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回シリーズの第2回目となる今回は、「システム分析」作業に的を絞って解説を行います。

MDD

2 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II〜システム分析編〜

前回までの内容と今回の記事内容

前回までの確認と注意点

3回シリーズの第1回目であった前回（『組込みプレスVol.5』にて掲載）は、連載の目的／狙いを説明し、ケーススタディである「エレベータ制御システム」の仕様と要件定義の作業から、成果物としてユースケースモデルを中心に見ていきました。

開発プロセス「Harmony」[注1]のような反復型開発を用いて作業を進める場合には、実際にはもっと多くの作業を実施します。また、成果物も数多くのものを作成することになりますが、今回の企画では誌面の都合からポイントを絞って重要な点を中心に解説し、ほかの活動は割愛しています[注2]。

作業を通じて作成した成果物も、反復を繰り返すたびに見直しをかけていきます。これは初期の段階では情報が少ないことや、少し作業を進めてみないと見えてこない部分もあるからです。実際筆者たちも、前回の解説で示した図や表などは必要に応じて修正をかけています。

今回の解説の流れ

第2回目の今回は「システム分析」作業に的を絞って解説しています。ユースケースドリブンによる反復開発を計画したときに、最初の反復開発として適切なユースケースを選び、その中からさらにシナリオを選定します。並行してシステムを構成するクラスを意識して「概念モデル」を作成していきます。

分析レベルのクラス図は、設計レベルと異なり、実装を考慮した設計の詳細化にはこだわりません。開発手法や方法論によって分析作業に求める成果物の完成度は異なりますが、分析は設計をするうえで必要な情報を獲得することです。そのため、クラス図の各クラスの持つ属性や振る舞いは段階的に明らかにしていき、最初から詳細な属性や振る舞いを定義することは少ないことが普通です。分析の主な作業は、最初はクラス間の関連、依存関係などに注目し作業を進めていくことになります。

今回の解説では、作業が終了した完成版の分析モデルを見てもらうというよりは、「クラスを見つけ出す過程」「クラス間の関連を定義していく過程」を理解してもらうことに意識を置いています。ユースケース記述からどのようにクラスを抽出していくのかを理解してください。この点はたいへん重要な部分です[注3]。

分析のもう一つの大きな作業として、並行性の検討があります。ビジネス系のようにリアルタイム性があまり問われないシステムは、ソフトウェアのアーキテクチャをオブジェクトとその集合であるサブシステムを中心に構築が可能ですが、組込みシステムでのリアルタイム性を満たすようなソフトウェアアーキクチャは、静的な構造の視点であるオブジェクトとその集合とは、また異なる視点で検討することが必要です。今回は、エレベータ制御システムで並行性の分析作業をどのように実施しているかを、一つのアプローチとしてお見せしたいと思います。

今回の解説の中で登場する各種作業成果物の多くは、前回と同様に開発環境「Rhapsody」[注4]を用いて作成した実際の作業成果物の一部です。誌面の都合ですべてを掲載できませんので、解説をするうえで必要不可欠な図を選択して紹介していきます。

注1●より詳しくは『組込みプレスVol.3』の「特別企画1　組込みリアルタイムソフトウェア開発のアプローチ[基礎]」をご参照ください。

注2●開発作業やそれ以外に必要な作業のうち、今回の解説で紹介できない作業として、CMMIやPMBOKなどで紹介される活動があります。一例を挙げると、要件管理、リスク管理、構成管理などです。要件管理作業は開発初期から実施することが重要ですが、これには要件管理表あるいは成果物間の依存関係を管理するための「追跡可能性マトリックス」などを利用することになります。また、成果物を構成管理することも必要になります。また品質保証活動として要件定義の作業から仕様書を含む各種成果物の構成管理作業も重要です。プロジェクトの計画立案と進捗管理についても同様です。

注3●本記事では、ユースケースによるソフトウェア開発のアプローチの基本となる考え方を紹介しています。ユースケースのアプローチについては、今日ではいろいろな著者や研究者から提唱されており、著者や研究者によってその内容に多少の違いがあります。また、ユースケース駆動のアプローチの発表者であるIvar Jacobson氏のアプローチも進歩し続けています。最近ではアスペクト指向による考え方を盛り込りこみ、上流からより効果的にソフトウェア開発を実施できるアプローチを提唱しています（参考文献④）。

注4●http://www.ilogix.com/sublevel.aspx?id=53、http://www.ctc-g.co.jp/~EDA/ilogix/rhapsody/03a_rhapsody.html

Chapter 1
ユースケースドリブンによるモデリング作業
シナリオ、シーケンス図、クラス図

シナリオの選別

シナリオについて簡単に説明すると、各ユースケースのユースケース記述には、メインフローと代替フローがあり、シナリオはその組み合わせになります。つまり、ユースケース記述は基本フローや代替フローから構成されていますが、実際にシステムが振る舞う、つまり動作するパターンは、基本フローや代替フローの中で記述されている分岐とその組み合わせになります。

この組み合わせをすべて網羅すれば、そのユースケース記述のすべての振る舞いを把握したことになります。これをすべてのユースケースについて行えば、すべてのユースケースの振る舞いを把握できることになります。

さて、前回作成したユースケース記述[注5]からシナリオを検討します。その後シナリオごとにシーケンス図を作成していきますが、実作業ではシナリオのすべての組み合わせのケースを考えることはあまりしません。組み合わせのパターンが非常に多くなるからです。そこで、重要なシナリオを注意深く検討し、重要でない代替フローなどは省略していきます。

重要か重要でないかの判断

このときの重要か重要でないかいう判断は、システムの振る舞いを検討するときに明確になります。たとえば、複数のシナリオの内容にあまり違いがない場合は、どれか1つのシナリオを分析することで事実上ほかのシナリオも包含されると判断することになります。

重要なシナリオとは、シナリオの内容を正しく処理できないとシステムとしての主要な目的（今回の例題では、目的の階に配車する部分）が満たせなくなるようなシナリオです。

ユースケースの処理の代表的な振る舞いを表現するようなフローを選択します。この時点で詳細を分析しなくても大きな問題にはならないような、オプション的な処理（室内灯を点灯するなど）については、それを選択することでシナリオの数が増えてしまうような場合には省略します。

抜き出したシナリオをもとにシーケンス図を作成しますが、このとき同時にユースケース記述に矛盾や改善するべき点がないかを検討します。シーケンスの図の作成の意味は、後述するクラスの抽出とクラス間の関係の洗い出しだけでなく、同時にユースケース記述の内容の洗練も含んでいます。

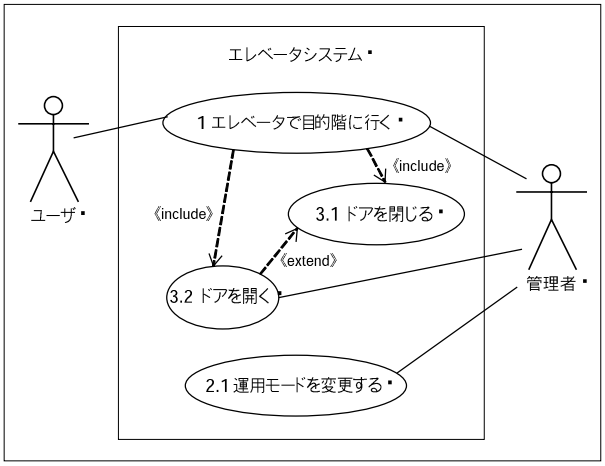
前回掲載した4つのユースケース記述も、今回2回目の連載の記事の作成にあたり、シーケンス図を作成しながら修正が加わっています。

たとえば、ユースケース名やユースケース記述の中で、「エレベータ」という名称は、個々のキャビンを考慮するために「キャビン」と変更してい

注5●『組込みプレス Vol.5』P.200～202。

ます。変更に伴い、ユースケース図のユースケース名やユースケース間の関係についても変更されています。

■図1■ユーザレベルのユースケース図（エレベータで目的階に行く）

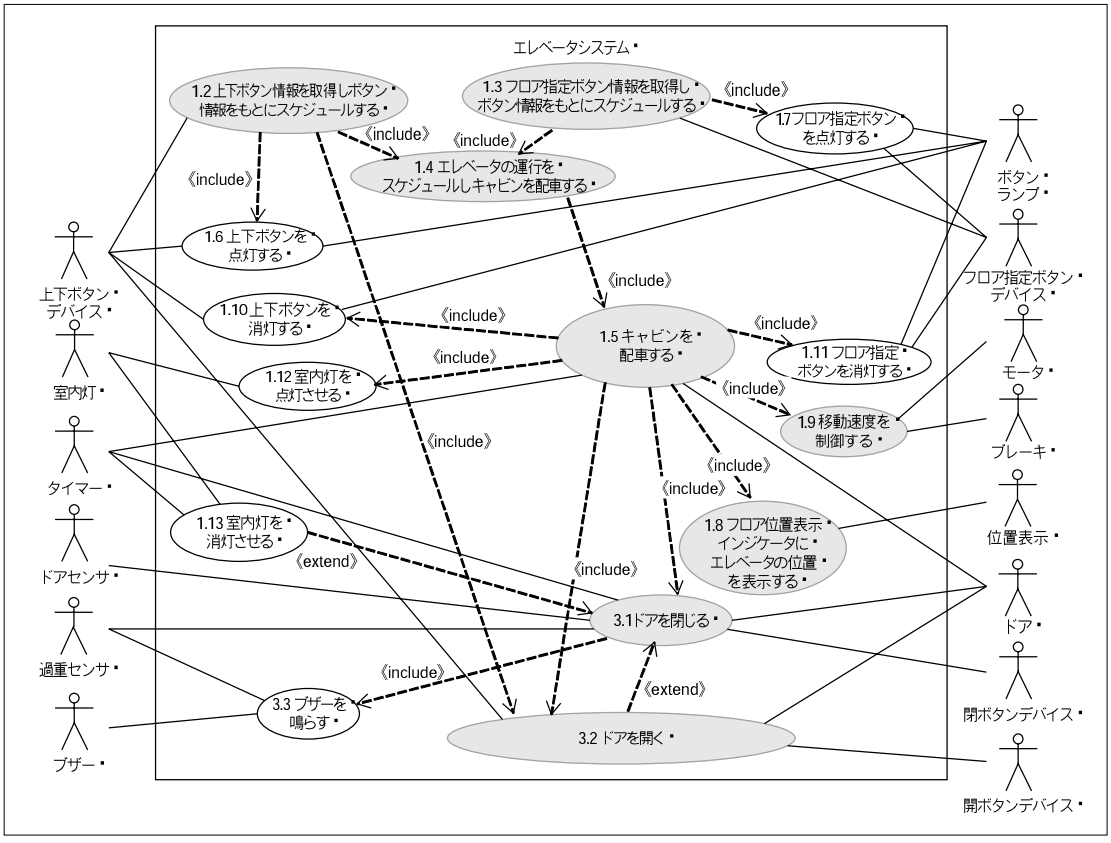


反復型開発におけるユースケースのシナリオの選定

反復型開発を実施することが多い最近のソフトウェア開発では、複数回の反復（以下、イテレーションと呼びます）により開発作業を進めていきます。実際の開発ではプロジェクトの計画時に反復回数と各反復でどのような作業を実施するかの計画を立案します。このときに重要な点は、各反復の中でどのユースケースのどのシナリオを分析、設計、実装、テスト、評価するかを検討しておくことです。今回の企画ではこの反復作業の計画方法については詳しくは説明しませんが、たいへん重要な意味を持ちます。

今回の解説でもエレベータ制御システムの中心的な機能を選択し、作業を進めていきます。その際、ユースケース図で<<extend>>で表現されているユースケースの機能の多くは、今回は対象外に

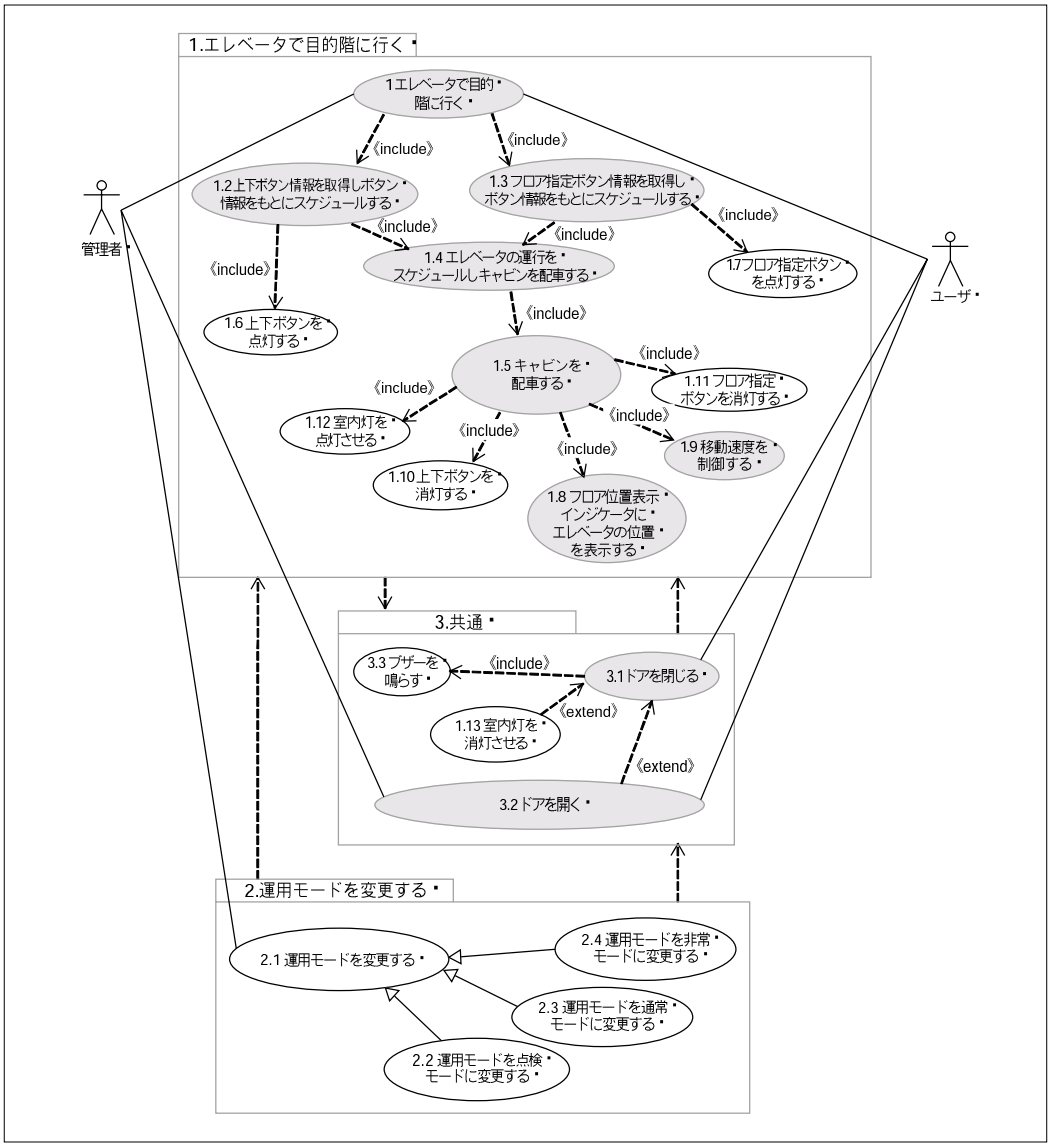
■図2■システムレベルのユースケース図（エレベータで目的階に行く）



なっています。これは、<<extend>>で表現されているユースケースはオプション的な振る舞いが多く、あとから実装しても間に合うからです[注6]。

ユースケース“エレベータで目的階に行く”（図1、図2、図3）は、このシステムの中心となるユースケースなので、最初のイテレーションで今回の作業の対象とします。このユースケースは、“上下ボタン情報を取得しボタン情報をもとにスケジュールする”“フロア指定ボタン情報を取得しボタン情報をもとにスケジュールする”という2つのユ

図3 ユースケースのパッケージ構成



注6● 反復型開発の中では、ユースケースやシナリオの選択が重要な意味を持ちます。開発初期からソフトウェアアーキテクチャを意識して反復計画を進めるからです。開発計画も、ソフトウェアアーキテクチャを効果的に開発していくようにユースケースやシナリオを選択し計画します。つまり、ソフトウェアアーキテクチャの実現で重要なユースケースは初期の反復作業の中で扱い、分析、実装、評価を通して実現していきます。このように、ユースケースやそのシナリオはソフトウェアアーキテクチャの実現上、重要か否かで扱うシナリオを決定していきます。アーキテクチャの検討に重要と判断された場合は、<<extend>>のユースケースも初期の反復で扱うことがあります。

ースケースを包括しており、さらにほかのユースケースも包括しています。シーケンス図のベースとなるシナリオを考えるときには、<<include>>と<<extend>>のパスをたどって、すべてのシナリオの組み合わせを考え、重要なものを抜粋してシーケンス図化します。今回考慮したユースケースの中で、比較的重要なものを図2の中で網掛け表示しています。

今回は、見やすさとわかりやすさのために、ユースケース“エレベータ目的階に行く”を、“上下ボタン情報を取得ボタン情報をもとにスケジュールする”と“フロア指定ボタン情報を取得しボタン情報をもとにスケジュールする”という2つのユースケースに分けて考え、“上下ボタン情報を取得しボタン情報をもとにスケジュールする”のほうを例にとって説明します。

シナリオの検討

それでは実際の作業に進んでいきましょう。ユースケースの選定からシナリオを検討しますが、いくつかポイントがあります。それはユースケース記述の中で<<include>>、<<extend>>のユースケースが登場する場合です。このユースケースを見てみると、次のユースケースを包含していることがわかります。

- 上下ボタンを点灯する
- エレベータの運行をスケジュールしキャビンを配車する
- ドアを開く

また、これらのユースケースはさらにほかにユースケースを包含しているので、ユースケース“上下ボタン情報を取得しボタン情報をもとにスケジュールする”のシナリオには、上の3つのユースケースのシナリオのほかに、

- キャビンを配車する
- 室内灯を点灯させる
- ドアを閉じる
- ブザーを鳴らす
- 移動速度を制御する
- フロア位置表示インジケータにキャビンの位置を表示する
- 上下ボタンを消灯する
- フロア指定ボタンを消灯する

といったユースケースのシナリオを含む可能性があります。すべてのシナリオの組み合わせを考えるのが原則ですが、先に述べたように、すべてを考慮するとシナリオの数が多くなってしまうので、重要なケースに限定して分析していきます。

たとえば、次のようなケースを考えます。

> シナリオ1.2_1
> 押された上下ボタン情報（方向とフロア）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運行をスケジュールする

このシナリオの具体的なフローは、

① 乗客の押した上下ボタン情報を取得する
② 上下ボタンを点灯する
③ 押された上下ボタンの情報から、指定されたフロアに配車するのに最適なキャビンを計算し、そのキャビンの運行スケジュールを更新する[注7]
④ そのキャビンに運行スケジュールが更新されたことを通知する
⑤ そのキャビンのドアの状態を確認する
⑥ 各キャビンの移動速度を制御し、目的の階に配車する
⑦ 目的の階に到着するまで、途中のフロアを通過するたびにフロア位置表示インジケータにキャビンの位置を表示する
⑧ 目的階に到着したことを確認する

注7●今回の仕様では、各階層行きのキャビンは1基ずつですが、ほかのビルに適用する場合も考慮し、配車するのに最適なキャビンを計算する処理を行わせています。

⑨ フロア指定ボタンを消灯する
⑩ 上下ボタンを消灯する
⑪ ドアを開く

となります。

このシナリオを考える作業の中で、あるいは、シーケンス図を作成していく過程で、ユースケース間のシナリオの流れが適切であるか、処理内容に矛盾や抜けがないかを並行して確認していきます。

たとえば、今回の作業の中で、Idleタイマーをセットし、Idleタイマーが30秒をカウントしたあとで室内灯を点灯する部分は、それを処理するユースケースを見直しました。前回のユースケース記述内では、この部分の処理を“キャビンを配車する”（前回は“エレベータ配車する”というユースケース）の中に入れていましたが、必ずドアを閉じたあとに処理されなければならないので、ユースケース“ドアを閉じる”のメインフローの最後に追加しています。この部分以外にも、前回作成したユースケース記述からシナリオシーケンス図を確認し、かなり修正しています。

この確認／修正作業は、分析フェーズにおいて、要求にあいまいな点がないかどうかや、各ユースケースが要求を満たしているかをチェックし、自分たちがどのようなシステムを開発するのかを明確にするために必要な作業です。

<<boundary>>、<<entity>>、

シナリオをベースに作成するシーケンス図は、クラス間の処理の流れを表現するものなので、シーケンス図を作成する前に、ユースケース記述から「バウンダリクラス」「コントロールクラス」「エンティティクラス」というカテゴリを用いてクラス（候補）の抽出を実施します。

今回対象としているユースケースから“上下ボタン情報を取得しボタン情報をもとにスケジュールする”で考えてみましょう。

バウンダリクラス

バウンダリクラスは、システム境界内外のインタフェースに該当するクラスです。たいていの場合、組込みシステムではユースケース図にアクタとして登場しているハードウェアがバウンダリクラスとして表現され、クラス図に登場します。バウンダリクラスには、クラスダイアグラムに<<boundary>>を用いて表現することもあります。

このユースケースでは、上下ボタンデバイスがアクタとして挙げられているので、この名前を使ってバウンダリクラス“上下ボタンデバイス”が考えられます。また、このユースケースが包含しているユースケースに対しても同様に考え、

- 上下ボタンデバイス
- ドア
- フロア表示インジケータ
- モータ
- ブレーキ
- タイマー

がバウンダリクラスとして挙げられます。

エンティティクラス

エンティティクラスは、データを管理するものや、特定の処理の計算をするものなどが取り上げられます。エンティティクラスは<<entity>>を用いて表現します。

エンティティクラスの抽出は、古典的な方法ですが、ユースケース記述の中の単語で「名詞」が候補となります。ただし、名詞として登場する単語には、クラスではなくあるクラスの属性やまったくクラスにも属性にも関係ない単語もあります。

シーケンス図を描きながら検討すれば、クラス（オブジェクト）として抽出される単語がすぐにわかるようになってきます。今回対象としているユースケースでは、

●運行スケジュール
●運用モード
●配車計算

がエンティティクラス(候補)として考えられます。

コントロールクラス

コントロールクラスは、特定のユースケースの中のエンティティクラスを利用して、バウンダリクラスからの要求に対する処理の制御を行うクラスです。<<controller>>を用います。

コントロールクラスの抽出の考え方は、ユースケースごとに（原則として）1つ用意し、そのユースケースで実施する処理の制御を担当させます。コントロールクラスが必要な理由は、処理に関するクラス間（オブジェクト間）の処理の流れのロジックを持たせるためです。「ビジネスロジック」と呼ぶこともあります。

コントロールクラスのオブジェクトにビジネスロジックを持たせなかった場合、処理に関する内容がエンティティクラスやバウンダリクラスに分散し、機能変更の際に多くのクラスに変更が波及しやすくなりがちです。

ただし、シーケンス図を作成して分析を進める中で、ユースケースによっては、コントロールクラスが不要な場合も出てきますし、逆に制御内容が異なる場合には複数のコントロールクラスが必要なときもあります。必要か不要かの判断は、今後の分析を通じて検討します。

このユースケースでは、配車コントローラを用意して、運行スケジュールを管理させるようにします。また、このユースケースが包括しているユースケースにもそれぞれコントロールクラスがあるので、ユースケース“上下ボタン情報を取得しボタン情報をもとにスケジュールする”では、

●配車コントローラ
●キャビン
●速度コントローラ

がコントロールクラスとして考えられます。

Rhapsodyのモデルデータに各クラスを登録する際に、バウンダリクラス、コントロールクラス、エンティティクラスを明示するために、各クラスにステレオタイプ<<boundary>>、<<controller>>、<<entity>>をセットします。

シーケンス図によるオブジェクト間のコミュニケーションの明確化

図4は、上述のシナリオ1.2_1に対するシーケンス図です。このシーケンス図は、シナリオに出てくるほかのユースケースシナリオに対するシーケンス図を組み合わせて作成できます。

最後のドアを開く処理の部分は、このシーケンス図だけを考えたときには、単にタイマーをセットするだけなので、コントローラにその処理を依頼するのではなく、“ドア”にドアタイマーをセットさせています。しかし、タイマーはドアタイマーだけではなくIdleタイマーもあります。

このIdleタイマーの処理は、別のユースケース“フロア指定ボタン情報を取得しボタン情報をもとにスケジュールする”のシーケンス図（図7）で分析されています。ドアが閉じられてからセットされ、さらに運行スケジュールを確認して、そのキャビンが配車をスケジュールされていない、つまり休止状態に移行することを確認しなければなりません。このため、図4のシーケンス図でも、コントローラであるキャビンにドアタイマーをセットさせるように変更します（図5）。

概念モデリング

各シナリオからシーケンス図を作成しますが、そのときに見つけたクラス（正確にはオブジェクト）間で、シナリオの振る舞いを実現するには、クラス（オブジェクト）間でどのようなメッセージのやり取りを実施しながら振る舞いを実現するかを検討します。

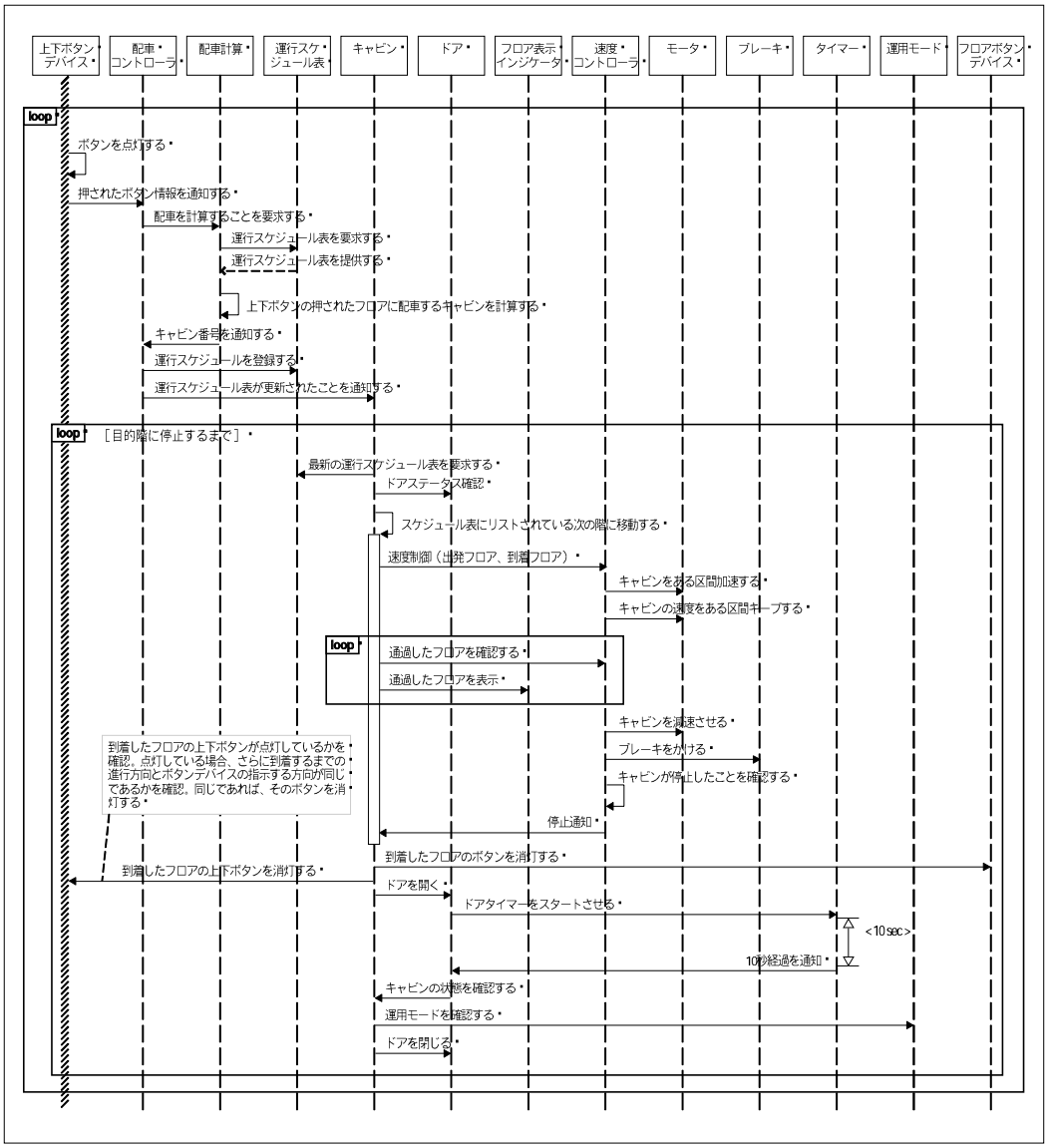
シーケンス図で登場したクラス（オブジェクト）とクラス（オブジェクト）間のメッセージから関連がわかります。このシーケンス図（図4および図5）から、シナリオ1.2_1に対するクラス図が作成できます（図6）。シーケンス図に記述されているクラスと、クラス間のメッセージのやり取りの内容から判断して関連を追加し、適切な関連名をつけます。

ここでは分析としての作業ですので、クラスとクラス間の関連の構造を明確にすることが狙いです。設計的な詳細さは不要です。

ほかのシナリオにも同様にシーケンス図を作成します（図7）。なお、図7からも図8のようなクラス図が作成されます。

繰り返し出現するユースケース“ドアを閉じる”“ドアを開く”についても検討しましょう。この2つのユースケースのある1つのシナリオに対するシ

図4 SD1.2_1：押された上下ボタン情報（方向とフロア）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



ーケンス図が、図9と図10です（この2つのシーケンス図に対するクラス図の紹介は省略します）。

作成したすべてのシーケンス図に対して、図6、図8に相当するようなクラス図を作成し、最終的にそれらをすべてマージしたクラス図を作成します。このとき、Rhapsodyのモデルデータには、各クラス要素やクラス間の関連が登録されています。クラス図をマージするときには、必要に応じて、クラス間の関連の見直しや関連名の洗練が必要になります。また、「全体 － 部分」関係や「汎化 － 特化」関係を定義することも必要に応じて実施します。さらに、クラス間の多重度も検討し、追加します。

Rhapsodyには、パッケージ単位やクラス単位でクラス図に配置するクラスを選択し、さらにそのクラス間のどのような関係をダイアグラム上に表示させるかを設定することで、自動的にクラス図を作成する機能が用意されています。今回は、クラス要素を保存しているパッケージ単位でクラスと、そのクラス間の関連を全体のクラス図表示させるように設定して図11を作成しました。ここで作成された全体のクラス図は、分析レベルのアーキテクチャ図を作成する際のベースとなります。

アーキテクチャ分析

シーケンス図とクラス図ができたあとは、シス

図5 変更後のシーケンス図（SD1.2_1）

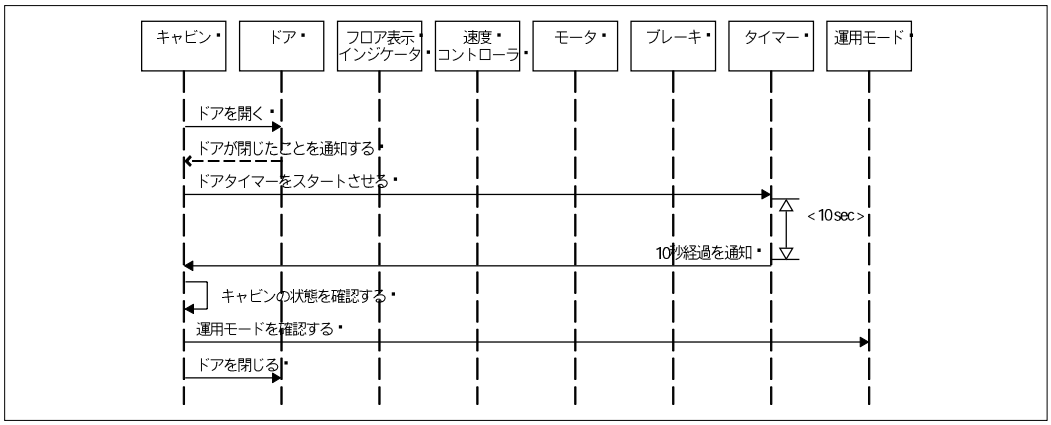


図6 SD1.2_1に対するクラス図

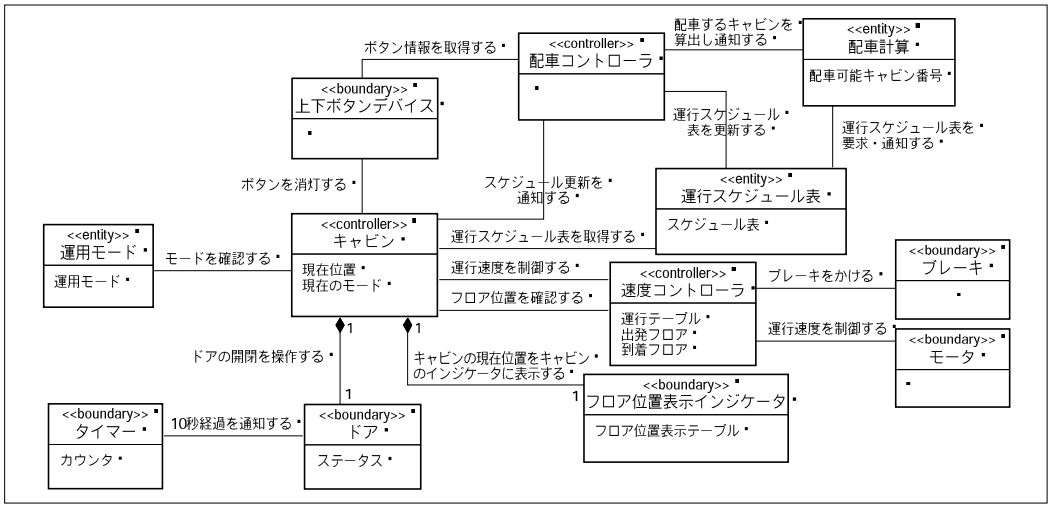
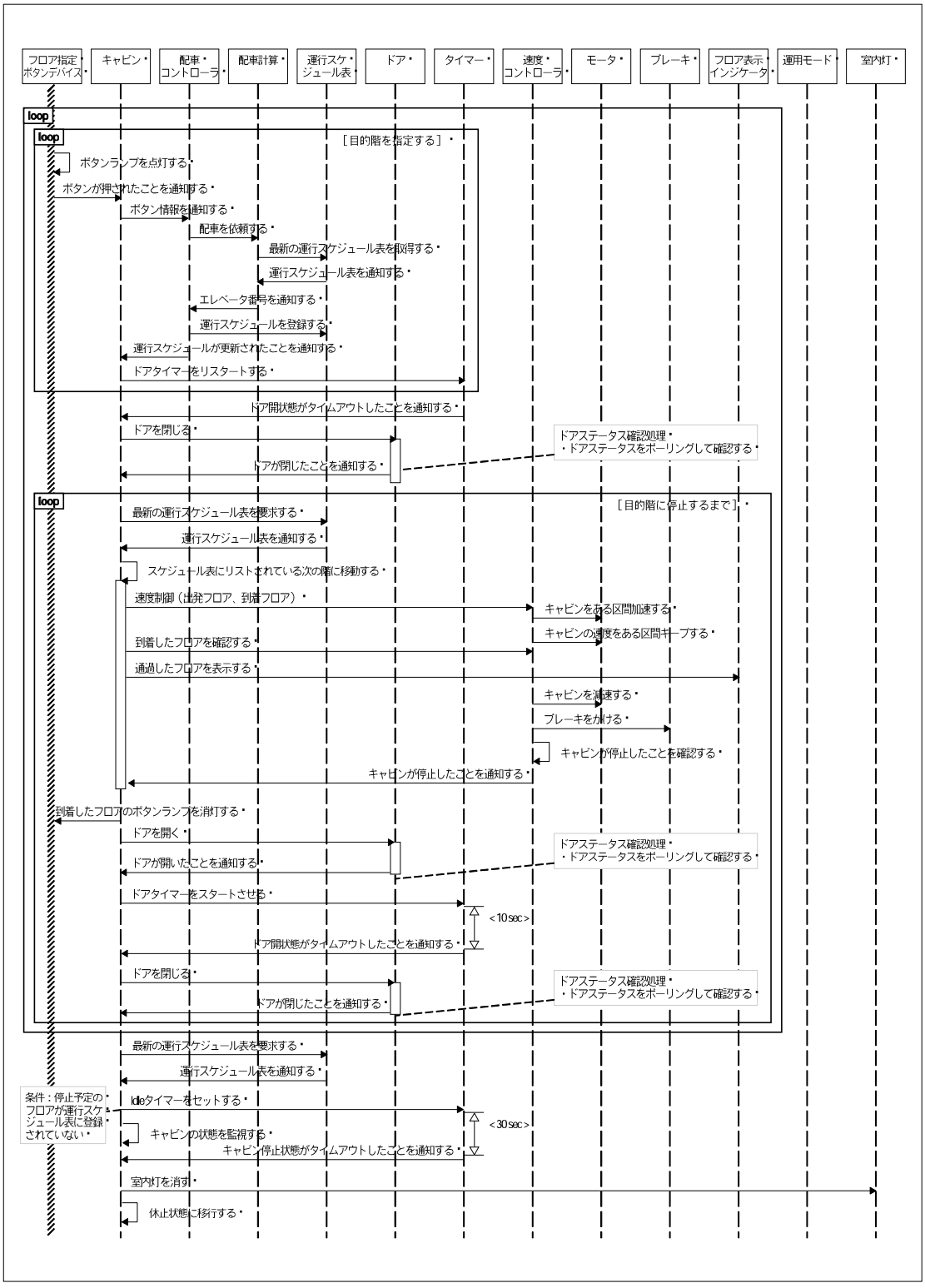


図7 SD1.3_1：押されたフロア指定ボタン情報(フロア)を取得し、点灯させ、取得したボタン情報をもとにエレベータの運行をスケジュールする



テムのソフトウェアアーキテクチャを検討します。ソフトウェアアーキテクチャには多くの考え方があり、画一的には決まりません。

今回は、アーキテクチャ図を作成する際に、全体のクラス図に出てくる各クラスを、バウンダリ、コントロール、エンティティのカテゴリをベースにレイヤ分けし、各レイヤのなかで密接な関係を持つクラス同士をパッケージにまとめます。

異なるパッケージに含まれるクラス間で関係がある場合、そのパッケージに依存関係を表記します。依存関係がレイヤをまたがる場合、インタフェースを定義することで、そのレイヤの中ではほかのレイヤに依存せずにモデリングすることができます[注8]。

図8 SD1.3_1に対するクラス図

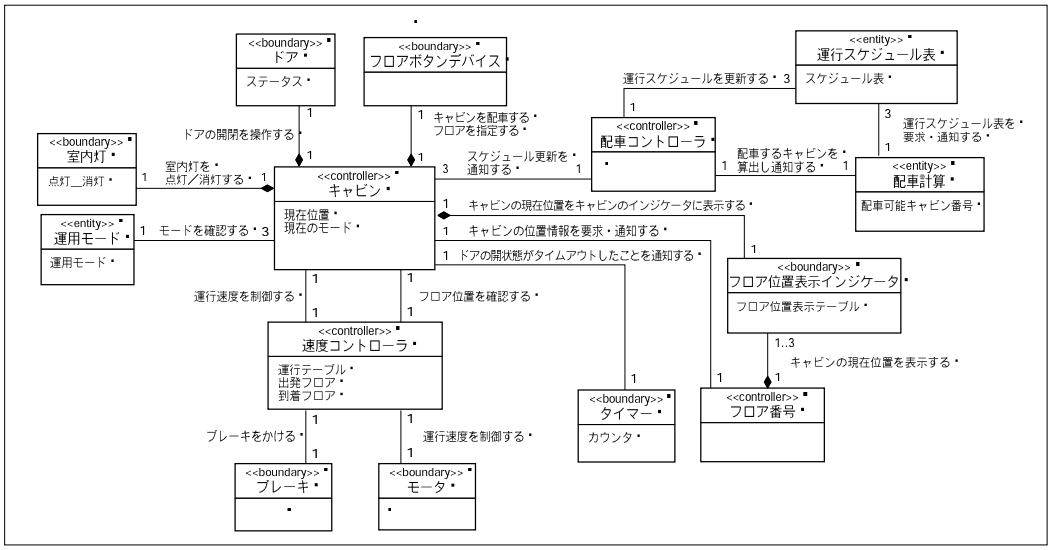


図9 SD3.1_1_1：閉ボタンが押されてドアを閉じるードアを閉じる操作を実行できることを確認してドアを閉じる

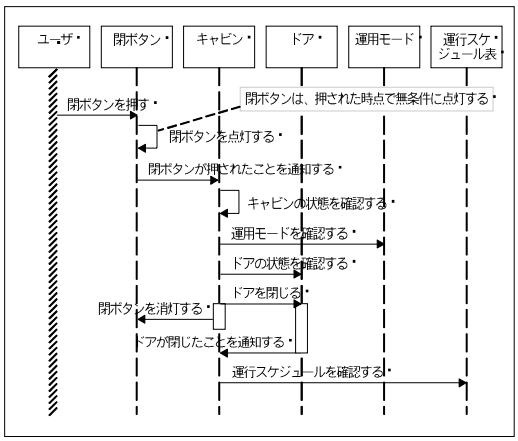
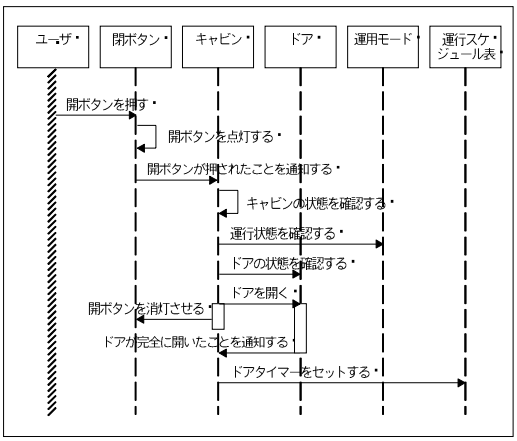


図10 SD3.2_1_1：開ボタンが押されたためドアを開くードアを開く操作を実行できることを確認してドアを開き、ドアタイマーをスタートさせる



注8● アーキテクチャはシステムに求められている要件に依存します。要件変更に柔軟に対応できるように、アーキテクチャの保守性／拡張性を可能な限り考慮しなければならない場合と、そうでない場合では大きくアーキテクチャの構造が異なる場合もあります。また、保守性／拡張性よりも、処理速度のパフォーマンスを最大限に重視した要求であれば、また異なるでしょう。さらに、要件変更がどの要求事項に発生しやすいか、ハードウェアの変更の影響をどこに受けやすいかによってもアーキテクチャは異なるでしょう。このことから、アーキテクチャの構造は画一的に決まるわけではなく、求められている要件や開発者のアーキテクチャの戦略に大きく依存することになります。

●レイヤ構成

各クラス（オブジェクト）をバウンダリ（境界）、コントロール（制御）、エンティティ（実体）に分けることにより、変更頻度の高いインタフェースや機能変更によるシステムの変更を局所化しています。

●サブシステム構成

各レイヤのパッケージ構成ですが、システムのオブジェクトの構成をもとにパッケージ分けします。

図12はここまでの作業から見出したクラスとアーキテクチャ構造ですが、必ずしもこれが正解とは限りません。設計／実装を行い、動作検証を通じてアーキテクチャ構造を評価しながら、クラス、クラス間の関係およびアーキテクチャ構造が適切かを検討します。

組込みシステムでは机上の検討だけでは見えてこない問題点や課題が多いため、実際に動作させての評価が特に重要となります。ここが反復型開発の重要な狙いです。

図11 全体のクラス図（概念モデル）

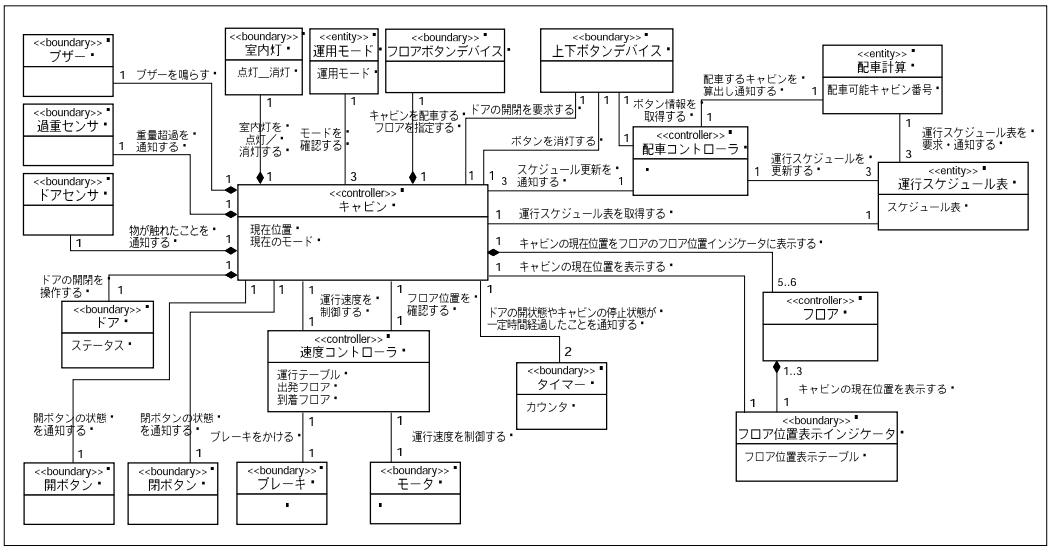
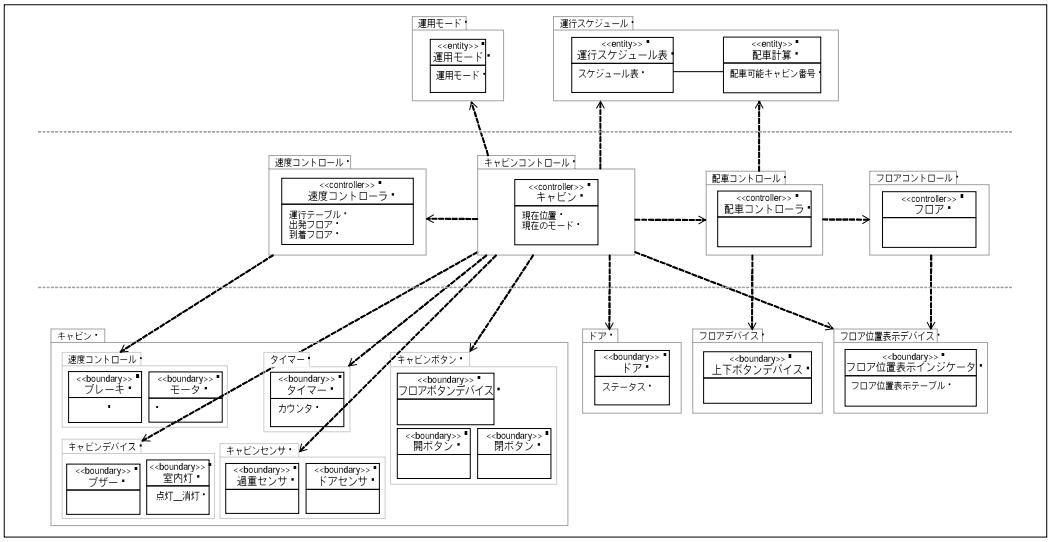


図12 分析レベルのアーキテクチャ図



MDD

Chapter 2 並行性の分析

タスク分割の考え方と実践

タスク分割とタスク間通信

さて、ここからは並行性の分析の検討に話を移しましょう。ビジネス系のシステムではリアルタイム性があまり要求されないため、並行性の分析作業は割愛されています。このため、オブジェクト指向開発の手法や方法論の中には、組込みシステムで求められるような厳しいリアルタイム性について、どのように分析／設計をするのかをまったく扱っていないものが多いのが残念です。

組込みシステムのような開発では、初期からシステムに要求されているリアルタイム性やフォルトトレランス性を適切に把握するために、分析作業が欠かせないものが多くあります。そこでここからは、分析作業としてどのように進めていくのかを紹介していきます。

なお、この並行性の分析の説明は、誌面で説明する都合上、シーケンス図、クラス図、アーキテクチャ図のあとになっていますが、分析作業では並行して実施することに注意してください[注1]。

タスク分割

複数の処理を並行して動作させるために機能分割することを「タスク分割」と言います。一般の構造化プログラミングでの機能分割では生産性や独立性、保守性などが重視されますが、リアルタイム処理を要求されるシステムの場合、特にリアルタイム性についても重視して分割する必要があります。

分割されたタスクには通常優先順位がつけられ、RTOSによるスケジューリングを行うことにより、リアルタイム性の要求を満たすことができます。タスク分割はシステムに要求されているリアルタイム性の要求を満たすために必要となる場合に通常複数のタスクに分割し、複数のタスクの並列動作を通じて、システムの処理を実施します。

タスク間通信

タスク間通信を使用する目的は、タスクとタスクの同期をとるためやタスクからタスクへデータを受け渡すためです。

「タスクとタスクの同期をとる」とは、あるタスクがある状態になった条件で関連するタスクを動かす、または関連するタスクがある状態になるまで待つなどの処理を指します。

タスク間通信の具体的な方法として、イベントフラグ、セマフォ、メッセージキュー、メールボックスなどがあります。どの方法を使用するかは、設計者や対象機器に依存しますが、ガイドラインとしては次のように分けることができます。

注1● リアルタイム性を要求されるシステムに対して、どのようにオブジェクト指向を適用するかを解説した手法／方法論は多数あります。各手法／方法論では、それぞれ独自のアプローチを解説していますが、本特集では比較的リアルタイム性が厳しいシステムに用いられることが多いアプローチを紹介します。

- イベントフラグ：1タスク対1タスク間同期のみの場合
- セマフォ：1タスク対複数タスク間同期またはリソース取得の場合
- メッセージキューやメールボックス：データを受け渡したい場合

また、対象機器のタスクのレスポンスを考慮した方法の選択が必要です。採用した方法のシステムコールの処理時間が、要求される応答時間を満足しているかどうかを詳細に検討する必要があります。タスク間通信設計の注意点としては、タスク分割と一体で設計を行うこと、必ず正常シーケンス、異常シーケンスも考慮することなどがあります[注2]。

タスク分割の考え方

タスクはどのような視点で切り分けるかを考えてみます。OSのタスクスケジューリングやプリエンプションの動作から、タスクに分ける必要があるパターンを考えてみます。

並列処理から見たタスク分割

ある時間帯に「～しながら～する」といった並列の処理が存在する場合、それぞれの処理は待ち合わせや時間間隔によって処理が切り替わり、断続的に処理が進みます。このような場合は並列する処理をタスクにする必要があります。逆に、並列に動作することのない処理をタスクに分ける必要はありません。この場合、関数にしておけばよいでしょう。

「どの処理を並列動作させたいか」の視点でタスクを分割することが必要です。タスク内の処理は優先度が同じであることも分割の指針になります。

優先処理から見たタスク分割

同じ優先度のタスクではFCFS（First Come First Service。後述）でタスクが切り替わるため、優先的に実行させたい処理が必ずしも優先されるとは限りません。プリエンプション（優先）させたいタスクが実行中のタスクと同じ優先度ではそのタスクが待ち状態になるまで切り替わりません。言い換えれば、プリエンプションさせたい処理はタスクにし、高い優先度を付けます。また、優先度を分けたい処理はタスクにします。

一般に、何らかのシステム異常への緊急処理や高速に処理しなければリアルタイム性を損ねるような処理は最高位の優先度を付け、システムのログをとるような、空き時間に処理すればよいものは低い優先度を付けます。

最下位の優先度のタスクは、ほかにどのタスクも実行されていないときに実行されるためにアイドルタスクと呼ばれます。何も機能しないループ処理をアイドリング用に置く場合もあります。

●優先処理の注意点

優先度の低いタスクは、どんなに重要な処理を実行中であろうと中断される場合があること、また、より高いタスクが実行されている限りは下位のタスクは永久に実行されないことに注意し、優先度を設定する必要があります。

以上が基本的な考え方ですが、タスク分割やタスクの優先度は、最終的にいろいろなことを検討してバランスを考えて決定します。

たとえば、論理的には並行に動作することが求められない場合でも、開発の行いやすさ、動作環境の都合などからタスクを分割したほうがよいと判断した

注2● 要件から求められる応答時間を満たすために、タスクの処理時間は最悪処理時間のケースを見積もります。この見積もりを可能な限り正確に行うのが、難しい作業の一つです。厳しいリアルタイム性を求められるシステムの開発に利用されるスケジューリング理論と、それをサポートするツールもいくつか存在します。このようなツールを用いれば、タスク分割の候補や各タスクがデッドラインを満たすか否か、優先度の検討もある程度可能になります。しかし、ツールを用いた場合でも、計算に必要な入力値は、利用者が事前に過去の実績値か机上で算出しておく必要があります。また、要件から求められる応答時間は仕様書に明示的に記述されていることは少なく、システム要求分析の中で、開発者が明らかにしていく必要があります。そのためにも、事前に十分に「イベント分析」を実施し、システムの入出力のイベントと求められる処理、応答時間、重要度などを検討しておくことが不可欠となります。

場合にはタスクを分割することがあります。その逆に、論理的にはタスクを分割したほうがよい場合でも、設計上、分割しないと判断する場合があります。

単純に並行性の視点からタスクの分割を実施すると、タスクの数が多くなることがありますが、安易なタスク分割は危険です。タスクの数が多くなると欠点も多くなります。たとえば、タスク間の通信が多くなると開発が複雑になったり、タスク間通信のオーバヘッドでシステムの処理速度が低下することがあります。これは利用するRTOSのシステムコールを利用することになるからです。また、タスク間の優先度を検討して適切な優先度を判断することも、タスクが多くなれば難しくなります。さらに、タスク間で共有するデータの排他制御などの問題もあります。

タスク間の優先度の問題と共有するデータの排他制御を正しく設計しないと、優先度逆転などのデリケートな不具合に悩まされることになります。

一般的なタスクのスケジューリング方式

●FCFS方式(First Come First Service)

最初に実行待ち状態に入ったタスクから実行します。実行待ちのタスクは、先に待ち状態に入ったタスク順に待ち行列を作ります。

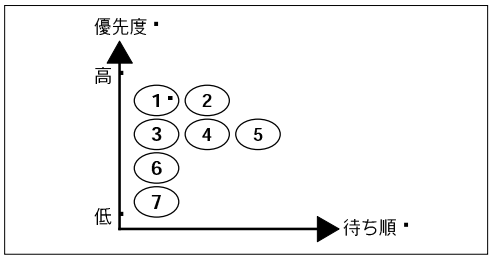
●優先度方式

各タスクに優先度を付けておき、優先度の高いものから先に実行します。特に、時間ごとにタスクを切り替え、順番に実行を繰り返す方式を「ラウンドロビン」と言います。ほかにも、デッドラインが最も近いタスクからCPUの実行権を割り当てる「デッドラインドリブン方式（Deadline Driven Scheduling Algorithm)」や、各タスクの処理時間が既知である場合に処理時間の短いものから順にCPUを割り当てる「SPT方式（Shortest Processing Time first)」などがあります[注3]。

日本でこれまで最もよく利用されているμITRONなどの組込みOSでは、優先度方式とFCFS方式が併用されています。優先度の高いタスクが実行され、同じ優先度のタスクはFCFSで実行されます(図1)。実行の待ち行列にあるタスクを一定時間ごとにループ実行させてラウンドロビン方式で実行させることもできます。

実行可能タスクの待ち行列がある場合、数字の順番に実行タスクが切り替ります。

図1 タスクの優先度



タスク候補の抽出

今回は、並列処理から見たタスク分割方法を主とし、優先度を考慮してタスク分割を行うことにします。ここではクラスやオブジェクトといった括りを考えずに、これまでのシステム分析（ブロック図、ユースケース図、シーケンス図、イベント分析表など）からタスク分割していきます[注4]。

●処理の性質別に、それぞれの処理を分類する

次の4種類の処理に分類します。

- 表示：出力系
- 処理：割り込み系
- 状態：データ系
- タイマー

注3● デッドラインドリブン方式にはいくつか種類が存在します。代表的なところでは、EDF(Earliest Deadline First)、LL(Least Laxity)などです。しかし、これらのスケジューリング方式を実装するには、原則利用するRTOSがタスク属性としてデッドラインを指定でき、これらのスケジューリング方式をサポートしていなければなりません。現在までのところ、組込み用の商用RTOSではサポートされていないのが現状です。

注4● これは、明示的並行性アプローチ（explicite concurrent approach）と呼ばれることがあります。

タイマー処理は、無条件にタスク候補として分類しました。その他の処理は、関連するデバイスとの通信の種類によって分類しています。分類した処理の中でそれぞれ機能をまとめます（図2）。

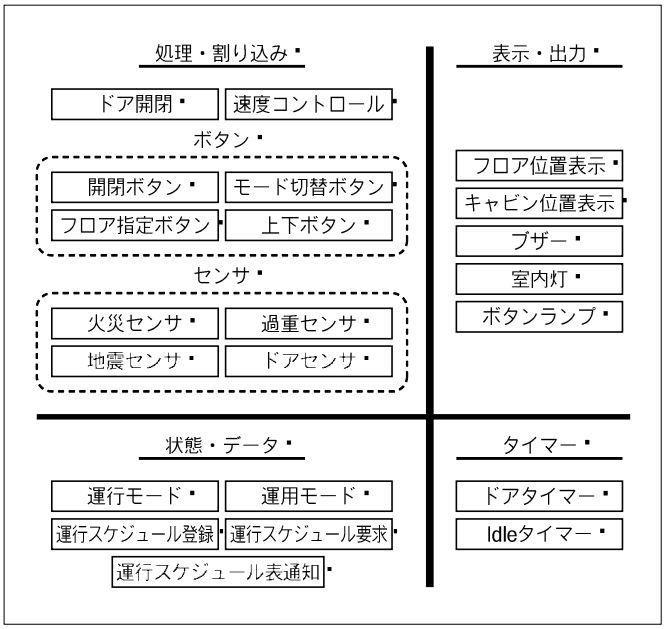
簡単にタスク分割の視点を補足しておきます。

システムが外部との入出力を行う処理（I/O処理）は、タスク分割の候補です。I/O処理を分けないと、ほかの処理（タスク）に影響を与えてしまうからです。I/O処理は外部から入ってくるイベントのタイミングに依存し、処理時間も外部のハードウェアに依存します。

複雑な計算、画像処理、暗号処理など時間がかかる処理もタスク分割の候補です。リアルタイムシステムであっても、本当にリアルタイム性を求められる処理は限定されていることが多いものです。厳しいリアルタイム性を求められる処理については、タスクを分割して高い優先度を付与するようにしておき、ほかの処理は要件から求められるパフォーマンスを満たすようにすれば問題ないことが多いものです。

ほかにもタスクを制御する処理が必要になる場合がありますが、このような処理もタスク候補の一つです。この場合、システムの状態やデータに応じて、タスク間の制御や呼び出す処理を判断します。

図2 機能分類図



タスク間コミュニケーションとタスク優先度の分析

タスク分割（第1段階）

機能分類図で分類した機能をそれぞれタスクにします。その際、各タスク間の関係も記述します。データや処理の流れの方向のみに注目し、タスク間通信の方法については後ほど考えることにします（図3）。

分割したタスクは次のとおりです。

- コントロールタスク
- ボタンタスク
- センサタスク
- ドアタスク
- 出力タスク
- 速度コントロールタスク

タスク分割（第2段階）

第1段階で分割したタスクそれぞれについて、さらに分割する必要があるか考えます。コントロールタスクとして分割したタスクの中の処理では、キャビンとフロアの処理の並列性が求められますので、キャビンタスク、フロアタスクとして分割します（図4）。

コントロールタスクを2つに分割したので、その他のタスクとの関係を再考します。構成を変更する必要があるタスク候補として次のタスクが考えられます。

- ボタンタスク
- 出力タスク
- ドアタスク
- センサタスク

タスク分割（第3段階）

構成を変更する必要がありそうなタスクに対し、以下のような変更を加えます。

①センサタスク

センサタスクについては構成の変更などは行いません。

②ボタンタスク

ボタンタスクもキャビンとフロアの並列性が求められますので、キャビンボタンタスクとフロアボタンタスクに分割します。

③出力タスク

まず、フロア用とキャビン用があるフロア位置表示をタスクとして分割します。

図3 タスク分割（第1段階）

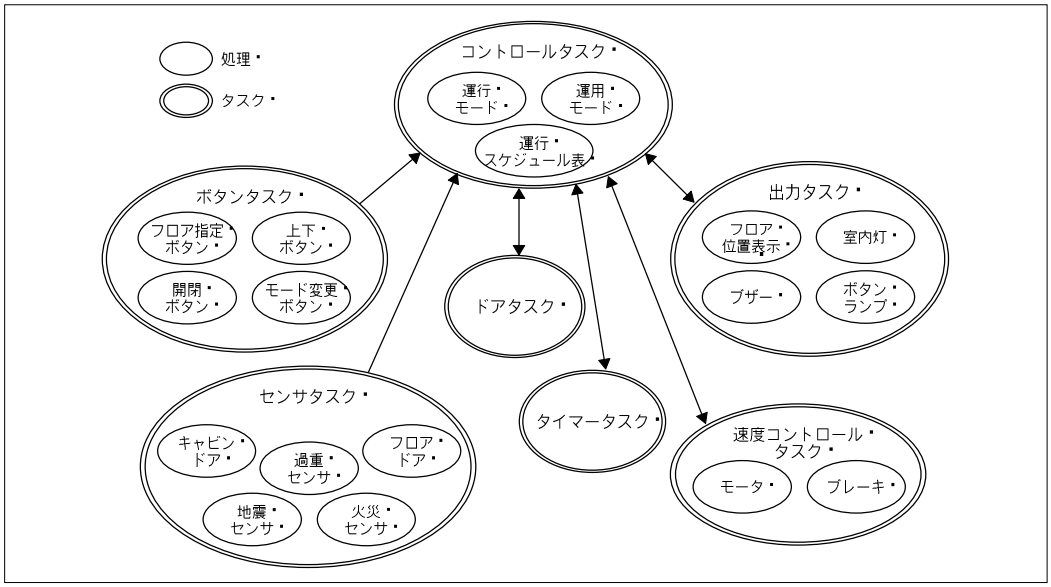
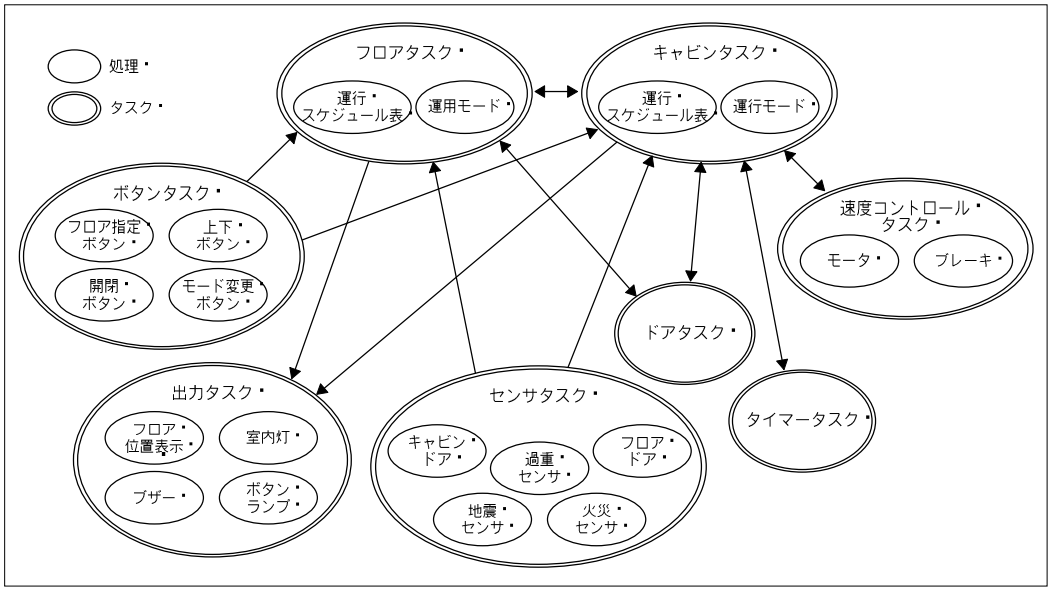


図4 タスク分割（第2段階）



出力タスクの残りの処理で、ボタンランプ処理は、キャビンボタンタスク、フロアボタンタスクにそれぞれ持たせます。

残りの室内灯処理、ブザー処理はキャビンタスクに持たせます（図5）。

タスク分割（最終段階）

各タスク間の通信方法を考慮します。データの受け渡しがあるタスク間はメッセージに、それ以外はイベントにしています（図6）。

最終的なタスク分割の結果も決して最適ではありません。冗長な部分やもっと細分化したほうがよいところもあると思います。

ここで行ったタスク分割は、これまで分析して抽出したクラスやオブジェクトに対して直接関連しているわけではありません。このあとに行う設計段階で、タスク分割とクラス、オブジェクトのマッピング作業を行うことになります。この点については次回に解説を予定しています。組

図5 タスク分割（第3段階）

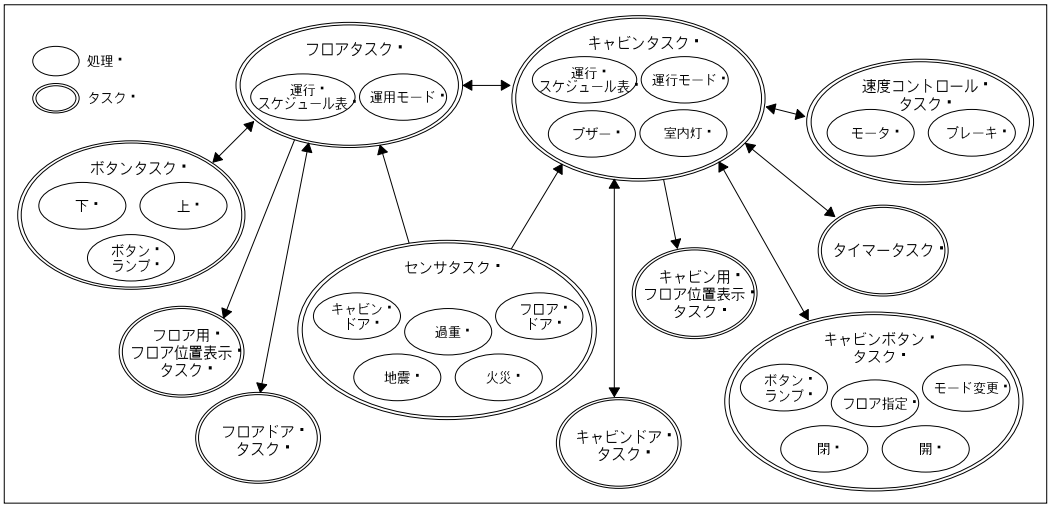
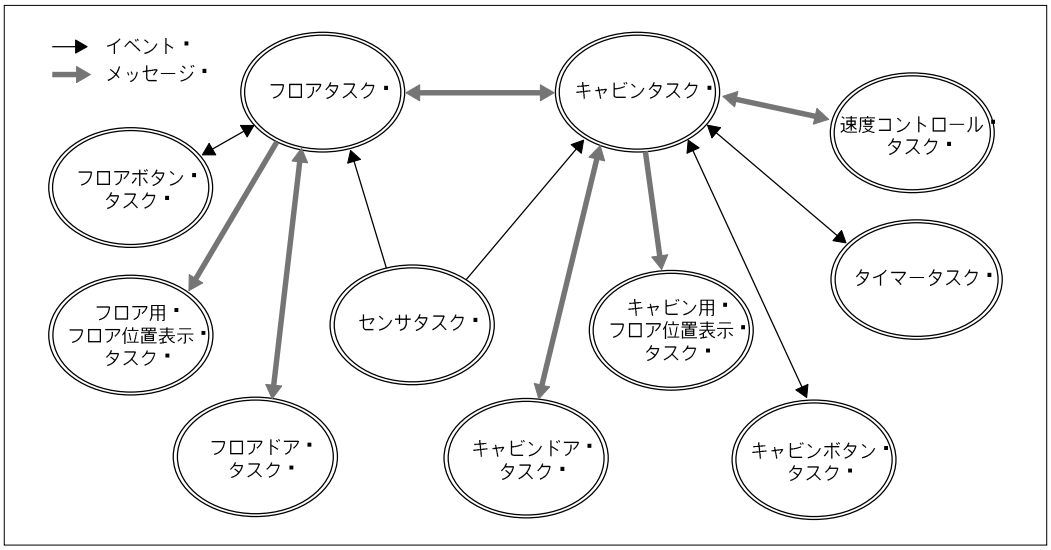


図6 タスク分割（最終段階）



MDD Chapter 3 今回解説したユースケース記述

今回の分析では、ユースケース記述に基づいていくつかのシーケンス図を作成しています。以下に、今回の作業で参照した重要なユースケース記述を紹介します。

ユースケース記述 エレベータで目的階に行く

ユースケース番号 1
ユースケース名 エレベータで目的階に行く
目　的 エレベータで目的階に行く
アクタ ユーザ、管理者
説　明 目的階に行く
事前条件 エレベータシステムが、通常モードで動作していること
メインフロー
1. 昇り降りを指定する。<<include>>1.2上下ボタン情報を取得しボタン情報をもとにスケジュールする
2. 配車されたキャビンに乗車する
3. 目的階を指定する。<<include>>1.3フロア指定ボタン情報を取得しボタン情報をもとにスケジュールする
4. 目的階で降車する
<<include>>ユースケースリスト
1.2 上下ボタン情報を取得しボタン情報をもとにスケジュールする
1.3 フロア指定ボタン情報を取得しボタン情報をもとにスケジュールする
制　約 止まらない階がある
シナリオリスト
S1_1 ユーザが上下ボタンを押し、到着したキャビンに乗車し、フロア指定ボタンを押して、目的の階に行く。

ユースケース記述 上下ボタン情報を取得しボタン情報をもとにスケジュールする

ユースケース番号 1.2
ユースケース名 上下ボタン情報を取得しボタン情報をもとにスケジュールする
アクタ 上下ボタンデバイス
目　的 ユーザが押した上下ボタン情報を取得する
説　明 ユーザが押した上下ボタン情報を取得し、そのボタンの押されたフロアにキャビンの配車をリクエストする
事前条件 エレベータシステムが動作している、かつ、通常モードである
メインフロー
1. 乗客が押した上下ボタン情報を取得する
2. 上下ボタンを点灯する。<<include>>上下ボタンを点灯する
3. 上下ボタンが押された階にキャビンを配車するようスケジュールする。<<include>>エレベータ運行をスケジュールする(A1)
代替フロー
A1. 上下ボタンが押された階にキャビンが停車中の場合：そのキャビンの<<include>>ド

アを開いて処理を終了する

事前条件　上下ボタンを押した乗客がいるフロアにキャビンを配車する

<<include>>ユースケースリスト

1.6 上下ボタンを点灯する

1.4 エレベータ運行をスケジュールする

3.2 ドアを開く

シナリオリスト

S1.2_1　押された上下ボタン情報(方向とフロア)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。

S1.2.3　押された上下ボタン情報(方向とフロア)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運行をスケジュールした結果、上下ボタンが押された階にキャビンが停止中だったため、そのままドアを開く。

ユースケース記述　フロア指定ボタン情報を取得しボタン情報をもとにスケジュールする

ユースケース番号　1.3

ユースケース名　フロア指定ボタン情報を取得しボタン情報をもとにスケジュールする

アクタ　フロア指定ボタンデバイス

目　的　乗客が押したフロア指定ボタンの情報を取得し、そのフロアに配車する

説　明　乗客が押したフロア指定ボタンの情報を取得し、そのフロアにキャビンを配車する

事前条件　エレベータシステムが動作している、かつ、通常モードである

メインフロー

1. 乗客が押したフロア指定ボタンの情報を取得する
2. フロア指定ボタンを点灯する。<<include>>フロア指定ボタンを点灯する
3. 指定された階にそのキャビンを配車するようスケジュールする。<<include>>エレベータ運行をスケジュールする

事前条件　そのキャビンを、乗客の指定した階に配車する

<<include>>ユースケースリスト

1.7 フロア指定ボタンを点灯する

1.4 エレベータ運行をスケジュールしキャビンを配車する

シナリオリスト

S1.3_1　押されたフロア指定ボタン情報(フロア)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。

ユースケース記述　エレベータ運行をスケジュールしキャビンを配車する

ユースケース番号　1.4

ユースケース名　エレベータ運行をスケジュールしキャビンを配車する

アクタ　上下ボタンデバイス、フロア指定ボタンデバイス

目　的　乗客からの要求に応じて、各キャビンの配車を計算する

説　明　ユーザの上下ボタン、フロア指定ボタンの操作から、3基のうちから最適なキャビンを、指示のあったフロアに配車するようにスケジュールする

事前条件　配車がリクエストされたこと

メインフロー

1. 押されたフロアボタンあるいは上下ボタンの情報から、指定されたフロアに配車するのに最適なキャビンを計算し、そのキャビンの運行スケジュールを更新する(A.1)
2. 運行スケジュールが更新されたキャビンに、運行スケジュールが更新されたことを通知する
3. <<include>>キャビンを配車する

代替フロー

A1. 非常モードの場合：最寄りのフロアに配車する
　　点検モードの場合：無視する。

事前条件　リクエストされた階に1基のキャビンを配車する

<<include>>ユースケースリスト
1.5 キャビンを配車する
制　約　各キャビンには、止まらない階がある
キャビンがフロアに停止中以外には、ドアは開かない（制約1）
運行ルールを参照すること
シナリオリスト
S1.4_1 押されたフロアボタンあるいは上下ボタンの情報から、指定されたフロアに配車するキャビンを決定して、そのキャビンの運行スケジュールを更新し、そのキャビンにスケジュールが更新されたことを通知してキャビンを配車する。

ユースケース記述　キャビンを配車する

ユースケース番号　1.5
ユースケース名　キャビンを配車する
アクタ　タイマー、ドア
目　的　キャビンを配車する
説　明　キャビンを、指示のあったフロアに配車する
事前条件　キャビンの運行スケジュール表に配車が予定されている
メインフロー
1. ドアが開いていたら<<include>>ドアを閉じる(A1)
2. 各キャビンの<<include>>移動速度を制御し、目的の階に配車する
3. <<include>>フロア位置インジケータに各キャビンの位置を表示する
4. 目的のフロアに到着したことを確認する
5. <<include>>フロア指定ボタンを消灯する
6. 到着したフロアの上下ボタンの状態を確認し、<<include>>上下ボタンを消灯する
7. <<include>>ドアを開く
代替フロー
A1. 室内灯が消灯している場合には、<<include>> 室内灯を点灯させる
事前条件　リクエストされた階に1基のキャビンを配車する
<<include>>ユースケースリスト
1.9 移動速度を制御する
1.8 フロア位置表示インジケータにエレベータ位置を表示する
1.11 フロア指定ボタンを消灯する
1.10 上下ボタンを消灯する
1.12 室内灯を点灯させる
3.1 ドアを閉じる
3.2 ドアを開く
制　約　止まらない階がある
キャビンがフロアに停止中以外には、ドアは開かない（制約1）
シナリオリスト
S1.5_1 ドアが閉じられていることを確認して、キャビンを目的階に移動させ、移動中に(キャビン内と各フロアの)フロア位置表示インジケータにキャビンの位置を表示し、目的の階に到着したら、到着フロアに対応するボタンと到着したフロアの上下ボタンを消灯してドアを開く。

ユースケース記述　ドアを閉じる

ユースケース番号　3.1
ユースケース名　ドアを閉じる
アクタ　ユーザ、管理者、ドア、ドアセンサ、過重センサ、閉ボタンデバイス、タイマー
目　的　ドアを閉じる
説　明　センサやユーザ、メインコントローラの要求に応じてドアを閉じる
事前条件　エレベータシステムが動作していて、非常モードではないこと
•トリガー:
閉ボタンデバイスからの割り込みイベントがあった。もしくは、ドアタイマーが10秒経過をキャビンに通知した
メインフロー
1. キャビンの状態を確認して、ドアを閉じる操作を実行できることを確認する(A1、A4)

2. Switch(運用モード)
   - Case1 運用モードが非常モードである場合:ユーザからの要求は受け付けない。処理終了
   - Case2 運用モードが点検モードである場合:ユーザからの要求は受け付けない。処理終了
   - Case3 運用モードが通常モードである場合:処理を続行する
3. ドアが閉まっている、もしくはドアが閉まる途中でないことを確認する(A2)
4. 過重センサからの入力を判断して、ドアを閉じる操作が実行できることを確認する(A3)
5. ドアを閉じる(A5、A6、A7)
6. If(運行スケジュールが空)
   Idleタイマーをセットし、Idleタイマーが30秒経過した時点で、Idleタイマーをリセットし、<<extend>>室内灯を消灯する
   Else 処理終了

代替フロー

A1. キャビンがフロアに停車中以外の場合:ドアを閉じる要求を無視する。処理終了
A2. ドアを閉じる要求を無視する。処理終了
A3. 過重センサが、積載量が重量制限を超えていることを感知した場合:積載量が重量制限以下になるまで、ドアを閉じる処理を中断して<<include>>ブザーを鳴らす
A4. 閉ボタンが押された場合:閉じるボタンを点灯させる
A5. ドアセンサが物に触れたことを感知した場合:ドアを閉じる動作を中断して、<<extend>>ドアを開く
A6. ドアタイマーがセットされている場合:ドアタイマーをリセットさせる
A7. 閉ボタンが点灯している場合:閉ボタンを消灯する

<<include>>ユースケースリスト

3.3 ブザーを鳴らす

<<extend>>ユースケースリスト

1.13 室内灯を消灯させる
3.2 ドアを開く

シナリオリスト

S3.1_1_1 閉ボタンが押されてドアを閉じる:閉ボタンを点灯し、ドアを閉じる操作を実行できることを確認してドアを閉じ、閉ボタンを消灯する。
S3.1_1_2(ドアタイマーから10秒経過が通知されたため)キャビンからドアを閉じる指示がある:ドアを閉じる操作を実行できることを確認してドアを閉じる。
S3.1_1_3(ドアタイマーから10秒経過が通知されたため)キャビンからドアを閉じる指示がある:ドアを閉じる操作を実行できることを確認してドアを閉じ、運行スケジュールが空であることを確認してIdleタイマーをセットし、Idleタイマーが30秒経過した時点で、室内灯を消灯する。
S3.1_2 ドアを閉じる途中で、積載量が重量制限以上になる。
S3.1_3 ドアを閉じる途中で、ドアセンサが物に触れたことを検出する。

ユースケース記述 ドアを開く

ユースケース番号 3.2
ユースケース名 ドアを開く
アクタ ユーザ、管理者、開ボタンデバイス、ドア、上下ボタンデバイス、ドアセンサ
目 的 ドアを開く
説 明 センサやユーザの要求に応じてドアを開く
事前条件 エレベータ電源が入っている
トリガー 開ボタンデバイス、上下ボタンデバイス、ドアセンサから割り込みイベントがあった、もしくはフロアに到着した

メインフロー

1. キャビンが移動中か停止中かを確認して、ドアを開く操作を実行できることを確認する

（A1、A2）

2. ドアが開いているもしくは、開く途中ではないことを確認する(A3)
3. ドアを開く(A4)
4. ドアが完全に開いた状態になったことを確認し、ドアタイマーをスタートさせる(A5)

代替フロー

A1. 開ボタンが押された場合：開ボタンを点灯する

A2. キャビンがフロアに停止中以外の場合：ドアは開かない。処理終了

A3. ドアを開く要求を無視する。処理終了

A4. Switch(開ボタン)

- Case1 点灯している場合：開ボタンを消灯する
- Case2 継続して押されている場合：開ボタンがはなされるまで、点灯し、ドアを開いた状態に保つ

A5. ドアタイマーがすでにスタートしている：処理終了

制　約　ドアが開いている途中は、ドアを閉じる操作は無効である

シナリオリスト

S3.2_1_1 開ボタンが押されたためドアを開く：開ボタンを点灯し、ドアを開く操作を実行できることを確認してドアを開き、開ボタンを消灯してドアタイマーをスタートさせる。

S3.2_1_2 キャビンがフロアに到着したためドアを開く:ドアを開く操作を実行できることを確認してドアを開き、開ボタンを消灯してドアタイマーをスタートさせる。

S3.2_1_3 ドアセンサから、ドアを開く指示がある:ドアを開く操作を実行できることを確認してドアを開き、ドアが完全に開いた状態になったらドアタイマーをスタートさせる。

参考文献

①『リアルタイムUML 第2版 オブジェクト指向による組込みシステム開発入門』Bruce Powel Douglass著／渡辺博之 監訳／㈱オージス総研 訳／翔泳社／2001年／ISBN4-8813-5979-2

②『ユースケース実践ガイド－効果的なユースケースの書き方』／Alistair Corkburn著／ウルシステムズ㈱、山崎耕二、矢崎博英、水谷雅宏、篠原明子 訳／ウルシステムズ㈱ 監修／翔泳社／2001年／ISBN4-7981-0127-3

③『ユースケース入門－ユーザマニュアルからプログラムを作る』／Doug Rosenberg、Kendall Scott著／長瀬嘉秀、今野 睦、テクノロジックアート 訳／ピアソンエデュケーション／2001年／ISBN4-8947-1377-2

④『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／鷲崎弘宜、太田健一郎、鹿糠秀行、立堀道昭 訳／翔泳社／2006年／ISBN4-7981-0896-0

まとめ

今回は「システム分析」作業について解説しました。本文中でも触れましたが、分析は設計を行ううえで必要な情報を修得する重要な作業です。要求仕様から、いかにシステム分析を行うかの過程を理解してください。

さて、次回（『組込みプレス』Vol.9にて掲載の予定）はシステム設計／実装について解説していきます。要求／分析モデルから設計モデルへのマッピング、シミュレーション技術の紹介、モデルからのコード生成の解説を予定しています。