デザインパターンを用いた オブジェクト分析/設計技法［実践編］

佐原伸
E-Mail≡ sahara@sra.co.jp
URL≡ http:://www.sra.co.jp/people/sahara
産業システム第2部第3グループオブジェクト指向グループ
E-Mail≡st-info@sran265.sra.co.jp
URL≡ http:://www.sra.co.jp/smalltalk
ＳＲＡ
URL≡ http:://www.sra.co.jp

セミナーの目的

- デザ インパターンをある程度実際に使える
- オブジェク ト指向分析・設計の「評価」を行うことができる
- 小規模なオブジェク ト指向分析・設計を行うことができる

前提条件

- オブジェクト指向によるアプリケーション開発を目指すソフトウェア技術者
- オブジェクト指向技術の基礎知識
- オブジェクト指向方法論の基礎知識
- オブジェクト指向プログラミングの基礎知識
- デザインパターンの基礎知識

実践編1-3

目次

実践編1-4

Ⅰ. デザインパターン技術解説

- 1.デザインパターンとは？
  - 1.1 デザインパターンの種類
  - 1.2 デザインパターンの分類
- 2.デザインパターン解説
  - 2.1生成パターン
  - 2.2構造パターン
  - 2.3振る舞いパターン

1.デザインパターン

- 問題の解決法
  - ある状況で開発する際に発生する問題の解決法
- 静的及び動的構造
  - 設計の主要項目の静的及び動的構造と協調方式
- 成功した設計の再利用促進
  - うまくいったアーキテクチャと設計の再利用を促進する

1.1 デザインパターンの種類

- デザインパターン
  - サブシステムやコンポーネントあるいはそれらの間の関係を洗練する構成を提供する。
    - ある文脈での一般的な設計上の問題を解く
    - コンポーネント間に繰り返し現れる協調構造を記述する
- 分析パターン
  - 対象問題領域(ドメイン)に存在するドメインオブジェクトとその間の関係を提供する。
    - 設計上の問題を「解く」訳ではなく、再利用できるモデルを提供する
- アーキテクチャパターン
  - ソフトウェアシステムの基本的で構造化された組織や構成を表す。
    - 既定義のサブシステムの集合とその責任、それらの間の関係や組織化のための規則・指標を提供する。

1. 1 デザインパターンの種類

- プロセスパターン
  - ソフトウェアの開発に関わる定石を提供する
    - 開発チームの組織化・開発方法論・プロジェクト管理法などを提供する
- イディオム
  - プログラミング言語に依存した詳細レベルの抽象パターン。
    - 特定の言語で、コンポーネントやその間の関係をどうやって実装するかを記述する。
  - 参考書
    - 青木淳　著、「Smalltalkイディオム」、SRC、1997年
    - Kent Beck、「Smalltalk Best Practice Patterns」、1997年

1.2 デザインパターンの分類

- 生成パターン
  - クラスとオブジェクトの初期化と設定を行う
- 構造パターン
  - インターフェースと、「クラスとオブジェクト」の実装とを切り離す
- 振る舞いパターン
  - クラスとオブジェクトの動的なやりとりを扱う

実践編1-9

2.デザインパターン解説

- 2.1生成パターン
- 2.2構造パターン
- 2.3振る舞いパターン

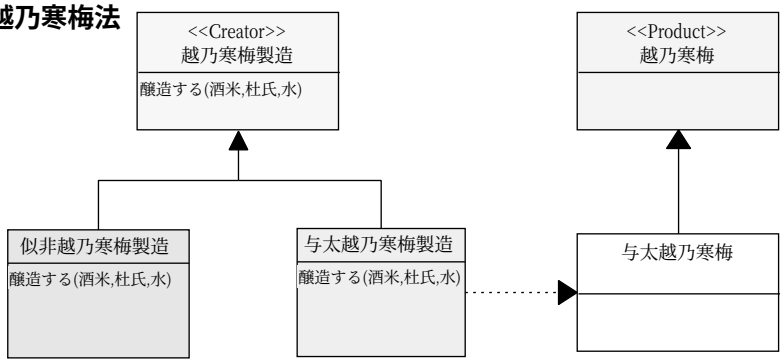
実践編1-10

2.1生成パターン

- Abstract Factory（**抽象工場**: Kit）
  - **互いに関連するオブジェクト群を、その具象クラスを明確にしないまま生成するインターフェースを提供する**
- Builder（**建築業**）
  - **複合オブジェクトの作成過程と表現形式を分離することにより、同じ作成過程で異なる表現形式のオブジェクトを生成する**
- Factory Method(**工場操作**：Virtual Constructor）
  - **オブジェクトを生成するためのインターフェースだけを規定して、実際のインスタンス生成をサブクラスにまかせる**
- Prototype(**模型**)
  - **模型（Prototype）からクローンとして新しいインスタンスを作る**
- Singleton(1**枚札**)
  - **クラスにインスタンスが一つしかないことを保証する**

Factory Method(**工場操作**）の比喩

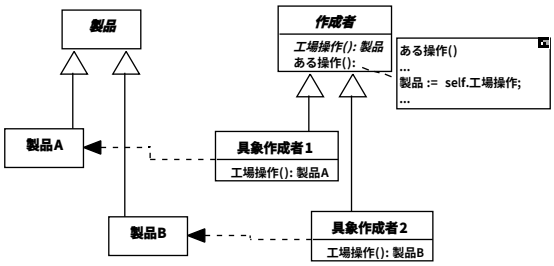
- **オブジェクトを生成するためのインターフェースだけを規定して、実際のインスタンス生成をサブクラスにまかせる**
  - **越乃寒梅法**



Factory Method(**工場操作**)

- **概要**
  - **オブジェクトを生成するためのインターフェースだけを規定して、実際のインスタンス生成をサブクラスにまかせる**
- **文脈(動機)**
  - **生成しなければならないオブジェクトのクラスを、生成する側のクラスが事前に知ることができない場合**
  - **クラスの責任をいくつかのサブクラスの一つに委譲するとき、どのサブクラスに委譲したかの知識を局所化したい場合**

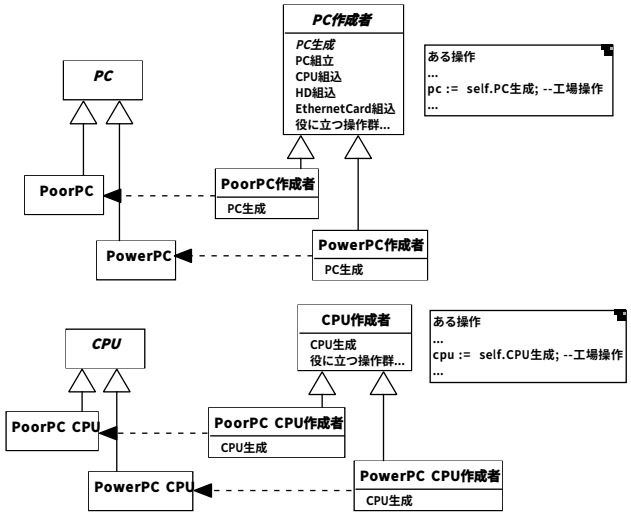
Factory Method**のクラス図**



Factory Methodの配役

- 製品Product
  - factoryMethod()が生成するオブジェクトの抽象操作を定義する
- 具象製品Concrete Product
  - <<製品>>の抽象操作を具現する
- 作成者Creator
  - <<製品>>オブジェクトを返すfactoryMethod()を定義する
    - factoryMethod()は抽象操作とは限らない
  - <<製品>>オブジェクトを生成し利用するために、factoryMethod()を呼ぶ
- 具象作成者Concrete Creator
  - << 具象製品>>オブジェクトを返すように、factoryMethod()を具現する

PC組立のFactory Methodクラス図



Factory Methodの結果

- 1. アプリケーションに特化したコードをクラス内に埋め込む必要がなくなる
- 2. <<作成者>>側のコードで<<製品>> クラスの操作しか呼び出さないため、任意の<<具象製品>>を追加することができる
- 3. 特定の<<具象製品>>のオブジェクトを作るためだけに、<<顧客>>が<<作成者>>のサブクラスを作らなければならないことがある
- 4. factoryMethod()を使ってオブジェクトを生成する方が、直接サブクラスを生成するよりも柔軟性を高める
  - 例えば、factoryMethod()を具象操作として実装すると、サブクラスで特殊化したfactoryMethod()を作成するときのヒントになる

Factory Methodの実装

- 通常、factoryMethod()は抽象操作であるが、前記4.で見るように具象操作として実現しても良い
- 生成するオブジェクトの種類を指定するための引数をfactoryMethod()に追加する
  - <<作成者>>が作成するオブジェクトの種類の増減や変更が容易になる
- パラメータ化クラス(テンプレート)を用いてサブクラス化を避ける
  - 前記3.のケースを避けることができる
  - Smalltalk ではクラスあるいはプログラム自身を引数として渡せるので必要ない

Factory Methodの使用例

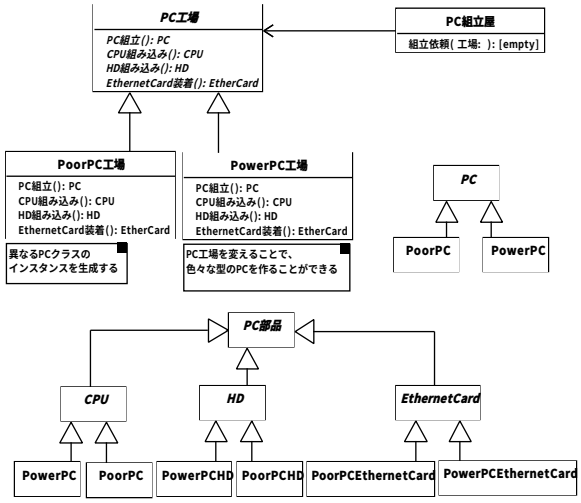
- Smalltalk のView クラスはfactoryMethod()であるdefaultControllerClass メソッドでControllerオブジェクトを生成する
- Smalltalk のBehavior クラスはfactoryMethod()であるparseClass を持つ
  - これにより、あるクラスが自分のソースコードを解析するための専用Parser を使うことができる
  - 例えばMartin Wollenweber は、Smalltalk コードの色付き清書にSyntaxHighlightingParser という専用Parser を作っている

Factory Methodの関連パターン

- Template Method パターン
  - factoryMethod は通常テンプレートメソッドの中で呼び出される
- Builderパターン
  - Factory Methodを使ってインスタンスを生成することができる
- Abstract Factoryパターン
  - Factory Methodを使ってインスタンスを生成することができる

Abstract Factory（抽象工場）の比喩

- PC組立屋



抽象工場（Abstract Factory、別名Kit）

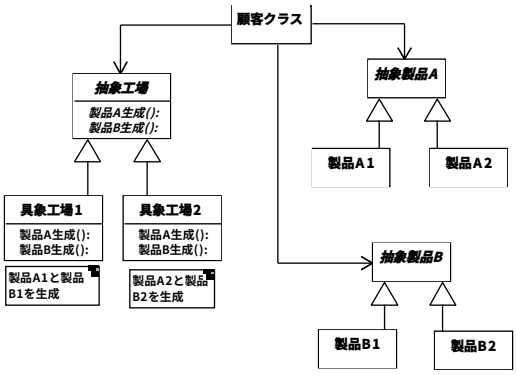
- 目的
  - 互いに関連するプロダクトとしてのオブジェクト群を、その具象クラスを明確にしないまま生成するインターフェースを提供することにより、顧客クラスが具象クラスを指定せずプロダクトを生成できるようにする
    - プロダクトを個々の部品から一歩一歩組み立てる必要のあるとき
    - 同じ部品ファミリーから１個のプロダクトに組み立てるとき
    - 部品の生成・組合せ・実装方法を、システムの他の部分から独立にしたいとき

Abstract Factoryの説明

- 部品による分岐の弊害回避
- if **ユーザーの選択** = #PoorPC then
- 　　pc = PoorPC.new
- elsif **ユーザーの選択** = #PowerPC then
- 　　pc = PowerPC.new
- end

- if **ユーザーの選択** = #PoorPC then
- 　　ethernetCard = PoorEthernetCard.new
- elsif **ユーザーの選択** = #PowerPC then
- 　　ethernetCard = PowerEthernetCard.new
- end

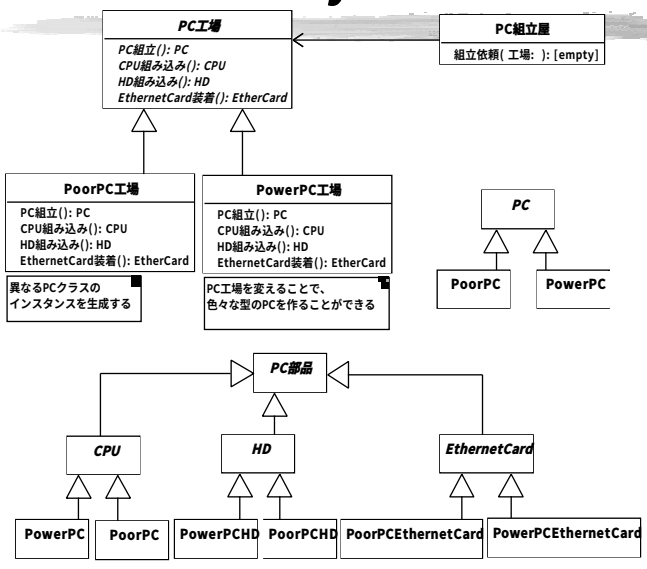
Abstract Factoryのクラス図



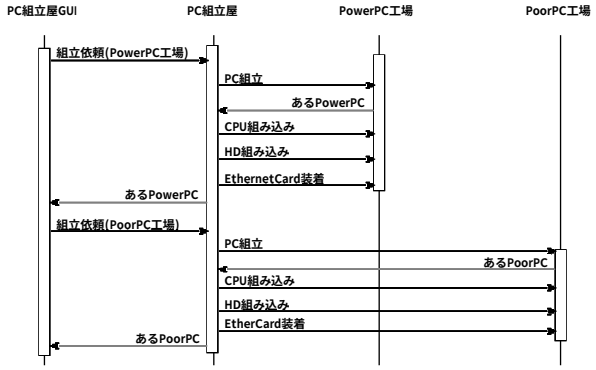
Abstract Factoryの配役

- **抽象工場**(Abstract Factory)
  - 抽象製品を生成する操作のインターフェースを宣言する。
- **具象工場**(Concrete Factory)
  - 具象製品を生成する操作を実装する。
- **抽象製品**(Abstract Product)
  - 製品オブジェクトのインターフェースを定義をする。
- **具象製品**(Concrete Productr)
  - 対応する具象工場オブジェクトによって生成される製品オブジェクトを定義し、抽象製品のインターフェースを実装する。
- **顧客**(Client)
  - 抽象工場と抽象製品のインターフェースのみを用いて、それらの提供するサービスを受ける。

Abstract Factoryの例: PC組立屋



Abstract Factoryの例: 順序図



Abstract Factoryの実装方法

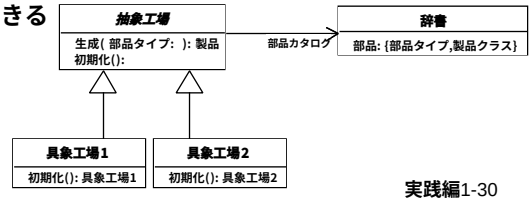
- 抽象工場の普通の実装方法
  - CPUを追加するとき修正を1カ所にできる
    - **context** PC**工場**::CPU**組み込み操作**
      - **抽象操作 -- サブクラスで定義**
    - **context** PowerPC**工場**:: CPU**組み込み操作**
      - CPU**のインスタンスを返す**
      - return = PowerPC.new
    - **context** PoorPC**工場**:: CPU**組み込み操作**
      - return = PoorPC.new

Abstract Factoryの実装方法

- 同一操作方式
  - Factory Methodの一種 -- 修正を普通より局所化できる
    - **context** PC工場::CPU組み込み操作
      - 修正する必要がない
      - return = self.cpuClass.new
    - **context** PowerPC工場:: cpuClass
      - return = PowerPC -- CPUのクラスを返す
    - **context** PoorPC工場::cpuClass
      - return = PoorPC

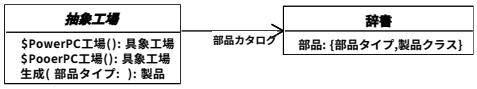
Abstract Factoryの実装方法

- 部品カタログ方式
  - 同一操作方式は工場内に多量の操作を必要とする
    - 新しいタイプの部品を追加する度に操作（CPU組み込み、HD組み込みなど）が増える
  - 「部品のカタログ」に部品クラスを登録し、一つの操作で各クラスの部品を生成できるようにする
    - 部品カタログの「辞書」を抽象工場クラスと関連付け、具象工場クラス毎にキーが部品タイプで値が対応する部品クラスであるように「辞書」を初期化する
  - 具象工場は部品カタログの初期化を行うだけで、他のすべての処理は抽象工場が行うことができる



Abstract Factoryの実装方法

- 1工場方式
  - 抽象工場のインスタンスとして具象工場を持つ
  - 抽象工場のクラス操作で、部品カタログの初期化を行った具象工場インスタンスの生成を行う



Abstract Factoryの使用例

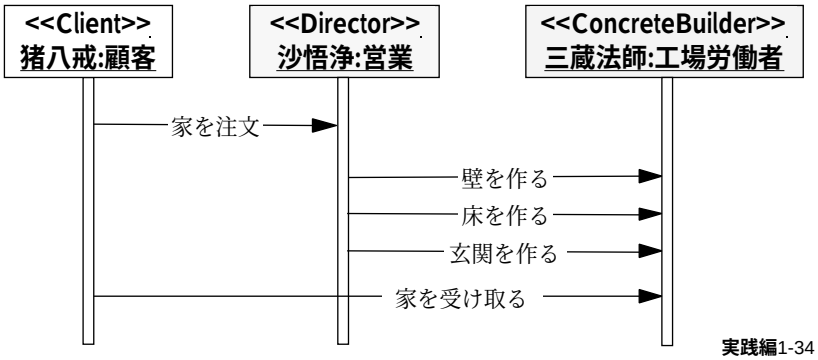
- InterViews
  - Kit
- ET++
  - WindowSystem
- VisualWorks
  - UILookPolicy

Abstract Factoryの関連パターン

- **建築業**(Builder)
  - **抽象工場と非常に似ているが...**
    - **複合オブジェクトを組み立てる主体が異なる**
      - **抽象工場**
        - **抽象工場の顧客オブジェクトが組立**
      - **建築業**
        - **建築業オブジェクトが組立**
- **工場操作**(Factory Method)
  - **抽象工場の対案になりやすいパターン**
    - **抽象工場は工場クラス階層が複雑**
    - **工場手法はアプリケーションクラス階層が複雑**

Builder（建築業）の比喩

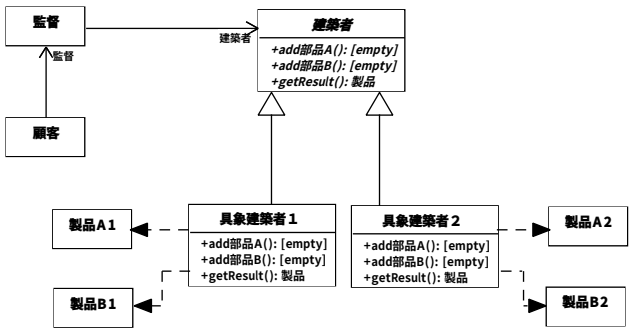
- 複合オブジェクトの作成過程と表現形式を分離することにより、同じ作成過程で異なる表現形式のオブジェクトを生成する
  - ツーバイフォー方式



Builder（建築業）

- 概要
  - 複合オブジェクトの作成過程と表現形式を分離することにより、同じ作成過程で異なる表現形式のオブジェクトを生成する
- 文脈(動機)
  - 多くの構成要素からなるオブジェクトを解釈・生成する部分と、構成要素やその構造とを独立にしておきたい場合
  - オブジェクトの作成プロセスが、オブジェクトに対する多様な表現を認めるようにしておかなければならない場合

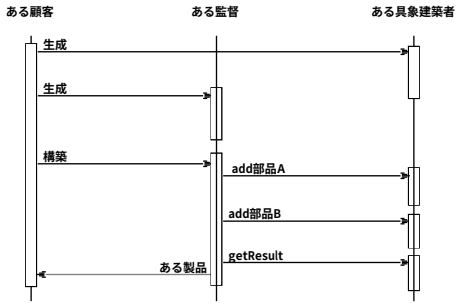
Builderのクラス図



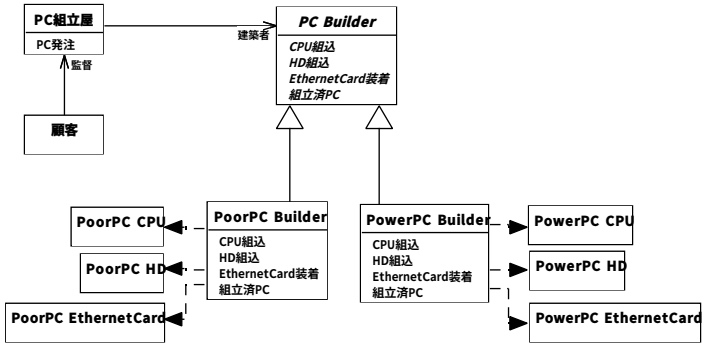
Builderの配役

- 建築者Builder
  - <<製品>>オブジェクトの構成要素の生成・組立を行う抽象操作を提供する
- 具象建築者Concrete Builder
  - <<建築者>>クラスの抽象操作を具現する
  - 自身が生成する「表現」を定義･管理する
  - <<製品>>オブジェクトを取り出すための操作を提供する
- 監督Director
  - <<建築者>>クラスの操作を使って、 <<製品>>オブジェクトを生成する
- 製品Product
  - 操作の対象となる各種の複合オブジェクトを表す
  - 自身の構成要素を定義するクラスや構成要素を<<製品>>オブジェクトに組み立てる操作を含む

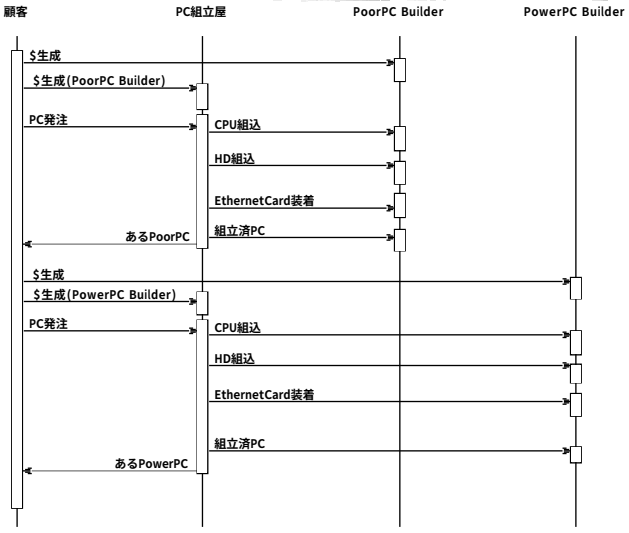
Builderの協調関係



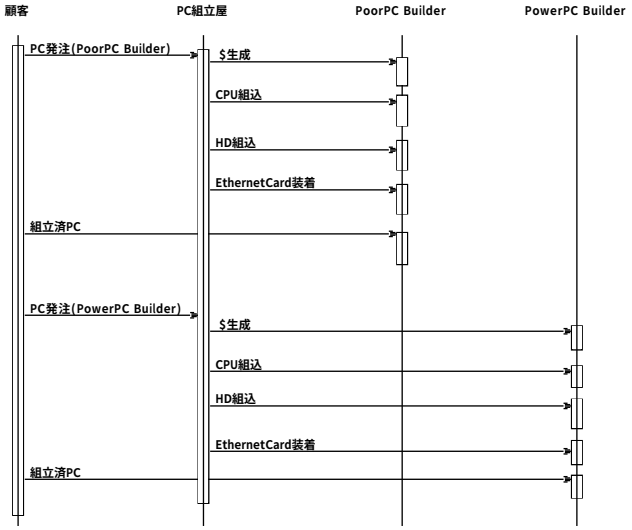
PC組立のBuilderクラス図例



PC組立のBuilder順序図例



PC組立のBuilder順序図例2



Builderの結果

- <<製品>>オブジェクトの内部表現変更が可能
  - <<建築者>>オブジェクトが<<製品>>オブジェクトの内部表現や内部構造や組立方を隠蔽している
- 生成や表現のためのコードを局所化でき、再利用につながる
  - <<製品>>クラスは、異なる種類の<<監督>>クラスに同じサービスを提供することができる
- <<製品>>オブジェクトの作成過程をより詳細に制御できる

Builderの実装

- <<具象建築者>>クラスがあらゆる種類の<<製品>>オブジェクトを生成できるように、<<建築者>>クラスのインターフェースは十分に一般的でなければならない
- 部品の生成にはFactory Methodパターンを使うことができる

Builderの使用例

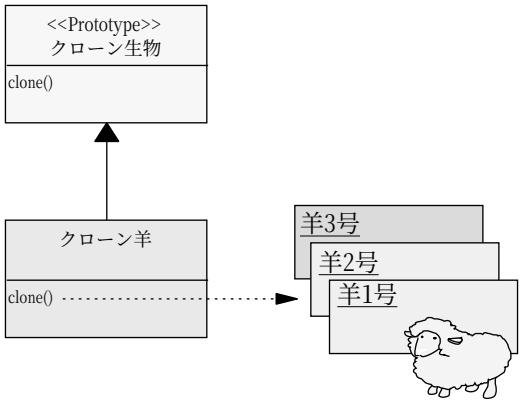
- FrameMakerやクラリスインパクトといったドキュメント作成プログラムは、各種のワープロやファイル形式を読み込み、逆に、色々なファイル形式で書き出すのにBuilderパターンを使っている
- フリーのSmalltalkであるSqueakコンパイラーのParseクラスは、<<建築者>> ParseNodeクラスを使う<<監督>>である
  - Parserは文法上のトークンを認識するたびに、ParseNodeオブジェクトにメッセージを送る
  - 構文解析が終了したら、ParserはParseNodeオブジェクトに、それまでに作成した構文解析木MethodNodeを要求し、<<顧客>>であるCompilerその他のオブジェクトに返す

Builderの関連パターン

- Compositeパターン
  - <<建築者>>は<<入れ物>>を作成することが多い
    - FrameMakerは図や表に<<入れ物>>を使用している
    - Smalltalkの構文解析木は<<入れ物>>を使用している
- Strategyパターン
  - BuilderパターンはStrategyパターンの特殊な形
- Abstract Factoryパターン
  - 複合オブジェクトを組み立てる主体が異なる
    - 抽象工場
      - 抽象工場の顧客オブジェクトが組立
    - 建築業
      - 建築業オブジェクトが組立

Prototype(模型)の比喩

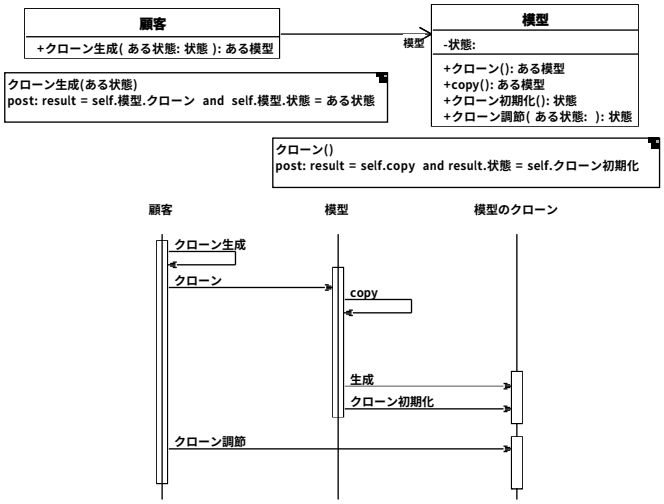
- 模型（Prototype）からクローンとして新しいインスタンスを作る
  - クローン羊方式



Prototype(**模型**)

- **概要**
  - **模型としてのインスタンスを使ってオブジェクトの種類を記述し、その模型をコピーして新しいオブジェクトを作る**
- **文脈(動機)**
  - **動的に責任を追加したい場合**
  - **生成したいオブジェクトのクラス階層と対称関係になる複雑なFactoryのクラス階層を作りたくない場合**
  - **インスタンスの初期化コストが高い場合**

Prototype**のクラス図**



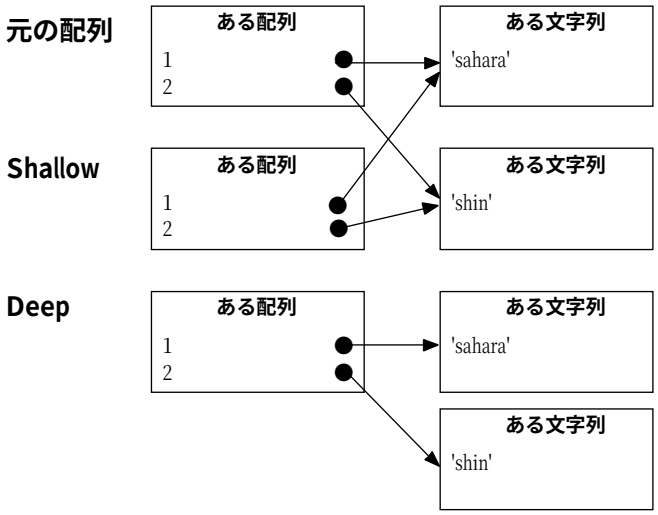
Prototypeの配役

- 模型(Prototype)
  - 自身の複製を行い、新しい状態を設定する
- 顧客(Client)
  - 模型に複製を依頼することで、模型の新たなインスタンスを生成し、それを利用する

Prototypeの結果

- 複雑なFactoryのクラス階層作成を避けることができる
- 動的に責任を追加できる
- インスタンスの初期化コスト安くできる
- Copy問題が発生する
  - Shallow Copy(浅いコピー)かDeep Copy(深いコピー)か?

Copy問題



Prototypeの実装

- 模型管理者
  - 模型クラスがクローン群を管理できないので、クローン群を管理するオブジェクト(模型管理者)が必要になる
    - 模型管理者は、クローンの登録・検索・削除・状態変更などを行う
- クローン操作の実装
  - Copy問題を注意する

Prototypeの使用例

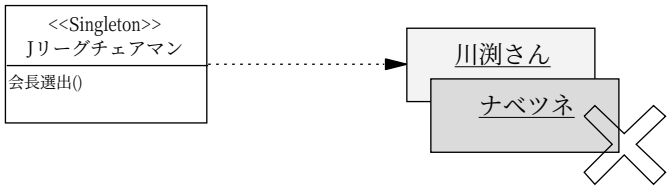
- ThingLab(制約指向図形エディター)
  - ユーザー定義のThingに対応したクラスを生成し、再利用するときにクローンとして取り出す
- VisualWorksのRDBフレームワーク
  - 空のオブジェクトのクローンに1行分のデータ
- 楽譜編集エディタ
  - 汎用のGraphicToolに楽譜編集機能を動的に付与
- Smalltalkのクラス
  - Metaclassのクローン

Prototypeの関連パターン

- Decoratorパターン
  - 動的に「差分」機能を追加する
  - Prototypeパターンは、全体をコピーし、機能を追加する
- Type Object
  - 新しいクラスをコンパイルなしに生成できる

Singleton(1**枚札**)**の比喩**

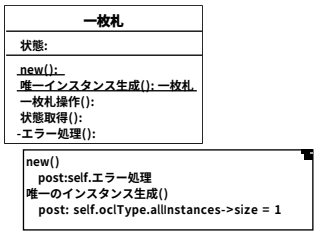
- **クラスにインスタンスが一つしかないことを保証する**
  - **「両雄並び立たず」**



Singleton(1**枚札**)

- **概要**
  - **あるクラスにインスタンスが一つしかないことを保証する**
- **文脈(動機)**
  - **唯一性が要求される機能を実行する場合**

Singletonのクラス図



Singletonの配役

- Singleton
  - インスタンスを一つだけ生成する
  - 唯一性を保証する

Singletonの結果

- 大域変数を使わずSingletonを使うことで保守性･再利用性が増す
- 唯一のインスタンスへのアクセス制御ができる
- インスタンスの数を後で増やすことができる
- クラス操作より柔軟性が増す
- サブクラス化して、操作や内部表現を詳細化できる

Singletonの実装

- 唯一性の保証
  - **if** instanceOfSingleton = {} **then** new() **else** error() **end**
- サブクラス化したときの問題点
  - どのサブクラスのインスタンスを使うか？
    - 環境変数などで指定する
    - インスタンス生成時に決める
      - 簡単だが柔軟性に欠ける
    - Singletonオブジェクトの登録機構を使う
      - 柔軟だがSingleton全クラスで登録機構のアクセス操作が必要になる

Singletonの使用例と関連パターン

- 使用例
  - SmalltalkのChangeSet
    - 唯一のインスタンスはソースコードの変更履歴
  - SmalltalkのMetaClass
    - 唯一のインスタンスはクラス
- 関連パターン
  - Abstract Factory、Builder、PrototypeパターンはSingletonパターンを使って実装することが多い

実践編1-61

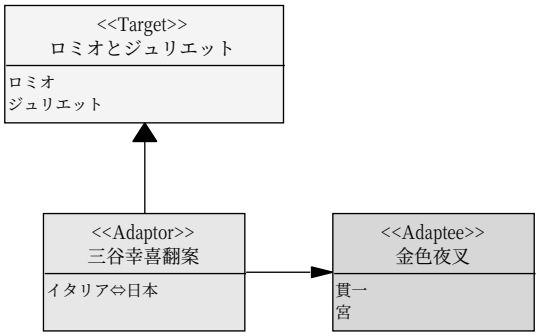
2.2構造パターン

- Adapter（翻案者：Wrapper）
  - あるクラスのインターフェースを、他のインターフェースへ変換する
- Atomizer(原始化)
  - 複雑なオブジェクト構造を、バックエンドの構造データ（普通のファイル、RDB、RPCバッファーなど）として格納する
- Bridge（橋：Handle/Body）
  - 論理クラスと実装クラスを分離し、それぞれの独立性を保つ
- Composite（混成）
  - 部分-全体構造を表現するために、オブジェクトを木構造で構成する

2.2構造パターン　続き

- Decorator（装飾者：Wrapper）
  - オブジェクトに動的に責任を追加する
- Facade(見せかけ)
  - サブシステムのインタフェースを単純化する
- Flyweight（フライ級選手）
  - 多数の小さなオブジェクトを共有し、空間効率を高める
- Proxy(代理人：Surrogate)
  - オブジェクトの代理を提供する

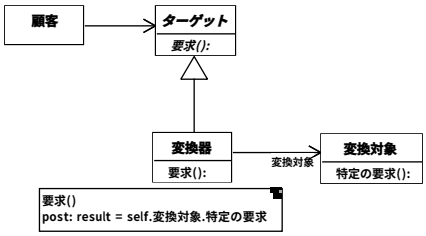
Adapter（翻案者：Wrapper）の比喩



Adapter （翻案者）

- 概要
  - あるクラスのインターフェースを、他のインターフェースへ変換する
- 文脈(動機)
  - 既存のクラスを利用したいがインターフェースが一致していない場合
  - 予想もしないようなクラスとも協調していける、再利用可能なクラスを作成したい場合
- 制約条件Forces
  - インターフェースの違いを<<Client>>に見せてはいけない
  - 既存クラスは、インターフェースの変更を予期してはならない
  - 既存クラスの実装は進化し続けることが可能でなければならない

Adapterのクラス図



実践編2-5

Adapterの配役

- Target(**ターゲット**)
  - **<<顧客>>が利用するドメインに特化したサービスを定義する**
- Client(**顧客**)
  - **<<変換対象>>クラスと協調したいオブジェクト**
- Adaptee(**変換対象**)
  - **インターフェースを適合させる必要のある既存クラス**
- Adapter(**変換器**)
  - **<<変換対象>>クラスのインターフェースを、<<ターゲット>>クラスのインターフェースに適合させる**

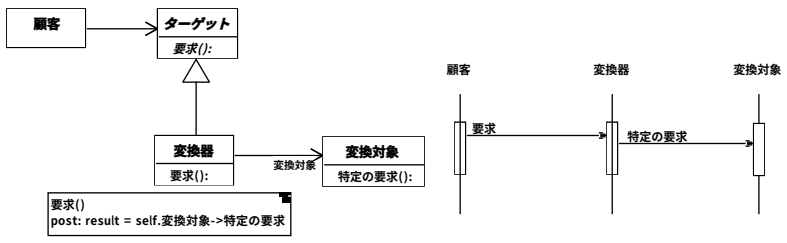
実践編2-6

Adapterの結果

- <<変換対象>>クラスは<<変換器>>によってブラックボックスとなる
  - <<変換対象>>はそれ自身、別の用途のために改良を続けることができる
- <<変換器>>は、<<変換対象>>にない機能を追加することができる
- <<変換器>>は、<<変換対象>>にない複数のインターフェースを提供できる
- インターフェースの適合機能を持つ<<Pluggable Adapter>>は汎用性が高い

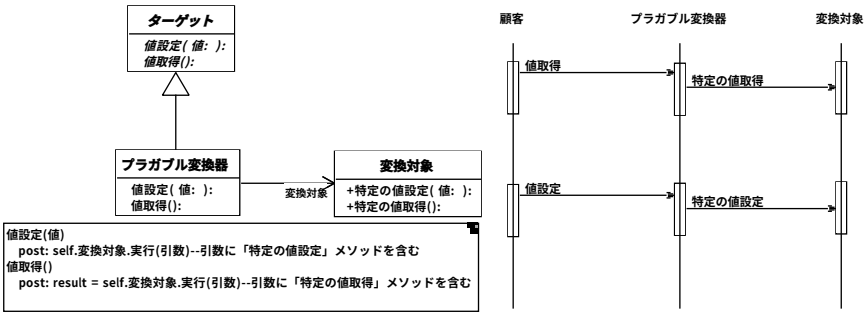
Adapterの実装

- 個別仕立てAdapter
  - 特定のメソッドをハードコードする



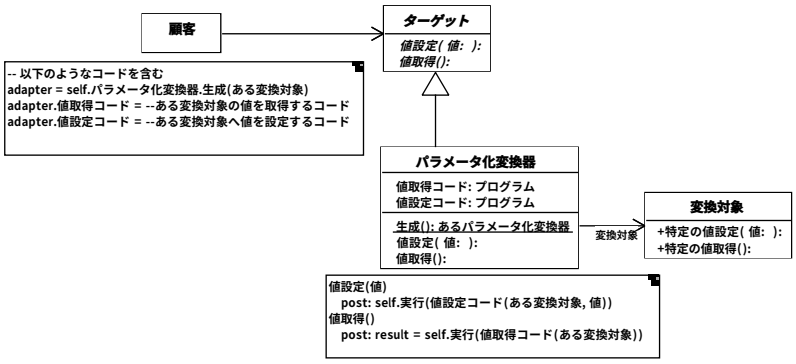
Adapterの実装

- メッセージ転送Pluggable Adapter
  - Adapteeのインタフェースが設計時には不明の時



Adapterの実装

- パラメータ化Adapter

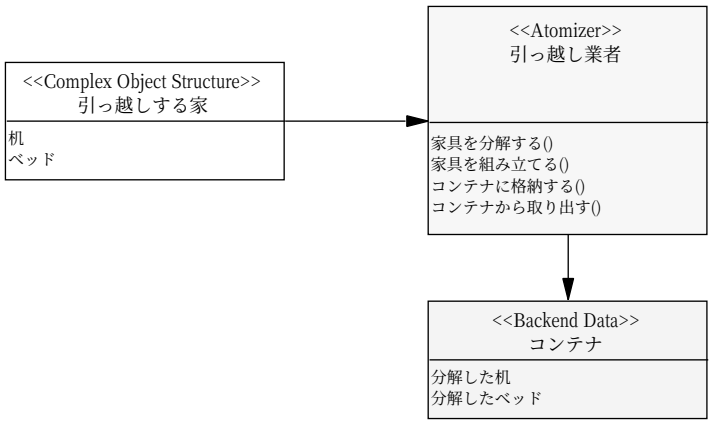


Adapter関連パターン

- Decoratorパターン
  - インターフェースを変更することなく、オブジェクトに機能を追加する
  - Adapterでは不可能な再帰構造を実現できる
- Proxyパターン
  - 他のオブジェクトを代表するか代理を務めるが、インターフェースを変更することはない

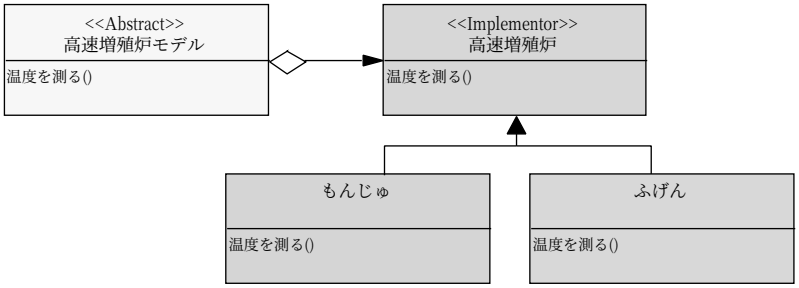
Atomizer(原始化)の比喩

- 複雑なオブジェクト構造を、バックエンドの構造データ（普通のファイル、RDB、RPCバッファーなど）として格納する

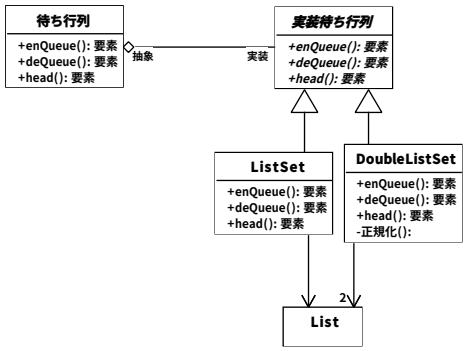


Bridge（橋：Handle/Body）の比喩

- 論理クラスと実装クラスを分離し、それぞれの独立性を保つ
  - 動燃方式

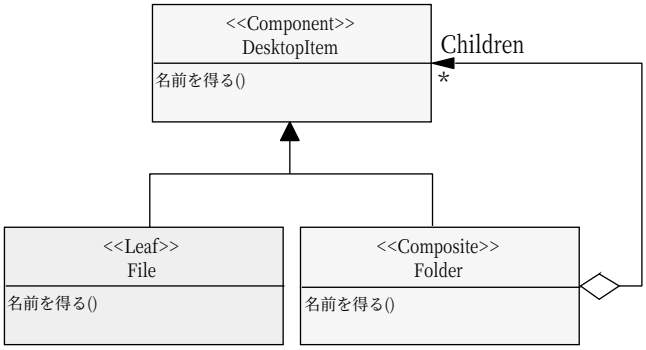


Bridgeの例



Composite（混成）の比喩

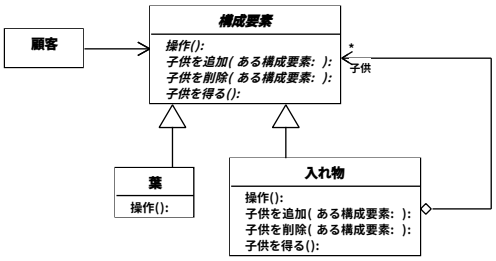
- パソコンのファイル・システム



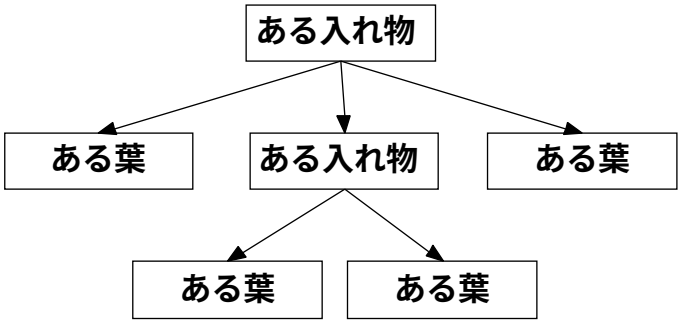
Composite（混成）

- 概要
  - 部分-全体構造を表現するために、オブジェクトを木構造で構成する
- 文脈(動機)
  - オブジェクトの部分-全体階層を実現したい場合
  - オブジェクトを包含したものとオブジェクト自身を同じように扱いたい場合

Compositeのクラス図



Compositeの構造例



Compositeの配役

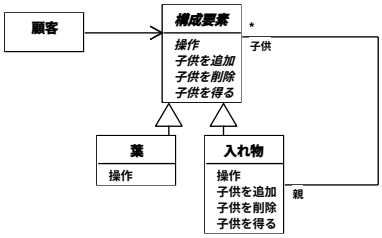
- 構成要素 Component
  - 構造内のオブジェクトのインターフェースとしての抽象操作を定義する
  - 構造内のすべてのクラスに共通な、標準インタフェースの操作を定義する
  - 子にあたる<<構成要素>>オブジェクトをアクセスするための操作を定義する
  - 親にあたる<<構成要素>>オブジェクトにアクセスするための操作を定義する
- 葉Leaf
  - 構造内のターミナル(葉)オブジェクトを表す
  - 子オブジェクトを持たない
  - 構造内のオブジェクトの基本操作を定義する
- 入れ物Composite
  - 子オブジェクトを持つ<<構成要素>>オブジェクトの振る舞いを定義する
  - 子<<構成要素>>オブジェクトを保持する
  - <<構成要素>>クラスの抽象操作を具現する
- 顧客Client
  - <<構成要素>>クラスの操作を通して、構造内のオブジェクトにアクセスする

Compositeの結果

- 再帰的で複雑なオブジェクトの構造に、統一的なインターフェースを与えることができる
- 再帰呼び出しのできる言語では、簡単なコードで全オブジェクトのアクセスや検索が行える
  - 例
    - extractElems(node(t1, e, t2) ≡extractElems(t1) **union** Sequence{e} **union** extractElems(t2)
- <<顧客>>のコードが単純で分かりやすくなる
  - <<顧客>>は<<葉>>と<<入れ物>>のどちらを扱っているか知る必要がない
- 新しい<<構成要素>>を追加しやすくなる
  - <<顧客>>での変更の必要がない
- <<構成要素>>の種類の制限は動的にしか行うことができない
- 性能改善のためのキャッシュを効かせやすい
  - SmalltalkのMethod Dictionaryや、図形エディターの表示領域内の図のキャッシュなど

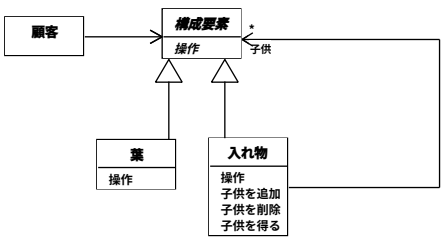
Compositeの実装

- 親オブジェクトへの明示的なリンクの効果
  - 構造内の走査や管理が簡単になる



Compositeの実装

- 子オブジェクトを管理する走査の定義方法
  - 子オブジェクトを管理するインターフェースをクラス階層の最上位クラスで定義する
    - すべての<<構成要素>>を統一的に扱えるが、<<葉>>に対して「子供の追加」や「子供を削除」などの意味のない操作を要求する危険性がある
  - 子オブジェクトを管理するインターフェースを<<入れ物>>クラスで定義する
    - 安全性が高まるが、<<葉>>と<<入れ物>>のインターフェースが異なってしまうため、統一性が失われる



Compositeの実装

- 性能改善のためのキャッシュ
  - 操作や検索を実際に行った結果やそのための近道情報をキャッシュする
- 誰が<<構成要素>>を削除するか
  - ガーベージコレクションをサポートしていない言語では、<<葉>>が多くのオブジェクトに共有されているときなどに問題が起こる
- <<構成要素>>を保持するための最適なデータ構造の選択
  - リスト、木(2分木、B-木、Trayなど)、ハッシュ表、配列などから最適なものを選択する

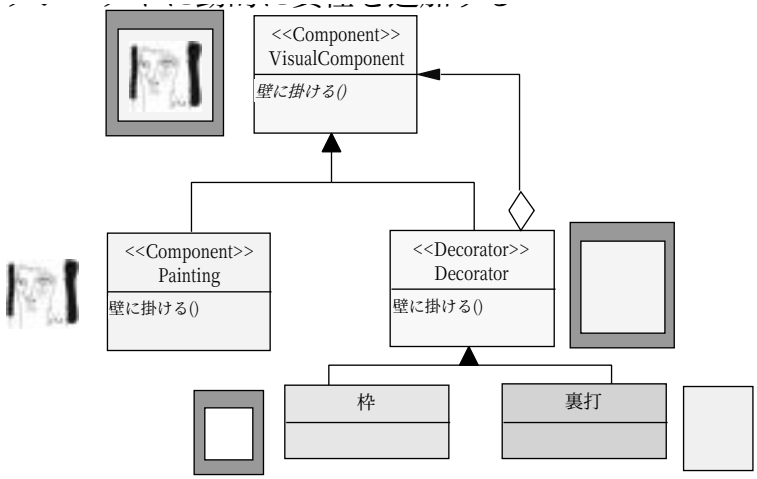
Compositeの使用例

- Interpreterパターンで使用
- OODBで使用
- ほぼすべてのOOシステムのフレームワークやユーザーインタフェースで使用
- Smalltalkの場合
  - MVCモデルのView、Compilerの構文解析木、SqueakのFileDirectoryとFileStreamなど

Compositeの関連パターン

- Chain Of Responsibilityパターン
  - いわば「潰れた木」にアクセスしているパターン
- Decoratorパターン
  - しばしばCompositeパターンと一緒に使われる
- Interpreterパターン
  - 構文解析木に使用

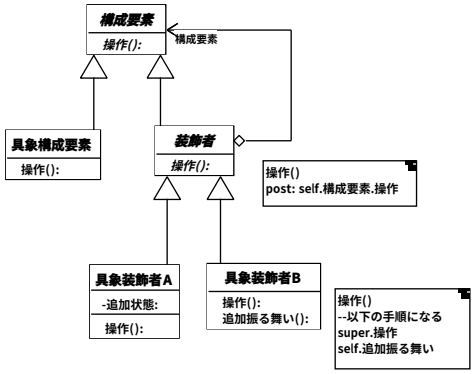
Decorator（装飾者：Wrapper）の比喩



Decorator （装飾者：Wrapper）

- **概要** Synopsis
  - オブジェクトに責任を動的に追加する事により、サブクラス化より柔軟な機能拡張法を提供する
- **文脈(動機)**Context
  - オブジェクトに責任を動的かつ透明に追加する場合
  - 責任を動的に削除することができるようにする場合
  - サブクラス化による拡張ができない場合
    - 拡張が非常に多い場合
    - クラス定義が隠蔽されている場合

Decoratorのクラス



Decoratorの配役

- 構成要素Component
  - 責任を動的に追加できるオブジェクトのためのインタフェースを定義する
- 具象構成要素Concrete Component
  - 責任を動的に追加できるオブジェクトを定義する
- 装飾者Decorator
  - <<構成要素>>への参照を持ち、<<構成要素>>のインターフェースと一致したインターフェースを定義する
- 具象装飾者Concrete Decorator
  - <<構成要素>>に責任を追加するオブジェクトを定義する

実践編2-29

Decoratorの結果

- 静的な継承より柔軟
- 機能を満載した(保守や再利用のしにくい）クラスを、クラス階層の上層で定義する危険を避けることができる
- 多くの似たようなオブジェクトができる
  - 構造を理解していないと、修正やデバッグが困難になる
- 同一性の問題
  - 装飾者とそれが装飾している構成要素は異なるオブジェクトになるため、同一性の判定は行えない

実践編2-30

Decoratorの実装

- インタフェースの一致
  - <<装飾者>>と<<構成要素>>のインターフェースは一致していなければならない
- 抽象クラス<<装飾者>>の省略
  - 単純な場合、<<具象装飾者>>だけでも良い
- 構成要素クラスの軽量化
  - 構成要素クラスで多くの事をやるべきでない
  - インターフェースの定義のみに機能を絞る

Decoratorの使用例

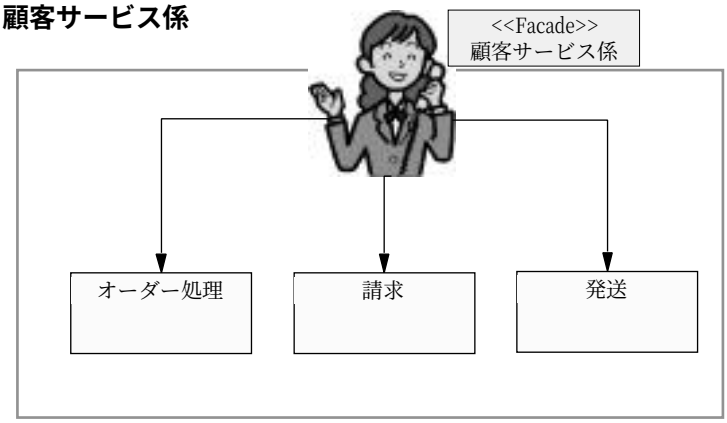
- 多くのGUIキットがWidgetに装飾を追加するためにDecoratorパターンを使用している
  - Smalltalk、InterViews、ET++など
- Streamにコード変換や圧縮アルゴリズムを適用するためにDecoratorパターンを使っている
  - Smalltalk、ET++など
    - Stream
      - EncodedStream
        - encoderを呼び出す

Decoratorの関連パターン

- Adapter
  - Decoratorはインターフェースを変えないが、Adapterは変える
- Proxy
  - Proxyは責任を追加せず、アクセス方法を変える
- Strategy
  - Decoratorのように表面的に装飾するのではなく、中身を変える

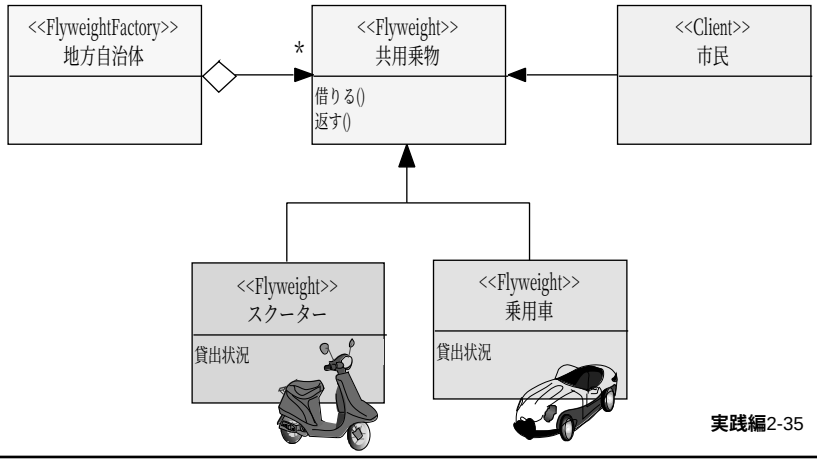
Facade(見せかけ)の比喩

- サブシステムのインタフェースを単純化する
  - 顧客サービス係



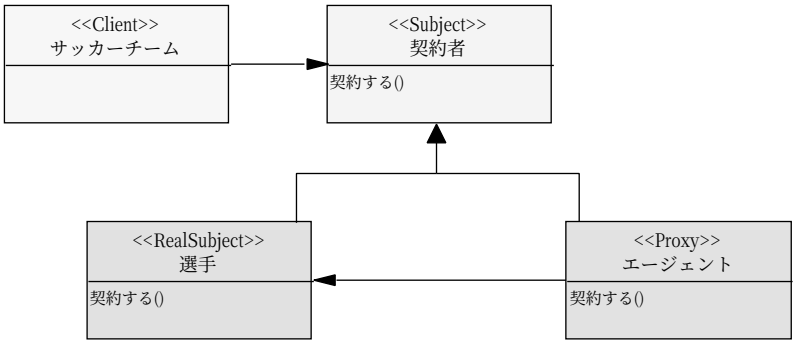
Flyweight（フライ級選手）の比喩

- 多数の小さなオブジェクトを共有し、空間効率を高める
  - 乗り物の共有



Proxy(代理人：Surrogate)の比喩

- オブジェクトへのアクセスの代理を提供する
  - 中田の代理人



2.3振る舞いパターン

- Chain of Responsibility（責任の連鎖）
  - 責任のあるサービス提供者へ要求を「連鎖的に」委任する
- Command（命令：Action, Transaction）
  - パラメータ化した要求をオブジェクトとしてカプセル化する
- Interpreter（通訳）
  - データを言語と考え、文法･意味を与え、それを解釈する
- Iterator（繰り返し）
  - オブジェクトの数を指定せず、順番に処理する
- Mediator（調停者）
  - オブジェクト群の相互作用をカプセル化するオブジェクトを定義する
- Memento（形見：Token）
  - オブジェクト個別の内部状態を捉え、後でその状態に戻す

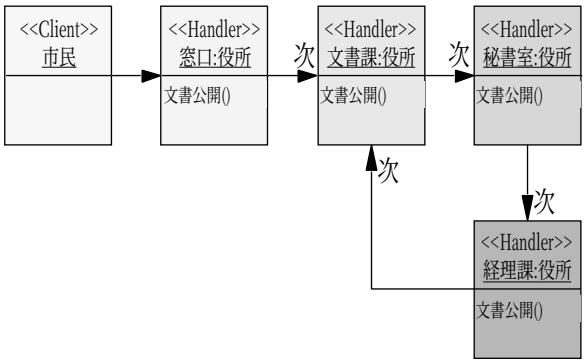
2.3振る舞いパターン　続き

- Observer（観察者：Dependents,Publish/Subscribe）
  - オブジェクトが状態を変えたとき、それに依存したオブジェクトが自動的に更新される
- State（状態）
  - オブジェクトの内部状態が変わったとき、振る舞いを変える
- Strategy(戦略：Policy)
  - アルゴリズム群の各々をカプセル化し、交換可能にする
- Stream（流れ：Pipes and Filters, Dataflow）
  - 順序のある情報の流れを処理し、再利用できる処理単位とする
- Template Method（型紙方式）
  - 一部をサブクラスで実装するアルゴリズム
- Visitor（訪問者）
  - オブジェクト構造上の要素で実行される操作を表現する

Chain of Responsibility（責任の連鎖）の比喩

■責任のあるサービス提供者へ要求を「連鎖的に」委任する

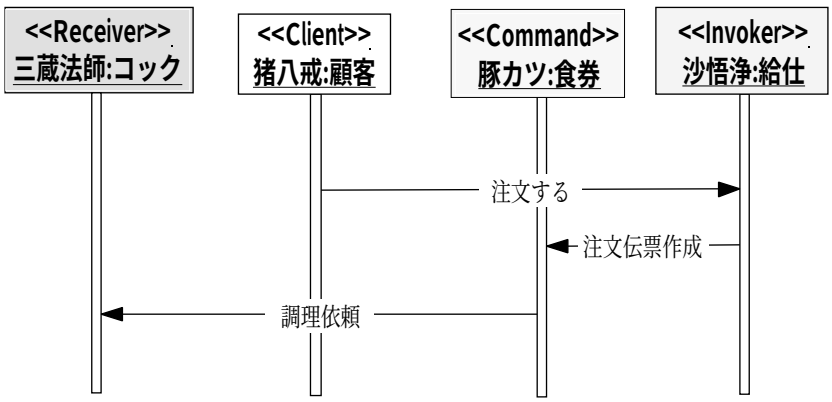
◆たらい回し



実践編3-3

Command（命令：Action, Transaction）の比喩

■パラメータ化した要求をオブジェクトとしてカプセル化する

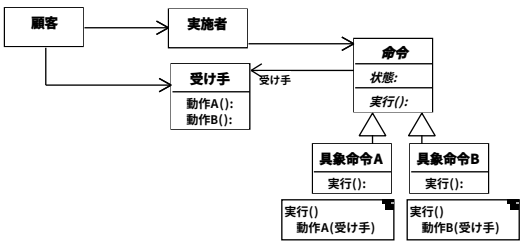


実践編3-4

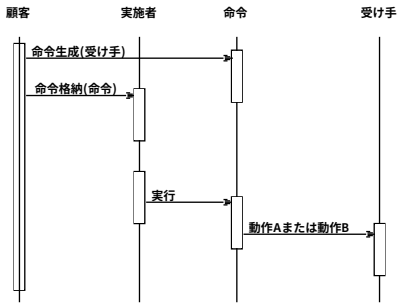
Command(**命令**)

- **概要**
  - **要求をカプセル化することによって、要求を受信するオブジェクトに関する知識を不要とし、要求に対する種々の操作を実現する**
- **文脈**
  - **要求の明確化・順序化・実行をそれぞれ独立化したい場合**
  - **要求の取り消しを行いたい場合**
  - **要求の再実行を必要とする場合**
  - **基本的な要求からなる高度な要求（トランザクションなど）によってシステムを構造化したい場合**

Command**のクラス**



Commandの順序図



Commandの配役

- 命令Command
  - 要求を実行するための操作のインターフェースを宣言する
- 具象命令Concrete Command
  - 要求を実行する
  - 受け手オブジェクトの該当動作()を呼び出す
- 顧客Client
  - <具象命令>>オブジェクトを生成し、その<<受け手>>を設定する
- 実施者Invoker
  - <<命令>>に要求実行を依頼する
- 受け手Receiver
  - 要求を実現するための具体的操作を知っている

Commandの結果

- コマンドの発行と実行を異なるオブジェクトに分離できる
- コマンドに関する種々の操作を用意できる
  - 取り消し、順序変更など
- 新しいコマンドを容易に追加できる
- 複数のコマンドを合成できる

Commandの実装

- 取り消しと再実行
  - コマンドの履歴が必要になる
  - Memento(形見)パターンを使うことが多い
- コマンドの汎用化
  - Interpreterパターンを使って、コマンドを「言語」化すると汎用性が増す

Command（命令）の使用例

- MacAppではフレームワークにCommandパターンを実装している
  - InterViews, ET++でも同様のクラスを実装している
- VisualSmalltalk
  - アプリケーションフレームワーク
- IBM Smalltalk
  - アプリケーションフレームワーク

Interpreter（通訳）の比喩

■クラスに関わるパターン

◆データを言語と考え、文法･意味を与え、それを解釈する

▸関数定義ができる計算機の構文例と実行例（BNF記法）

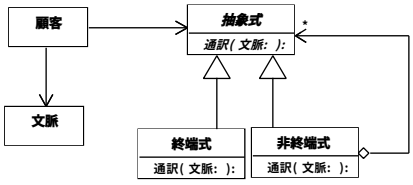
```
式 ::= NUM                      pi = 3.141592653589
     | VAR                      3.141592653589
     | VAR '=' 式               sin(pi)
     | FNCT '(' 式 ')'          0.0000000000
     | 式 '+' 式                a = b = 2.3
     | 式 '-' 式                2.3000000000
     | 式 '*' 式                ln(a)
     | 式 '/' 式                0.8329091229
     | '-' 式                   exp(ln(b))
     | 式 '^' 式                2.3000000000
     | '(' 式 ')'
```

Charles Donnerlly, Richard Stallman著
「BISON The YACC-compatible Parser Generator」Free Software Foundation, 1991より引用

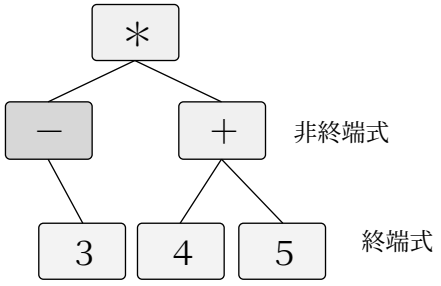
Interpreter（通訳）

- **概要** Synopsis
  - **データを言語と考え、文法･意味を与え、それを解釈する**
- **文脈(動機)**Context
  - **文脈自由文法に従う言語を処理したい場合**
    - **ほとんどのデータはちょっとした工夫で「文脈自由文法に従う言語」と見なせる**
    - **正規表現、図形表現、ドキュメント、プログラミング言語、通信文・プロトコルなど、固定幅以外のデータのほとんどが対象**
- **制約条件**Forces
  - **問題の複雑さは文法で表すべきで、個々の要素や組み合わせ規則で表すべきではない**
  - **個々の要素や組み合わせ規則に制限を設けるべきではない**
    - **不要な制限の例：**
      - **ある汎用大型機の電卓ソフト**
      - **Wangの電卓ソフト**

Interpreterのクラス



抽象構文木の例



Interpreterの配役

- **抽象式** Abstract Expression
  - **抽象構文木のすべての頂点に共通な抽象操作を定義する**
- **終端式** Terminal Expression
  - **文法中の終端記号に関する操作を具現する**
  - **終端記号毎にインスタンスが生成される**
- **非終端式** Non-Terminal Expression
  - **非終端記号それぞれについてこのクラスが定義される**
  - **非終端記号それぞれについて抽象式型のインスタンス変数を保持する**
  - **非終端記号それぞれについて**interpret()**操作を具現する**
- **文脈**Context
  - **インタープリター実行時の環境、すなわちグローバルな情報を持つ**
- **顧客**Client
  - **文脈自由文法で与えられた言語の構文解析木を作る（または与えられる）**
  - interpret()**操作を呼び出す**

Interpreterの結果

- 文法の変更と拡張が容易である
- 構文解析部を実装するのも容易である
- 複雑な文法を保守していくのは難しい
  - 本格的な「言語」を処理する場合は、OOにこだわらず、Parser生成ツール(yacc, bisonなど)を使用し、<<Adapter>>でラップして使った方がよい
- 言語表現を新しい方法で評価することを容易にする
  - 言語を処理するだけでなく、清書したりチェックをする操作を簡単に定義できる

Interpreterの実装

- Compositeパターンと同様の実装上の問題がある
- 構文解析
  - Interpreterパターンでは、抽象構文木の生成方法すなわち構文解析法までは説明していない
    - コンパイラーの教科書またはSmalltalkの実装を参考にすべし
  - Interpret()操作の定義
    - 必ずしも<<抽象式>>クラスの一族で実装する必要はない
    - Visitorパターンを使って、インタープリターを<<Visitor>>として実装する方が汎用性が高い
  - Flyweightパターンを使って、終端記号を共有する
    - SmalltalkのCharacterなど
- キャッシュ
  - 構文解析木の部分木をキャッシュする事で、効率を改善できる

Interpreterの実装 その2

- 部分実行
  - 一度実行した時に、効率化のための情報を保存し、次にそれを使う
- 並行実行
  - 部分木は、ある条件下で並行実行できる
  - 副作用がなければ、理論的には並行実行できる
- 遅延評価（Lazy Evaluation）
  - 実際に必要になるまで式の評価を行わないことで、プログラムの自由度を増す
    - 例：関数型言語Haskell, ConcurrentClean
  - 無限のリストなども扱えるようになる
- 状態遷移マシン
  - 有限状態遷移機械を実行する
  - インタープリターとして実装しておくと、状態遷移図の実装に役立つ

Interpreterの使用例

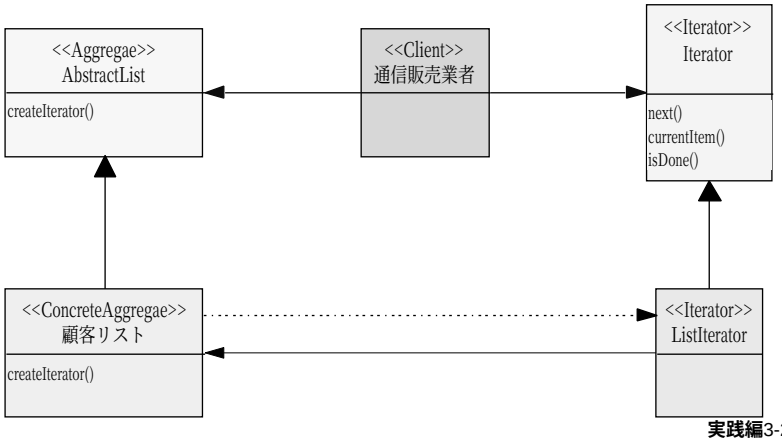
- ほとんどのOO言語のコンパイラー
- 野村証券第2次オンライン汎用大型機データ交換
- Smalltalkによるビジネスルール・エンジン
  - 保険商品仕様記述言語
- Dynamic Query Language
  - SQLに変換

Interpreterの関連パターン

- Compositeパターン
  - 抽象構文木はCompositeパターンのインスタンスである
- Flyweightパターン
  - 抽象構文木内で終端記号を共有する方法を示す
- Iteratorパターン
  - 抽象構文木内を操作するときに使える
- Visitorパターン
  - 抽象構文木内の各頂点の振る舞いを、一つのクラスにカプセル化する

Iterator（繰り返し）の比喩

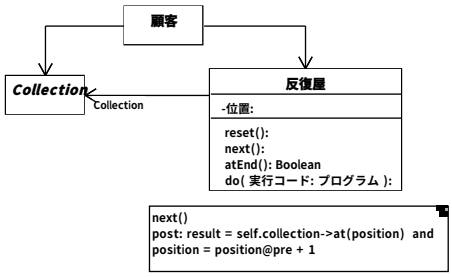
■オブジェクトの数を指定せず、順番に処理する



Iterator（繰り返し）

- 概要
  - オブジェクトの集まりを表すCollectionオブジェクトで、その内部表現を公開せずに、その要素に順番にアクセスするインターフェースを提供する
- 文脈
  - Collectionオブジェクトの内部表現を公開せず、その中のオブジェクトにアクセスしたい場合
  - 異なる構造のCollectionオブジェクトに対して、単一のインターフェース(多相性)を提供したい場合

Iteratorのクラス



Iteratorの配役

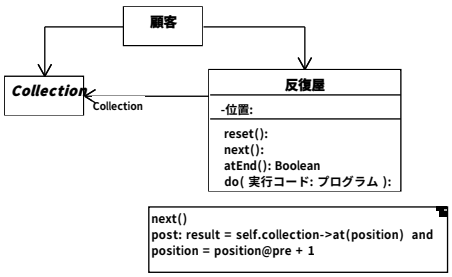
- 反復屋 Iterator
  - 要素にアクセスしたり操作するためのインターフェースを定義する
- Collection
  - <<反復屋>>を生成するためのインターフェースを定義する

Iteratorの結果

- <<Collection>>に対する種々の走査を提供する
- <<Collection>>クラスのインターフェースを単純化する
  - 走査のためのインターフェースは<<反復屋>>で提供するため
- ひとつの<<Collection>>に対して、複数の走査を実行できる

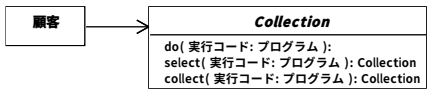
Iteratorの実装

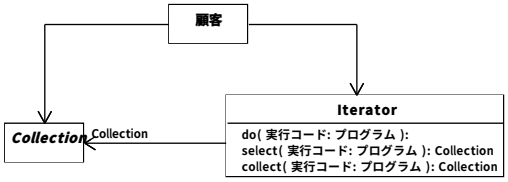
- 外部Iterator
  - <<顧客>>が制御する
    - <<顧客>>は<<反復屋>>に次の要素を明示的に要求せねばならない



Iteratorの実装

- 内部Iterator
  - 反復屋自身が制御する
    - <<顧客>>は<<反復屋>>に実行すべき操作を渡せばよい





Iteratorの実装

- Smalltalkでは<<反復屋>>を実装する必要がない
  - Collectionクラスにdo:を標準装備
- Robust　Iterator
  - 走査中に<<Collection>>の要素が変化しても、<<Collection>>のコピーをせずに、影響が出ないようにしたIterator

Iteratorの使用例

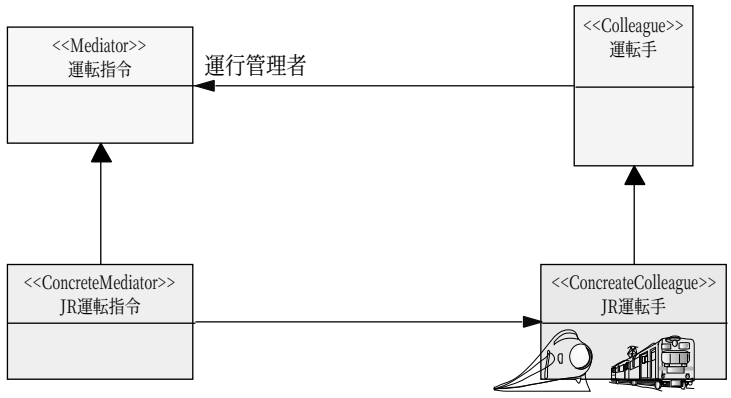
- SmalltalkのCollectionクラス群
- ET++のコンテナクラス
- Object Windowのコンテナクラス

Iteratorの関連パターン

- Command
  - **履歴リストにIteratorパターンを使う**
- Composite
  - **Iteratorパターンを使うことが多い**

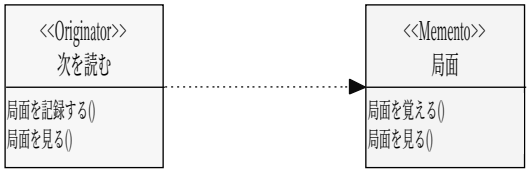
Mediator（調停者）の比喩

■オブジェクト群の相互作用をカプセル化するオブジェクトを定義する



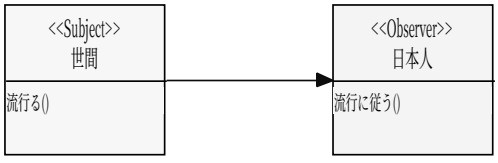
Memento（形見：Token）の比喩

- オブジェクト個別の内部状態を捉え、後でその状態に戻す



Observer（観察者）の比喩

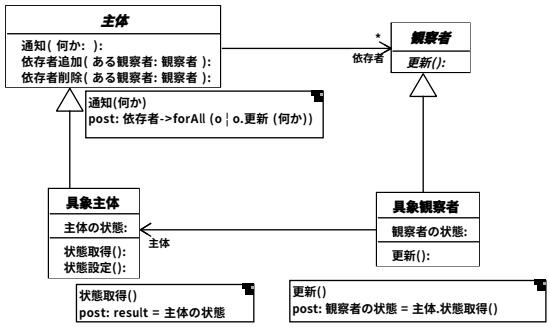
- オブジェクトが状態を変えたとき、それに依存したオブジェクトが自動的に更新される
  - 日本人方式



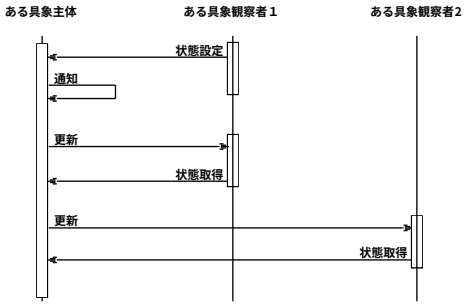
Observer（観察者）

- **概要** Synopsis
  - **あるオブジェクトが状態を変えたとき、それに依存するすべてのオブジェクトに通知する**
- **文脈(動機)**Context
  - **関連あるオブジェクト同士を、密に結合せずに連動させたい場合**
  - **あるオブジェクトに依存するオブジェクトを固定的に決められない場合**

Observerのクラス



Observerの順序図



Observerの配役

- 主体Subject
  - <<観察者>>を知っている
  - <<観察者>>を登録･削除するための操作を提供する
- 観察者Observer
  - <<主体>>の変化を受け取ったときの更新操作を定義する
- 具象主体Concrete Subject
  - <<具象観察者>>に影響する｢状態｣を持っている
  - 「状態」が変化したとき、<<具象観察者>>オブジェクト群に通知する
- 具象観察者Concrete Observer
  - <<具象主体>>への参照を持つ
  - <<具象主体> >>の｢状態｣と矛盾しないように、自身の「状態」を追従させる
  - <<観察者>>クラスで宣言した更新操作を具現する

Observerの結果

- <<主体>>と<<観察者>>クラス同士の抽象的結合
  - それぞれの<<観察者>>は、異なるGUI・アーキテクチャであっても問題がない
  - <<主体>>は可能な限り<<観察者>>から分離されているので、<<観察者>>は<<主体>>と独立に変更できる
- ブロードキャスト通信のサポート
  - メッセージの受け手を明確にしておく必要がない
  - <<観察者>>が多くなっても問題が少ない
- 不慮の更新の危険性
  - <<主体>>の変化に伴うコストの総計を<<観察者>>が予測することはできない

Observerの実装

- <<主体>>を<<観察者>>にマップする方法
  - <<観察者>>への参照を<<主体>>毎に持つと空間効率が悪くなる場合
  - ハッシュ表などを使って共同の検索表を実現する
- <<観察者>>が複数の<<主体>>に依存している場合
  - 更新()の引数に<<主体>>自身を追加し、どこからの更新()メッセージか分かるようにする
- 通知()操作をブロードキャストすべきか?
  - <<観察者>>側からポーリングする手もある
  - 不要な更新が少なくなり、効率はよいが更新タイミングがずれる可能性がある
- 削除された<<主体>>へのダングリング参照
  - <<主体>>を削除したとき、<<観察者>>内にダングリング参照(参照先がないリンク)が残らないようにする
  - <<主体>>が削除されたことを通知する手がある

Observerの実装 その２

- 更新プロトコル
  - pushモデル
    - 更新時に<<観察者>>にすべての情報を送る
      - <<主体>>と<<観察者>>の結合度が大きくなり、再利用性を低くする
  - pullモデル
    - 更新時に<<観察者>>に最小の情報を送り、後で<<観察者>>が<<主体>>に問い合わせる
      - 効率が悪くなる恐れがある
- 通知頻度の減少
  - <<観察者>>を特定の事象に対して登録できるようにする
    - 不要な更新が減るため効率が良くなる

Observerの実装 その３

- ChangeManagerオブジェクト
  - 複雑な更新をカプセル化する
    - 例えば、操作が複数の依存し合う<<主体>>に影響するとき、<<観察者>>に対する通知が何度も繰り返されないように、すべての<<主体>>が修正された後に1回だけ通知を送るようにする
  - ChangeManagerの責任
    - <<主体>>を<<観察者>>にマップし、これを維持していく操作を提供する
      - <<主体>>は<<観察者>>への参照を保持する必要がなくなる
    - 更新のための戦略を定義する
    - <<主体>>からの要求により、依存するすべての<<観察者>>を更新する
- 多重継承がない言語の場合
  - 例えばSmalltalkでは、Objectクラスで<<主体>>と<<観察者>>の両方の操作が定義されているため、すべてのクラスがどちらの役割でもこなせる

Observerの使用例

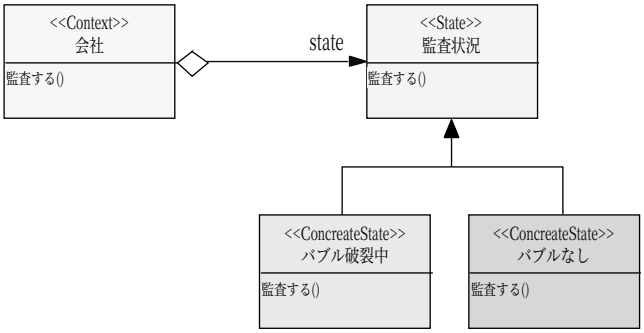
- MVCパターン
  - Modelが<<主体>>、Viewが<<観察者>>と対応する
- クライアント/サーバーパターン
  - サーバーが<<主体>>、クライアントが<<観察者>>と対応する
    - ただし、両者は異なるマシン上に実装されていることが多いので、エラー処理などでObserverパターンよりかなり複雑になる

Observerの関連パターン

- Mediatorパターン
  - ChangeManagerオブジェクトは<<Mediator>>として働く
- Singletonパターン
  - ChangeManagerオブジェクトは、<<主体>>と<<観察者>>の関係を統一的に制御するため、Singletonパターンを使って、自身を唯一のインスタンスとしている

State（状態）の比喩

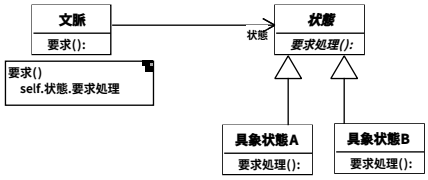
■オブジェクトの内部状態が変わったとき、振る舞いを変える



State（状態）

- 概要
  - オブジェクトの状態変化を、別なオブジェクトにカプセル化する
- 文脈
  - オブジェクトの状態変化が複雑な場合
  - オブジェクト状態の変化要求が多い場合

Stateのクラス



実践編3-47

Stateの配役

- 文脈Context
  - <<顧客>>に必要なインターフェースを定義する
  - 状態を表す<<具象状態>>クラスのインスタンスを保持する
- 状態State
  - <<文脈>>オブジェクトの個々の状態をカプセル化するためのインターフェースを提供する
- 具象状態Concrete State
  - <<文脈>>オブジェクトの一つの状態に関する振る舞いを具現する

実践編3-48

Stateの結果

- 状態に依存した振る舞いを局所化できる
- 状態が明確になる
  - オブジェクトの内部状態は、通常、変数やプログラムポインターに分散する

Stateの実装

- どのオブジェクトで状態遷移を実装するか？
  - <<文脈>>で実装すると柔軟性に問題が出る
  - <<状態>>のサブクラスで実装すると、サブクラス間の依存性が発生する
- 状態遷移マシンとの比較
  - 状態遷移マシンをシミュレートする実装との利害を評価する
    - 状態遷移マシンによる実装の方が柔軟性が高いが、実行速度で問題が出る場合がある
    - 言語により、状態遷移マシンを実装しづらい
- 状態オブジェクトの生成と破壊
  - 必要なときだけ生成する場合
    - 利用しないオブジェクトを生成することを避けることができる
  - 最初に生成しておく場合
    - 状態遷移が頻繁に起こる場合、生成･破壊の回数が減る

Stateの使用例

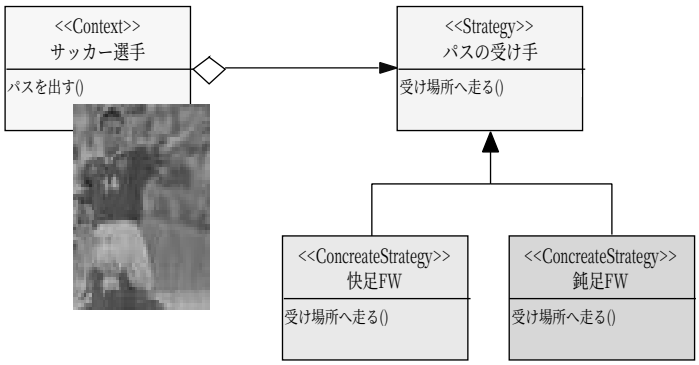
- HotDrawなどの描画ツールで、選択しているツールによってエディターの振る舞いを変えるために、このパターンを利用している
- SmalltalkのSMTPサーバー

Stateの関連パターン

- Singletonパターン
  - <<具象状態>>が<<Singleton>>であることが多い
- Strategyパターン
  - 似ているが...
    - 状態が少ない場合に選択することが多い
      - 状態が多い場合はStateパターンを使う
    - 状態を自分で変化させることが多い
      - Stateパターンでは、状態は外から変化させられる
    - <<文脈>>はStrategyの内容を隠蔽する
      - Stateパターンでは、内容が隠蔽されない

Strategy(**戦略**：Policy)**の比喩**

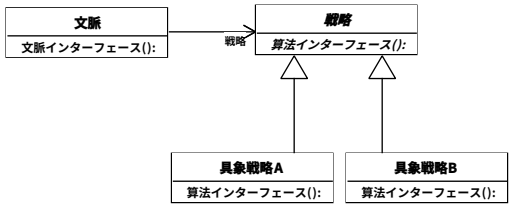
- アルゴリズム群の各々をカプセル化し、交換可能にする



Strategy(**戦略**)

- **概要**
  - **アルゴリズム(算法)をカプセル化し、交換可能にする**
- **文脈**
  - **複数の異なるアルゴリズムを使用する場合**
  - **アルゴリズムの変化が予測される場合**
  - **アルゴリズムが、<<顧客>>が知るべきでないデータを利用している場合**

Strategyのクラス



Strategyの配役

- **戦略** Strategy
  - サポートするアルゴリズムに共通のインターフェースを定義する
- **具象戦略** Concrete Strategy
  - アルゴリズムを具現する
- **文脈** Context
  - <<具象戦略>>オブジェクトを保持する
  - アルゴリズムを<<戦略>>に隔離する

Strategyの結果

- アルゴリズムの局所化が図れる
- インターフェースのオーバーヘッド
  - ある<<具象戦略>>に関係ないインターフェースを定義しなければならないことがある
- <<文脈>>クラス中の条件文を排除できる
- 異なる実装を提供できる
  - 時間と空間のトレードオフにより、適当なアルゴリズムの実装を選択できる

Strategyの実装

- <<戦略>>と<<文脈>>間のインターフェース
  - <<具象戦略>から<<文脈>>オブジェクトのデータを参照できなければならない
    - <<戦略>>操作の引数として渡す
    - <<文脈>>オブジェクト自身を引数として渡す
    - <<戦略>>オブジェクトから<<文脈>>への参照を持つ

Strategyの使用例

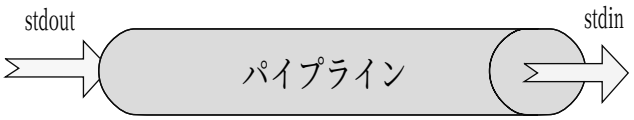
- 保険ポリシー
  - ビジネスロジックの戦略を切り替え
- 改行アルゴリズム
  - ET++、InterViewsで<<戦略>>を使って実装している
    - 例
      - 段落全体のバランスを考えた改行アルゴリズム
      - 各行が固定の項目数を持つような改行アルゴリズム
      - 各行に単純なテキストを入れるための改行アルゴリズム

Strategyの関連パターン

- Builder
  - Strategyパターンの特殊な形と見なせる
- Abstract Factory
  - Strategyパターンの特殊な形と見なせる
- State
  - 似ているが...
    - 状態が多い場合はStateパターンを使う
    - Stateパターンでは、状態は外から変化させられる
    - Stateパターンでは、内容が隠蔽されない

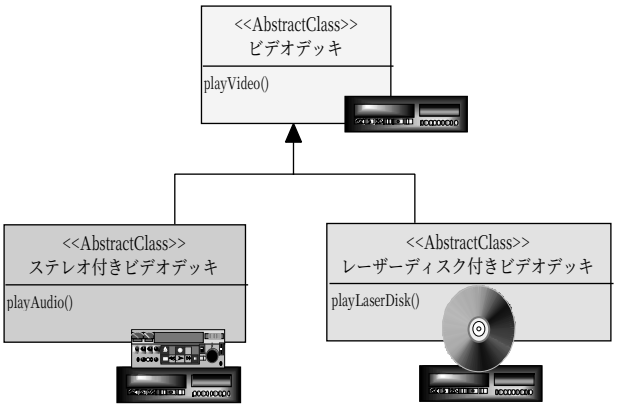
Stream（流れ）の比喩

■順序のある情報の流れを処理し、再利用できる処理単位とする
- ♦例
  - ▸UNIX pipeline
  - ▸HTML解釈
    - -Squeak HTML Server
    - -VisualWave
  - ▸永続的Object変換



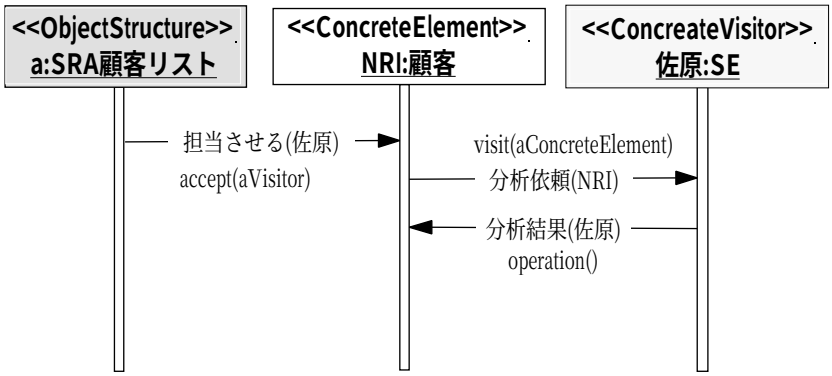
Template Method（型紙方式）の比喩

■クラスに関わるパターン
- ♦一部をサブクラスで実装するアルゴリズム



Visitor（訪問者）の比喩

■オブジェクト構造上の要素で実行される操作を表現する
◆アウトソーシング



II. 制約記述言語OCL

概要

Ver. 1.3 α

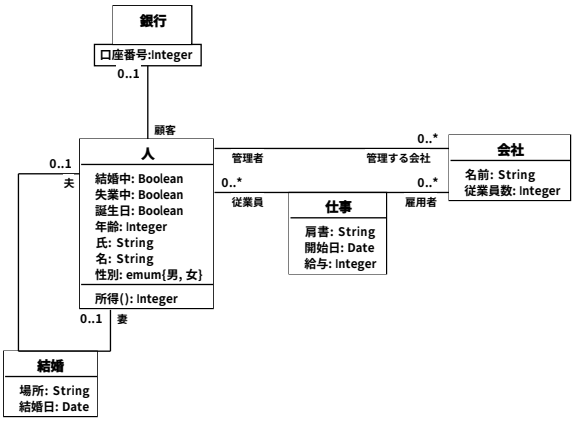
制約記述言語(OCL)

- IBMの保険部門で開発された
  - 起源はSyntropyメソッド
- 特徴
  - 参照の透過性
    - ･式や記述の一部をそれと等価なものに書き換えても、全体の評価値や意味が変わらないこと
      - ･正しさの証明やチェックがやりやすい
  - プログラミング言語ではない
    - プログラムロジックとフロー制御は書けない
    - 実装に関わることは表現できない
  - 型のある言語
  - 再帰可能
  - ASCII文字だけで構成される

使う場所

- クラスモデルのクラスおよび型の不変条件記述
  - self.allConnections->forAll
    (r1, r2 ¦ r1.名前 = r2.名前 **implies** r1 = r2)
- 操作とメソッドの前件と後件の記述
  - b **implies** (b2 : Boolean) : Boolean
    post: (not b) **or** (b and b2)
- ガードの記述
  - [**not** doorOpen]
- ナビゲーション言語として
  - **context** 人 **inv:**
    self.雇用者->isEmpty

説明に使用するクラス図の例



UMLメタモデルとの関係

- self
  - 着目しているインスタンス(contextual instance)の参照を示す
- 不変条件の一部として使う場合
  - **context** 会社 **inv:**
    self.従業員数 > 1000
  - **context** c : 会社 **inv:**
    c.従業員数 > 1000
- 前件と後件の一部として使う場合
  - **context** 型名::操作名(param1:Type1,...):返値
    pre : param1 > …
    post : result = …
  - **context** 人::所得(d : Date) : Integer
    post : **result** = self.年齢 * 200000

基本の型と値

型	値の例	演算子の例
Boolean	true, false	and, or, xor, not, implies, if-then-else
Integer	1,2,34,26524,...	*,+,-,/,abs
Real	1.5, 3.14,...	*,+,-,/,floor
String	'To eat or not to eat ...'	toUpper, concat

型について

- UMLモデルからの型
  - モデル上のすべての型とクラスは、OCLの型となる
- 列挙形
  - **enum**{#value1,#value2,#value3}
- 型の一致
  - 異なる種類の型同士の演算はできない
  - 型の一致規則
    - ある型はスーパー型と一致する
    - 型の一致は推移律を満たす
- 型の変換
  - object.oclAsType(Type2)
    -- objectをType2の型として評価

型	一致する型
Set	Collection
Sequence	Collection
Bag	Collection
Integer	Real

演算子の優先順位

- 優先順位の高い順に
  .演算子と->演算子
  単項notと単項ー
  ＊と／
  ＋と2項演算子ー
  and, or, xor
  implies
  if-then-else-endif
  <,>,<=,>=,=

未定義の値

- 照会結果が未定義の場合はocl式も未定義
  - ただし、
    - true or 式　なら、常にtrue
    - false and 式　なら、常にfalse

Let文

- context 人 inv:
  let 所得 : Integer = self.仕事.給与->sum in
  if 失業中 then
  　　所得 < 100
  else
  　　所得 >= 100
  endif

特性(property)

- 属性
  - context 人 inv:
    self.年齢 >= 0
- 関連の端(AssociationEnd)
  - ロール名など
    - context 会社
      inv: self.管理者.失業中 = false -- 人のインスタンス
      inv: self.従業員->size > 1000 -- size = インスタンスの集合の個数
      inv: self.従業員->notEmpty -- 従業員が居なければtrue
- 照会操作または照会メソッド
  - 照会(isQuery)≡状態変化(副作用)のないこと
    - 人.所得(aDate)
  - 操作を実装したものがメソッド

特性(property)の結合

- 日本の結婚条件
  - context 人 inv:
    self.妻->notEmpty implies
    (self.妻.年齢 >= 16 and self.妻.性別 = #女 and
    self.妻->size = 1) and
    self.夫->notEmpty implies
    (self.夫.年齢 >= 18 and self.夫.性別 = #男 and
    self.夫->size = 1)
- 会社には高々50人の従業員しかいない
  - context 会社 inv:
    self.従業員->size <= 50

ナビゲーション

- 関連クラスへのナビゲーション
  - context 人 inv:
    self.仕事.給与->sum = 9100000
- 関連クラスからのナビゲーション
  - context 仕事
    inv: self.雇用者.従業員数 > 0
    inv: self.従業員.年齢 >= 18
- 限定子のナビゲーション
  - context 銀行 inv:
    self.顧客 -- 銀行顧客の集合
  - context 銀行 inv:
    self.顧客[123456] -- 口座番号123456の人

Collection

- oclAny
  - Collection {abstract}
    - Bag -- 重複のある物の集まり(順序はない)
      - Bag {1, 3, 4, 3, 5, 1}
    - Set -- 数学の集合（重複はない）
      - Set{1, 2, 3, 4, 5, 6} ≡ Set{ Set{1, 2}, Set{3, 4}, Set{5, 6}}
    - Sequence -- 要素に順番がある物の集まり（重複がある）
      - Sequence{ 1, 3, 45, 2, 3}
        Sequence{1..10}≡
        Sequence{1..(6+4)}≡
        Sequence{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

後件中の「前の値」

- context 人::誕生日到来()
  post : age = age@pre + 1
  - age@preは、誕生日到来()開始時の年齢の値を参照
  - @pre記法は後件の定義中でのみ許される
- context 人::ダイエット(w)
  post : weight() = weight@pre() - w

selectとreject演算子

- select
  - 集合から抽出する
    - context 会社 inv:
    - -- 50歳より上の従業員の集合を求める
      - self.従業員->select(年齢 > 50)
      - self.従業員->select(p ¦ p.年齢 > 50) -- 同上
      - self.従業員->select(p : 人 ¦ p.年齢 > 50) -- 同上
    - -- 50歳以下の従業員の集合を求める
      - self.従業員->reject(p.年齢 > 50)
    - -- 18以上の未婚従業員
      - self.従業員->reject(p.年齢 < 18 or p.結婚中)

collect演算子

- context 会社 inv:
  - -- 全従業員の誕生日のBag(重複有り)
    - self.従業員->collect(誕生日)
  - -- 全従業員の誕生日のSet(重複無し)
    - self.従業員->collect(誕生日)->asSet
- Collectの省略
  - collection->collect(propertyname) = collection.propertyname
    - self.従業員->collect(誕生日) = self.従業員.誕生日

限量子（forAll, exists）

- forAll（∀）
  - context 会社 inv:
  - -- 全従業員の姓が異なっていればtrue
    - self.従業員->forAll
      (e1, e2 | e1<>e2 implies e1.氏 <> e2.氏)
- exitsts（∃）
  - context 会社 inv:
  - -- 一人でも'伸'がいればtrue
    - self.従業員->exists(p : 人 | p.名 = '伸')

繰り返し(iterator)

- collection->iterate
  (elem : Type; acc : Type = 式 ¦ elemとaccを含む式)
- collection->collect
  (x : T ¦ x.property) = collection->iterate
  (x : T; acc : T2 = Bag{} ¦ acc->including(x.property))

定義済みの型(OclType)

- OclType
  - type.name :String -- typeの名前
  - type.attributes : Set(String) -- typeの属性名の集合
  - -- typeのassociationEnd名の集合
    type.associationEnds : Set(String)
  - type.operation : Set(String) -- typeの操作名の集合
  - -- typeのすべての直接のスーパー型の集合
    type.supertype : Set(OclType)
  - -- typeのすべてのスーパー型の集合の推移閉包
    type.allSupertype : Set(OclType)
  - -- typeとサブ型の全インスタンスの集合
    type.allInstances : Set(type)
    - Person.allInstances->forAll(p1, p2 ¦ p1 <> p2 implies p1.name <> p2.name)

OclAny

- OclAny
  - -- objectとobject2が同一オブジェクトならtrue
    object = (object2 : OclAny) : Boolean
  - -- objectとobject2が同一オブジェクトでなければtrue
    object <> (object2 : OclAny) : Boolean
    - post : result = not (object = object2)
  - object.oclType : OclType -- objectの型
  - -- typeがオブジェクトの型のスーパー型であればtrue
    object.oclIsKindOf(type : OclType) : Boolean
    - post : result = type.allSupertypes->includes(object.oclType) or
      type = object->oclType
  - object.oclIsTypeOf(type : OclType) : Boolean
    - post : result = (object.oclType = type)
  - -- オブジェクトが状態sになっていればtrue
  - oclIsInState(s : Enumeration) : Boolean
  - -- 後件中のみ使え、その操作中でオブジェクトが生成されていればtrue
  - oclIsNew : Boolean

Real

- Real
  - =,+,-,*,/,<.>,<=.>=
  - r.abs : Real
  - r.floor : Integer
  - r.max(r2 : Real) : Real
  - r.min(r2 : Real) : Real

Integer

- Integer
  - =,+,-, *,/
  - i.abs : Integer
  - i.div(i2 : Integer) : Integer -- 整数演算の商
  - i.mod(i2 : Integer) : Integer -- 余り
  - i.max(i2 : Integer) : Integer
  - i.min(i2 : Integer) : Integer

String

- String
  - =
  - string.size : Integer -- 文字列の字数
  - string.concat(string2 : String) : String -- stringとstring2の連接
  - string.toUpper : String
  - string:toLower : String
  - string.substring(lower : Integer, upper : Integer) : String

Boolean, Enumeration

- **Boolean**
  - **=, or, xor, and, not, implies**
  - **if b then (expression1 : OclExpression)**
    else (expression2 : OclExpression) endif :
    expression1.evaluationType
- **Enumeration**
  - **enumeration = (enumeration2 : Boolean) : Boolean**
  - **enumeration <> (enumeration2 : Boolean) : Boolean**

Collection

- **Collection**
  - **collection->size : Integer -- collectionの個数**
  - **-- objectがcollectionに含まれているか？**
    collection->includes(object : OclAny) : Boolean
  - **-- collectionに含まれているobjectの個数**
    collection->count(object : OclAny) : Integer
  - **collection->isEmpty : Boolean -- collectionが空か？**
  - **collection->notEmpty : Boolean -- collectionが空でないか？**
  - **collection->sum : T -- collectionの全要素の合計**
  - **collection->exists(expr : OclExpression) : Boolean**
  - **collection->forAll(expr : OclExpression) : Boolean**
  - **collection->isUnique(expr : OclExpression) : Boolean**
    - **post: result = collection->collect(expr)->forAll(e1, e2 ¦ e1 <> e2)**
  - **collection->sortedBy(expr : OclExpression) : Boolean**

Set（Bagも同様）

- Set
  - set->union(set2 : Set(T)) : Set(T) -- set ∪set2
  - set->union(bag : Bag(T)) : Bag(T) -- set asBag∪bag
  - set = (set2 : Set) : Boolean -- set = set2
  - set->intersection(set2 : Set(T)) : Set(T) -- set ∩set2
  - set->intersection(bag : Bag(T)) : Set(T) -- set ∩bag asSet
  - set - (set2 : Set(T)) : Set(T) -- set￥set2。set2の要素を除くset
  - set->including(object : T) : Set(T) -- set ∪ {object}
  - set->excluding(object : T) : Set(T) -- set￥{object}
  - -- (set ∪set2) ￥(set ∩set2)
    set->symmetricDifference(set2 : Set(T)) : Set(T)
  - set->asSequence : Sequence(T)
  - set->asBag : Bag(T)

Sequence

- Sequence
  - Setと同様の演算の他に
    - sequence->append(object : T) : Sequence(T) -- 後ろへ追加
    - sequence->prepend(object : T) : Sequence(T) -- 前へ追加
    - sequence->subSequence(lower : Integer, upper : Integer) : Sequence(T)
    - sequence->at(i : Integer) : T -- i番目の要素
    - sequence->first : T -- 最初の要素
    - sequence->last : T -- 最後の要素

例1(Singletonパターン)

- context Singleton::唯一インスタンス生成()
  - post: self.oclType.allInstances->size =1

例2(平方根)

- 平方根を求める操作の仕様
  - context Real::root(x : Real) : Real
  - pre: x >= 0.0
  - post: Real->exists(r : Real | r * r = x and r >= 0.0 and result = r)

例3(全順序)

- 要素がIntegerであるCollectionクラスのインスタンスが{sorted}(全順序)である
- **context** Collection **inv:**
  Collection->allInstances->exists(c ¦ Integer->forAll(i1, i2 ¦
  i1 > 0 **and** i2 > 0 **and** i1 <= c->size **and** i2 <= c->size **and**
  i1 < i2 **implies** c->at(i1) <= c->at(i2)))

例4(図書館の貸出管理)

- **context** 利用者::返却(ある本実体 : 本実体):本実体
- **pre:**
  self.借用本->notEmpty **and** ある本実体->notEmpty **and**
  self.借用本->includes(ある本実体)
- **post:**
  Set(本実体) = Set(本実体)@pre->including(ある本実体) **and**
  self.貸出->size = self.貸出@pre->size ー 1 **and**
  ある本実体.貸出可能か = true **and**
  result = ある本実体

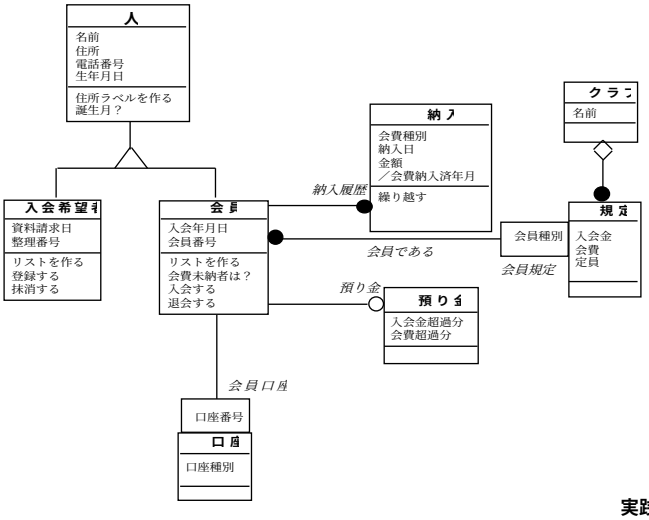
例5(図書館の貸出管理その2)

- context 利用者::借用可能か : Boolean
- pre: --貸出あり
  self.借用本->notEmpty
- post: --貸出あり
  result = self.図書館.最大貸出数 >= self.借用本->size and
  self.貸出->forAll(返却日 >= Date.現在日)
- pre: --貸出なし
  - self.借用本->isEmpty
- post: --貸出なし
  - result = true

OCL演習

- 排他的論理和　b xor (b2 : Boolean) : Booleanの後件を書け
- 継承は循環しない
  - あるクラスのスーパークラスは、self.oclType.allSupertypesで指定できる
- sequenceの後ろへobjectを追加する
  - sequence->append (object: T) : Sequence(T)の後件を書け

OCL演習



OCL演習

- 以下の制約をOCLで記述せよ
  - 振り込まれた入会金の内、余分な入会金は預り金とする
  - 「入会希望者でありかつ会員である」ということはない
  - 会員は定員以内
  - 「規定」で、会員種別の重複はない
  - 正会員は正会員の定員以内である

III. デザインパターンによるオブジェクト指向分析設計

エレベータ問題

エレベータ問題

- N個のエレベーターシステムがM階のビルに設置されている。エレベーターと制御メカニズムはメーカーから供給されている。内部のメカニズムは下に与えられている。
- 以下のルール（簡易版UseCase）にしたがってビル内の各階を移動するエレベータのロジックを設計せよ。
- 各エレベータは、各階に対応したボタンの集合を持つ。
  - これらのボタンを押すと点灯し、対応する階へ行く。ある階で止まると、その階に対応するボタンは消灯する。
- 各階は２つのボタンを持つ（最上階と最下階を除く）。
  - １つは上に行くエレベータの要求で、もう一つは下に行くエレベータの要求である。これらのボタンは押されると点灯する。エレベータがその階で止まると、ボタンは消灯する。エレベータはこの後、要求された方向に向かうか、どちらに向かうか未解決のままその階に留まる。後者のケースで、その階の両方の要求ボタンが点灯していたら、１つだけが消灯する。どちらの要求を最初に受け付けるかは、どちらが待ち時間が少ないかによる。

エレベータ問題　続き

- エレベータが何の要求も受けていないとき、最後の訪問階でドアを締めて、待機する。
- 各階からの要求は全部実行しなければならず、各階は同じプライオリティを持つ。
- エレベータからの各階行きの要求は全部実行しなければならず、エレベータの行く方向に順番に行く。
- 各エレベータは緊急ボタンを持ち、これを押すと管理者に警報が送られる。そして、エレベータは「サービス停止」状態になる。各エレベータは、「サービス停止」を解除する機構を持つ。

オブジェクト指向分析/設計の手順

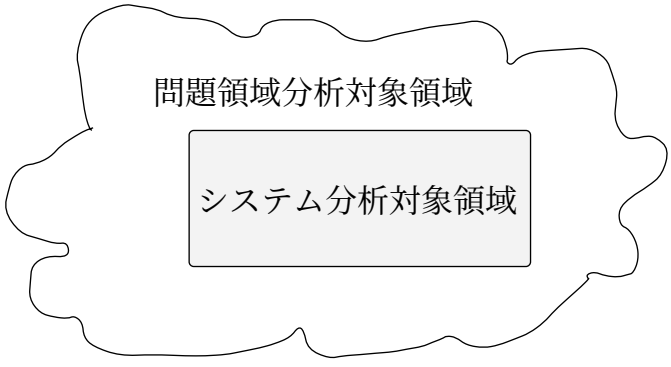
問題領域分析
システム分析
設計

各プロセス共通の主なステップ

- モデルの作成&修正
  - クラスとオブジェクトの認識
  - パターンがあればパターン適用
    - デザインパターン
    - 分析パターン
    - プロセス･パターン
    - アーキテクチャパターン
    - ...
- 勘所(Hotspot)の発見
  - 不変箇所と変更多発箇所(Hotspot)
- 仕様記述
- 場合によってはプロトタイプ作成
- 検証

1. 問題領域(ドメイン)分析

- 対象問題領域では「何をやっているか」をモデル化する
  - システム化する対象よりやや大きい対象領域を考える



1. 問題領域分析の手順

- 1.1振る舞い要求の発見と記述
  - 今の場合は、すでに問題に記述されている
    - 通常は
      - UseCaseなどで要求を発見していく
      - 制約の発見
- 1.2ドメインオブジェクトを見つけ、定義する
  - オブジェクトの名前･責任・役割を見つける
- 1.3ドメインオブジェクトを分類し、関連･包含関係を見つけ定義する
- 1.4クラス図を書く

1. 問題領域分析の手順　続き

- 1.5オブジェクトの状態変化の記述
  - 順序図の作成
  - 状態遷移図の作成
- 1.6オブジェクトの類型(ステレオタイプ)を発見する
  - 一種のパターン
- 1.7制約の発見と記述
  - 仕様記述言語による記述
- 1.8要求の検証

1.2ドメインオブジェクトの発見

- N個のエレベーターシステムがM階のビルに設置されている。エレベーターと制御メカニズム（<<Controller>>?）はメーカーから供給されている。内部のメカニズムは下に与えられている。
- 以下のルール（<<Strategy>>?）（簡易版UseCase）にしたがってビル内の各階を移動するエレベータのロジックを設計せよ。
- 各エレベータは、各階に対応したボタンの集合を持つ。
  - これらのボタンを押すと点灯し、対応する階へ行く。ある階で止まると、その階に対応するボタンは消灯する。
- 各階は２つのボタンを持つ（最上階と最下階を除く）。
  - １つは上に行くエレベータの要求で、もう一つは下に行くエレベータの要求である。これらのボタンは押されると点灯する。エレベータがその階で止まると、ボタンは消灯する。エレベータはこの後、要求された方向に向かうか、どちらに向かうか未解決のままその階に留まる。後者のケースで、その階の両方の要求ボタンが点灯していたら、１つだけが消灯する。どちらの要求を最初に受け付けるかは、どちらが待ち時間が少ないかによる。

1.2ドメインオブジェクトの発見続き

- エレベータが何の要求も受けていないとき、最後の訪問階でドアを締めて、待機する。
- 各階からの要求（<<Command>>?）は全部実行しなければならず、各階は同じプライオリティ（<<Strategy>>）を持つ。
- エレベータからの各階行きの要求は全部実行しなければならず、エレベータの行く方向に順番に行く。
- 各エレベータは緊急ボタンを持ち、これを押すと管理者に警報が送られる。そして、エレベータは「サービス停止」状態になる。各エレベータは、「サービス停止」を解除する機構を持つ。

1.2.1クラスの候補

- 問題記述文から得られたクラス
  - エレベーター、階、制御メカニズム、ルール、（エレベータ内の）行き先ボタン、（各階にある）上行きボタン、（各階にある）下行きボタン、最上階、最下階、要求、ドア、プライオリティ、方向、緊急ボタン、管理者、警報、機構
- 問題領域の知識から得られたクラス
  - コントローラー

1.2.1クラスの吟味

- 冗長なクラスの排除
  - 制御メカニズムとコントローラーでは、コントローラの方が記述的
- 無関係なクラスの排除
  - 問題の中でほとんど使われないクラスを除く
    - 「ドア」「管理者」「警報」を除く
      - 本格的なエレベーター･システムでは必要だろう
- 曖昧なクラスの排除
  - 対象範囲が曖昧なものは除く
    - 「機構」「ルール」
      - 「ルール」は、設計時に<<Strategy>>として復活するかもしれない

1.2.1クラスの吟味　続き

- 属性の排除
  - 属性になるものは除く
    - 「方向」(おそらく状態を表す属性になる)「プライオリティ」(将来は、使うかもしれない)
- 正しそうなクラス
  - エレベーター、階、行き先ボタン、上行きボタン、下行きボタン、最上階、最下階、緊急ボタン、コントローラー

1.2.2クラスの責任と役割

- コントローラー
  - エレベータの制御を行う
    - エレベータの選択を行う
    - 各エレベータの動きを決定する
- エレベーター
  - 客を運ぶ
- 階
  - 客の所在を表す
- ボタン
  - 客の要求を受け、他に通知する

1.3.1関連を見つける

- N個のエレベーターシステムがM階のビルに設置されている。エレベーターと制御メカニズム（<<Controller>>?）はメーカーから供給されている。内部のメカニズムは下に与えられている。
- 以下のルール（<<Strategy>>?）（簡易版UseCase）にしたがってビル内の各階を移動するエレベータのロジックを設計せよ。
- 各エレベータは、各階に対応したボタンの集合を持つ。
- これらのボタンを押すと点灯し、対応する階へ行く。ある階で止まると、その階に対応するボタンは消灯する。
- 各階は２つのボタンを持つ（最上階と最下階を除く）。
- １つは上に行くエレベータの要求で、もう一つは下に行くエレベータの要求である。これらのボタンは押されると点灯する。エレベータがその階で止まると、ボタンは消灯する。エレベータはこの後、要求された方向に向かうか、どちらに向かうか未解決のままその階に留まる。後者のケースで、その階の両方の要求ボタンが点灯していたら、１つだけが消灯する。どちらの要求を最初に受け付けるかは、どちらが待ち時間が少ないかによる。

1.3.1関連を見つける　続き

- エレベータが何の要求も受けていないとき、最後の訪問階でドアを締めて、待機する。
- 各階からの要求（<<Command>>?）は全部実行しなければならず、各階は同じプライオリティ（<<Strategy>>）を持つ。
- エレベータからの各階行きの要求は全部実行しなければならず、エレベータの行く方向に順番に行く。
- 各エレベータは緊急ボタンを持ち、これを押すと管理者に警報が送られる。そして、エレベータは「サービス停止」状態になる。各エレベータは、「サービス停止」を解除する機構を持つ。

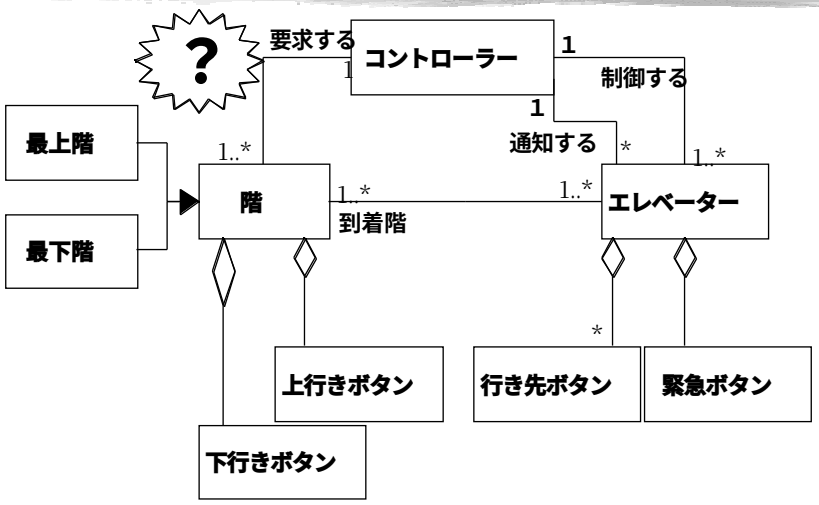
1.3.1関連の吟味

- N個のエレベーターシステムがM階のビルに設置されている
- 各階を移動するエレベータ
  - 上の２つは同じ関連と見なせる（エレベータは各階を移動する）
- エレベーターと制御メカニズムはメーカーから供給されている
- エレベータが何の要求も受けていない
  - 上の２つは同じ関連と見なせる（コントローラーはエレベーターを制御する）
- 各エレベータは、各階に対応したボタンの集合を持つ
  - エレベータは行き先ボタンを持つ
- 各階は２つのボタンを持つ
  - 階は上行きボタンを持つ
  - 階は下行きボタンを持つ
- 各階からの要求は全部実行しなければならず
  - 各階からの要求を、コントローラに通知する
- エレベータからの各階行きの要求は全部実行しなければならず
  - エレベータからの要求を、コントローラに通知する
- 各エレベータは緊急ボタンを持ち
  - エレベータは緊急ボタンを持つ

1.3.2関連の主要な特徴を記述する

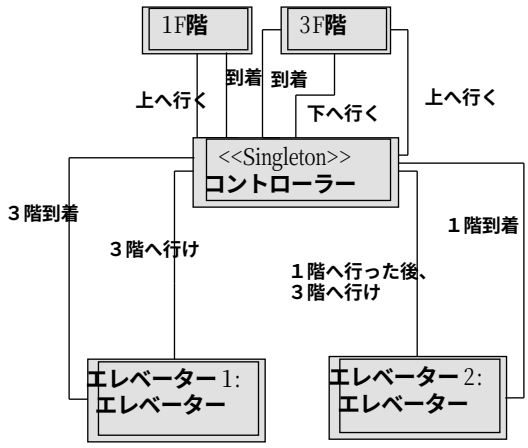
- 多重度
  - 両方のクラスのインスタンス同士の多重度を記述する
  - 良く分からないときはオブジェクト図を使って説明し、ユーザーの知識を得る
- ロール
  - そのクラスのオブジェクトの役割を記述する
- 限定子
  - 相手のオブジェクトの集合を限定できる名前などの限定子がないか調べる

1.4クラス図のたたき台を書く



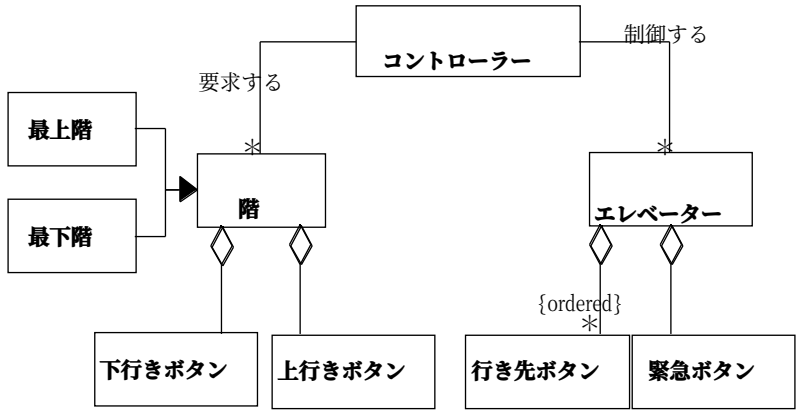
実践編5エレベータ問題-19

1.4オブジェクト図で ドメインエキスパートに確認



実践編5エレベータ問題-20

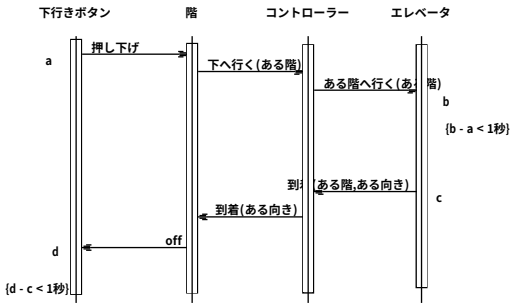
1.4クラス図の改良



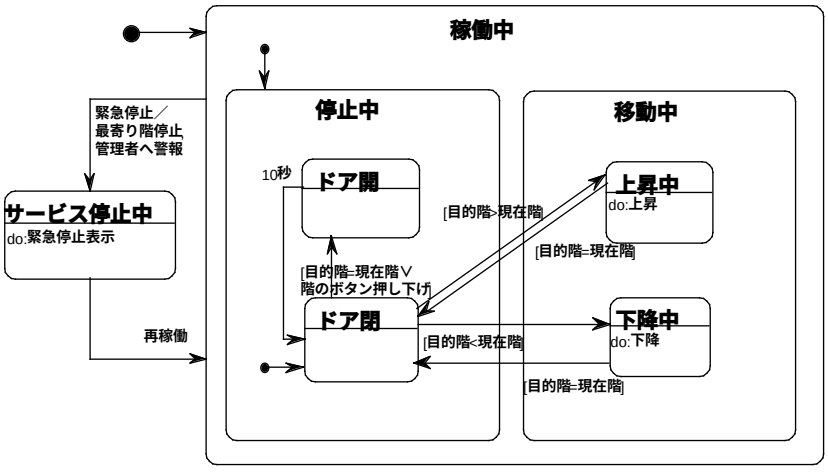
1.5オブジェクトの状態変化の記述

- 1.5.1順序図の作成
  - Rational Unified Processでは...
    - 1.1振る舞い要求の発見と記述で順序図を使う
- 1.5.2状態遷移図の作成
  - オブジェクトの内部状態の変化を記述する
    - オブジェクトは一種の仮想マシン≡状態マシン
      - 状態遷移図の記述は、仮想マシンの形式仕様

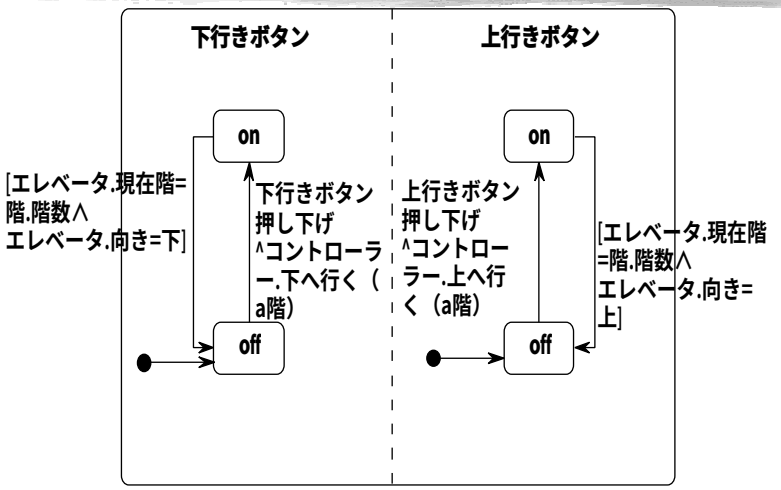
1.5.1順序図の作成



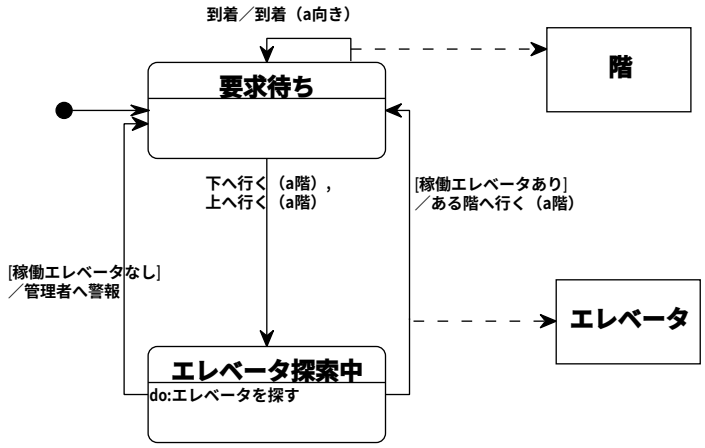
1.5.2状態遷移図の作成　エレベーター



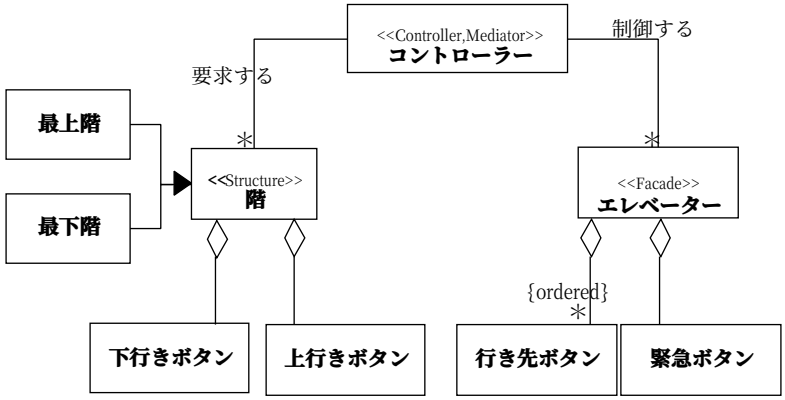
1.5.2状態遷移図の作成　階



1.5.2状態遷移図の作成 コントローラー



1.6類型(ステレオタイプ）を発見する



1.7制約の発見と記述

- 境界条件
  - isFloor(f:Integer) : Boolean
  - **post**: f >= 最下階 **and** f <= 最上階
- 安全性(エレベータの不変条件)
  - エレベータの移動中は、ドアが閉まっている
  - エレベータが階に停止中は、ドアが開いている
- 有用性
  - 上の階への要求がない　向き = 下
  - 下の階への要求がない　向き = 上

制約に関する演習問題

- 前に記述した状態遷移図の中で、前ページの制約に反するものがある
  - どこがおかしいか指摘せよ

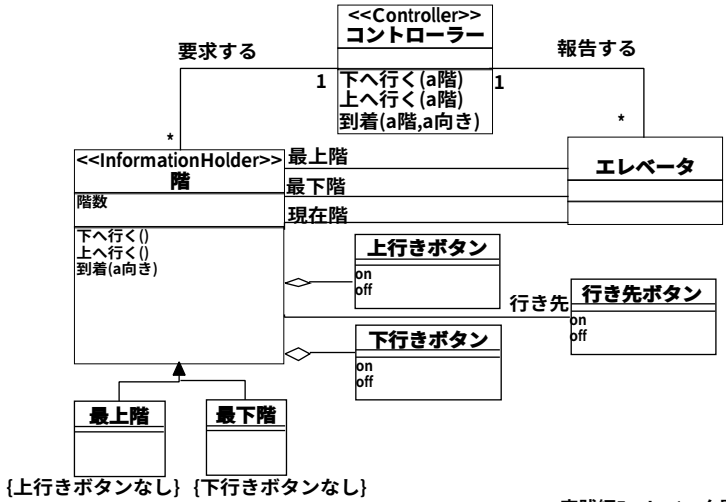
1.8要求の検証

- シナリオに基づきテストする
  - クラス図のアクセス経路をテストする
    - 情報にアクセスするパスは存在するか？
- 机上またはCASEツールで状態遷移図を実行し、テストする
  - Rational Rose for Realtime
  - Statecharts

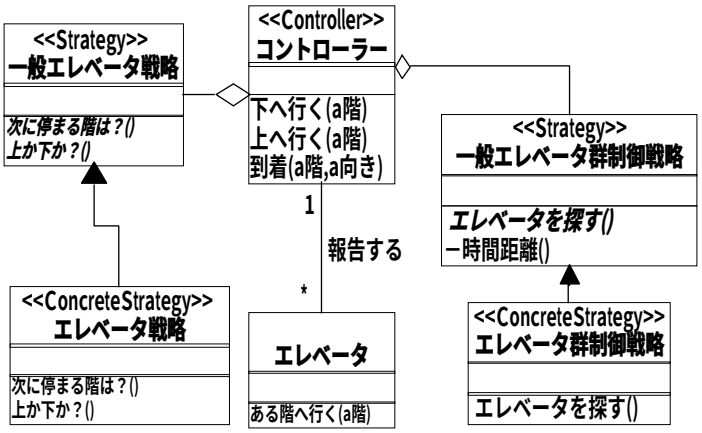
2.システム分析

- 2.1クラス図(静的構造図）の精密化
  - デザインパターンの適用を検討する
  - ドメインモデルより詳細な要求仕様の作成
    - 操作仕様に制約を記述する
- 2.2状態遷移図の精密化
  - この例の場合は、ドメイン分析の場合と同じ
- 2.3分析モデルの検証
  - 机上または仕様記述言語のインタープリターで操作仕様を実行しテストする。
    - VDM-SL

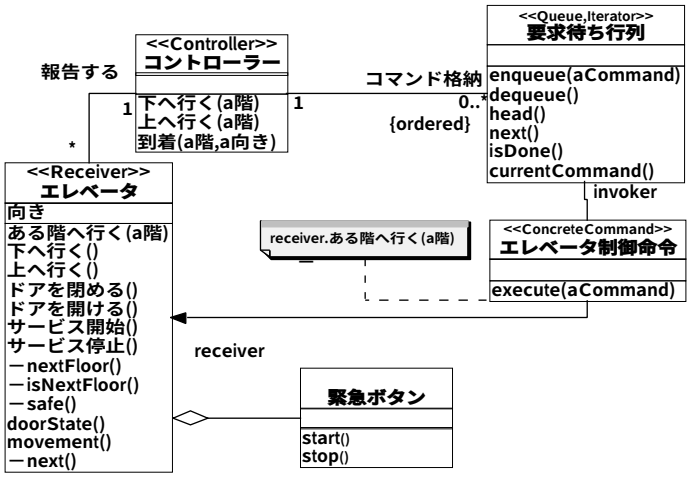
2.1クラス図の精密化　全体図



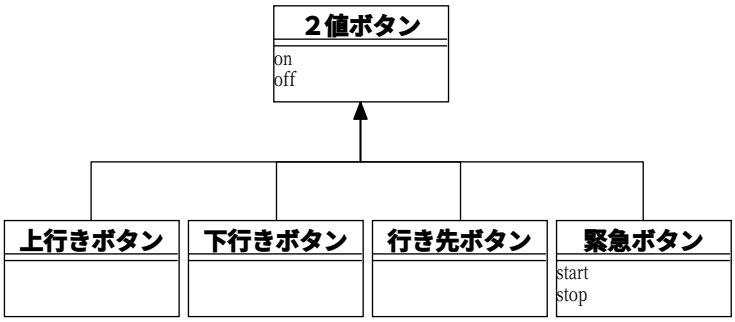
2.1クラス図の精密化　コントローラー



2.1クラス図の精密化　エレベーター



2.1クラス図の精密化　ボタン



2.1クラス図の精密化　操作仕様例

- **context** エレベータ群制御戦略::
  エレベータを探す(es : Set(エレベータ), f : 階) : エレベータ
- **pre**: es->size > 1
- **post**: es->size = 1 **and**
  es->forAll(e : エレベータ ¦ es->exists
  (e1 : エレベータ ¦ 時間距離(e1,f) <= 時間距離(e,f)))
- -- e1は、すべてのエレベータの中で、時間距離が最小である

2.1クラス図の精密化 操作仕様例 その2

- context 一般エレベータ群制御戦略::
- 時間距離(e : エレベータ, f : 階) : 時間
- post: if e.現在階 < f then
- if e.向き= 上 then return = 移動時間(f - e.現在階)
- else -- そのエレベータの最下階まで行って戻ってくる可能性がある
- return = 移動時間(e.現在階 + f - 2 * e.最下階)
- endif
- else if e.現在階 = f then return = 0
- else if e.向き= 下 then return = 移動時間(e.現在階 - f)
- else -- そのエレベータの最上階まで行って戻ってくる可能性がある
- return = 移動時間(2 * e.最上階 - e.現在階 + f)
- endif
- endif

仕様の検証に関する演習問題

- 「時間距離」操作の仕様を検証せよ

2.1クラス図の精密化 操作仕様例 その３

- **context** エレベータ::safe(e : エレベータ) : Boolean
- **post:** 階->forAll(f : 階 ¦ (doorState(e, f) = #開) **implies** (movement(e) = #停止中 **and** e.現在階 = f))
- -- 「ある階でエレベータのドアが開いているならば、エレベータが停止中で、かつ、エレベータの現在階とある階が 等しい 」時、safe()は真になる

- **context** エレベータ:: doorState(e : エレベータ, f : 階) : ドア状態
- **context** エレベータ:: movement(e : エレベータ) : 動作

2.1クラス図の精密化 操作仕様例 その４

- **context** エレベータ::nextFloor(d : 向き, f : 階) : 階
- **pre:** isNextFloor(d,f)
- **post:** isNextFloor(d,f) **and**
- **if** d = #上 **then** return = f + 1 **else** return = f - 1 **endif**

- **context** エレベータ:: isNextFloor(d : 向き, f : 階) : Boolean
- -- 最上階で上へは行けない、最下階でも同様 */
- **post** : **if** d = #上 **then**
- return = (f < 最上階)
- **else**
- return = (f > 最下階)
- **endif**

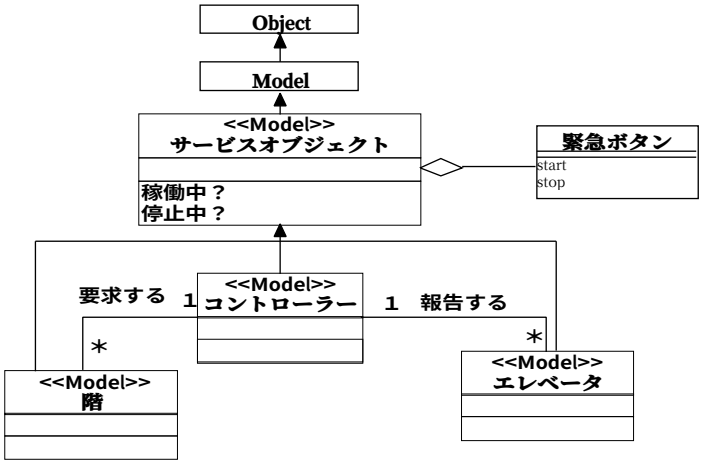
3.設計

- 3.1設計モデルを作成する
  - 3.1.1分析モデルをHowToの視点で書き直す
  - 3.1.2効率の検討を行う
  - 3.1.3再利用性･保守性を分析する
- 3.2設計モデルを検証する

3.1.1分析モデルをHowToの視点で書き直す

- デザインパターン適用を検討する
  - この例題では、すでにやりすぎ？
- 状態遷移の実装方法を決める
- 関連の実装方法を決める
  - 1対1関連は、オブジェクトへのポインター
  - 1対多関連は、Collectionのサブクラス
    - {Sorted}の場合はSortedCollection
- 宣言的仕様を段階的に詳細化する

3.1.1クラス図の変更



3.1.1状態遷移の実装方法を決める

- 状態遷移を行う操作next()で実装する
  - 特に大きなオブジェクトではなく、効率上も特に問題は考えられないので、Stateパターンの使用は見送る
- next(r : 要求, e : エレベータ) : エレベータ
  - enum {#ここに, #後へ, #前に} -- 要求

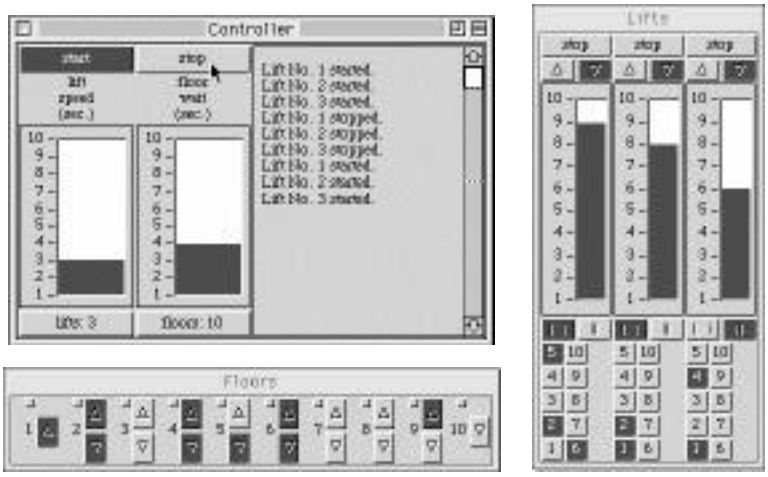
3.1.1宣言的仕様を段階的に洗練する

- variable 最小距離 : 時間, result : エレベータ -- RSLによる手続き的仕様
- operation
- エレベータを探す : エレベータω×階 →write 最小距離, result Unit,
- ∀e :エレベータω, f : 階・エレベータを探す(e, f) ≡
- 最小距離 := 時間距離(hd e,f); result := hd e
- while tl e ≠ 〈〉 do
- if 時間距離(hd tl e,f) ≤ 最小距離 then
- 最小距離 := 時間距離(hd tl e,f); result := hd tl e
- end
- e := tl e
- end
- pre card e > 1
- post card e = 1 ∧ 時間距離(e1,f) ≤ 時間距離(e,f)
- end

3.1.2効率の検討を行う

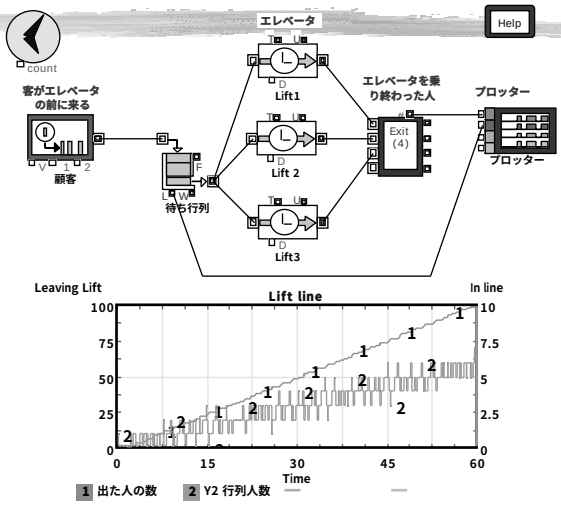
- 各オブジェクトの件数
  - エレベータのように高々10件、ボタンから発生する要求は高々数百件であり、さして効率上の問題は発生しない
  - 各エレベータ毎にどの程度の時間粒度で時間距離()の計算を行うかは、最適組み合わせ問題となり、効率上の問題が発生する可能性が強い
    - ここでは、要求のある階へ行くエレベータを一度近似解で決定したら、最適解を求めるための「再計算」をせず、効率向上を図る

3.1.2効率の検討を行うシミュレーション



3.1.2効率の検討を行うシミュレーション

- ある階での待ち行列の分析例
  - 待ち行列の長さ
    - 2.93
  - 平均待ち時間
    - 1.60
  - 最大の長さ
    - 7
  - 最大の待ち時間
    - 3.3



3.1.3再利用性･保守性を分析する

- 汎用性のある抽象データ型ができているか？
  - 抽象クラス
    - 「一般エレベータ群制御戦略」「一般エレベータ戦略」
      - エレベータとその群管理の移動戦略を切り替えたり保守するのが容易
    - 「エレベータ制御命令」のスーパークラスとして「制御命令」といった抽象クラスを考えることもできる
    - 要求待ち行列は、抽象データ型Queueを具象している
  - コントローラー
    - 通常は機能が多く複雑になりがち
      - 今回の設計では各種の「戦略」と「要求待ち行列」と「エレベータ制御命令」とに役割分担をしているので、再利用性と保守性が高い
  - デザインパターン
    - Strategy, Iterator, Commandの3種類、合計４つのパターンを使用
    - コントローラー・エレベーターなどはMVCパターンのモデルにマップできる

3.1.3再利用性･保守性を分析する

- 勘所
  - 機能の変更が多いであろうエレベータの動き方は、「戦略」クラスにカプセル化している
  - 他のクラスはほとんど変化する可能性がないし、変化するとしてもそのクラス内の実装を変えるだけで良さそうである
- 構造化されたモジュール
  - 各クラスは、高々操作数１０個程度であり、実装時にも２０～３０個程度に収まる
  - インターフェースは単純で、共通データのようなものはない

3.2設計モデルを検証する

- UseCase・シナリオに基づき、テストする
  - 机上で操作仕様をチェックする
    - 具体的データを用意してインスペクションを行う
    - 机上で状態遷移図をチェックする
    - クラス図に、状態遷移図では動作や活動で表されている、すべての操作が反映されているかチェックする
  - または、インタープリタで操作仕様をチェックする
    - VDM-SL, Concurrent Clean, Smalltalk

Ⅳ. デザインパターンの適用「例題による総合演習」

- 1.図書館問題
- 2.巡航制御システム

1.図書館問題

- 以下のトランザクションが発生する図書館の貸出管理処理を考えよ。
  - 本の貸出および返却
  - 図書館への本の登録および削除
  - 特定の作者の本、あるいは特定の分野の本に関するリストの作成
  - 特定の利用者が借りている本のリストの作成
  - 特定の本を最後に借りた人の検索
- ユーザー要求
  - 利用者は図書館の職員と一般の利用者で、職員は１～５のすべてを行えるが、一般の利用者は４の機能のうち、自分で借りている本のリストの作成しか行えない。
  - 本の貸し出し記録は長期間保存したい
- 制約条件
  - 図書館のすべての本は、貸出可能かすでに貸し出されているかのどちらかである。
  - ある本が貸出可能であり、かつ貸し出されているということはない。
  - 利用者はある決まった冊数以上の本を借り出すことはできない。

1.1UseCase

- **図書館貸出管理の**UseCase**を作成せよ**
- **利用者管理の**UseCase**を作成せよ**
- **蔵書管理の**UseCase**を作成せよ**
- **貸出の**UseCase**を作成せよ**
- **貸出状況の**UseCase**を作成せよ**

1.2**順序図**

- **蔵書検索の順序図を作成せよ**
- **貸出の順序図を作成せよ**
- **返却の順序図を作成せよ**

1.3インスタンス図

- 題名検索のインスタンス図を作成せよ
- 貸出記録のインスタンス図を作成せよ

1.4クラス図

- 全体のクラス図を作成せよ
  - 制約を書き込め
  - デザインパターンを利用して改良せよ

1.5状態遷移図

- 「利用者」の状態遷移図を作成せよ
- 「本」の状態遷移図を作成せよ
- 「本実体」の状態遷移図を作成せよ
- 「貸出記録」の状態遷移図を作成せよ

2.巡航制御システム

- 巡航制御システムは、運転者の指示にしたがって、速度を一定に保つ。運転者は、「巡航開始」「巡航停止」「加速開始」「加速停止」「巡航再開」を指示できる。
- 巡航制御システムは、エンジンが稼働していて、トップギアに入っていれば稼働可能である。巡航制御システムの運転者が「巡航開始」すると、システムは現在のスピードを維持する。「巡航停止」を行うと、速度の維持は運転者にまかせられる。「加速」すると加速後の速度（「加速停止」のときの速度）が新たな巡航速度になる。運転者がブレーキを踏んだり、ギアをトップ以外にしたときは、巡航制御システムは中断する。中断したあとに、「巡航再開」を指示すると元の巡航速度に戻る。
- システムが走行距離を正確に把握するように、運転者は「距離計測開始」「距離計測停止」を指示できる。ただし、この指示が有効なのは、巡航停止の状態の時のみである。
- 運転者は「平均速度計測開始」を指示したあと、いつでも「平均速度算出」を指示することによって、「平均速度計測開始」時からの平均速度を、表示パネルに表示することができる。

2.巡航制御システム 速度の求め方

- 車の速度計は不正確であるとする
  - そのため、システムは独自に速度を計測しなければならない
  - そこで、前述の距離計測機能を使って、シャフトの回転率と距離の比率を計測する
    - 1km走行したときどれだけシャフトが回転していたかが分かるので、この速度変換係数（回転数/km）を保存して、速度計算に使用する
- 速度の設定
  - スロットル位置を制御することによって速度を設定する
    - スロットル : (0..9)とする。９がフルスロットル、０は燃料をカットする

実践編6-9

2.1UseCase・シナリオ

- 上位レベルUseCaseを作成せよ
- 主要なシナリオを作成せよ
  - 現在速度を求める
    - km/hで求めたい
  - 速度維持
    - 急激な速度変化を避けたい、速度の誤差は2%程度にしたい
  - 加速
    - 急激な加速は避けたい

実践編6-10

2.2状態遷移図

- 最上位レベルの状態遷移図を作成せよ
  - 問題文に記述されている大きな事象を書き、詳細は子の状態遷移図に書くつもりで作成する

2.3順序図

- 巡航開始の順序図を作成せよ
- 距離計測の順序図を作成せよ
- 平均速度算出の順序図を作成せよ

2.4オブジェクト図

- オブジェクトの候補を描き、その間のリンクを書いて見よ

2.5クラス図

- クラス図のたたき台を作成せよ

2.6状態遷移図

- コントローラークラスの状態遷移図を作成せよ
  - コントローラークラスは、巡航制御に責任を持つ
- 速度センサークラスの状態遷移図を作成せよ
  - 速度センサークラスは、速度を提供する

2.7デザインパターン

- デザインパターンを考慮してクラス図を改良せよ
- 解答例を批評せよ

Ⅴ. 今後の展望と導入の課題

今後の展望

- 将来のある有力な技術は何か？
  - 形式技術
    - 生命に関わるシステム向き(Safety Critical)
    - 仕様記述言語と開発方法論
    - 関数型言語
      - 高信頼性、並行処理
  - オブジェクト指向技術
    - 生命に関わらないシステム向き(Mission Critical)
    - 仕様記述言語には形式技術導入
    - オブジェクト指向言語、分散処理（CORBA）、OODB
  - いずれにせよデザインパターンが必要

導入の課題

- デザインパターンの前にやっておくべきこと
  - ソフトウェア科学･ソフトウェア工学
    - アルゴリズムやピープルウェアは進化しているが、開発現場で使われていない
    - すでに解かれた問題を解くな
  - 構造化･抽象データ型
    - オブジェクト指向は、構造化の発展形
    - デザインパターンは20年以上の技術の集積
- 体制の変革
  - 形式技術・オブジェクト指向技術共に、現在の開発体制では通用しない。ソフトウェアの作り方は全く変わった！
  - 信頼性の高いソフトウェアが要求される
    - 管理者の意識変革こそ最も重要

導入の課題　続き

- デザインパターン作成･管理チームの必要性
  - デザインパターン、分析パターン、アーキテクチャパターン、イディオム、プロセスパターンの収集･作成･改良
  - ドメイン・フレームワーク、ドメインモデルの構築
    - アプリケーションの開発に先立ち、ドメイン向けのフレームワーク・ドメインモデルを収集･作成･改良する
- 最善・万能のモデルは存在しない
  - 間違ったモデルは存在する
  - モデルの良さ悪さは、常に相対的

V 1. ビジネスオブジェクト

- ドメインモデル
  - ビジネスオブジェクト＋ビジネスルール＝ドメインモデル
  - 誰がドメインモデルを作るか？
    - （ドメイン、モデル、プログラミング）のエキスパートの協力
  - デザインパターン・分析パターンの助けがあれば、モデルの品質が向上する
    - しかし、モデルそのものより、モデルを評価し作成できる「メタ知識」の方が大事
- CORBA
  - 分散オブジェクトと互換性の維持
    - ただし、現在は製品の互換性にまだ問題がある

V 2. 形式仕様とデザインパターン

- デザインパターンだけでは、問題は解決しない
  - 曖昧すぎる場合が多い
  - コーディング例では冗長すぎる
- 形式仕様記述言語と組み合わせたときに威力
  - 宣言的・手続き的両方の記述が可能
    - 厳密なデザインパターンの構築が可能
    - デザインパターンの蓄積が多い
  - 高階関数・パターンマッチ･再帰呼び出しなどで簡潔に記述可能
  - RSL,VDM-SL, B-Methodなど汎用の本格的形式仕様記述言語がある
    - 構文チェック、検証支援ツールなどと連動
  - OCLのようなオブジェクト指向用制約記述言語もある

参考文献

- オブジェクト指向
  - Bertrand Meyer. 酒匂寛, 酒匂順子 訳. Object-Oriented Software Construction オブジェクト指向入門. アスキー, 1990
    - ＯＯの必要性をプログラミング技術面から分かりやすく解説
  - Jacobson, Ivar et al. Object-Oriented Software Engineering - A Use Case Driven Approach. Addison-Wesley, 1992.
    - UseCaseの教科書
  - 青木淳. オブジェクト指向システム分析設計入門. ソフト・リサーチ・センター, 1992
    - OOA/OOD/OOPの分かりやすい解説
  - 青木 淳。例題による！！ オブジェクト指向分析設計テクニック。ソフト・リサーチ・センター, 1994
    - OOD/OOPの分かりやすい解説
  - 佐原伸。オブジェクト指向システム分析／設計　Ｑ＆Ａ。ソフト・リサーチ・センター, １９９５年１１月

参考文献

- デザインパターン
  - C.アレグザンダー、パタン・ランゲージ、鹿島出版会、1992年
    Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1994.
    - デザインパターン教科書の定番
  - Sherman R.Alpert, Kyle Brown, Bobby Woolf、The Design Patterns Smalltalk Companion, Addison Wesley, 1998
    - Smalltalkに最適化したデザインパターン
  - 佐原伸。デザインパターン オブジェクト指向システム分析／設計技法。ソフト・リサーチ・センター, 1999年5月

参考文献

- 分析パターン
  - Martin Fowler, "Analysis Patterns: Reusable Object Models"Addison-Wesley, 1997
    David Hay, "Data Model Patterns: Convention of Thought", DorsetHouse, 1996
- イディオム
  - 青木淳、Smalltalkイディオム、SRC、1997年
  - Kent Beck, "Smalltalk Best Practice Patterns", Prentice Hall, 1997
- プロセスパターン
  - Tom DeMarco, Timothy Lister、ピープルウェア、日経BP社、1989
  - J.Rumbaugh, M. Blaha , W.Premerlani, F.Eddy, and W.Lorensen. 羽生田訳. オブジェクト指向方法論：ＯＭＴ. トッパン, 1992
    - 現時点で最も完成されたOOA/OOD技法OMTの教科書
  - J.Rumbaugh, M. Blaha , W.Premerlani, F.Eddy, and W.Lorensen。Solution Manual Object-Oriented Modeling and Design。Prentice Hall, 1991
    - 上の本の解答集

参考文献

- 仕様記述
  - OMG Unified Modeling Language Specification (draft) . Object Management Group, Inc.他, Version 1.3 alpha R5, March 1999
    - UMLの仕様書。制約仕様記述言語OCLの仕様を含んでいる。
  - The RAISE Language Group著. The RAISE Specification Language. Prentice-Hall, 1992
    - RAISE/RSLの仕様書
  - John Fitzgerald, Peter Gorm Larsen, Dines Bjゥrner, Cliff Jones. Modelling Systems: Practical Tools and Techniques in Software Development, Cambridge University Press, 1998
    - VDM Tool簡易版を使った形式手法のCD-ROM付き入門書。
  - 中川 中著. 代数的仕様記述言語 CafeOBJ. SRA, 1993
    - Cafe OBJの解説
  - Jean-Raymond Abrial. The B-Book : Assigning Programs to Meanings. Cambridge University Press, 1996
    - B Methodの教科書

参考文献

- 記法
  - OMG Unified Modeling Language Specification (draft) . Object Management Group, Inc.他, Version 1.3 alpha R5, March 1999
- パターン・ホームページ(http://hillside.net/patterns/)
  - パターンに関連するすべての情報があるWebページ
- 形式手法ホームページ(http://hello.to/fm/)
  - 仕様記述を含む、形式手法すべてについての情報があるWebページ