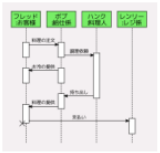
Class 7 - 7

開発設計

UML 概要

- **統一モデリング言語**[Unified Modeling Language, UML]は、ソフトウェアエンジニア開発の世界において、ビジネスユーザーを始め、システムを理解しようとするあらゆるユーザーが理解できる視覚化のための共通言語を打ち立てることを目的のために作られたモデリング手法です。
- UML は、ソフトウェアの可視化、統一を実現させ、説明書に対する理解力を向上させる効果があり、システム分析の可視化によりコミュニケーションが効率化できる効果があります。



UML 2.0 における変更

- UML は継続的に改良されています。アジャイルを含め、開発のより多くの側面に対応するよう、UML の仕様を拡張したものが UML 2.0 で、使いやすさや実装、導入を簡素化するため、UMLを再構築し、改良することを目的として作られました。
  UML 図については、以下のような点が変更されています:
  - 構造モデルと行動モデルの間の統合強化、
  - 階層の定義とソフトウェアシステムのコンポーネントとサブコンポーネントへの分類が可能に、
  - UML 2.0 では図の数が 9 から 13 に拡大。

UML とは

Unified Modeling Language の略語です。日本語では「**統一モデリング言語**」と呼ばれています。

システムの振る舞いや構造をオブジェクト指向で分析したり設計したりする際、図を用いることで視覚的に把握できるようになり、効果的に表現できます。しかし、その図の描き方が人によって違っては困るので、標準規格として作られた図の記法（モデリング言語）が UML です。

UML は 2021 年現在、バージョン 2.0 が公開されており、有名な「**クラス図**」「**シーケンス図**」等を含む 14 種類の図（ダイアグラム）が定義されています。

UML を使って何をするのか

オブジェクト指向でシステムの仕様を分析、設計、記述するのに使います。

すでに存在して動いているシステムであれ、これから開発するシステムであれ、コンピューターシステムにはさまざまな側面があります。

たとえば「ユーザーがシステムを通じてどんな行動ができるのか」といった「ユースケース」と、「開発者はどのサーバー上にどんなコンポーネントをアップロードするのか」という「配置」は、同じシステムの全く別の側面です。

UML で定義されている 14 種類の図は、これらシステムのさまざまな側面を記述するのに適した図法を持っています。

設計者はそれらの図を用いて、既存システムの仕様を分析して記述したり、これから開発するシステムの仕様についてチームで議論しつつ構築し、記述します。

UML を使用するメリット

システムを記述するための図法は UML 以外にも存在しますが、もっとも広い範囲をカバーしていて、もっとも普及しているのが UML です。

システムの構造や振る舞いが UML で記述してあれば、数年先になって担当者が変わり、誰も仕様がわからないといった事態は防げるでしょう。特に、ユースケース図、シーケンス図といったシステムの振る舞いを記述する図はプログラム開発者以外にも理解しやすいため、開発者、営業担当者、ユーザー、マネージャー等、システムに関わるさまざまな役割の人の間で仕様を共有するのにも役立ちます。また、UML が規定するさまざまな図は、設計者や開発者が「こういう側面で、こういう点について気にかけるべきである」という方法論にも通じています。UML を学ぶことは設計の方法論を学習する上でも有用です。

UML の視点

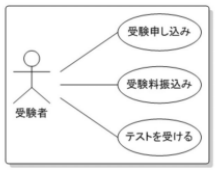
- システムの開発においては、以下の 3 つの全体的なシステムモデルに重点が置かれます:
  1. **機能**: ユーザーの視点からシステムの機能性を記述する***ユースケース図***が該当します。
  2. **オブジェクト**: オブジェクト、属性、関連と操作の観点からシステムの構造を記述する***クラス図***が該当します。
  3. **動的**: ***相互作用図***、***ステートマシン図***や***アクティビティ図***は、システムの内部動作を記述するために用いられます。

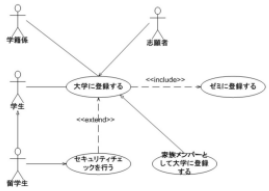
UML 図一覧

名　前	説　明
ユースケース図	システムのユーザ要求を表現
クラス図	部品（クラス）の内容と関係を表現
オブジェクト図	部品（オブジェクト）の関係を表現
シーケンス図	時間軸のオブジェクトの流れを表現
ステートマシン図	オブジェクトの状態や遷移を表現
アクティビティ図	処理の流れを表現
パッケージ図	グループ化した単位の関係を表現
コミュニケーション図	オブジェクト間のメッセージのやり取りを表現
タイミング・チャート	時間軸の変化を表現
コンポーネント図	部品の構造を表現
コンポジット・ストラクチャ図	クラスやコンポーネントの内部構造を表現
配置図	システムの物理的な配置を表現

ユースケース図

- **ユースケース図**[Use Case Diagram]は、システムの**ユーザー**（別名アクター）と**システム**との相互作用を表すことができます。
- ユースケース図を作成するためには、一連の特別な記号とコネクターが必要となります。





ユースケース図の役割

- 効果的なユースケース図は、チームにおける以下の内容の議論や表現に役立ちます：
  - システムやアプリケーションが人、組織や外部システムと相互作用するシナリオ、
  - システムやアプリケーションがこれらの実体（アクター）の達成する内容を支援する目標、
  - システムのスコープ。

ユースケース図の目的

- UML ユースケース図は、以下の目的に適しています：
  - システムとユーザーの相互作用の目標を示す。
  - システムの機能要件を定義し、整理する。
  - システムのコンテキストと要件を指定する。
  - ユースケースにおけるイベントの基本的な流れをモデル化する。

ユースケース図は、システムで何ができるかをユーザー目線で表現する図解術です。

システムを利用する人の目線で、具体的なシステムを利用する場面を、視覚的に図示することがユースケース図を使う目的です。

下記の図のようにとてもシンプルな図であり、一目でシステムの機能やシステム範囲を理解することができます。

ユースケース図は要求定義フェーズで主に使われます。

要求定義フェーズでは、顧客の要求を分析しユースケース図などで視覚化します。そうすることで、顧客と要求を明確にすることができます。

ユースケース図は、ユーザーの視点でシステムの動作やシステム範囲を見ることができるため、顧客とのコミュニケーションで大いに活躍します。

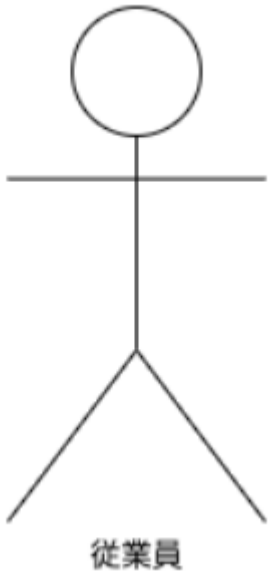
ユースケース図の構成要素

- **アクター**: システムと相互作用を行うユーザーを指します。検討対象のアプリケーションやシステムとやり取りを行う人、組織や外部システムなどがアクターとなり得ます。アクターは、データを生成または消費する外部オブジェクトである必要があります。
- **システム**: アクターとシステム間の特定のアクションや相互作用のシーケンスを指します。システムはシナリオと呼ばれる場合もあります。
- **目標**: 大半のユースケースの最終結果を指します。この目標に到達するまでのアクティビティとバリアントを表現できるのが優れたユースケース図の条件となります。

ユースケース図の基本コンポーネント

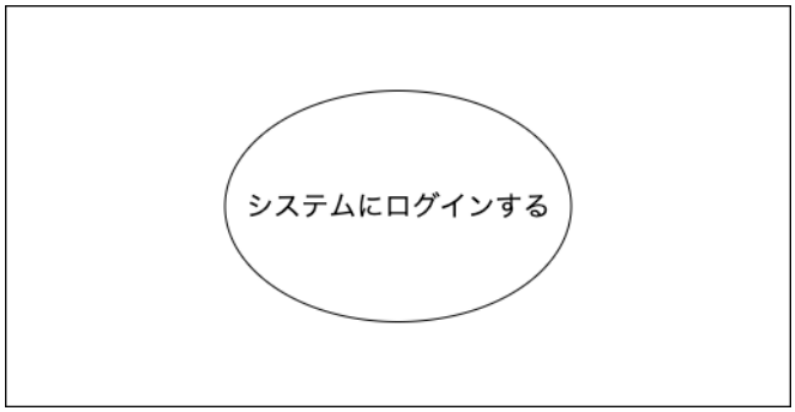
- **ユースケース**: 水平方向に長い楕円形で、ユーザーの有するさまざまな利用例を示します。
- **アクター**: 棒人間の形状で、実際にユースケースを利用する人を表します。
- **関連**: アクターとユースケースを接続する線です。
- **システム境界枠**: ユースケースにつきシステムのスコープを設定するボックスです。このボックス外のユースケースはすべて、対象のシステムのスコープ外であるとみなされます。
- **パッケージ**: 異なる要素をグループ化するために使える UML 図形。コンポーネント図と同様、これらのグループはファイルフォルダーとして示されます。

・アクター



アクターはアプリケーションと何らかの関わりを持つ存在でアプリケーションの外に位置します。
人型の図形が書かれていることからも分かる通り、アプリケーションを実際に操作するユーザーはもちろんアクターです。
しかし、アクターは必ずしも人とは限りません。
例えば、アプリケーションに何らかのデータを送信する別のアプリケーションもまた、ユースケース図ではアクターとして表されます。

ユースケース



ユースケースは、アクターが実際に行う事柄です。
最もイメージしやすいのは、上記のシステムにログインするなどの、アクターがアプリケーションに対して行う操作そのものでしょう。
しかし、アクターがアプリケーションを使わずに行う事柄もまたユースケースです。
アプリケーションでは行わないことを表現することもまた、ユースケース図の役目であるためです。

システム境界

ユースケースを囲うように書かれた外枠は、システム境界です。

- 境界線の中に、アプリケーションを使うユースケース
- 境界線の外に、アプリケーションを使わないユースケース

が存在します。

ユースケース図の書き方

ユースケース図の書き方は以下の 4 つのステップで構成されています。

アクターを洗い出す

まずはユースケース図に登場するアクターを洗い出しましょう。

粒度を意識すると難しくなるので、最初はとにかく箇条書きで書き出すことから始めましょう。

粒度については洗い出してから協議して決めます。

アクター別のユースケースを洗い出す

次にアクター別のユースケースを洗い出しましょう。

ユースケースは「〇〇を〇〇する」という、シンプルな書き方が良いです。

エクセルなどで箇条書きにすると頭が整理されます。

ユースケース図の構成と粒度を決める

次に、ここまでで洗い出したアクターとユースケースをもとに、ユースケース図の構成と粒度を決めます。

顧客が理解しやすいように、粒度が大きい大枠のユースケース図と粒度をブレイクダウンした詳細ユースケース図の 2 種類で構成することがおすすめです。

ユースケース図に落とし込む

ここまでくれば後はユースケース図に落とし込むだけです。手を動かしてユースケース図を書いていきます。

「アクター」、「ユースケース」、「システム境界」を書きましょう。

せっかくなので、みんなで書いてみましょう

Drow.io を紹介

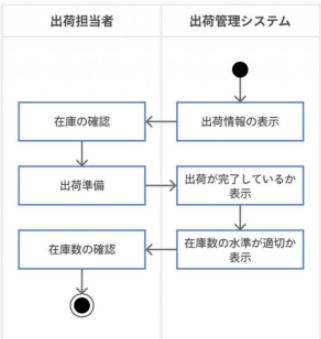
https://app.diagrams.net/

ユースケース図紹介

https://it-koala.com/usecasediagrams-1832

アクティビティ図

- **アクティビティ図**[Activity Diagram]とは、連続する「実行」の遷移、つまり一連の「手続き」を表現するための図です。
- ある事象の開始から終了までの機能を実行される順序にしたがって記述します。
- 状態マシン図が実体の状態遷移を表すのに対し、アクティビティ図では実体の制御の流れを描写します。



アクティビティ図の役割

- アルゴリズムのロジックを示す。
- UML ユースケースで実行される手順を説明する。
- ユーザーとシステムの間の業務プロセスやワークフローを図示する。
- 複雑なユースケースを明確化し、プロセスの簡素化と改善を行う。
- メソッド、関数、操作などのソフトウェアアーキテクチャの要素をモデル化する。

アクティビティ図の基本コンポーネント

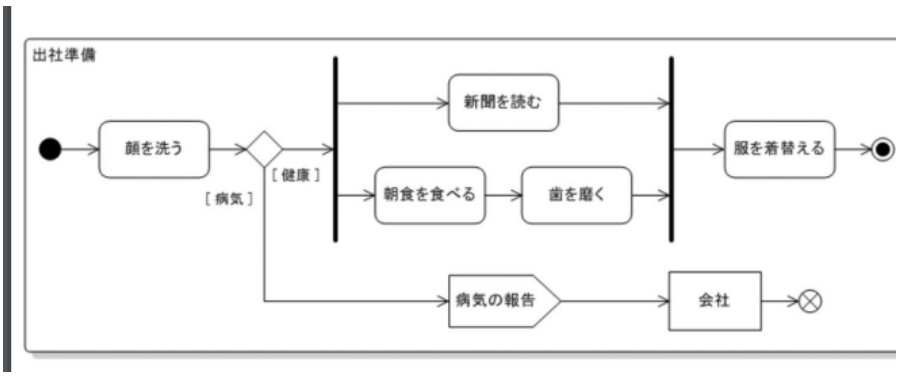
- **アクション**：ユーザーまたはソフトウェアが所定のタスクを実行するアクティビティの手順を指します。アクションは角丸の長方形で示されます。
- **判断ノード**：フロー内の条件分岐を意味し、ひし形で示されます。単一の入力と複数の出力を含みます。
- **制御フロー**：コネクターの別名で、図内の手順間のフローを示します。
- **開始ノード**：アクティビティの開始を意味し、黒い丸で示されます。
- **終了ノード**：アクティビティの最終段階を意味し、白い枠付きの黒い丸で示されます。

次へ

要素	表示形式	意味
初期ノード	●	スコープ内で開始を表します。
最終ノード	◉	スコープ内で終了を表します。
アクションノード	制御名	制御を表します。
デシジョンノード / マージノード	◇	条件によるフロー分岐（デシジョンノード）もしくは、複数のフローの合流（マージノード）を表します。
フォークノード / ジョインノード	━━	複数のフローが非同期に実行される（フォークノード）、もしくは、複数の非同期処理が終了する（ジョインノード）ことを表します。
データストア	<< datastore >> データストア名	データーを保持するための記憶領域や装置を表します。

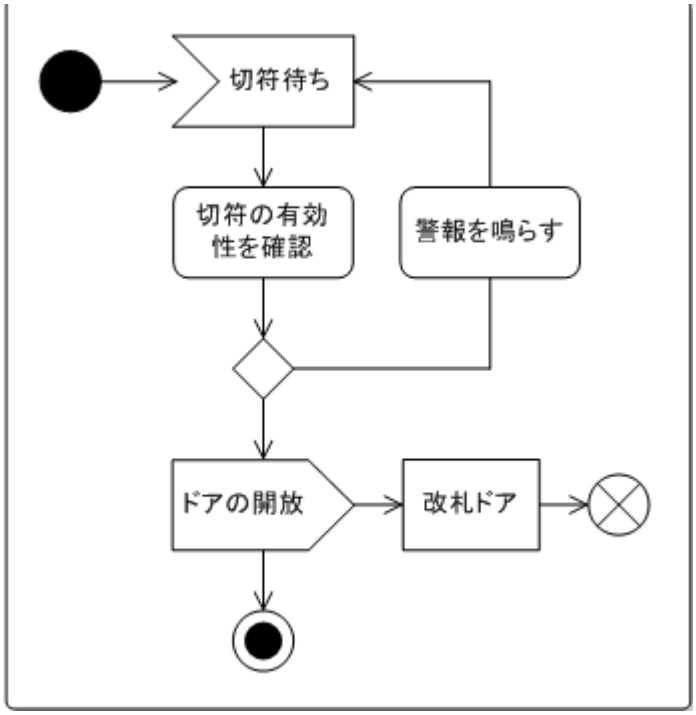
アクティビティ図は必ず開始状態から何らかの終了状態へ、矢印で示された手順に従って実行を行います。 下の図はある会社社員が起床して出社するまでの手続きを示した例です。

【要件定義】まず朝起きて顔を洗います。健康ならば新聞を読みながら朝食を食べ、歯を磨き、そのあと服を着替えて出社します。もし体調が悪ければ「病気である」と会社に連絡して処理終了です



練習問題

自動改札機は、利用者から切符の投入を待ちます。もし切符が入れば、「その切符が有効かどうか」を確認し、次に改札機のドアを開けて一連の動作を終了します。有効でない切符であるならば、ドアを開放することなく再び切符待ち状態になります。



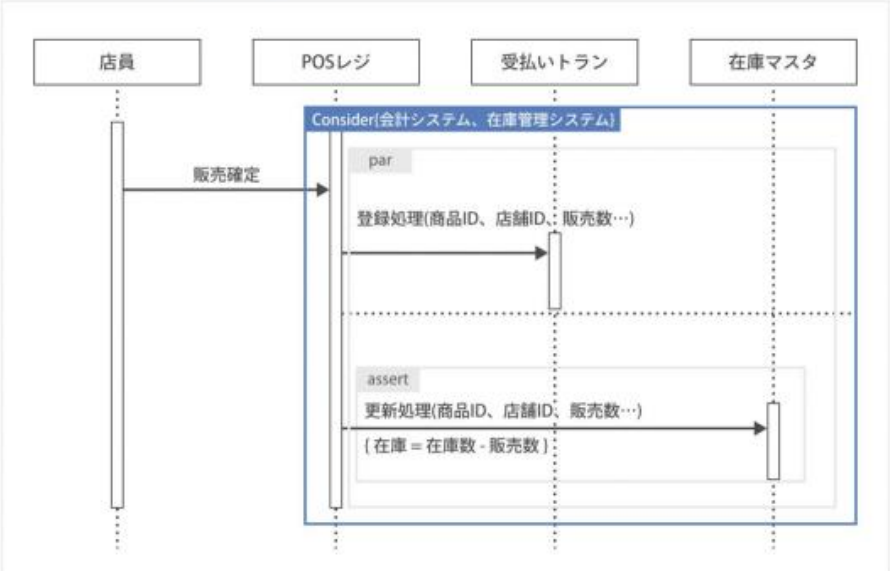
シーケンス図

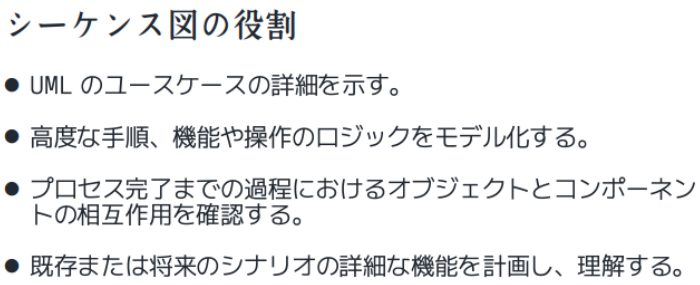
- シーケンス図[Sequence Diagram]は、相互作用を経時的に示すものです。
- プログラムの処理の流れや概要について、具体的には**クラスやオブジェクト間のやり取り**を**時間軸に沿って**、図で表現します。
- UML の中では「*相互作用図*」の1つに位置付けられています。

シーケンス図とは「プログラムの処理の流れや概要」を設計する際に使われます。オブジェクト指向のソフトウェア設計においては、グローバルスタンダードの設計手法と言っても過言ではありません。プログラムの処理の流れや概要について、具体的には「クラスやオブジェクト間のやり取り」を「時間軸に沿って」、図で表現します。UML の中では「相互作用図」の 1 つに位置付けられています。なお、シーケンス図はシステム設計時のみならず設計書の無い既存システムの分析（リバースエンジニアリング）にも使われることがあります。

1. メッセージと呼ばれる矢印で各オブジェクト間の応答を表し、縦軸（上から下）を時系列として応答の順序を表現しています。
2. これにより、ある機能（例では在庫一覧）を実現する各オブジェクトが時間に沿ってどのように相互作用しているかがわかります。

シーケンス図の例





プログラムの処理概要が整理できる

システムが巨大化・複雑化するほど実行手順が増えていきます。仕様書無しは論外ですし、文字ベースの仕様書にしても読み解くのに時間がかかります。また、記述漏れやミスの発見が難しいのも難点です。

シーケンス図は視覚的に、共通の記述ルールにもとづいてシステム処理の流れを把握することができます。ロジックの流れが明確になるので、プログラムの概要および処理手順について、早く正確に理解することができます。

仕様レビューと顧客へのエビデンス

共通のルールに基づいて記述されるシーケンス図は、システムの仕様レビューを効率的に実施するのに役立ちます。また、意外なことにクライアントへの説明にもシーケンス図は有用です。

グラフィカルにシステムの処理手順を示しているシーケンス図は、非エンジニアや IT について詳しくない人でも比較的理解しやすいです。

クライアントとシステムのイメージを共有しておくことは、後のトラブルを防ぐ意味でも重要です。設計初期の段階でクライアントと成果物に求める役割を明確化し、開発終盤での仕様変更リスクを下げましょう。

保守・追加開発時に仕様が確認できる

シーケンス図が活躍するのはシステム開発の時だけではありません。システム納品後の保守運用フェーズにおいてもシーケンス図は重宝します。

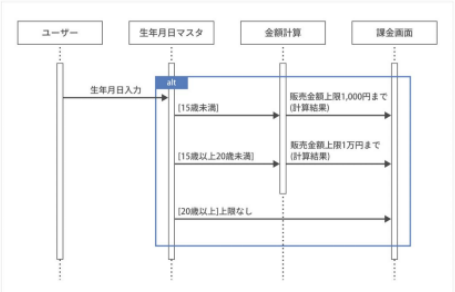
保守運用担当者の多くは、開発者とは別人です。トラブル発生時にコードを 1 から解読するのはとても非効率です。シーケンス図があればシステムの仕様を早期把握し、解決に向けての施策をスムーズに実施することができます。また同じ理由で、システムを仕様追加する際にもシーケンス図は有効です。

シーケンス図の適用場面

- **利用シナリオ**：システムを使用しうる方法を示す図が利用シナリオです。システムのあらゆる利用シナリオのロジックを想定できているかを確認する上で適した方法です。
- **メソッドのロジック**：UML シーケンス図でユースケースのロジックの検討に用いるのと同様に、あらゆる機能、手順や複雑なプロセスのロジックの検討にシーケンス図を利用することができます。
- **サービスのロジック**：異なるクライアントが使用する高次のメソッドとしてサービスを捉える場合に、そのサービスをマッピングする上でシーケンス図が役立ちます。

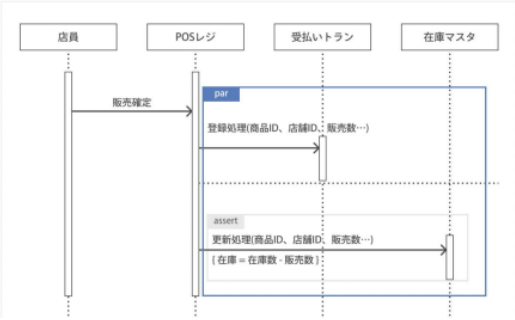
条件分岐

- **条件分岐**（alt）は複合フラグメントを使用します。「[]」（ガードという括弧）内に分岐条件を記載し、各処理を点線で区切って記載します。



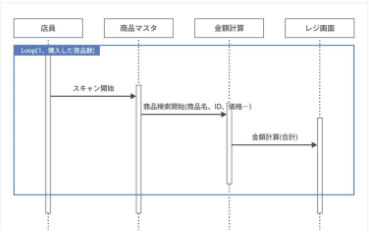
並列処理

- **並列処理**（par）はその名の通り、１つのアクションに対して並列で処理が実行される場合に使用します。



ループ処理

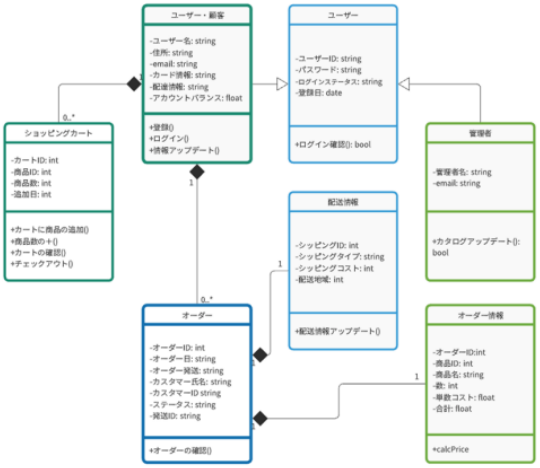
- 複合フラグメント「**ループ処理**（loop）」も名前通り、ループ処理するシステムを表すシーケンス図に使用されます。
- ループ条件の書き方は、「loop[開始条件、終了条件]」です。



LightHouseIT

クラス図

- **クラス図**[Class Diagram]は、ソフトウェアアーキテクチャの文書化を目的としてソフトウェア技術者の間で広く用いられています。
- モデリングの対象となるシステム内に存在すべき構成要素を図式化したもので、*構造図*の一種です。



クラス図は「システムの静的な構造・関係性を視覚的に表現するための図」です。

視覚的に表現、というのがポイントです。たとえばシステム開発の仕様書がすべて文字ベースだと、読み解く人はもちろん作成する人も大変な労力です。さらに文字ベースの仕様書だと、システムの抜け漏れにも気付きにくいです。

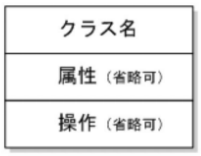
文字ベースの仕様書に対して、クラス図はシステム間の静的構造・関係性を図で表現したものです。記述ルールも統一されており、汎用性および保守性に優れています。

クラス図の役割

- 複雑度にかかわらず、情報システムのデータモデルを図式化する。
- アプリケーションの概略図の全体的な概要について理解を深める。
- システムに特有のニーズを視覚的に表現し、その情報を組織全体に伝達する。
- 記述された構造を対象とした、プログラムや実装を要する特定のコードをハイライトした詳細な図を作成する。
- システムで使用され、後に構成要素間で渡される実体型を実装に依存しない形で説明する。
- システム全体を可視化で表現できる。

クラス図の基本的な構成要素

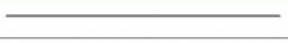
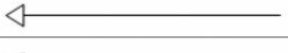
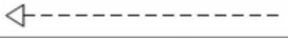
- 上段：クラスの**名前**が含まれます。分類子の場合でも、オブジェクトの場合でも、このセクションは常に必須です。
- 中段：クラスの**属性**が含まれます。クラスの性質の説明に用いるセクションです。クラスの特定のインスタンスを記述する際にのみ必要となります。
- 下段：クラスの**操作**（メソッド）が含まれます。リスト形式で表示され、各行に1つずつ操作を記述します。これらの操作は、クラスがデータと相互作用する方法を示します。



メンバーのアクセス修飾子

- 各クラスには、**アクセス修飾子**（可視性）によりレベルの異なるアクセス権が付与されます。アクセスレベルと対応する記号は以下のとおりです：
  - 公開（+）
  - 非公開（-）
  - 保護（#）
  - パッケージ（~）
  - 派生（/）
  - 静的（_）

クラス間相互関係の表現

関係	線形
関連（association）	
集約（aggregation）	
コンポジション（composition）	
依存（dependency）	
汎化（generalization）	
実現（realization）	

多重度の表現

表記	意味
1	1個
'0..n' or '*'	0以上
1..n	1以上
2..5	2から5

どういう風に書くのかというのを下記のサイトから見て一緒に書き方を見ていく

https://www.itsenka.com/contents/development/uml/class.html

OpenAPI

- UML 以外に、API を設計する時よく使っているツールを紹介します。
- OpenAPI とは、RESTful API を記述するためのフォーマットのこと。
- https://spec.openapis.org/oas/v3.1.0
- https://petstore.swagger.io

補足資料

前回まで、Springboot を利用してログイン画面や登録画面等、簡単な WEB アプリケーションを作成できるようになりました。　今回からは、近年 WEB 業界ではやりだしている SPA について触れていきます。ここで Web サイトの通信パターンには大きく分けて以下の 2 つがあることを抑えておきましょう。

・MPA の仕組み

こちらは従来の Web サイトの通信方法となります。

前回まで扱っていた Springboot 単体での開発も MPA になります。

通信パターンとしては以下のようになります。

1. Web サイトを表示するためにリクエストをサーバーに送る
2. サーバーからレスポンスとして、HTML が返ってくる
3. ブラウザで返ってきた HTML を表示する

・SPA の仕組み

こちらは近年流行り始めている Web サイトの通信方法となります。

MPA との大きい違いは、SPA は毎回 HTML を返すのではなく、差分を更新していくという新しい方法です。

通信パターンとしては以下のようになります。

1. Web サイトを表示するためにリクエストをサーバーに送る
2. サーバーからレスポンスとして、HTML が返ってくる

3. ブラウザで返ってきた HTML を表示する

4. ２回目以降は Ajax（一部の情報のみをリクエストできる仕組み）で差分情報のリクエストをサーバーに送る

5. JSON 形式（JavaScript と相性が良いシンプルなフォーマット）が返ってくる

6. ブラウザで返ってきた差分箇所の JSON を JavaScript で HTML に追加して表示する

SPA の特徴

SPA のメリットとデメリットは以下のようなものが挙げられます。

メリット① 通常の Web ページでは実現できないユーザー体験(UX)を実現できる

ブラウザの挙動に縛られないことから、より幅広い UI を実現することができるため、ユーザー体験（UX）を向上させることができます。

例えば、Facebook のように「チャットルームを表示したまま表示しているコンテンツを変更する」などの機能を SPA では実現することができます。

メリット② 高速なページ遷移を実現できる

SPA によって、ページの読み込み時間などが短縮され、ユーザーの満足度が高くなります。その上、ブラウザによるページ遷移が発生しないため、独自にキャッシュ機能などを実装することでさらなる速度向上を図ることも可能です。

メリット③ ネイティブアプリの代わりとして提供することができる

SPA では Web アプリとして作成したものをネイティブアプリとして流用することができます。最近では、Web の最新の技術を用いて Web とネイティブアプリの両方の利点を持ったアプリを作ることが着目されています。

デメリット① 実装コストが大幅に増える

SPA では、ブラウザが行っていた処理(履歴管理、URL の割り当て、ローディングの表示)を自ら実装する必要があるため、開発コストがかかってしまいます。

このような機能は UI 上のささいな点である場合がほとんどですが、実装を怠ると使い勝手に大きく影響してしまうため、SPA を採用する上では対応は必須です。

デメリット② 初期ローディングにかかる時間が増える

ページの切り替えは高速になりますが、JavaScript のコード量が増えるのに加えて今までサーバー側で行っていた HTML の生成をブラウザで行うことになるため、 初期ローディングにかかる時間は増えます。

デメリット③ 開発者が少ない

SPA を実装できる開発者は一般的な Web ページ製作者と比べて圧倒的に少なくなります。また、SPA を開発するには規模の大きいコードを書く必要があるので、Web ページの制作スキル

に加えて、JavaScript やそのフレームワークの幅広い知識と高度な設計スキルが必要になります。

しかし、開発者が少ないということは逆に言うと案件に入れるチャンスが増えるということでもあります

SPA の開発ではフロントエンドとバックエンドで言語を分けるため、これまで Springboot が行っていた MVC の View（画面表示）の部分がなくなります。

フロントエンドで View の部分を実装します。

この際にバックエンドは API として提供されます。API（Application Programming Interface）とは、サービスのデータを外部のアプリケーションやプログラムから扱うための機能を提供するインターフェースです。中でも、HTTP 通信によってやりとりを行う API を Web API と呼びます。

チーム開発では、Vue.js をフロントエンドで Springboot をバックエンドとして利用します。

API を作成する上で REST という概念は非常に大事になります。
RESTful API とは、API の種類の 1 つで、REST と呼ばれる設計原則に従って策定されたものです。

RESTful API を使うメリット

1. URI に規律が生まれることで、API を利用するサービス開発者が楽になる。
2. URI に規律が生まれることで、API 開発者も URI からソースのどの部分なのかが容易にわかる。
3. ブラウザのアドレスバーに URI を入力すればリソースが参照できる。
4. サーバ、クライアント間で何も共有しないことにより、負荷に応じたスケーラビリティが向上する。ステートレス性に値するもので、一番のメリットされている。
5. GET、POST、PUT、DELETE 等の HTTP 標準のメソッドを使うことで、シンプルで一貫性のあるリクエスト標準化が円滑に行える。統一インターフェースに値する。