（メッセージ）が同時に発生した場合、キューにイベント（メッセージ）をバッファリングすることにより、発生したイベント（メッセージ）を発生順に処理できるようにし、排他制御の代わりとして競合を回避しています。

Rhapsodyを使用した場合、タスクとして設定するとそのタスクに対して専用のイベント（メッセージ）キューが割り当てられます。割り当てられたイベント（メッセージ）キューを使用してイベントを発生順に処理できるようにします（図9）。

タイマとキャビンの間で相互に通信を行っていますが、あらかじめタスク分割の候補としてあります。したがって、相互に通信を行っているオブジェクト間で、新たにタスク分割を行う必要があ

図7 上下ボタンが押されて配車をスケジュールする

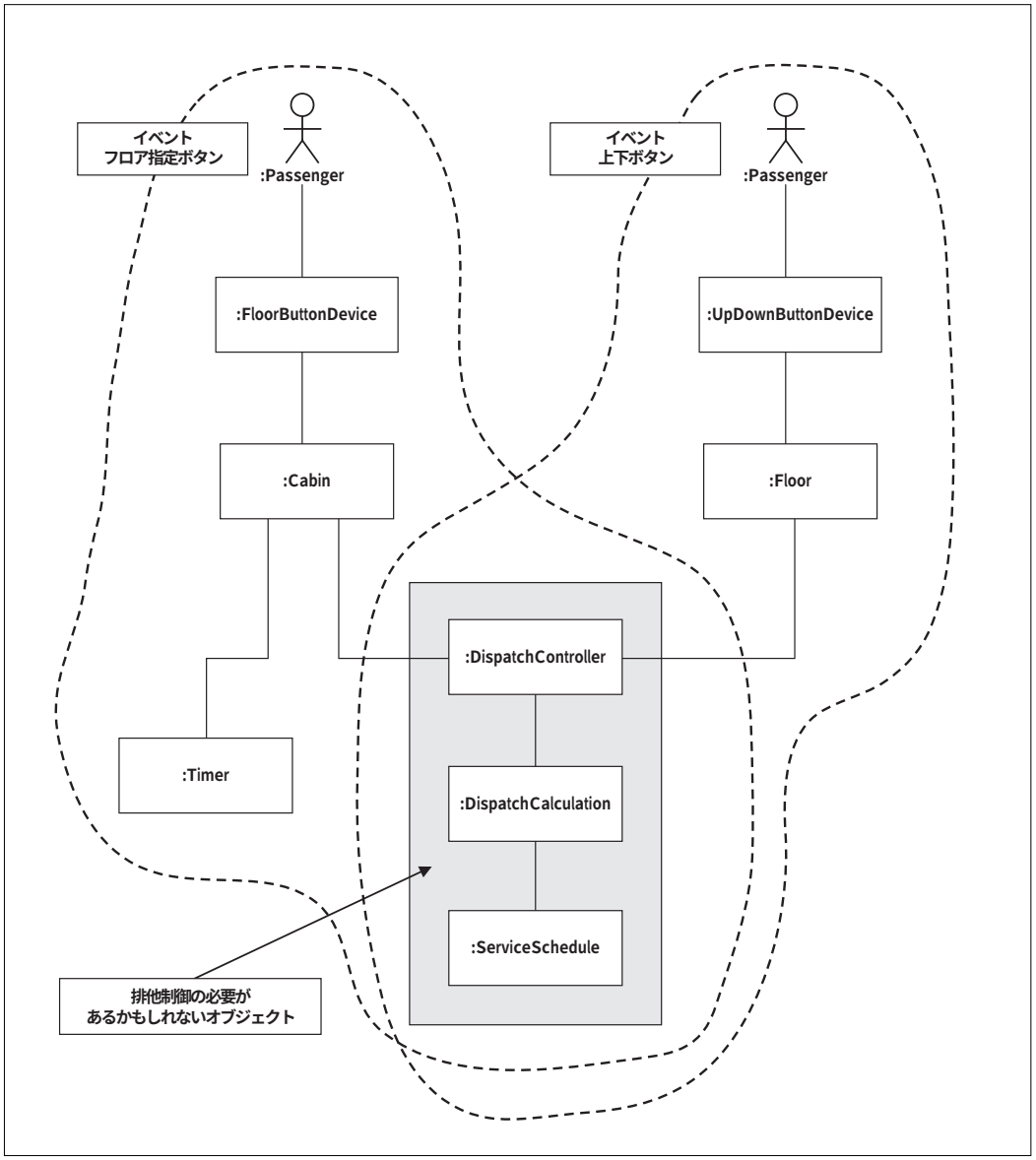


りそうなオブジェクトは、ありませんでした。

タスクマッピング図

前回（Vol.7）の分析時に行ったタスク分割を指標として、コミュニケーション図によるオブジェクト抽出の結果などからオブジェクトのグループ化を行い、さまざまなグループ化を検討した結果、最終的に最もバランスのよいグループ化をもとに、各タスクに対してオブジェクトをマッピングしていきます。タスクマッピング図は、クラス図やオブジェクト図がクラスやオブジェクトの静的関係を表しているのに対して、オブジェクトの動的な関係を表しています。つまり実際にプログラムとして動作する際の関係（どのオブジェクトはどのオブジェクトと関係があり、どのタスクに属するかなど）を表しています。まず、タスクの中心と

図8 別イベントから同じオブジェクトが呼ばれている図



なるオブジェクト（タスク起動の起点となるオブジェクト）を決定します。オブジェクトの候補として以下のものが挙げられます。

- イベントの起点になっているオブジェクト
- コントローラの機能を持つオブジェクト
- ほかのオブジェクトと独立して動作する必要があるオブジェクト（並行性）
- そのほか特別な理由があるオブジェクト（排他制御など）

これらの条件を考慮してタスクにマッピングしていきます。イベントの起点となるオブジェクトとしては、次のオブジェクトをタスク候補としました。これらはユースケースでアクタとして登場しているオブジェクトです。

- 上下ボタン
- フロア指定ボタン
- 開ボタン
- 閉ボタン

図9 メッセージ（イベント）キューの働き

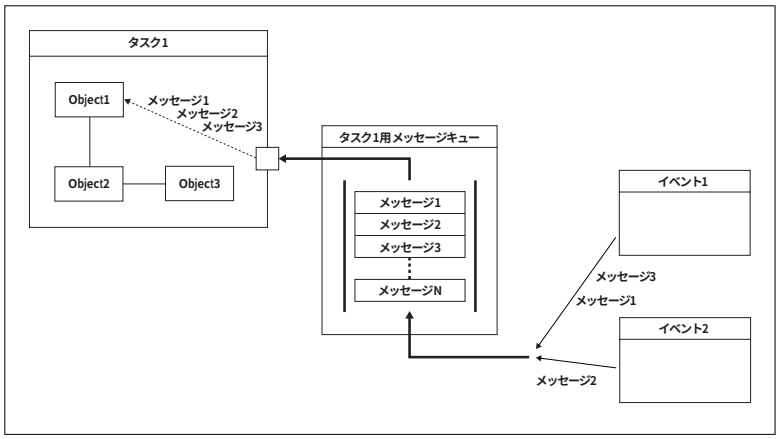
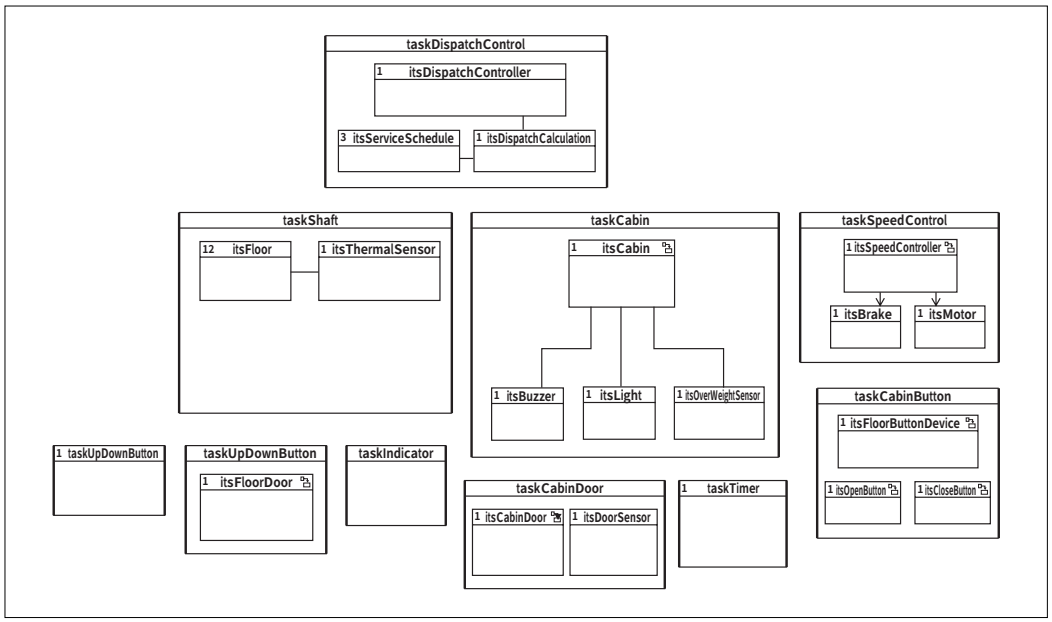


図10 タスクマッピング



● ドアセンサ

次にコントローラの機能を持ったオブジェクトは以下のオブジェクトを候補としました。

● キャビン
● 速度コントローラ
● フロア

次に独立して動作するオブジェクトのタスク候補です。

● ドア
● フロア位置表示インジケータ
● タイマー

最後に、特別な理由があるオブジェクトのタスク候補です。

● 配車コントローラ

これらのオブジェクトをタスクにマッピングしていきます。ここでオブジェクト個々には関連はないが、1つのタスクに含めることができそうなオブジェクトは1つにまとめています（図10）。今回は、フロア指定ボタン、開ボタン、閉ボタンとドアドアセンサをそれぞれタスクとしてまとめました。

次にそれぞれタスクにマッピングしたオブジェクトと関連のあるオブジェクトをタスクにマッピングしていきます。キャビンと関係のある室内灯、ブザー、過重センサをキャビンと同じタスクにマッピングします。同じように速度コントローラと関係のあるモーター、ブレーキを速度コントローラと同じタスクに、フロアと関係のある火災センサをフロアと同じタスクにマッピングします。マッピングが完了したら、タスク内のオブジェクト同士の関係も記述していきます。

分析時のタスク分割と大きく異なるのは、配車コントロールタスクを作ったことにあります。前項で説明したとおり、排他制御の代わりにタスクとして処理を分割しました。

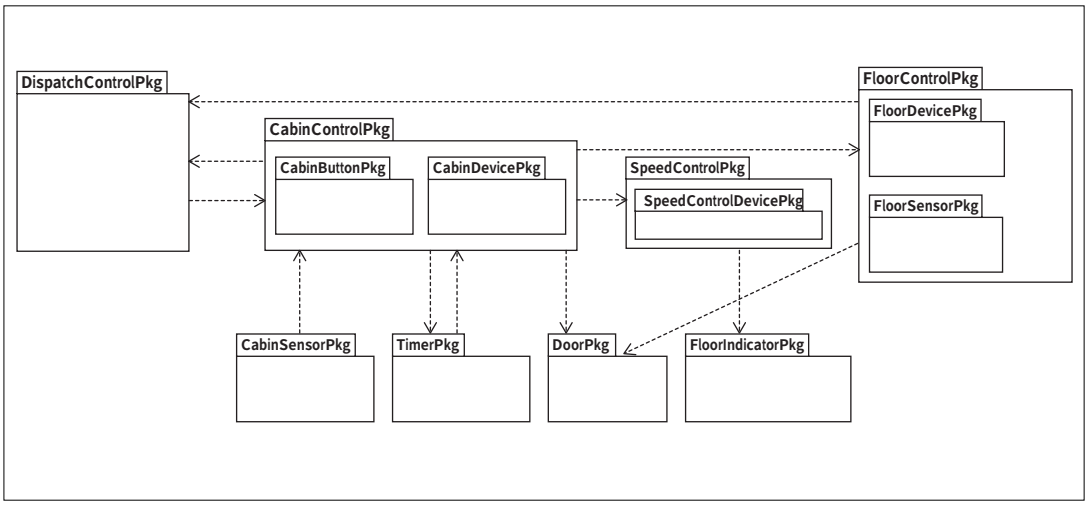
タスク構成表

タスクマッピング図を作成したら、決定した各タスクに対しタスクIDと優先順位を割り当てます。優先順位の決定方法についてはいろいろと方法がありますが、今回の場合、まずボタンやセンサなどハードウェアからの入力（割り込み）で起動し、処理を行うタスクを抜粋します。

タスクとしては、フロアボタンタスク、キャビンボタンタスク、キャビンドアタスク、フロアドアタスク、速度コントロールタスクが該当します。これらのタスクのうち、より緊急度の高い処理、ドアセンサからの入力によりドアを開け、人や物が挟まらないようにする処理を持つキャビンドアタスクとキャビンの移動を制御している速度コントロールタスクの優先度を上げます。また、それらハードウェアからの入力により起動するタスクからの入力を受け付け、その入力により実際に処理を行うのは、キャビンタスクになります。そこでキャビンタスクの優先度も高くしておく必要があると思われます。各フロアからの入力を処理するシャストタスクもこれにあたります。

一方、ハードウェアからの入力により起動するタスクでボタンからの入力により起動するフロアボタンタスク、キャビンボタンタスクは、それほどボタンの入力に対して即座に反応する必要もないと思いますので、キャビンタスク、シャフトタスクよりも優先度を下げます。また、フロア位置表示タスクなどは、キャビンの実際の位置と大きく食い違うことが起こらなければ問題はないと思いますので優先度を下げます。一通り優先順位がつけ終わったあとにそれぞれのタスクの責務を考慮し、タスクの優先度を調整します。配車コントロールタスクは、各エレベータの運行スケジュールの計算・管理を行うタスクです。もし配車コントロールタスクの優先度が低く、スケジューリングされないとイベントがキューに溜まるばかりで

図11 設計アーキテクチャ



エレベータの配車スケジュールを決める処理が行われない可能性があります。そこで配車タスクに関しては、ある程度優先度を上げる必要があると考えられます。また、タイマタスクなどは、スケジューリング過多になるとほかのタスクがスケジューリングされないなどの問題が発生する場合もありますので、優先順位の微調整は必要になるかもしれません。最終的に表1のタスク構成表のようにタスクIDと優先度を決めました。

表1 タスク構成表

No.	タスク名	タスクID	優先度
1	キャビンタスク	1	8
2	シャフトタスク	2	4
3	キャビンボタンタスク	3	5
4	フロアボタンタスク	4	9
5	キャビンドアタスク	5	1
6	フロアドアタスク	6	2
7	速度コントロールタスク	7	3
8	配車コントロールタスク	8	6
9	フロア位置表示インジケータタスク	9	10
10	タイマタスク	10	7

アーキテクチャ設計の結果

静的な構造

最終的に決定したパッケージ分割をもとにパッケージ間の関係を表します（図11）。分析時のパッケージ分割と比較すると、依存関係のあるオブジェクトのパッケージ構成を整理することによりパッケージ間の関係が疎になるように、パッケージ分割が再構成されています。キャビンボタンパッケージとキャビンデバイスパッケージは、キャビンコントロールパッケージにまとめることでパッケージ間の依存関係をなくしています。また、フロアデバイスパッケージとフロアセンサパッケージもフロアコントロールパッケージとしてまとめました。

最終的な設計アーキテクチャをもとにパッケージ間のインタフェースをコンポジット構造図（図12）で表します。

コンポジット構造図の特徴的な要素としてパート、コネクタ、ポートがあります。

- パート
  システムの内部構造を役割の観点から表す。この図では、パッケージ間の通信を表しているため、パートはない
- コネクタ
  ポートとポートをつなぐ関係を表している。この図では、パッケージ間の関係を表している
- ポート
  小さな正方形（□）で表されているのがポート。

パッケージが外部とやり取りする場合のインタフェースを表す。ポートが提供するインタフェース（提供インタフェース）は、白丸に実線で表す。また、ポートが要求するインタフェース（要求インタフェース）は、半円に実線で表す

各パッケージがほかのパッケージに対してどのようなインタフェースを要求し（要求インタフェース）、ほかのパッケージに対してどのようなインタフェースを提供するのか（提供インタフェース）を明確にします。パッケージ間の通信で、パッケージが違う場合には、インタフェースを分けています。それぞれのポートがインタフェースを提供するのか、それともインタフェースを要求するのかを各パッケージ間の関係を見ながら記述していきます。各パッケージが相手のパッケージに対して何を要求し、何を提供するのかを詳細化（デザインレベル）したシーケンス図やコミュニケーション図をもとにインタフェースを決めていきます。

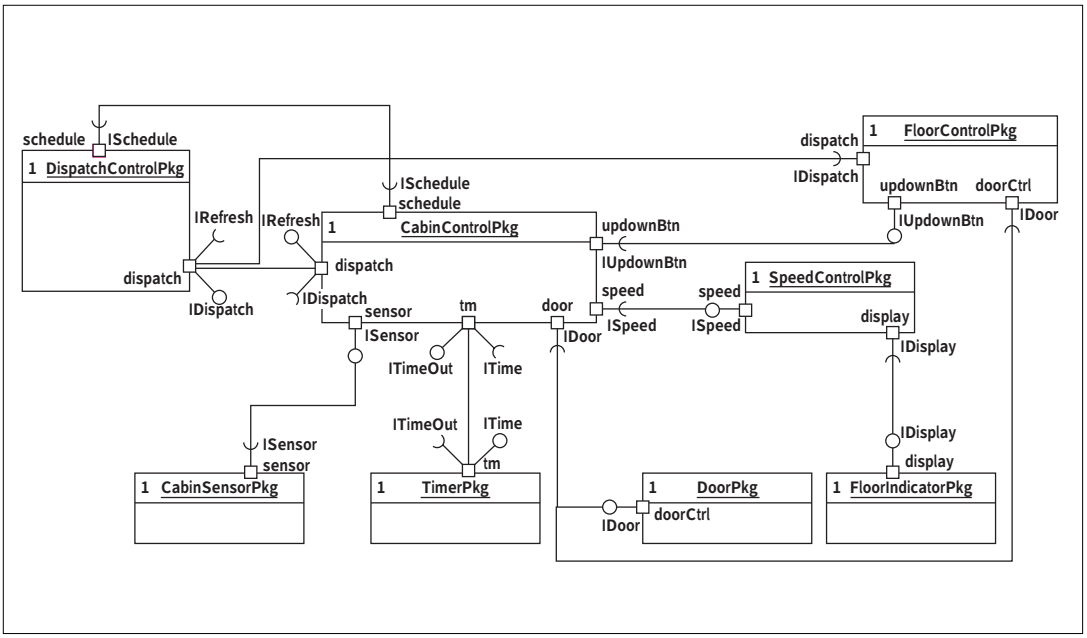
動的な構造（タスク分割とタスク間通信）

タスクマッピング図をもとに動的な構造図（コンポジット構造図）を作成します（図13）。動的な構造図は、静的な構造図ではわからない、実際にタスク分割された際のタスク間の関係、つまりどのタスクがどのタスクと通信するかなどを動的な観点で表します

コンポジット構造図の特徴的な要素としてパート、コネクタ、ポートがあります。

- パート
  システムの内部構造を役割の観点から表す。この図では、オブジェクトを表している
- コネクタ
  パートとパート、またはポートを繋ぐ関係を表している
- ポート
  小さな正方形（口）で表されているのがポート。パートとポートをコネクタで関連づけることにより、タスクが外部とやり取りする場合の境界

図12 静的な構造図

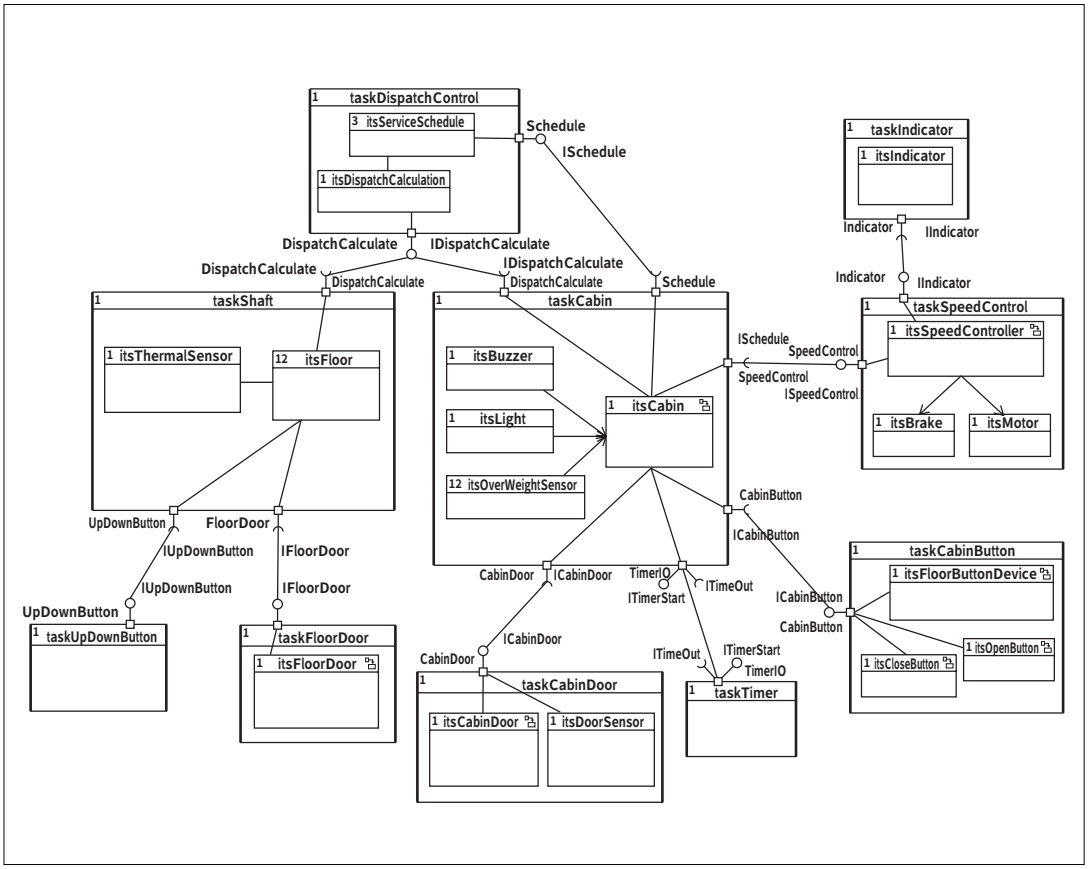


線を明確に表現できる。またポートがやり取りする場合のインタフェースを提供する。ポートが提供するインタフェース（提供インタフェース）は、白丸に実線で表す。また、ポートが要求するインタフェース（要求インタフェース）は、半円に実線で表す

まず、タスク内のオブジェクト間の関係に注目し、関係を明確にします。次に各タスク間の関係を明確にしていきますが、その際には、シーケンス図やコミュニケーション図などを参照します。単に関係を明確にするだけではなく、そのタスクがほかのタスクに対してどのようなインタフェースを要求し（要求インタフェース）、ほかのタスクに対してどのようなインタフェースを提供しているのか（提供インタフェース）を明確にします。

各タスク間の通信で、タスクが異なる場合には、別のポートを設け、インタフェースを分けています。各タスクのそれぞれのポートが、インタフェースを提供するのか、それともインタフェースを要求するのかを各タスク間の関係を見ながら記述していきます。たとえば、キャビンタスクは、配車コントロールタスクに対してエレベータの配車スケジュールを計算した結果を要求し、配車コントロールタスクは、エレベータの配車スケジュールを計算した結果を提供します。このように各タスクが相手のタスクに対して何を要求し、何を提供するのかを詳細化（デザインレベル）したシーケンス図やコミュニケーション図をもとにインタフェースを決めていきます。

図13 動的な構造図



MDD Chapter 3

詳細設計とMDD
クラス図とステートチャート図

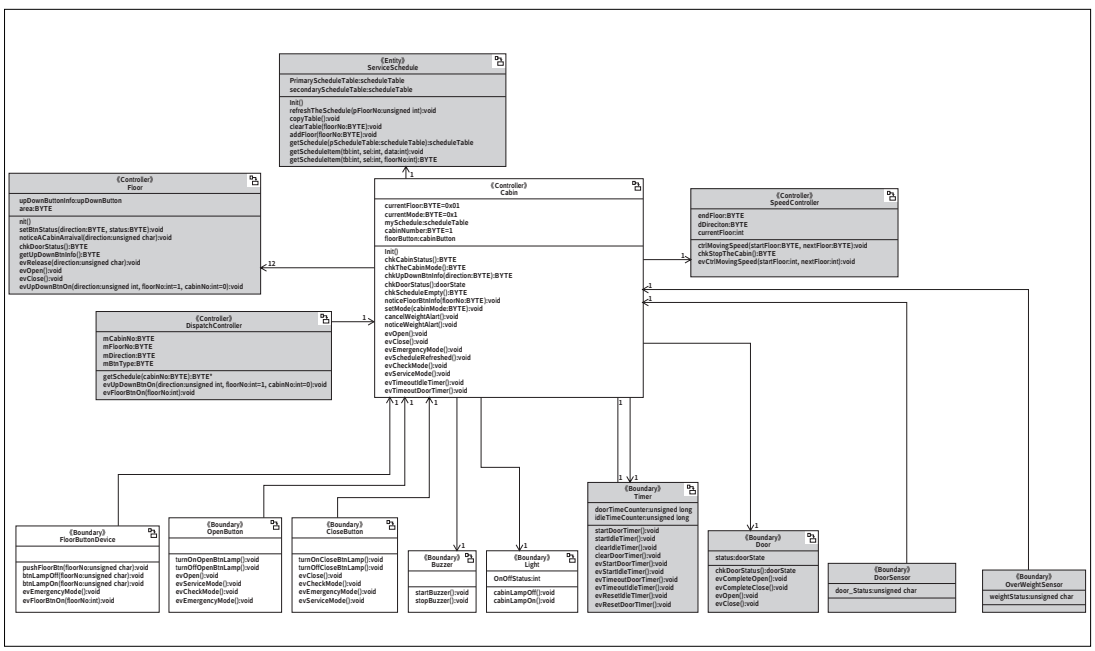
詳細設計とMDD

サブシステム内のクラス図

最終的なアーキテクチャ構成から、設計レベルのクラス図を作成します（図1）。設計レベルのクラス図では、クラスの属性や操作、操作の引数や戻り値などを詳細に表記します。そしてメッセージを送るためのクラス間の関連とその方向、多重度も表現します。サブシステム内のクラスに注目していますが、この中のクラスが関係しているほかのパッケージのクラスをグレーで表示して、ほかのパッケージとの依存関係があることを把握できるようにします。

Rhapsodyでは、クラス間に関連を引くと、自動的にロール名（itsClassName）がつけられるようになっています。このロール名は、実装時に相手のオブジェクトを参照するためのポインタ変数や配列の名前として利用されます[注1]。また、オブジ

図1 キャビンパッケージのクラス図



注1●本来は、オブジェクトの役割をイメージした名前を設定するべきですが、今回の実例では、Rhapsodyが自動的に命名したロール名を使っています。

ェクト間にリンクを引いた場合、自動的に、相手のオブジェクトのアドレスをポインタ変数や配列に代入するコードが生成されます。

図2は、DispatchControlPkgに対するクラス図であり、運行スケジュールを計算する配車アルゴリズムを実現するためのDispatchCalculationクラスを含んでいます。今回実現する各階層行きのキャビンが1つずつであるケースと、同じ階層行きのキャビンが複数基あるようなケース、あるいは、混雑時と閑散時では、配車アルゴリズムは大きく異なることが考えられます。今回対象としているエレベータシステムには、“ビルが変わりエレベータの数が変わっても再利用可能にする”という要件があるので、この要件に柔軟に対応できるようにするために、デザインパターンのStrategyパターンを使って、配車アルゴリズムを簡単に変更できるように設計します。Strategyパターンでは、アルゴリズムを表すインタフェースクラス（Strategyインタフェース）を用意し、それを実装するConcreteStrategyクラスを用意します。例題の中の配車アルゴリズムの部分では、DispatchCalculationクラス（配車計算）クラスとStrategyインタフェースを集約関係で結ぶことで、実際に採用するアルゴリズムを簡単に切り替えることができるようにします。また、ConcreteStrategyクラスとして、各階層行きのキャビンが1つずつの場合のアルゴリズム（SingularStrategyクラス）や、各階層行きのキャビンが複数の場合のアルゴリズ

図2 配車コントロールパッケージのクラス図

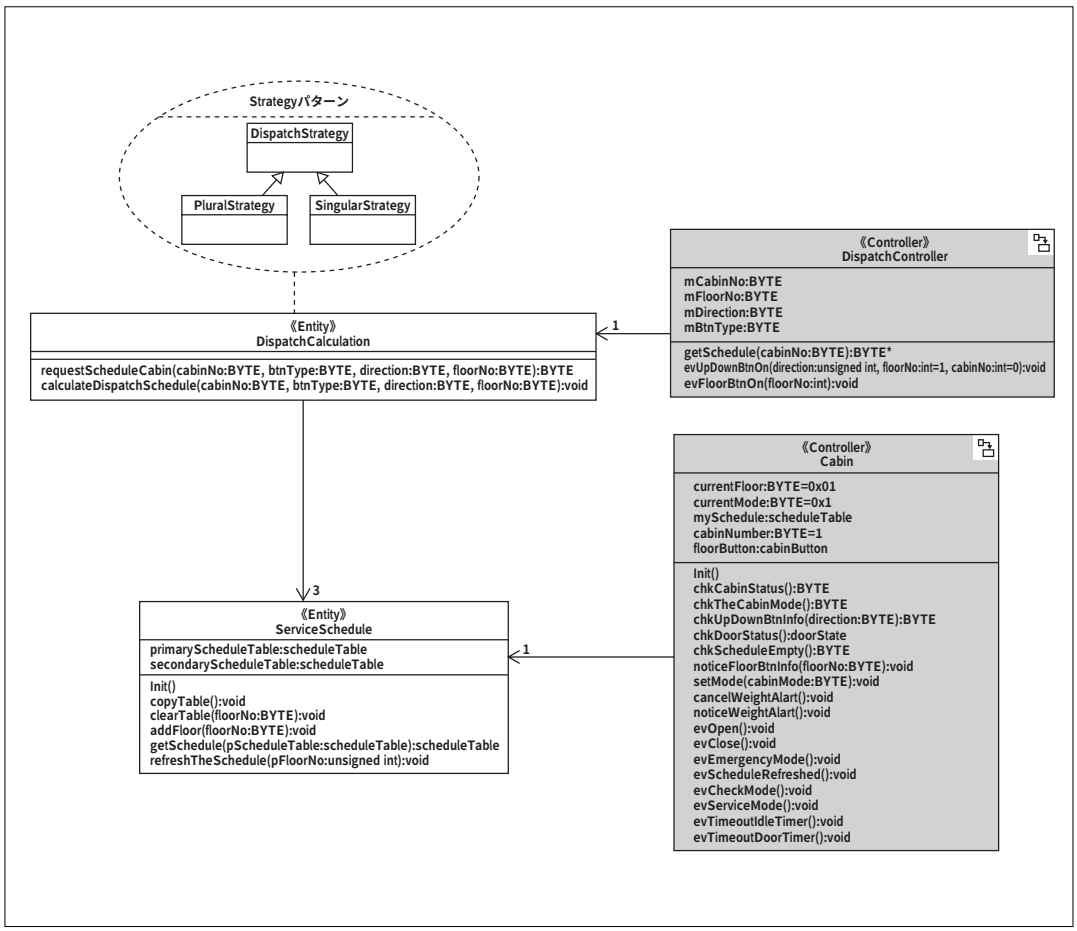


図3 キャビンのステートチャート[注2]

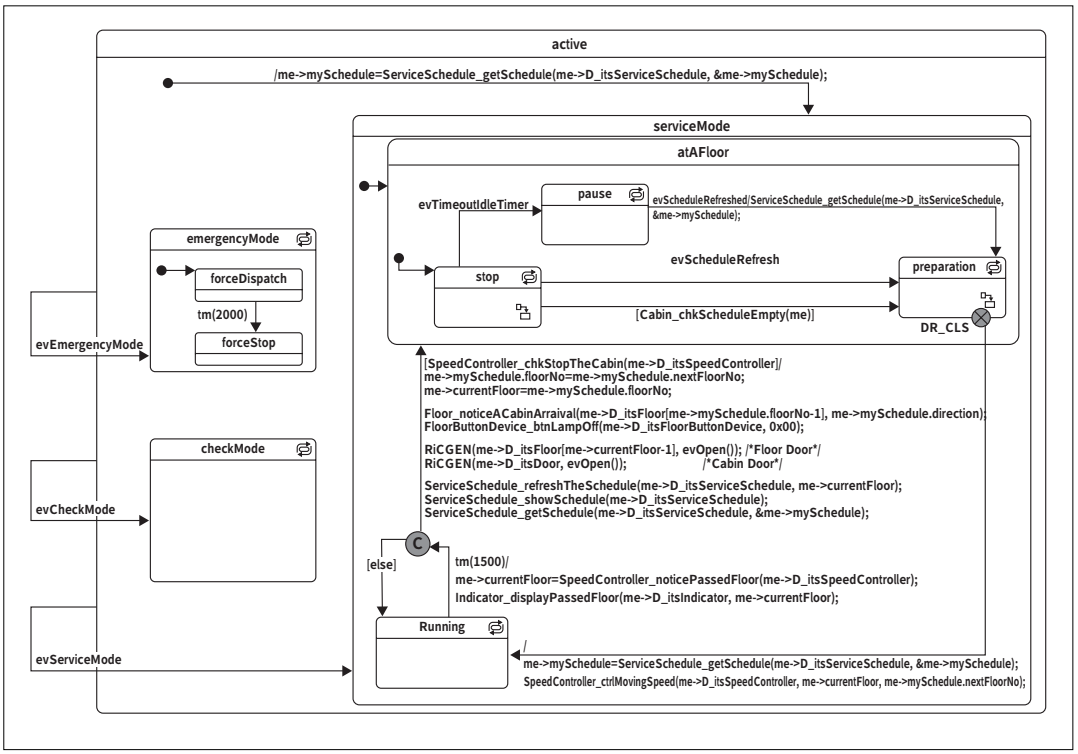
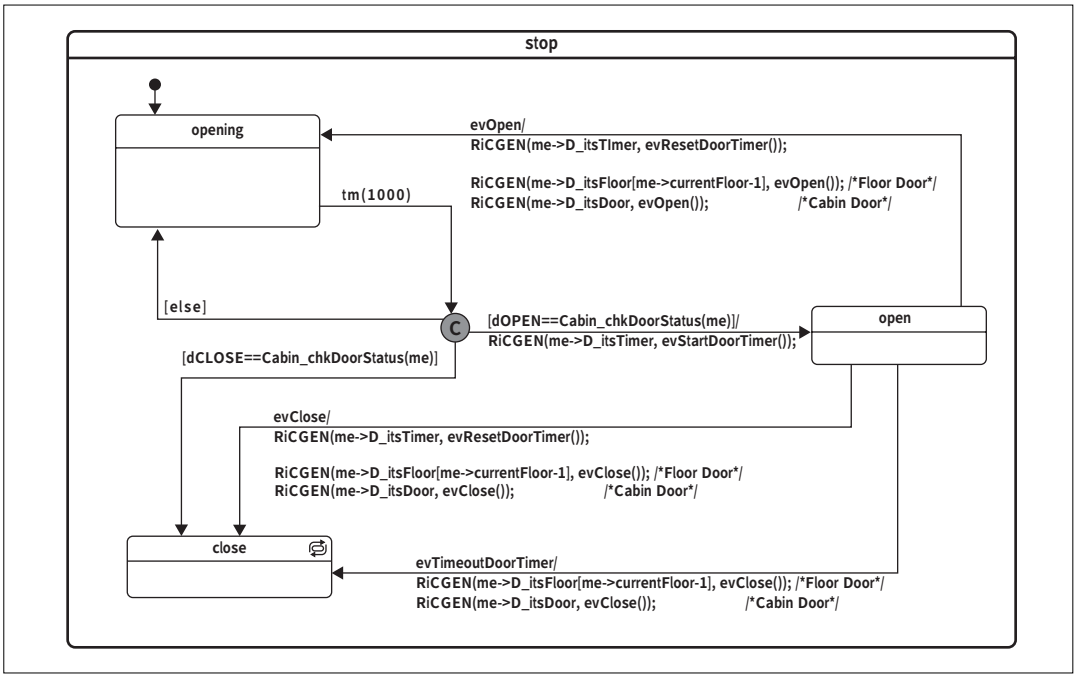


図4 停止状態（キャビンクラスのStop状態）のサブステートチャート



注2●ステートチャート内で用いられているtm()はタイムアウトトリガであり、RiCGENはイベント送信用のマクロです。

ム（PluralStrategyクラス）が考えられます。

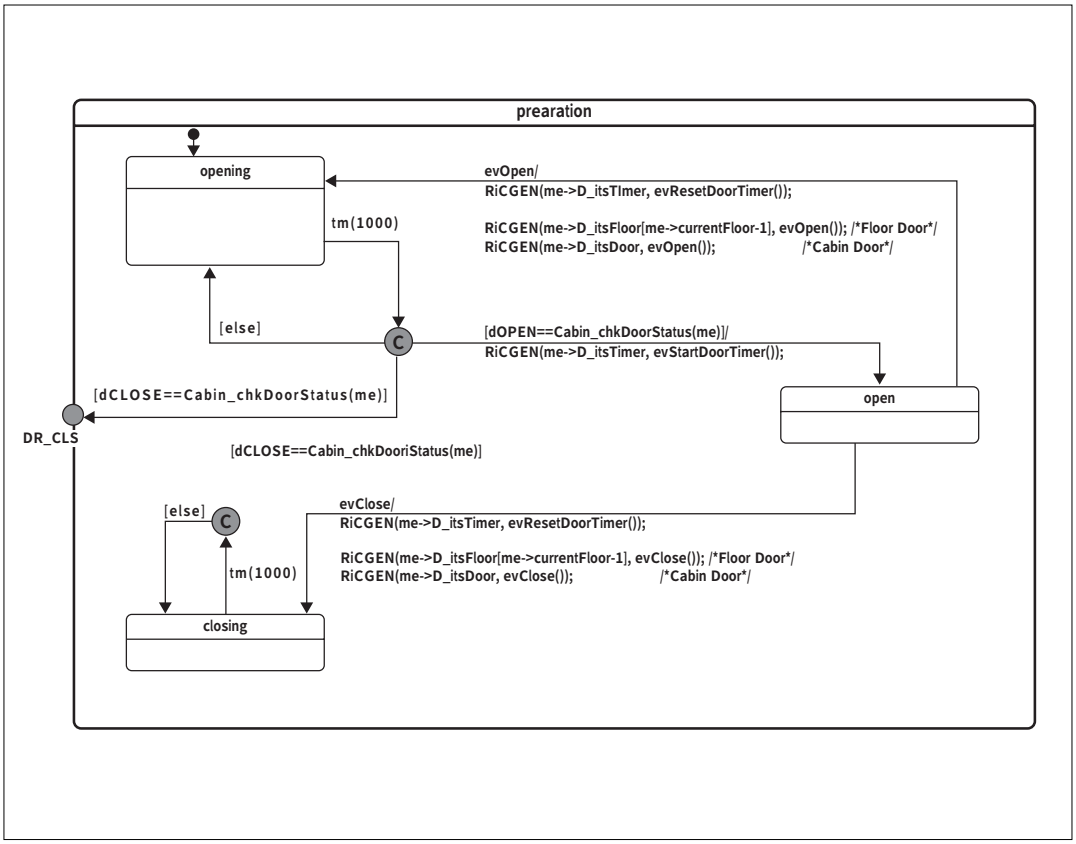
ステートチャートによる振る舞いモデリング（キャビン、ドア）

Cabinクラスは、受信したイベントに応じて処理を行ったり、定期的にほかのオブジェクトの状態を確認したりする必要があり、複雑な振る舞いを持つクラスです。いくつかのシーケンス図で表現されているCabinの振る舞いから、このクラスにステートチャートをモデリングしています（図3）。

ステートチャートは、状態とその状態からほかの状態に移る遷移で表現される動的なダイアグラムです。遷移の条件として、トリガやガードがあり、遷移するときに実行するアクションを記述します。Cabinクラスのステートチャートでは、各モード（非常／点検／運行）に対してemergency Mode、checkMode、serviceModeという状態を用意し、各モードで異なる処理を実行できるようにしています。serviceModeでは、休止、停止、運行準備、移動中を意味するPause、Stop、Preparation、Runningというサブ状態が入れ子で表現され、このうちstopとpreparationには、階層化されたサブステートチャート（図4、図5）が表現されています。この部分は、キャビンがある階に止まっている（停止／運行準備状態）時のドアの開閉を制御するための処理を表現しています。

Doorクラスは、Cabinクラスからドアの開閉を指示するメッセージ（evOpen、evCloseイベント）を受信することで、ドアを開けている（＝Opening）、ドアが開いた（＝Opened）、ドアを閉じている（＝Closing）、ドアが閉じられた（＝Closed）という状態を表現しています（図6）。

図5 運行準備状態のステートチャート



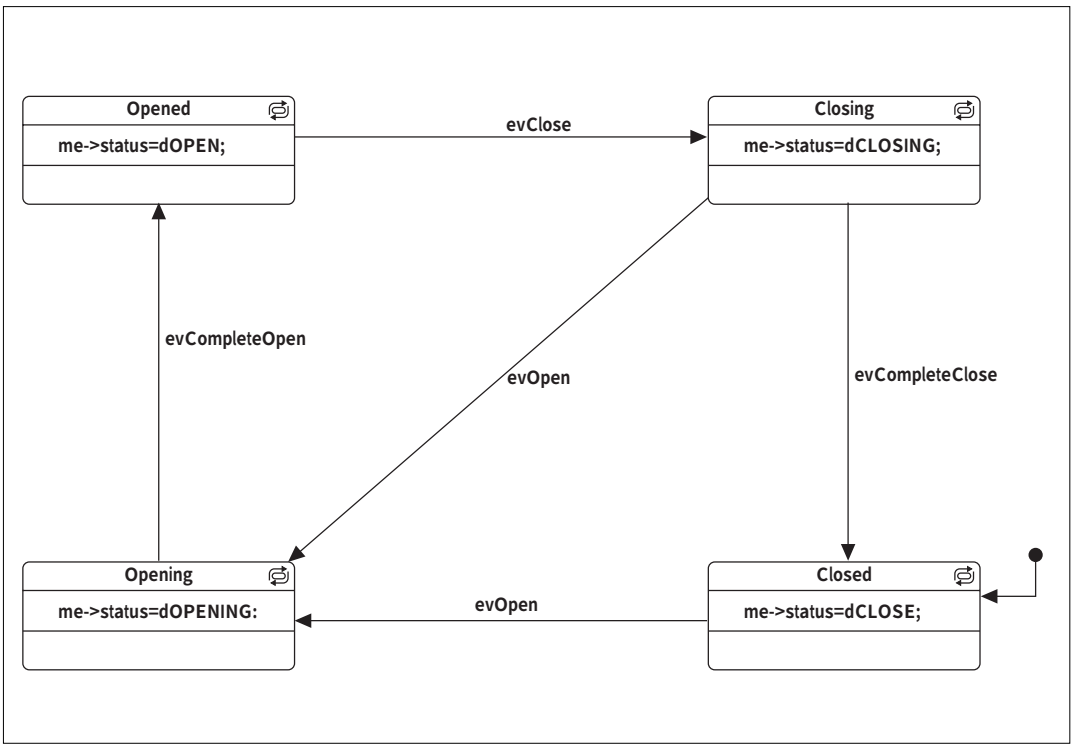
モデル駆動型開発――設計から実装へ

Rhapsodyでは、ステートチャートのアクション、つまり状態に入ったときに実行する処理や、別の状態に遷移するときに実行する処理などは、CやC++、Javaといったプログラミング言語で実装します（図7）。また、クラスの操作定義もプログラミング言語を用いて実装します。このような、コードを実装するフィールドでは、ほかのオブジェクトの操作を呼ぶ、あるいは属性にアクセスするなど、モデル内のさまざまな設計情報（クラス名、ロール名、操作名、属性名など）を入力する必要があります。Rhapsodyでは、このようなモデル内の設計情報を簡単に参照するために、インテリバイザーという機能提供しており、コードの入力ミスを軽減できます。

さて、モデルからコードを生成しましょう。コードを生成する手順は簡単です。図7にある「GMRボタン[注3]」を押すだけです。この操作で、RhapsodyはMDD（モデル駆動型開発）環境として、クラス図、オブジェクト図、ステートマシン図、アクティビティ図のモデル情報をもとに、実装コードを生成します。組込みシステムでは、ターゲットプラットフォームとして、μITRONやVxWorksなどの数多くのRTOSがありますが、Rhapsodyでは、リアルタイムフレームワークを切り替えることで、さまざまなRTOS環境に簡単にポーティングすることができるしくみを提供しています。このリアルタイムフレームワークは、ライブラリとして提供されています[注4]。

リアルタイムフレームワークには、アニメーションを実行するためのフレームワークも含まれま

図6 ドアのステートチャート



注3●GMRはGenerate/Build/Runの略です。つまり、コード生成、ビルド、実行を操作するためのボタンです。アプリケーションをビルドするのに必要なmakeファイルもコード生成と同時に生成されます。
注4●ライブラリのもととなるソースコードも提供されています。

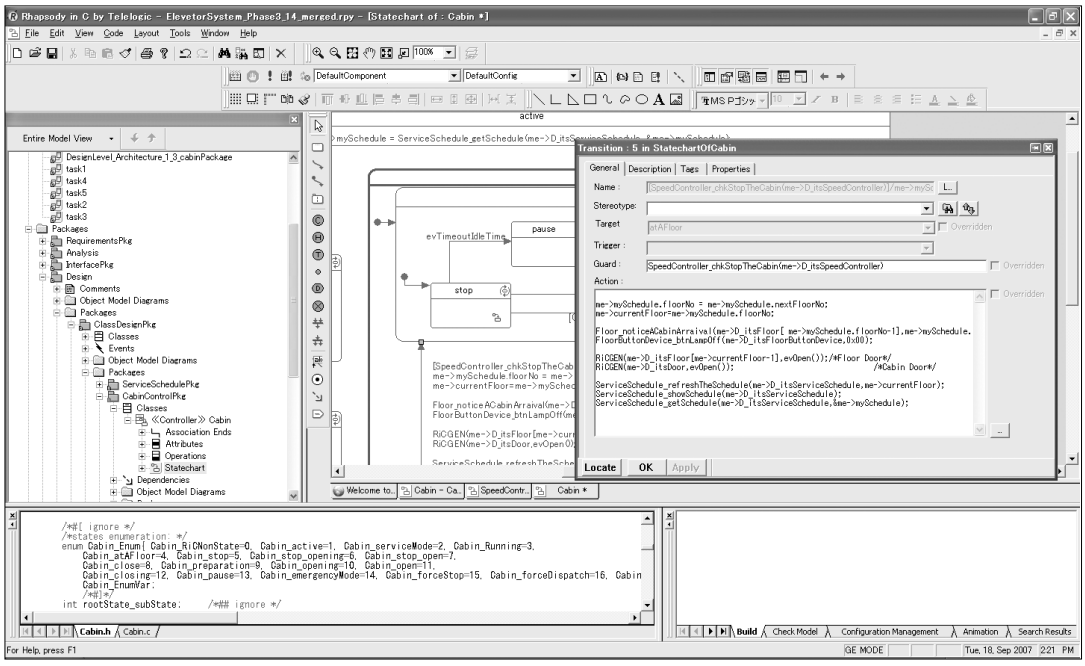
すが、コアとなるのはオブジェクト実行フレームワーク（Object Executable Framework：OXF）と呼ばれる部分で、最終的な製品コードにリンクされるライブラリです。OXFは、その名のとおり、Rhapsodyでモデリングしたオブジェクトを実行するために必要な実装を含むフレームワークで、イベント駆動フレームワーク、RTOSのポーティングするためのOSアダプタ層、多重度が多の場合の関連の実装に使われるコンテナクラスの3つのパーツから構成されます。

イベント駆動フレームワークでは、タスクのベースクラスやイベントディスパッチ機能、つまりイベントをメッセージキューに入れる処理、メッセージキューの先頭のイベントを取り出してイベントを処理するオブジェクトに渡す処理など、ステートチャートを持つオブジェクトに共通の処理が実装されています。

OSアダプタ層では、さまざまなRTOSにポーティングするためのレイヤで、この中には、RTOSごとにOSラッパーファイルが用意され、タスクやメッセージキューやイベントフラグなどのRTOSサービスを使って実装されています。そのため、このアダプタ層をカスタマイズすることで、Rhapsodyが未対応のRTOSにも柔軟に対応させることができます。コンテナクラスは、多重度が多の関連の実装に使われます。

Rhapsodyから生成されたコードは、ステートチャート（あるいはアクティビティ図）で表現されたクラスの動的な設計情報も実装されるため、スケルトンではない、実際に動かすことのできるコードです。さらに、PC上での動作検証から実機上でデバッグをする段階に移行する場合や、RTOSを変更するなど、ターゲット環境を変更する場合には、リアルタイムフレームワークのライブラリをターゲット環境に対応したライブラリに置き換えることで、そのターゲット環境で動作するバイナリを生成させることができます。リアルタイムフレームワークのライブラリの置き換えは、メニュー1つで操作できます。

図7 状態アクション記述



MDD Chapter 4

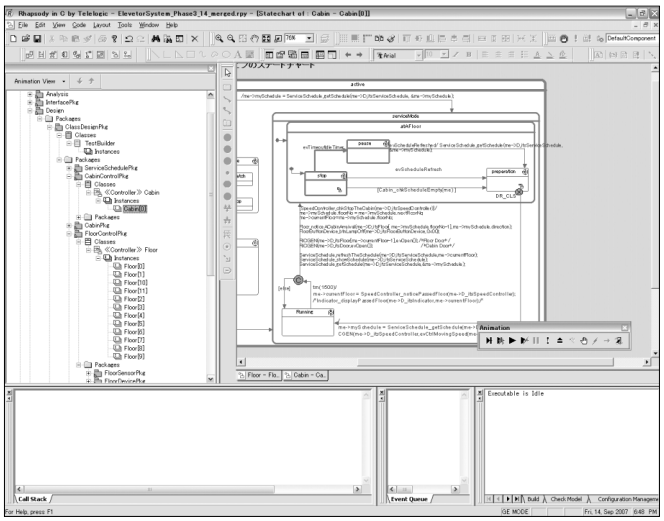
シミュレーション

設計仕様どおりに動作するか確認

Rhapsodyには、設計モデル上で動作検証する機能（アニメーション機能）が備わっています。このアニメーション機能は、実際にホスト環境あるいはターゲット環境上で実行されているアプリケーションの情報を、Rhapsody上に表示させる機能であり、ステートチャート（あるいはアクティビティ図）上でオブジェクトの動作を確認したり、シーケンス図上で各インスタンス間のメッセージのやり取りを記録し、モデルが正しく動作しているかどうかを確認することができます。このような、振る舞いをモデリングしているダイアグラム上で実行の様子を確認するだけではなく、モデル情報を管理しているブラウザ上で、各クラスのオブジェクトが、正しく作られているか（図1[注1]）、インスタンスの属性の値や、関連を持つオブジェクトが認識できているか（図2[注2]）を確認することもできます。

アニメーション機能には、ブレークポイント機能やステップ実行機能、メッセージキューに入っているイベントを表示するイベントキューや、関数の呼び出し履歴を表示するコールスタックウインドウがあります。またシステムの外からのイベントを発生させるためのイベントジェネレータが用意されているので、たとえば“ある階で上ボタンが押された”ことに対するシステムの反応をシミュレーションできますが、今回はより実際のエレベータシステムの動きをイメージしやすくするために、GUIを用意し、そこからボタン操作を行ったり、キャビンの動きやインジケータ情報を表示するようにしています。ステートチャートやシーケンス図などの設計モデルが実行結果と一致しているかどうかは、Rhapsodyのアニメーションシーケンス図や、アニメーションステー

図1 ブラウザ



注1●画面左側にあるブラウザ上で、Cabinオブジェクト Cabin[0]とFloorオブジェクト Floor[0]-Floor[11]が確認できます。
注2●Timerオブジェクトの情報です。Attributesフィールドで、属性doorTimeCounter、idleTimeCounterがそれぞれ0（ゼロ）であることが確認できます。さらに、Relationsフィールドでは、関連を持つCabinクラスのオブジェクト Cabin[0]が参照できていることが確認できます。
注3●今回は誌面の都合上、アニメーションシーケンス図の掲載は省略します。

トチャートで確認します[注3]。

シミュレーションで確認するのは、次のケースです。

> イベント：9階で下ボタンを押された
> ボタンが押された時点のキャビンの状態：
> 1階で休止状態（運行スケジュールが空）

今回シミュレーションするサンプルでは、高階層・中階層・低階層それぞれ昇降路が1つずつで各昇降路に対応する形で上下ボタンが存在しています。そのため、上下ボタンを操作した時点で配車するキャビンは決定しています。また、キャビンは1階で休止状態、つまり運行スケジュールは空であるため、システムの動きは、“高階層行きのキャビンが、1階から9階に移動し、ドアを開ける”というシナリオになります。

①9階で下ボタンを押す

Chapter2の図1(p.141）にあるように、FloorオブジェクトがUpDownBottunDeviceのturnOnButtonLamp( )を呼び、下ボタンを点灯します。さらにFloorオブジェクトはDispatchControllerオブジェクトにイベントevUpDownBtnOn[注4]を送信し、“9階で高階層キャビン（Cabin[0]オブジェクト）の下ボタンが押された”ことを通知します（図3）。DispatchControllerオブジェクトは、DispatchCalculationオブジェクトを使ってCabin[0]の運行スケジュール＝ServiceSchedule[0]を変更します（図4）。

②休止状態→運行準備状態→運行状態へ移行

①の処理の後、DispatchControllerオブジェクトはCabinオブジェクトにイベントevScheduleRefreshedを送信し、運行スケジュールが更新さ

図2 クラスFeatureダイアログ

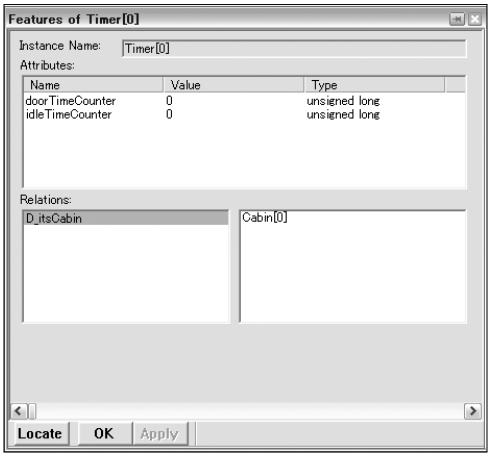


図3 シミュレーション（9階で下ボタンを押す）

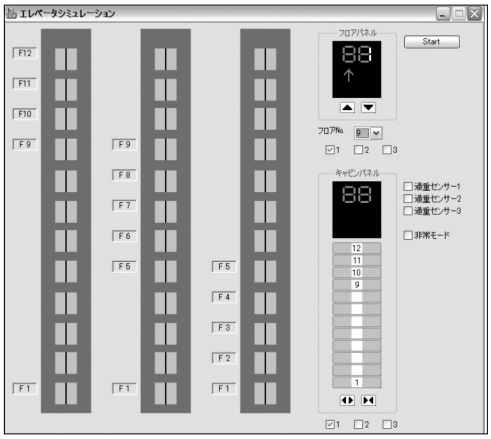
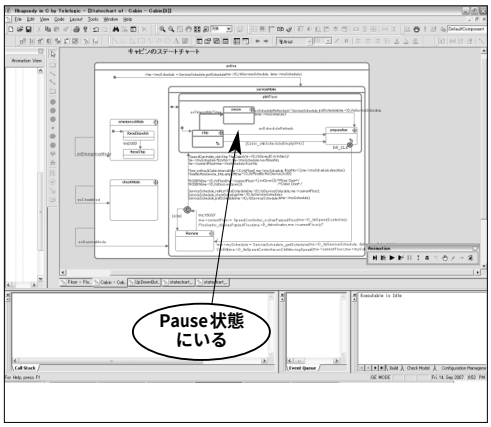


図4 キャビン休止状態



注4●パラメータとして“フロア＝9、キャビン番号＝0、方向＝下”の情報を渡しています。

図5 シミュレーション（休止モードから運航モードへ移行）

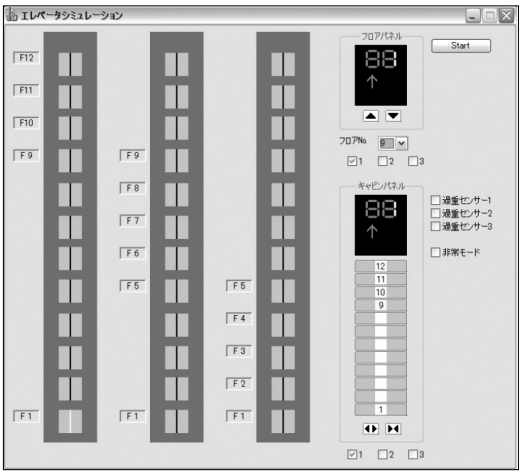


図6 運行準備状態に移行

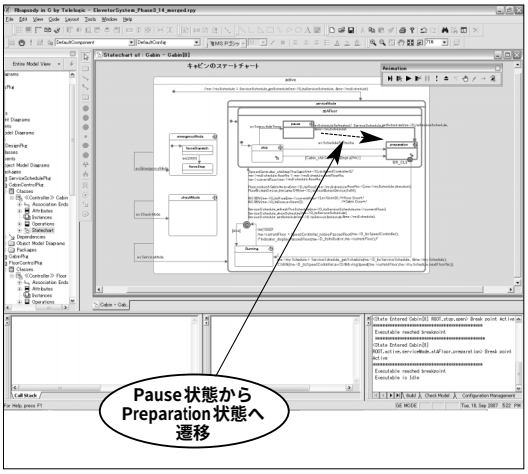


図7 シミュレーション（移動中）

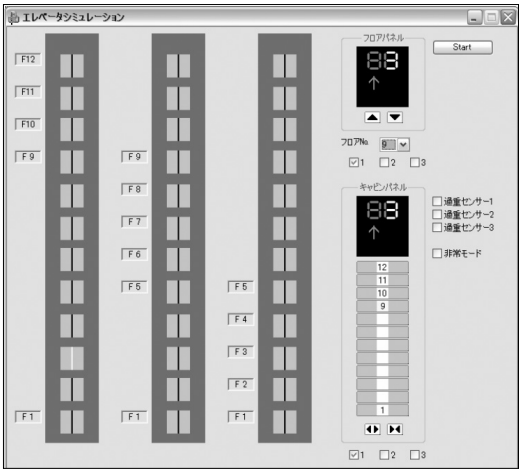


図8 運行状態：9階に移動中

図9 シミュレーション（9階へ到着）

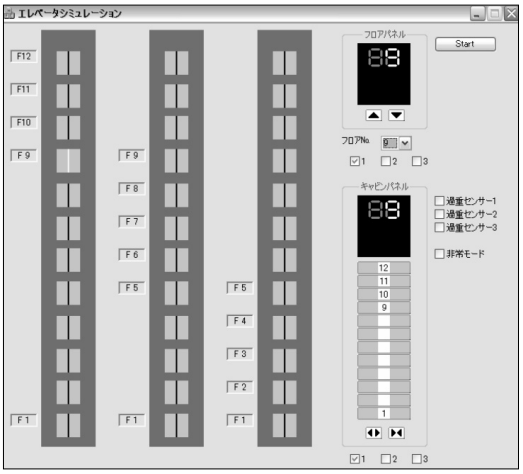
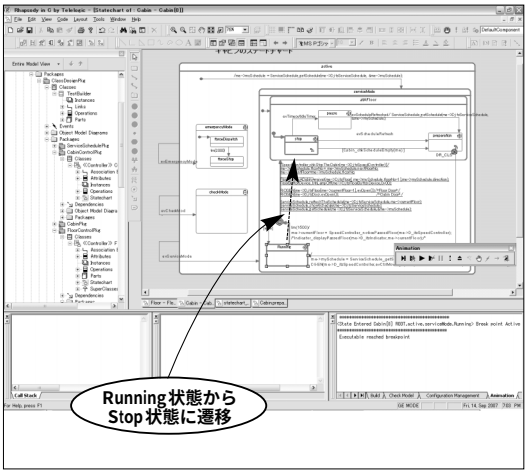


図10 運行状態：9階に到着



れたことを通知します。Cabin[0]は自分の属性に更新されたスケジュールをコピーし、運行状態に移行して9階に移動を開始します（図5、図6）。

③移動中

Cabin[0]は運行準備状態（preparation）から運行（running）に移行するときに、SpeedControllerオブジェクトにイベントevCtrlMovingSpeedを送信し、キャビンを移動させます（図7、図8）。SpeedControllerオブジェクトは、移動中に現在位置を計算し、Indicatorオブジェクトに表示させます（displayPassedFloor( )）。指定された階に到着したかどうかは、一定時間ごとにCabin[0]がSpeedControllerオブジェクトに確認しています。

④9階へ到着

9階に到着すると、Cabin[0]はFloorオブジェクトに到着したことを通知します。Floorオブジェクトはスケジュールに登録されているキャビンの進行方向と同じ方向のボタンを消灯し、インジケータの方向表示も消します（図9、図10）。

⑤ドアを開く

Cabin[0]は、Running状態からstop状態に遷移するときに、CabinDoorとFloorDoor[8]（9階フロアドアオブジェクト）に対してイベントevOpenを送信し、ドアを開けます。Stop状態の階層化されたサブステートチャートもアニメーション表示されます（図11～13）。

◇ ◇ ◇

このように、アニメーション機能を使って、

- モデルを実行するために必要なオブジェクトが生成されているか
- オブジェクト間のリンクが設定されているか
- ほかのオブジェクトから送られたメッセージを受信して処理できるか
- ステートチャートが正しく遷移するか
- オブジェクト間のメッセージのやり取りが適切

図11 シミュレーション（ドアを開く）

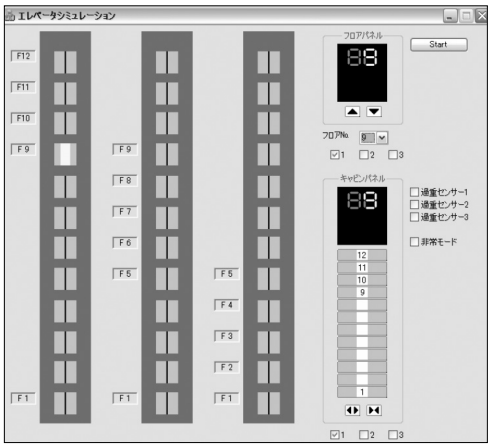
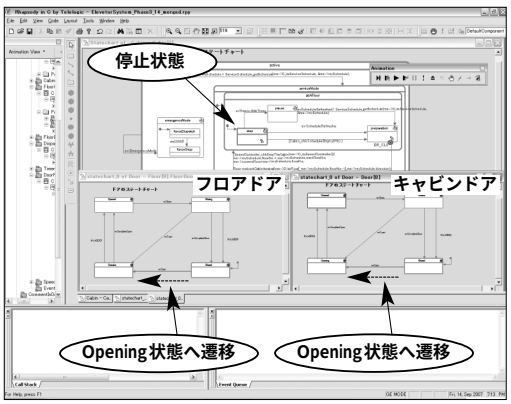


図12 停止状態（サブステートチャート表示）ドアを開く

図13 ドア（キャビンドア、9階フロアドア）が開く



な順序で処理されているか

などを確認できます。つまり、設計モデル上で、システムの動きを確認し、設計仕様どおりに動作するかどうかを確認できます。

本企画の最後に

3回の連載を通じてケーススタディ「エレベータ制御システム」により、組込みシステムをオブジェクト指向およびRhapsodyによるMDDアプローチを紹介してきました。限られた誌面の中でなるべく開発プロセスである手順や成果物、およびRhapsodyによるMDDの具体的な作業とメリットを読者の方に明快に理解していただければと解説を進めてきました。

ただし、実際のシステムを記事のために簡略化した今回のケーススタディ「エレベータ制御システム」でも、けっして作業量や成果物は少なくなく、本企画の作業を通じて数多くの検討時間必要としましたし、分析・設計作業を通じて多くの成果物を作成しています。

そのため、本来であれば読者の方にすべての作業の解説やユースケースモデリング、アーキテクチャ分析・設計や並行性の分析・設計であるタスク分割、タスク間通信、タスク優先度などの技術的なテクニックの詳しい解説を連載で紹介できれば理想的でした。

しかし、雑誌の誌面で詳細なソフトウェア開発上のテクニカルな解説を実施するには相当数のページを必要とします。加えて、初めてオブジェクト指向やMDDによる開発アプローチに触れる方にとって、数多くの成果物や分析・設計手法の解説を詰め込むのは混乱する危険もあると判断しました。そこで、要件定義の作業である上流からRhapsodyによるコード生成とシミュレーションまでの作業を組込みシステム特有の課題をオブジェクト指向技術でどのように進めていくかの「作業の大きな流れ」を解説し、イメージしていただくことを中心に置き解説を進めることにしました。

読者の方の中にはより詳細な解説やすべての成果物を見たいと考えている方も相当数存在すると思います。

またRhapsodyによるMDDによるUMLモデルのシミュレーションによる検証、コードの生成の実際の様子やアニメーションによる「エレベータ制御システム」が実際に要求仕様どおりに動作するデモンスレーションや分析・設計手法、テクニックのさらに詳細な解説の機会を検討しています。

成果物のダウンロード、デモンスレーションや分析・設計手法、テクニックのさらに詳細な解説の機会は、伊藤忠テクノソリューション㈱のWebサイト（http://www.ctc-g.co.jp/）にて、今後ご案内する予定ですので、ご期待ください。

参考文献

[1]『UML2.0クイックリファレンス』／Dan Pilone、Neil Pitman 著／原隆文 訳／オライリー・ジャパン／ISBN4-8731-1284-2

[2]『リアルタイムUML 第2版』／Bruce Powel Douglas 著／渡辺博之 監訳／オージス総研 訳／翔泳社／ISBN4-8813-5979-2

[3]『ユースケースによるアスペクト指向ソフトウェア開発』／Ivar Jacobson、Pan-Wei Ng 著／鷲崎弘宣、太田健一郎、鹿糠秀行、立堀道昭 訳／翔泳社／ISBN4-7981-0896-0

[4]『Java言語で学ぶ デザインパターン入門』／結城浩 著／ソフトバンククリエイティブ／ISBN4-7973-2703-0

[5]『独習UML 第3版』／テクノロジックアート 著／翔泳社／ISBN4-7981-0763-8

[6]『実践ソフトウェアエンジニアリング』／RogerS. Pressman 著／西康晴、榊原彰、内藤裕史 監訳／日科技連出版社／ISBN4-8171-6148-5