デザインパターンによる
オブジェクト指向分析/設計入門

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チーム
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

- オブジェク ト指向分析・設計をより深く理解する
- デザ インパターンを応用するための基礎を身につける

前提条件

- オブジェクト指向による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

目次

Ⅰ. オブジェクト指向システム開発の現状と問題点

ソフトウェア開発の現状と問題点
オブジェクト指向開発の登場

ソフトウェア開発の現状と問題点

ソフトウェア危機
プログラミングは難しい

ソフトウェア危機

- ハードウェア・ソフトウェアの進化により、ユーザーの要求が高度化しシステムが複雑になってきている
  - マルティメディア＋ネットワーク＋リアルタイム
- 従来の手法では、開発要員が多すぎて開発コストと開発期間が増大している
  - １０００人以上・１０００万ステップ以上
- 開発規模の拡大にともなってソフトウェアの品質低下が著しい
  - 中華航空機・もんじゅ・第３次オンライン・Y2K
- 保守のコストが増大している
  - 大手ユーザーでは８０%以上のリソースを保守に投入

プログラミングは難しい

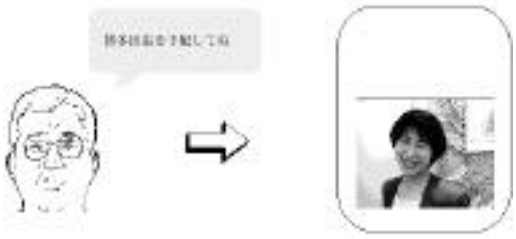
- 人間の思考に合わない
  - 人間に仕事を頼むときと異なる頼み方をする
- 言語のレベルがまだ低い
  - 特殊な言語（プログラミング）を使ってコンピュータに指示する
- 原理的に難しい
  - そもそも「プログラミング」は難しい
  - 数学基礎論・コンピュータ科学などで「難しさ」は証明されている
    - プログラムが正しいことは「実用的な大きさのプログラムでは証明できない」
- プログラムでは解けない問題がある

オブジェクト指向技術の登場

- 構造化技法の一種
  - 保守性と再利用性の向上を目指している
  - 人間のコミュニケーション方式を模倣
  - 従来の「手続型」指向は自然でない
  - フォン・ノイマン型計算モデルは、偶然の産物
    - オブジェクト指向
    - 関数指向
    - 制約指向...などいろいろな考え方がある
- オブジェクト指向の提案
  - コンポーネント（＝オブジェクト）
  - 抽象化
  - フォン・ノイマン型思考からの脱皮

プログラムの主人公が異なる

- 従来
  - フォン・ノイマン型コンピュータのように考える
    - 順序を追って操作を考える
      - 手続的思考
- オブジェクト指向
  - 人間の自然な概念に従って考える
    - 「誰に」「何を」頼むかを考える



オブジェクト指向の効果

- 品質の向上が期待できる
  - オブジェクト指向技術本来の狙い
- 情報表現力が向上する
  - すべてがオブジェクト
    - 例えば、テキストと図形とプログラムとを区別する必要がない
- 結果として
  - 保守が容易になる
  - 開発費用が削減できる
  - 開発期間の短縮が可能になる

オブジェクト指向の課題

- 教育の不足・難しさ
  - ソフトウェア工学３０年の遅れを一挙に取り戻さなければならないため
    - 抽象データ型・構造化などの概念も覚える必要がある
- 技術を理解した管理職の不足
  - ノモンハン症候群
- プログラマの不足
- 原理的不安定さ
  - 所詮、手続き型プログラミングであり、参照の透明性を保証できない

II. よい設計とは？

外的品質要因
モジュール性の原則
再利用へのアプローチ

外的品質要因

- 正確さ
  - 要求された通りに仕事を行う能力
- 頑丈さ
  - 異常な状態においても機能する能力
- 拡張性
  - 仕様の変更に容易に適応できる能力
- 再利用性
  - 新しい応用にどの程度再利用できるかを示すもの
- 互換性
  - ソフトウェア相互の組み合わせやすさ

モジュール性の原則

- 言語としてのモジュール単位
  - ほとんどのオブジェクト指向言語は合格
- 少ないインタフェース
- 小さいインタフェース
- 明示的なインタフェース
- 情報隠蔽
- 開放／閉鎖の両立
  - 以下を同時に満たさなければならない
    - モジュールが拡張可能である（開放）
    - モジュールが他のモジュールから使用できる（閉鎖）

再利用可能なモジュール構造の要件

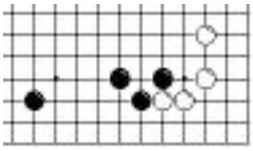
- 型の変化に対応できる
- データ構造とアルゴリズムの変化に対応できる
- 関連した操作がまとまって定義されている
- 顧客モジュールが実現方法を知ることなく操作を要求することができる
  - 動的束縛で解決できる
- 実現方法の間の共通部分がうまくまとめられている
  - サブグループ間の共通性をうまく記述できる
    - 表検索の場合では、順次形式の表がひとつのサブグループになる
      - 順次配列、連結リスト、順次ファイルの共通点をうまくまとめられるか？

そしてデザインパターンへ

- 建築家C.Alexanderの著書からの影響
  - 253パターンを使って設計
    - 日本の下町はこのパターンにかなり適合
    - 見分けやすい近隣(14)、買い物街路(32)、連続住宅(38）、小さな広場(61)、屋台(93)、玄関先のベンチ(242)
- 日本でも、東野高校、名古屋の団地設計案など

門編1-17

II. デザインパターンとは？



- 定石･手筋・大局観
  - 囲碁の定石は約2万個
    - アマチュア初段で大体マスターしている
- 公理･定理・公式
  - 数学公式の主なもので約2000個
- 抽象データ型･アルゴリズム
  - 基本的なものだけで約1000のアルゴリズム
  - 形式仕様手法RAISEの検証公式で約2000個
- フレームワーク・再利用可能クラス群
  - Smalltalkクラス･ライブラリー、MacApp、ET++など

囲碁の強豪になる法

- 最初にルールと用語を学ぶ
  - 例：石の名前、打ち方、勝敗の決め方、碁盤の構成など
- 定石や手筋を学ぶ
  - 例：手順、石の価値、大局観など
- 強い人の棋譜を仲間で勉強する
  - 定石・手筋・大局観を理解し、覚え、繰り返し適用してみる
  - 1万局ほど打ってアマチュア初段になるのが標準的
    - アマチュア五段＝プロ10級
- 定石や手筋を作り出し、実戦で評価し、独自の大局観を磨く

優秀なSoftware Engineerになる法

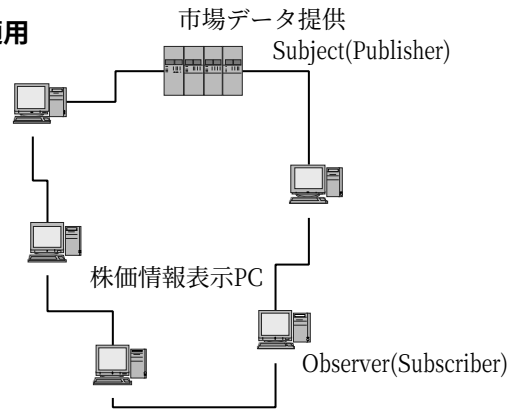
- 最初にルールと用語を学ぶ
  - 例：系統的プログラミング、仕様記述言語、プログラミング言語など
- デザインパターンを学ぶ
  - 例：アルゴリズム、ソフトウェア工学方法論、離散数学など
- 優秀なSEの設計(結果と過程)を、仲間で勉強する
  - デザインパターンを理解し、覚え、繰り返し適用してみる
- デザインパターンを作り出し、実践して評価する
- デザインパターンを発表し、世間の批評を受ける

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

III. デザインパターンの例

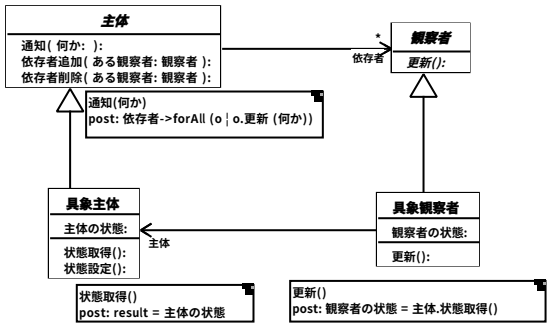
- 株価情報サービス
  - Observerパターンを適用



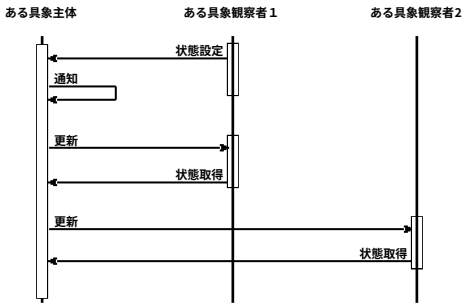
Observerパターン適用

- 概要
  - 主体(Subject)側の株価が変わったとき、それを見ている観察者(Observer)に通知し、自動的に最新状態に更新する
- 結果
  - 観察者(Observer)が多くなっても問題が少ない
  - それぞれの観察者(Observer)は、異なるGUI・アーキテクチャであっても問題がない
  - 主体(Subject)は可能な限り観察者(Observer)から分離していて、観察者(Observer)は主体(Subject)と独立に変更できる

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

Ⅴ. デザインパターンの分類

- デザインパターンの種類
  - 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
  - プロセスパターン
  - 算法パターン
  - イディオム
  - ...

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。

デザインパターンの種類(続き)

- 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。
- 算法パターン
  - ある問題の解法を、データ構造と算法(アルゴリズム)として提供する。
    - すでに解かれた問題を解くな!
- イディオム
  - プログラミング言語に依存した詳細レベル抽象パターン。
    - 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。

デザインパターン教科書

- 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
    - それまでは、Alexanderの「パターン」の利用はあまり知られていなかった
  - 23パターンのカタログ
    - 再利用可能なOOソフトウェアに共通するパターンを説明
  - 設計とコミュニケーションを改良する効果がある

IV.デザインパターンの分類

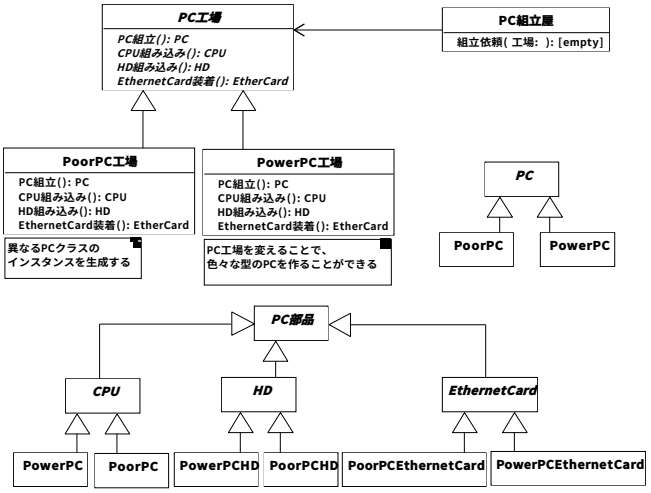
- 生成パターン
  - クラスとオブジェクトの初期化と設定を行う
- 構造パターン
  - インターフェースと、「クラスとオブジェクト」の実装とを切り離す
- 振る舞いパターン
  - クラスとオブジェクトの動的なやりとりを扱う

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

Abstract Factory（抽象工場）の比喩

- PC組立屋



抽象工場（Abstract Factory、別名Kit）

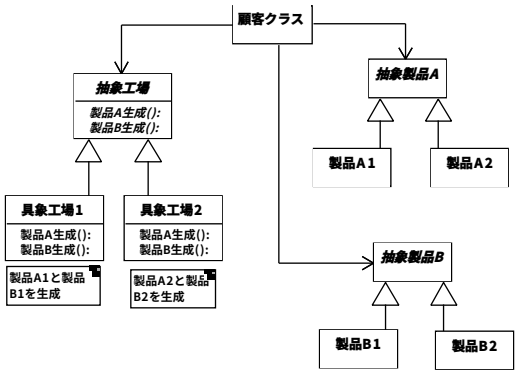
- 目的
  - 互いに関連するプロダクトとしてのオブジェクト群を、その具象クラスを明確にしないまま生成するインターフェースを提供することにより、顧客クラスが具象クラスを指定せずプロダクトを生成できるようにする
    - プロダクトを個々の部品から一歩一歩組み立てる必要のあるとき
    - 同じ部品ファミリーから１個のプロダクトに組み立てるとき
    - 部品の生成・組合せ・実装方法を、システムの他の部分から独立にしたいとき

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

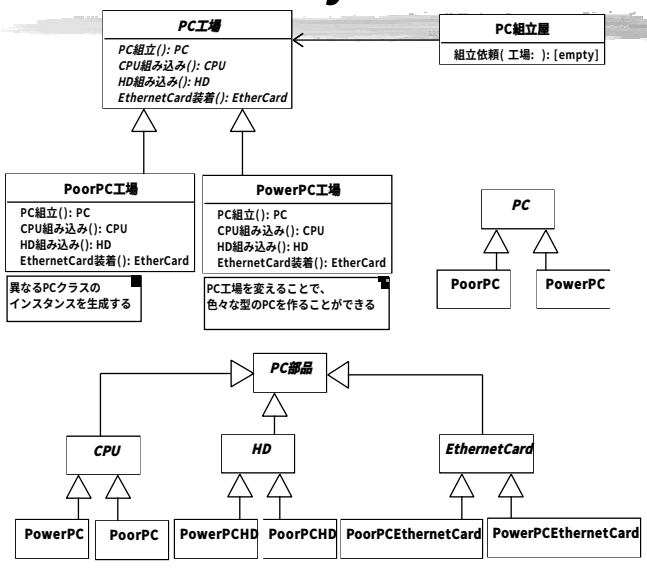
Abstract Factoryのクラス図



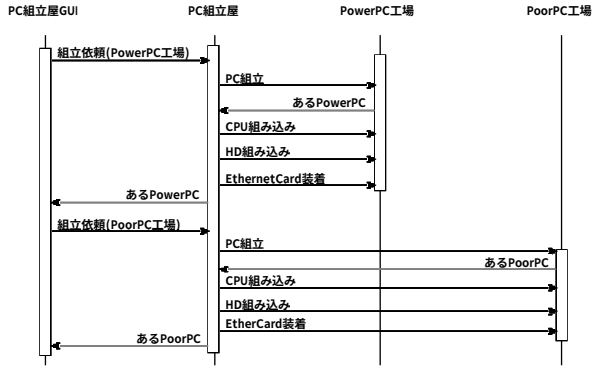
Abstract Factoryの配役

- 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

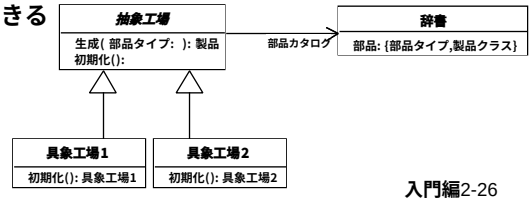
- 抽象工場の普通の実装方法
  - CPUを追加するとき修正を1カ所にできる
    - **context** PC工場::CPU組み込み操作
      - 抽象操作 -- サブクラスで定義
    - **context** PowerPC工場::CPU組み込み操作
      - CPUのインスタンスを返す
      - **post**: return = PowerPC.new
    - **context** PoorPC工場::CPU組み込み操作
      - **post**: return = PoorPC.new

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
      - **post**: return = self.cpuClass.new
    - **context** PowerPC工場::cpuClass
      - **post**: return = PowerPC -- CPUのクラスを返す
    - **context** PoorPC工場::cpuClass
      - **post**: return = PoorPC

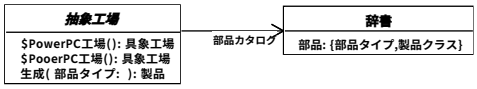
Abstract Factoryの実装方法

- 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

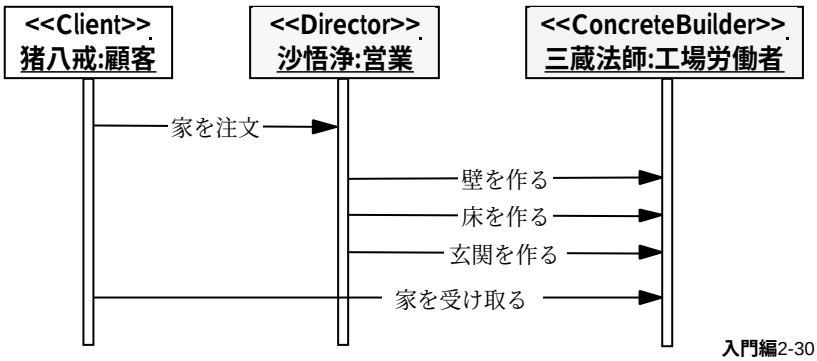
- InterViews
  - Kit
- ET++
  - WindowSystem
- VisualWorks
  - UILookPolicy

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

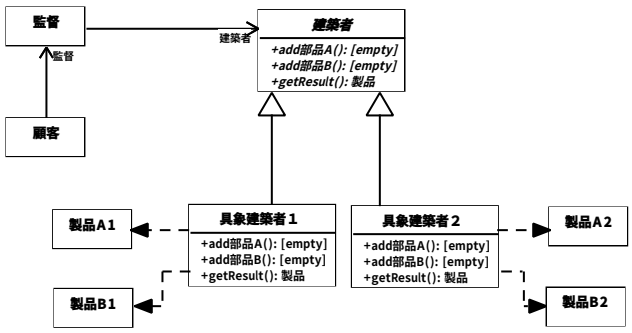
- 複合オブジェクトの作成過程と表現形式を分離することにより、同じ作成過程で異なる表現形式のオブジェクトを生成する
  - ツーバイフォー方式



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

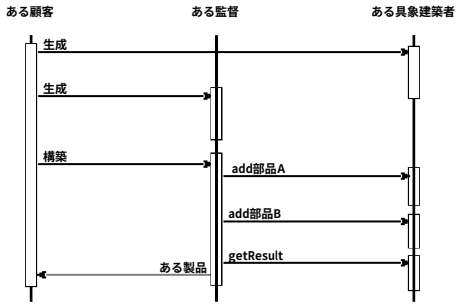
Builderのクラス図



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



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

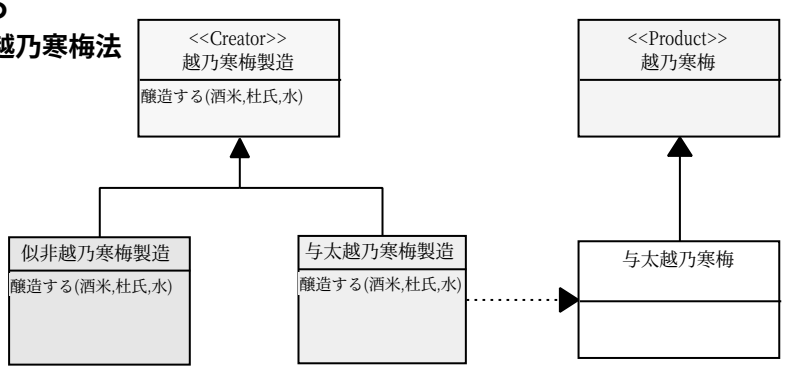
- FrameMakerやクラリスインパクトといったドキュメント作成プログラムは、各種のワープロやファイル形式を読み込み、逆に、色々なファイル形式で書き出すのにBuilderパターンを使っている
- フリーのSmalltalkであるSqueakコンパイラーのParseクラスは、<<建築者>> ParseNodeクラスを使う<<監督>>である
  - Parserは文法上のトークンを認識するたびに、ParseNodeオブジェクトにメッセージを送る
  - 構文解析が終了したら、ParserはParseNodeオブジェクトに、それまでに作成した構文解析木MethodNodeを要求し、<<顧客>>で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

Factory Method(工場操作）の比喩

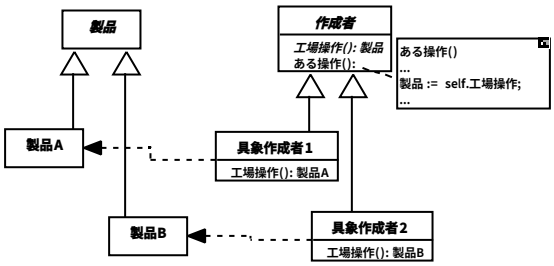
- オブジェクトを生成するためのインターフェースだけを規定して、実際のインスタンス生成をサブクラスにまかせる
  - 越乃寒梅法



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

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

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

PC組立のBuilderパターン例

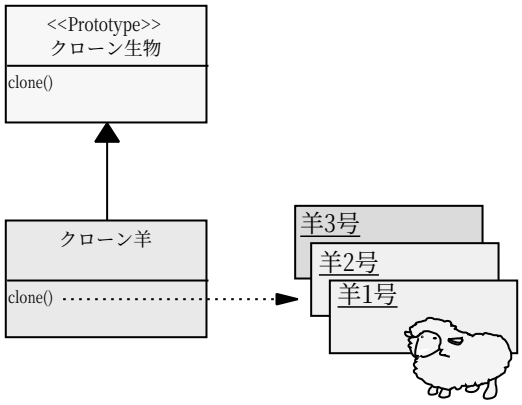
- 抽象工場(AbstractFactory)の例に出てきた、PC組立のモデルをBuilderパターンで作成して下さい
  - クラス図を書いてみて下さい
  - 順序図を書いてみて下さい

PC組立のFactory Methodパターン例

- 抽象工場(AbstractFactory)の例に出てきた、PC組立のモデルをFactory Methodパターンで作成して下さい
  - クラス図を書いてみて下さい

Prototype(模型)の比喩

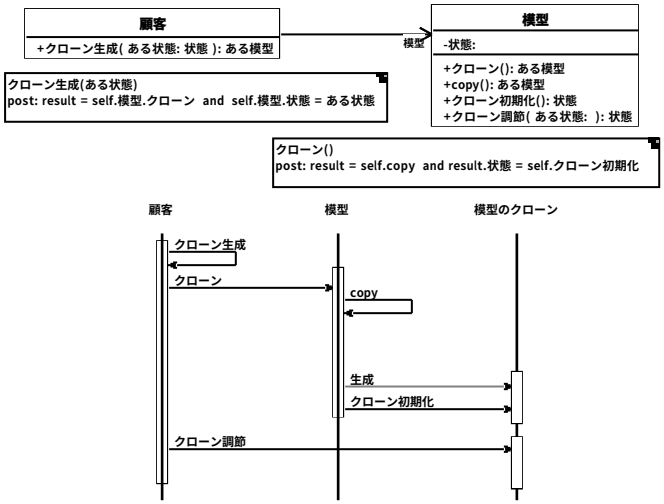
- 模型（Prototype）からクローンとして新しいインスタンスを作る
  - クローン羊方式



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



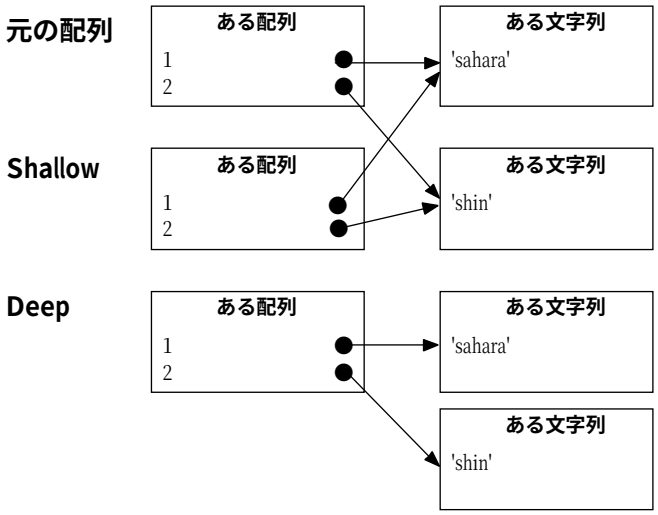
Prototypeの配役

- 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

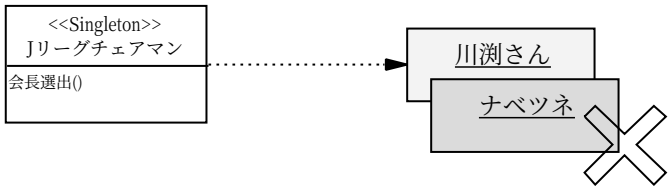
- ThingLab(制約指向図形エディター)
  - ユーザー定義のThingに対応したクラスを生成し、再利用するときにクローンとして取り出す
- VisualWorksのRDBフレームワーク
  - 空のオブジェクトのクローンに1行分のデータ
- 楽譜編集エディタ
  - 汎用のGraphicToolに楽譜編集機能を動的に付与
- Smalltalkのクラス
  - Metaclassのクローン

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

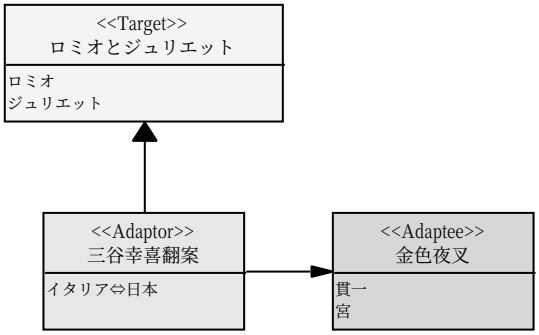
2.構造パターン

- 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(原子化)
  - 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

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

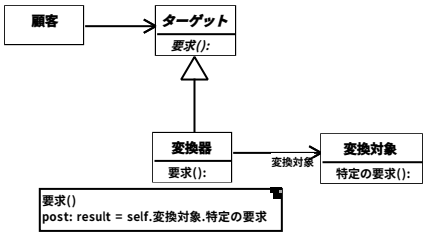
Adapter（翻案者：Wrapper）の比喩



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



Adapterの配役

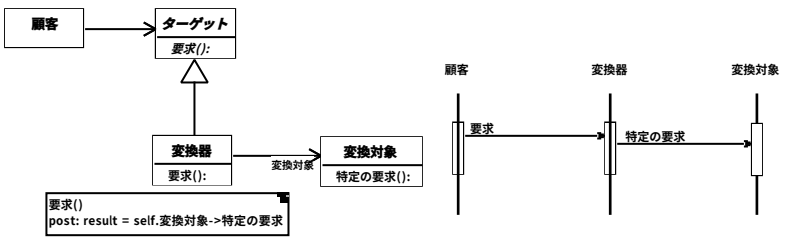
- Target(ターゲット)
  - <<顧客>>が利用するドメインに特化したサービスを定義する
- Client(顧客)
  - <<変換対象>>クラスと協調したいオブジェクト
- Adaptee(変換対象)
  - インターフェースを適合させる必要のある既存クラス
- Adapter(変換器)
  - <<変換対象>>クラスのインターフェースを、<<ターゲット>>クラスのインターフェースに適合させる

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

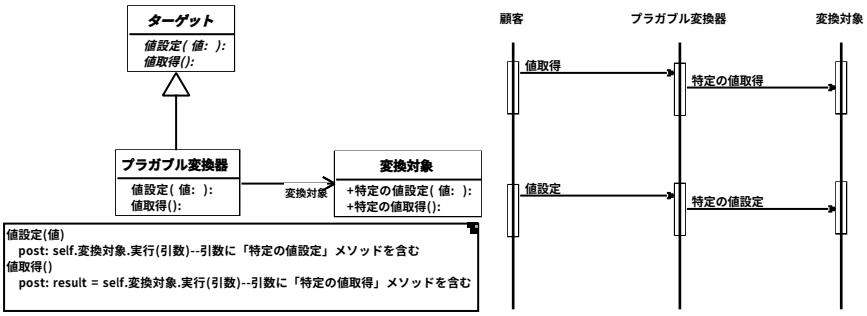
Adapterの実装

- 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

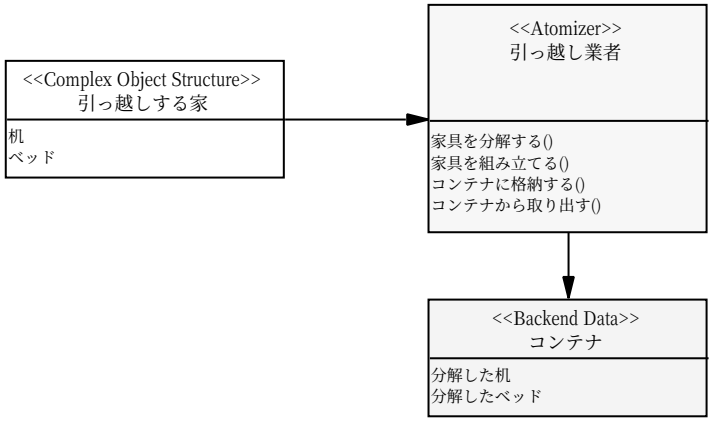


Adapter関連パターン

- 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(原子化)の比喩

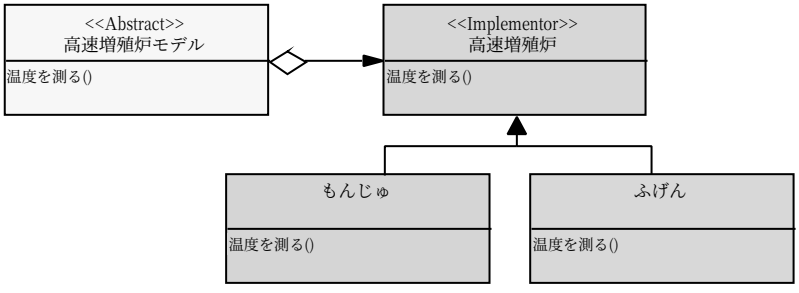
ζ 複雑なオブジェクト構造を、バックエンドの構造データ（普通のファイル、RDB、RPCバッファーなど）として格納する



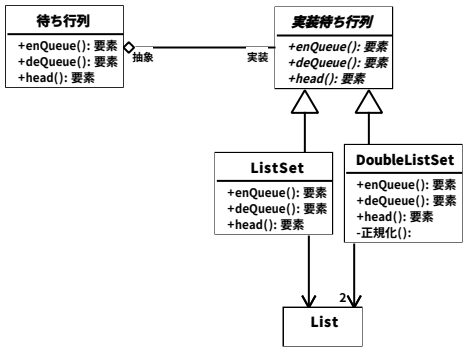
Bridge（橋：Handle/Body）の比喩

ζ 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

ψ動燃方式

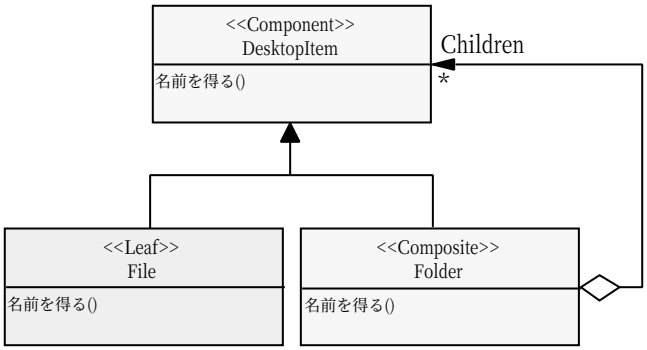


Bridgeの例



Composite（混成）の比喩

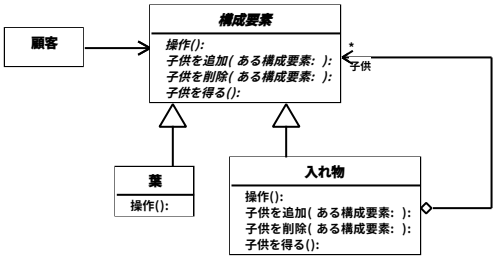
ζ パソコンのファイル・システム



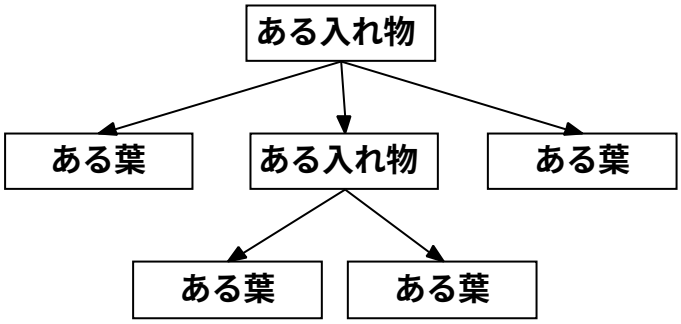
Composite（混成）

- 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

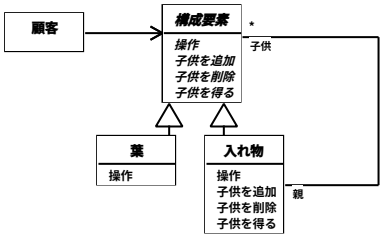
- 構成要素 Component
  - 構造内のオブジェクトのインターフェースとしての抽象操作を定義する
  - 構造内のすべてのクラスに共通な、標準インタフェースの操作を定義する
  - 子にあたる<<構成要素>>オブジェクトをアクセスするための操作を定義する
  - 親にあたる<<構成要素>>オブジェクトにアクセスするための操作を定義する
- 葉Leaf
  - 構造内のターミナル(葉)オブジェクトを表す
  - 子オブジェクトを持たない
  - 構造内のオブジェクトの基本操作を定義する
- 入れ物Composite
  - 子オブジェクトを持つ<<構成要素>>オブジェクトの振る舞いを定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

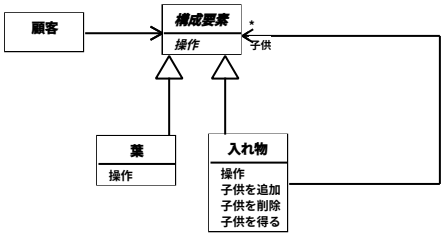
Compositeの実装

- 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

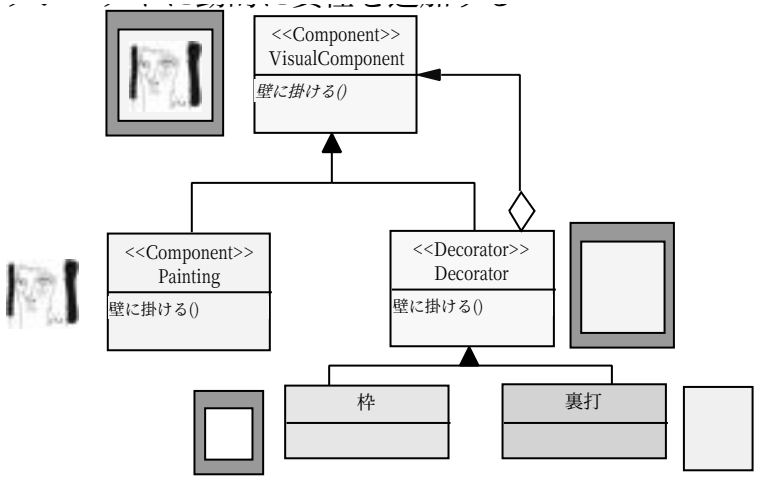
Compositeの使用例

- 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

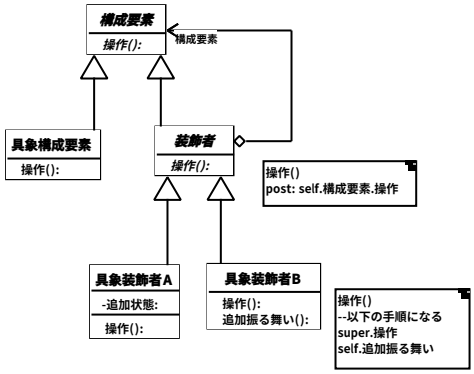
Decorator（装飾者：Wrapper）の比喩



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

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

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

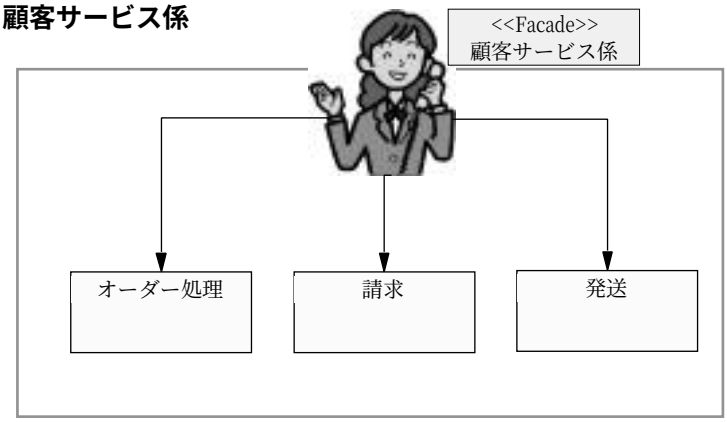
- 多くのGUIキットがWidgetに装飾を追加するためにDecoratorパターンを使用している
  - Smalltalk、InterViews、ET++など
- Streamにコード変換や圧縮アルゴリズムを適用するためにDecoratorパターンを使っている
  - Smalltalk、ET++など
    - Stream
      - EncodedStream
        - encoderを呼び出す

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

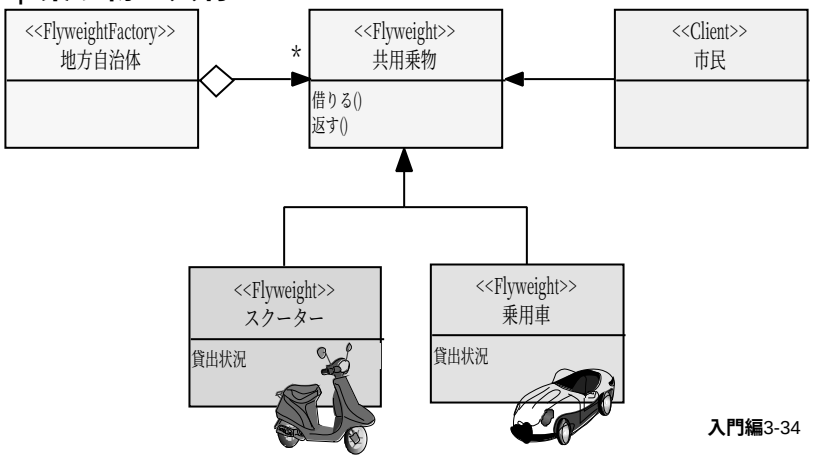
ζ サブシステムのインタフェースを単純化する
  ψ 顧客サービス係



入門編3-33

Flyweight（フライ級選手）の比喩

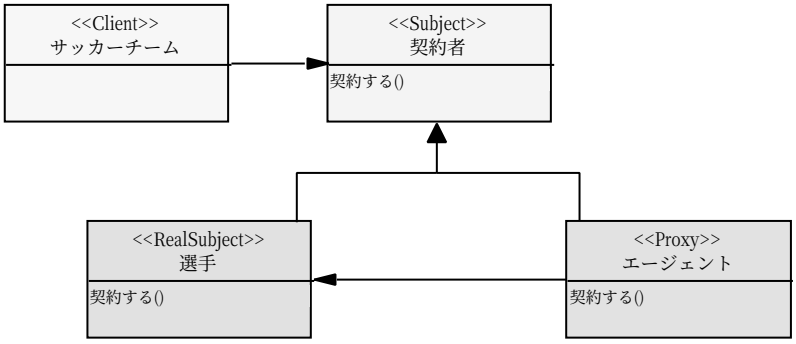
ζ 多数の小さなオブジェクトを共有し、空間効率を高める
  ψ 乗り物の共有



入門編3-34

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



入門編3-35

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

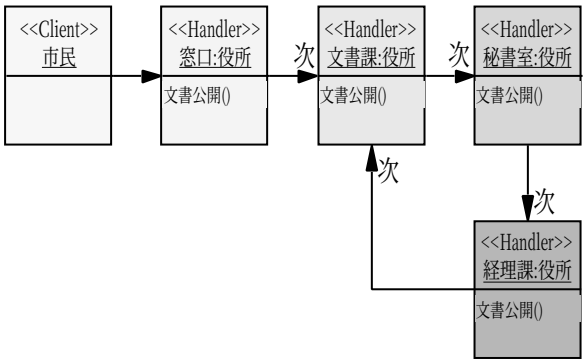
振る舞いパターン　続き

- 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

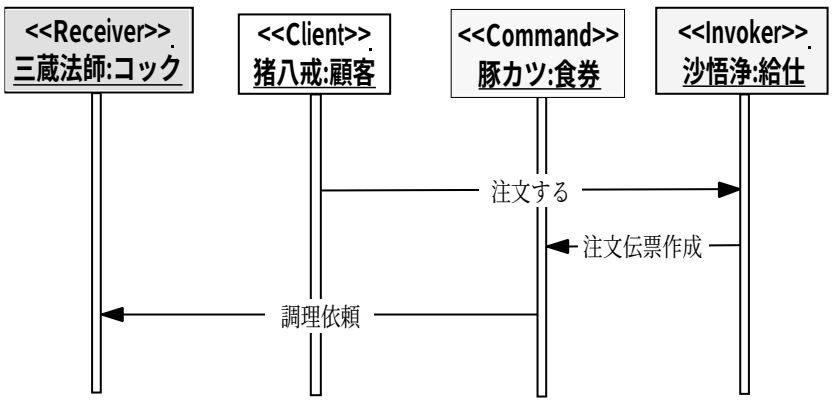
■責任のあるサービス提供者へ要求を「連鎖的に」委任する

◆たらい回し



Command（命令：Action, Transaction）の比喩

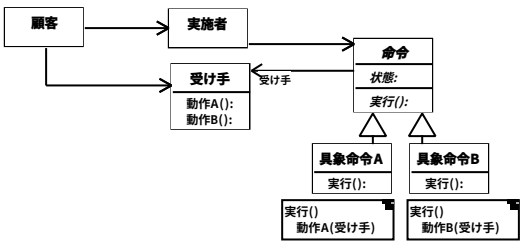
■パラメータ化した要求をオブジェクトとしてカプセル化する



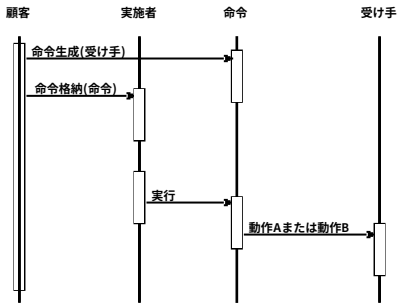
Command(命令)

- 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）

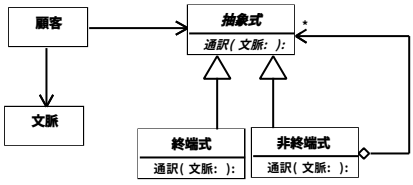
```
式 ::= NUM                      pi = 3.141592653589
     | VAR                      3.141592653589
     | VAR '=' 式               sin(pi)
     | FNCT '(' 式 ')'          0.0000000000
     | 式 '+' 式                a = b = 2.3
     | 式 '-' 式                2.3000000000
     | 式 '*' 式                ln(a)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

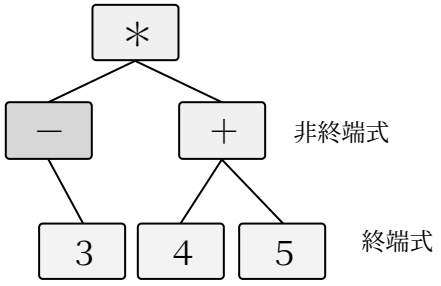
Interpreter（通訳）

- 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

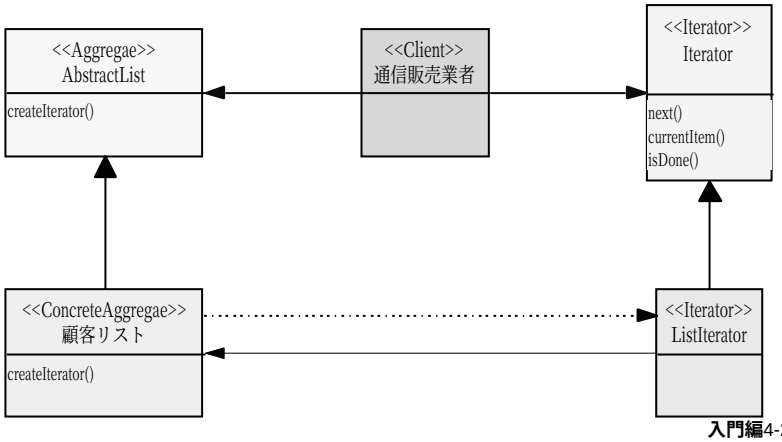
- ほとんどのOO言語のコンパイラー
- 野村証券第2次オンライン汎用大型機データ交換
- Smalltalkによるビジネスルール・エンジン
  - 保険商品仕様記述言語
- Dynamic Query Language
  - SQLに変換

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
  - 抽象構文木内を走査するときに使える
- 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

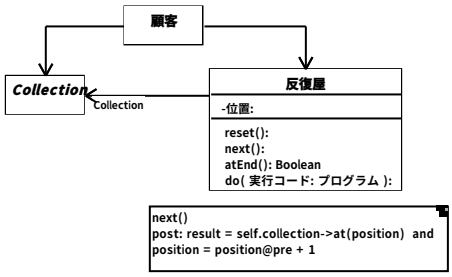
■オブジェクトの数を指定せず、順番に処理する



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

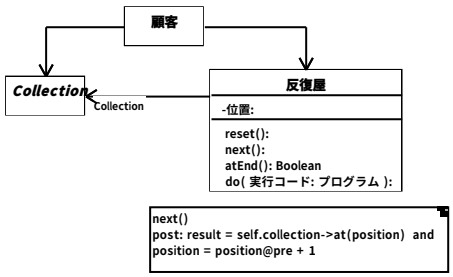
- 反復屋 Iterator
  - 要素にアクセスしたり走査するためのインターフェースを定義する
- Collection
  - <<反復屋>>を生成するためのインターフェースを定義する

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

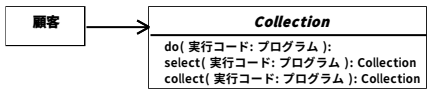
Iteratorの実装

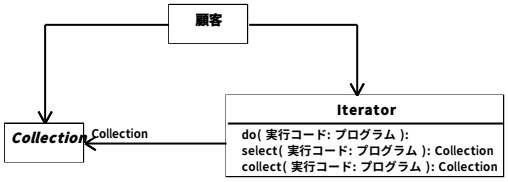
- 外部Iterator
  - <<顧客>>が制御する
    - <<顧客>>は<<反復屋>>に次の要素を明示的に要求せねばならない



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

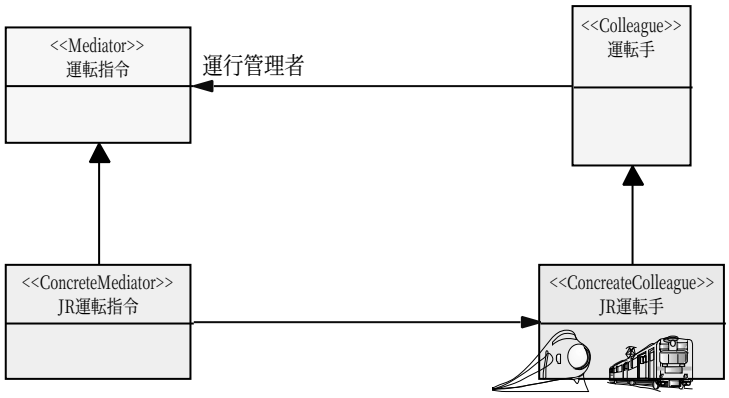
- SmalltalkのCollectionクラス群
- ET++のコンテナクラス
- Object Windowのコンテナクラス

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

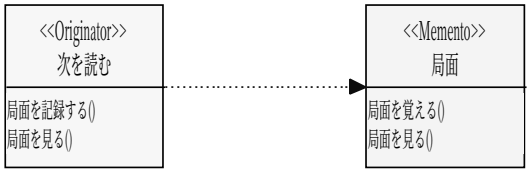
Mediator（調停者）の比喩

■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



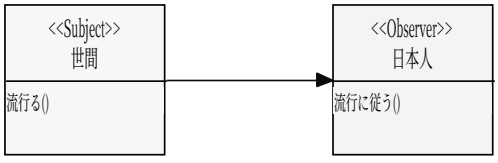
Memento（形見：Token）の比喩

- 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

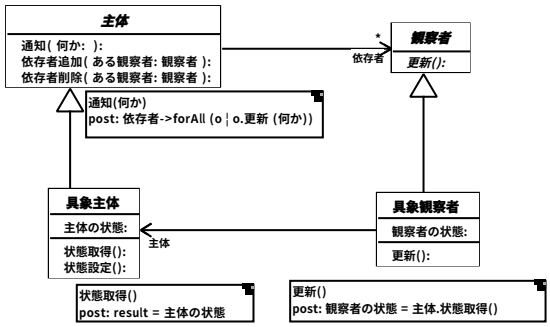
- オブジェクトが状態を変えたとき、それに依存したオブジェクトが自動的に更新される
  - 日本人方式



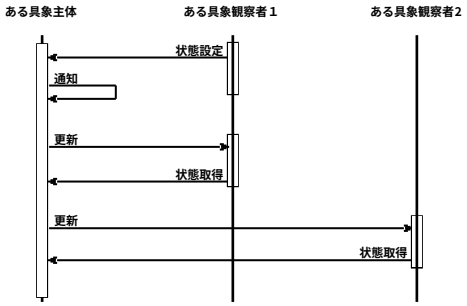
Observer（観察者）

- 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

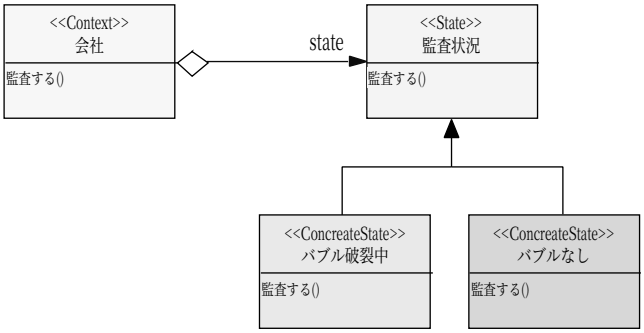
- MVCパターン
  - Modelが<<主体>>、Viewが<<観察者>>と対応する
- クライアント/サーバーパターン
  - サーバーが<<主体>>、クライアントが<<観察者>>と対応する
    - ただし、両者は異なるマシン上に実装されていることが多いので、エラー処理などでObserverパターンよりかなり複雑になる

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

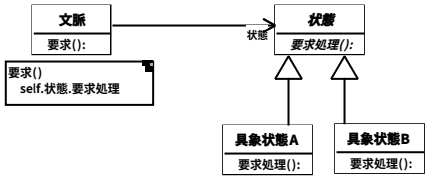
■オブジェクトの内部状態が変わったとき、振る舞いを変える



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



入門編4-47

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

入門編4-48

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

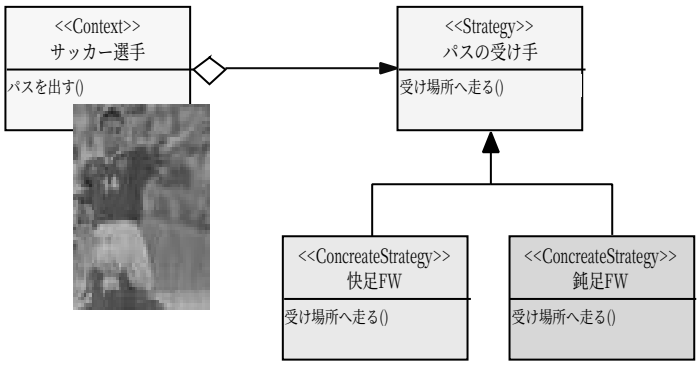
- HotDrawなどの描画ツールで、選択しているツールによってエディターの振る舞いを変えるために、このパターンを利用している
- SmalltalkのSMTPサーバー

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

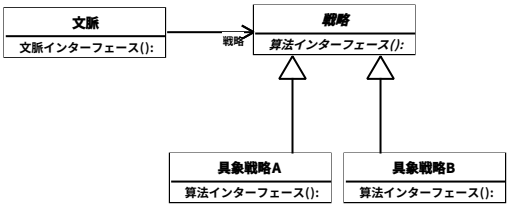
■アルゴリズム群の各々をカプセル化し、交換可能にする



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

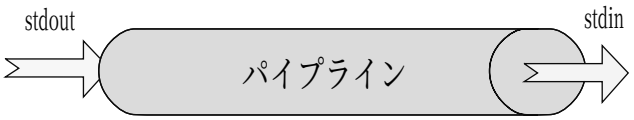
- 保険ポリシー
  - ビジネスロジックの戦略を切り替え
- 改行アルゴリズム
  - ET++、InterViewsで<<戦略>>を使って実装している
    - 例
      - 段落全体のバランスを考えた改行アルゴリズム
      - 各行が固定の項目数を持つような改行アルゴリズム
      - 各行に単純なテキストを入れるための改行アルゴリズム

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

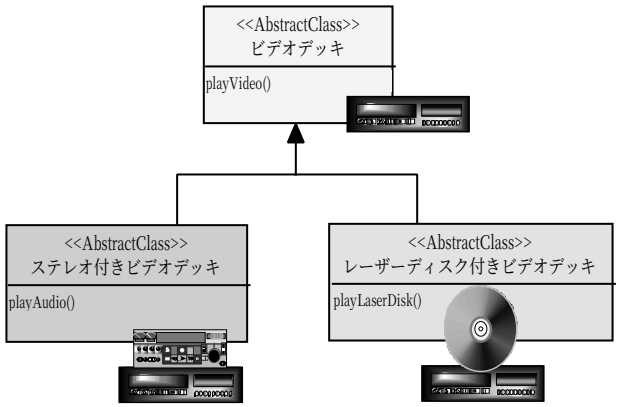
Stream（流れ）の比喩

■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



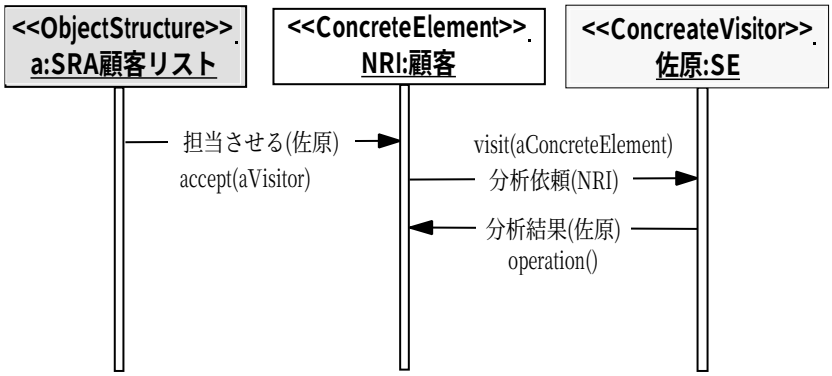
Template Method（型紙方式）の比喩

■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



Ⅴ.分析パターン

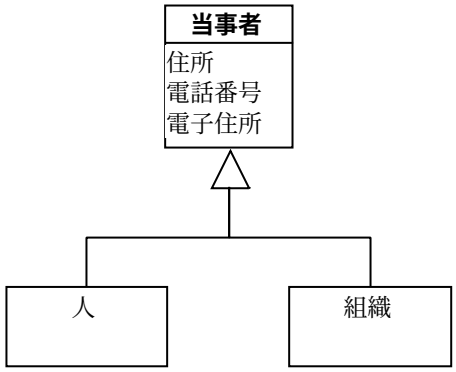
- 分析パターンの分類
  - 責任(Accountability)
    - 当事者(Party)、組織(Organization)
  - 観察(Observation)
    - 量(Quantity)、観察(Observation)
  - 計画(Planning)
    - 活動(Action)、計画(Plan)、資源割当(Resource Allocation)
  - ...

責任(Accountability)

- 責任パターンを構成するパターン
  - 当事者(Party)
    - 人と組織の役割の上での同一性を提供する
  - 組織構造(Organizational Structure)
    - 組織構造の変化にも影響を受けにくいモデルを提供する
  - 責任(Accountability)
    - 組織を責任から見た抽象モデルを提供する

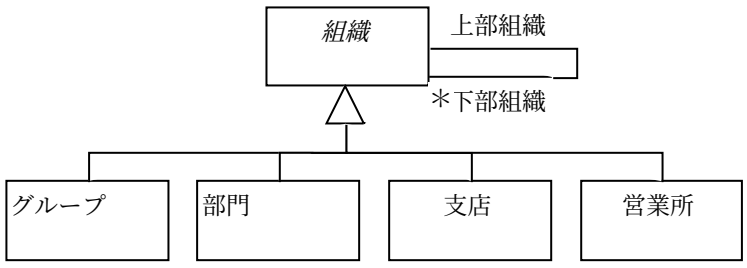
当事者(Party)

- 人と組織の共通点を抽出



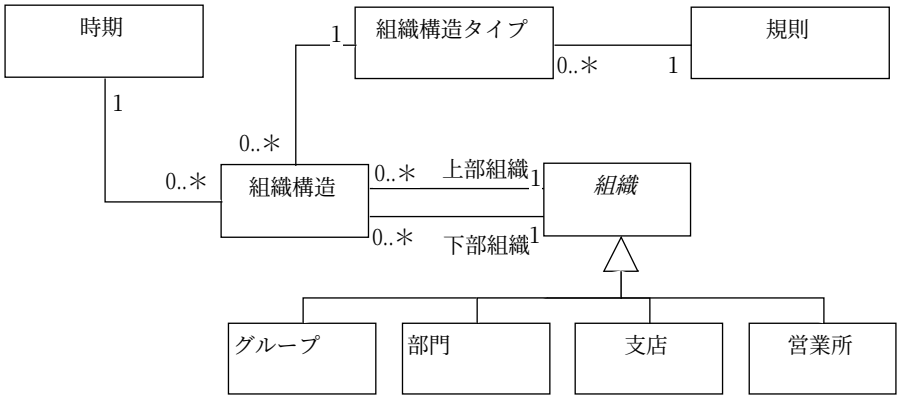
組織構造(Organizational Structure)

- 常識的な構造



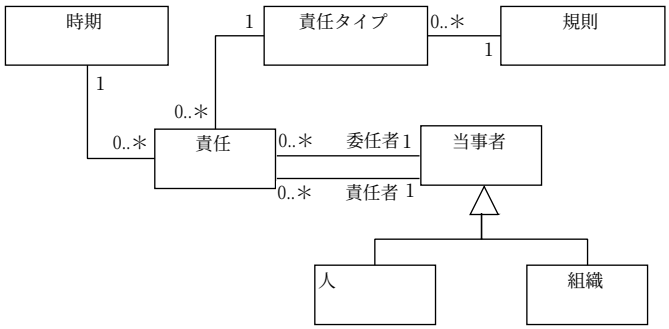
組織構造(Organizational Structure)

- 組織は変更されやすい



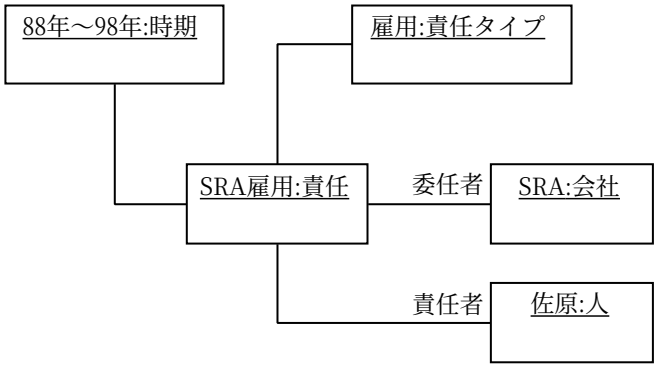
責任(Accountability)

- 組織の概念を一般化し、責任として捉えたモデルを提供する



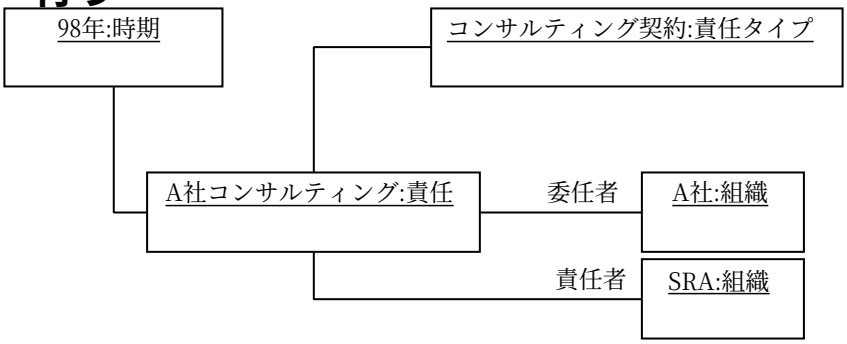
責任(Accountability)の例

- 佐原がSRA で働いている場合



責任(Accountability)の例

- SRA が98 年にA 社のコンサルティングを行う



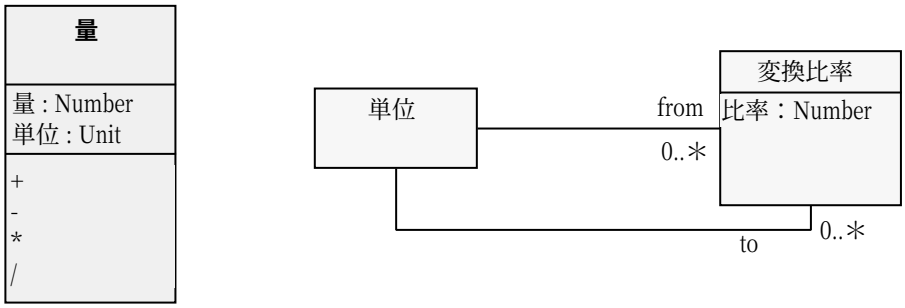
タイプ・オブジェクトの演習

- 責任パターンのところで「責任タイプ」「組織構造タイプ」といったクラスが出てきた。
  - こういったタイプ・オブジェクトはシステムの変更をプログラミング無しで行える利点がある
  - タイプ・オブジェクトを使わないとサブクラスを使いコーディングしなければならない
    - 例
      - 弁当屋が弁当の新メニューを作るたびに、サブクラスを追加するわけには行かない
        - 「弁当」と「弁当メニュー」
- タイプオブジェクトが必要な例を挙げよ

観察(Observation)

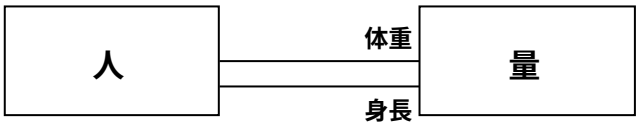
- 観察パターンを構成するパターン
  - 量(Quantity)
  - 測定(Mesurement)
  - 観察(Observation)

量(Quantity)と変換比率(Conversion Ratio)



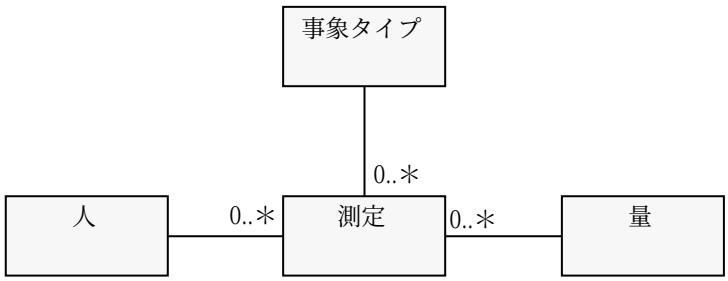
測定(Mesurement)

- 素直なモデル



測定(Mesurement)

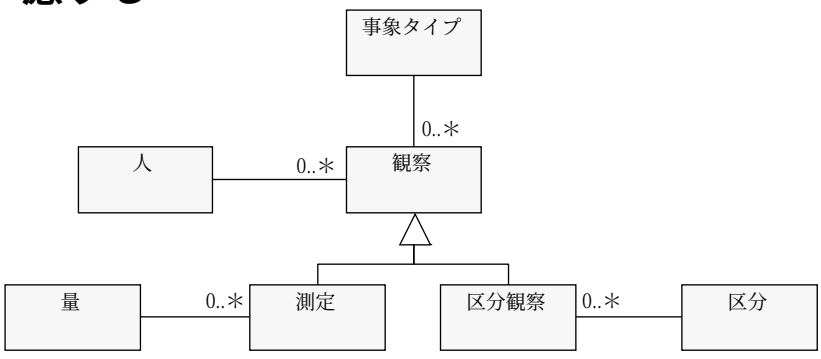
- より汎用的なモデル



入門編5-13

観察(Observation)

- 定量的測定だけでなく、定性的観察も考慮する

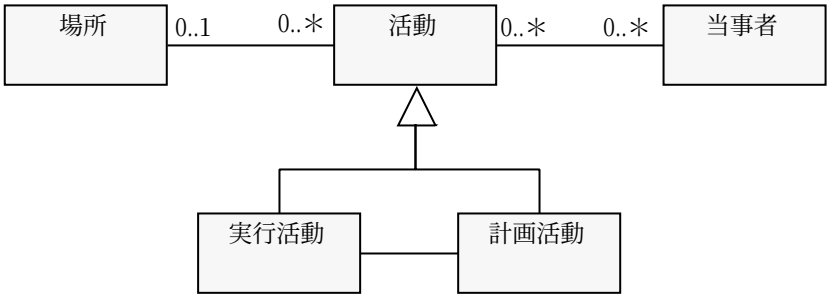


入門編5-14

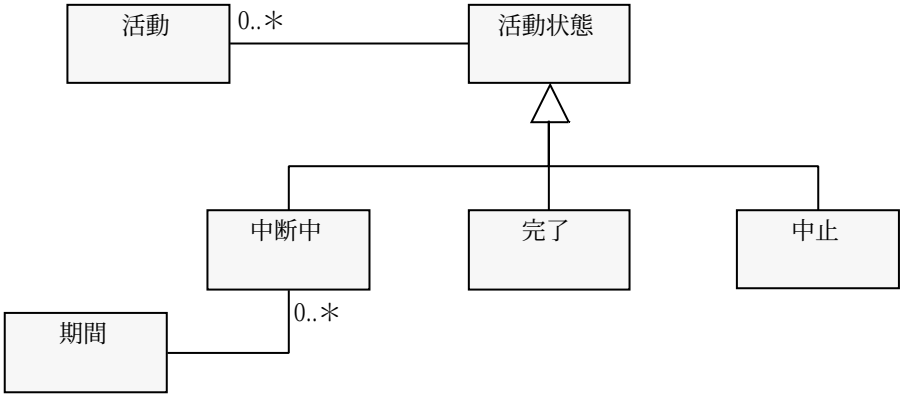
計画(Planning)

- 計画パターンを構成するパターン
  - 活動(Action)
  - 計画(Plan)
  - 資源割当(Resource Allocation)

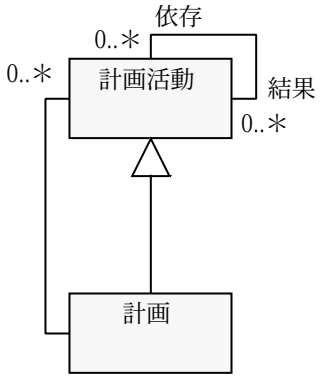
活動(Action)



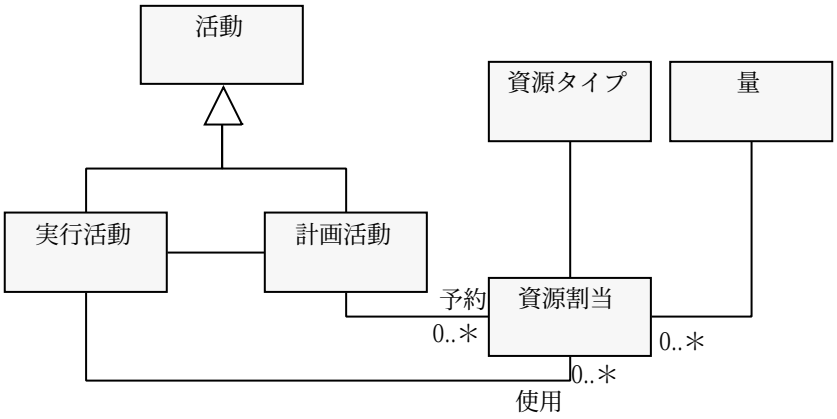
活動(Action)の状態



計画(Plan)



資源割当(Resource Allocation)



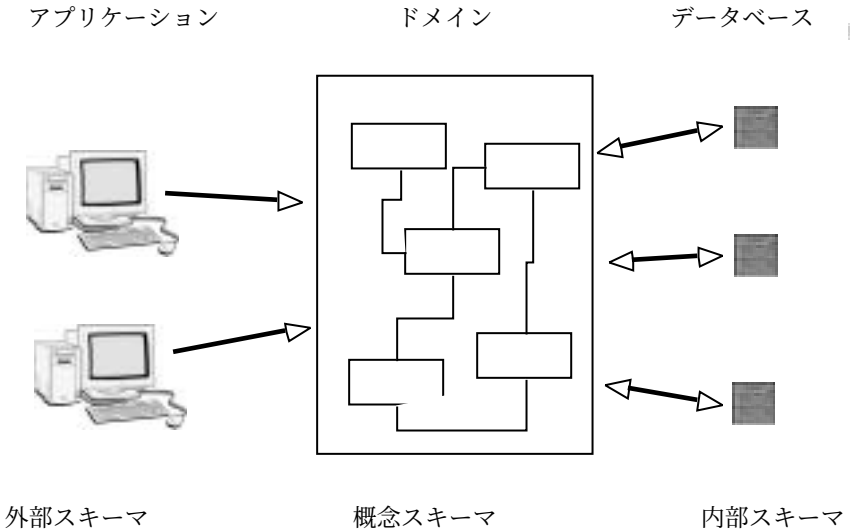
VI.アーキテクチャパターン

- アーキテクチャパターンの例
  - 分散処理パターン
    - 重要な機能を果たすサーバーとその機能を利用するクライアントのマシンを分離する
  - レイヤーパターン
    - システムを何階層もの(一種の)仮想マシンの上に構築する

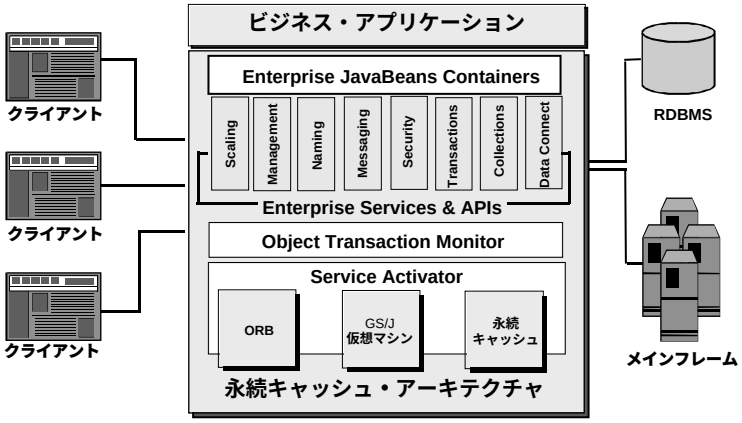
分散処理パターン

- 階層
  - 2 - tier
  - 3 - tier
- データパス
  - RPC
- トランザクション処理
  - 2 フェーズコミット
- 並行制御
  - 2 フェーズロック
  - 楽観的トランザクション、悲観的トランザクション
- ...

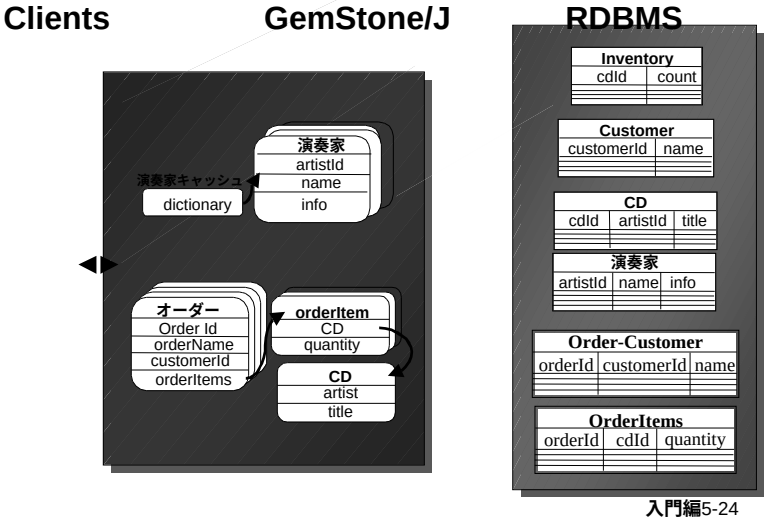
3-tier



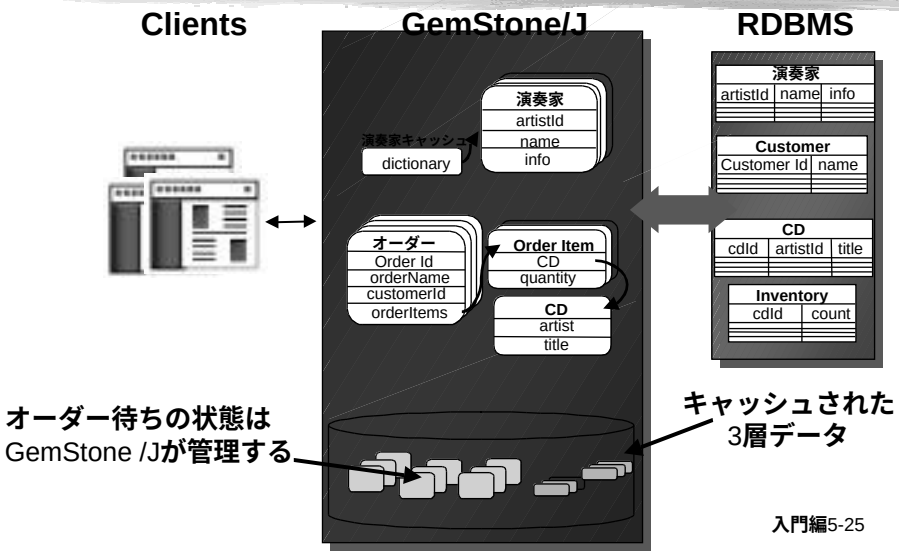
GemStone の例



ビジネス・オブジェクトのイメージ



永続キャッシュ・アーキテクチャのイメージ



レイヤーパターン

- Macintosh OSの例

Windows 98
Virtual PC
Mac OS
Mac OS X サーバー(4.4 BSD UNIX)
Machカーネル

レイヤーパターン演習

- レイヤーパターンを使った設計例を挙げよ

VII.プロセスパターン

- プロセスパターンの例
  - 開発方法論
    - OMT、RAISE、...
  - プロジェクト管理
    - 開発チームの組織化やプロジェクト管理法
      - ピープルウェア

1.オブジェクト指向分析/設計の手順

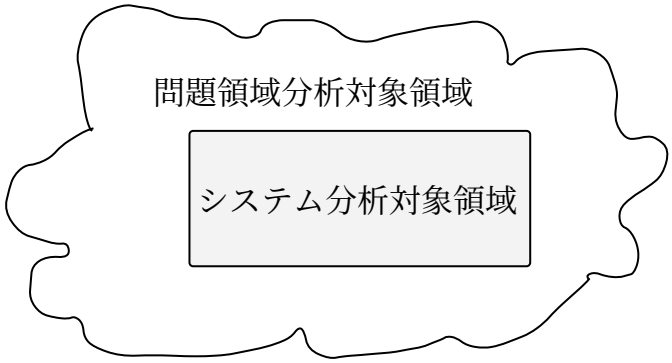
問題領域分析
システム分析
設計

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

1.1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



問題領域分析の手順

- オブジェクト指向分析でなくとも必要な項目
  - ビジネスゴールの策定
  - 用語集作成
- 振る舞い要求の発見と記述
  - UseCase作成
  - 制約の発見
- ドメインオブジェクトを見つける
  - オブジェクトの名前・責任・役割を見つける
  - オブジェクトを分類し、関連･包含関係を見つける
  - オブジェクトの類型（ステレオタイプ）発見
- オブジェクトの状態変化の記述
  - 状態遷移図の作成
- 要求仕様記述
  - 仕様記述言語による記述
- 要求の検証

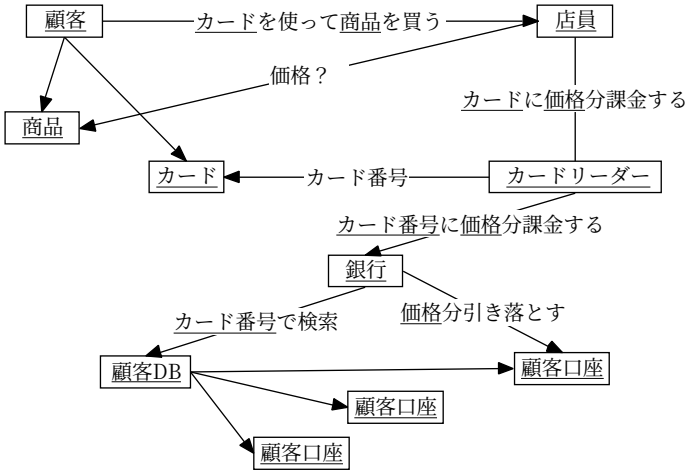
用語集

- 分析工程では、主要な用語を定義する
  - 例
    - 「ユーザー」「組織」「～情報」「～種別」
- 分析工程での成功の指標
  - 用語集を見ればそのプロジェクトの状態が分かる
    - 用語集がない＝絶望的
    - 用語集の出来が悪い＝成功は困難
- 特にオブジェクト指向では、「用語」はオブジェクトに直結するので大事

ドメインオブジェクトの発見

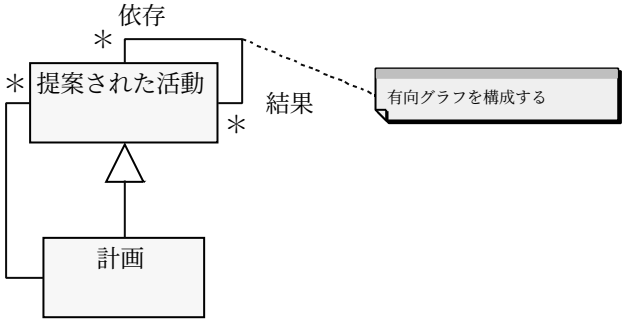
- ドメインオブジェクト
  - 対象問題領域の主要なオブジェクト
    - 明確に区別できること
    - 変化が少ないこと
    - 役割や一時的構造は変化しやすい

オブジェクトの認識



クラスとオブジェクトの関係の演習

- 次のクラス図に対応するインスタンスの関係を示せ
  - ただし、メッセージの詳細は記述しなくて良い



仕様記述言語

- 自然言語による定義は曖昧になる
  - 例
    - 「6時に京都に行くから」
    - 「医師はオーダーを発行することができる」
- 仕様記述言語
  - UMLの場合
    - OCL
  - RAISEの場合
    - RSL
  - VDMの場合
    - VDM-SL
  - B-Methodの場合
    - AMN

宣言的仕様記述の例

- rateOfInterest(倍: Real, 年: Integer, 誤: Real) : Real
- -- 利回りを求める操作
- -- rateOfInterest(2.0, 10, 0.0000001) = 0.0717735 */

- **pre:** 年 > 0 **and** 倍 > 0.0 -- pre: 前件
- **post:** Real->exists(利 : Real ¦
- 倍 ー 誤 < (1 + 利).raisedTo(年) < 倍 + 誤 **and**
- **result** = 利) -- post: 後件

attach(Observer)と detach(Observer)の場合

- **context** Subject **inv:**
- self.dependents = observer.allInstances

- **context** Subject::attach(o:Observer) : Subject
- **post:** self.dependents = self.dependents@pre->including(o)
- **and** result = self

- **context** Subject::detach(o:Observer) : Subject
- **post:** self.dependents = self.dependents@pre->excluding(o)
- **and** result = self

宣言的仕様記述の演習

- 平方根を求める操作の仕様を記述せよ
- 要素がIntegerであるCollectionクラスのインスタンスが{sorted}であることを記述せよ

1.2システム分析

- 対象システムで「何をやるか」をモデル化する
  - 「どうやって作るか」は我慢して書かない

システム分析の手順

- システム分析
  - 静的構造図を作成する
    - クラス図など
  - 動的構造図を作成する
    - 状態遷移図・順序図など
  - ドメインモデルより詳細な要求仕様を記述する
    - 制約を「操作仕様」中に記述する
  - 上記作業中、常に、以下を考えておく
    - 分析パターンが適用できないか検討する
    - タイプ・オブジェクトを探す
  - 勘所(Hotspot)の発見
    - 不変箇所と変更多発箇所(Hotspot)
  - その他の要求記述
    - プロジェクトの制約・運用要求などを記述
  - 分析の結果である分析モデルを検証

1.3設計

- 対象システムを「どうやって作るか」をモデル化する
  - 分析モデルを効率・保守性･再利用性などを考えて修正する
  - アーキテクチャを決める

設計の手順

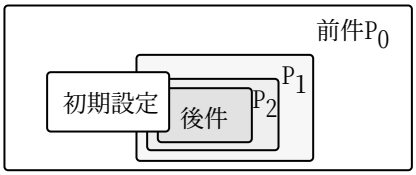
- アーキテクチャを決める
  - アーキテクチャパターンを適用する
- 設計モデルを作成する
  - 分析モデルと同じ構造だが、視点が異なるため構造が大きく変わることがある
  - デザインパターンを適用する
  - 状態遷移の実装方法を決める
  - 関連の実装方法を決める
  - 宣言的仕様の段階的洗練を行う
- 効率を検討する
  - ミクロの効率化でなく、マクロの効率化を図る
- 再利用性･保守性分析を行う
- 設計モデルを検証する

仕様の段階的洗練

- 宣言的仕様から手続き的仕様への変換公式を使って段階的に洗練する
  - グリース 著、筧 訳。プログラミングの科学。培風館、1991
  - ポター 他著、田中 監訳。ソフトウェア仕様記述　先進技法-Z言語。トッパン、1993
  - CRI 著。RAISE Method Manual。1994
- アルゴリズムがすでに存在しているときは、それを使った方が早い
  - 島内剛一、有沢誠、野下浩平、他編集。アルゴリズム辞典。共立出版、1994

仕様の段階的洗練手順

- 問題は何であるかを把握する
  - ここでは分析は終わっているはずなので省略
- 前件・後件を見つける
  - プログラムは前件を満たすどんな状態の下で実行しても、有限時間内に実行が終わり、後件を満たす状態をもたらす
- 再帰または繰り返しにより前件を後件に近づけていく
  - 実際には、後件を緩めて、前件P 0から後件へ至る途中の「条件」（P 1 , P 2など）を導出する



後件を弱める方法

- 積項を取り除く
  - 項A, B, Cがあって、A and B and Cのような形をしているとき「積項」と呼ぶ。そのうちのひとつの項を取り除いて条件を緩めようと言うのだ。
    - 例
      - post : ある曜日の集合->forAll(d ¦ from <= d and d ≦to and 曜日番号(d)=指定曜日) implies 曜日回数（指定曜日,from,to）= ある曜日の集合->size
      - d ≦toを取り除いてみる
- 定数を変数に置き換える
  - 例えば、1≦d≦10というような条件があったら、1≦d≦I and 1≦i≦10に置き換えてみるという方法である。
- 変数の領域を広げる
  - 例えば、1≦d≦10という条件があったら、0≦d≦100に置き換えてみるという方法である。

不変条件と限度関数の開発

- 反復の上限を決める「蓋」の条件
  - 先ほど取り除いた積項d ≦toの否定d ＞toを使えばよいことが分かっている。
- 反復の最中変わらない不変条件
  - 残りの積項すなわちfrom ≦d and dayNumber(d)=dayOfTheWeekとなる
- 反復が終了することを保証する「限度関数」
  - 「限度関数」tをto _ dとすれば、tは常に正であり、かつdは増加するのだから、段々0に近づいていくことが分かり、反復が終了することを保証できる
- プログラムの大筋
  - d := from;
  - do
  - ?
  - d := d +1;
  - until d ＞ to end

一応完成したプログラムの例

- 前のプログラムで？の部分は、dの曜日が指定された曜日と等しいことを保証するコードということになる。すなわち以下のようになる。
- 曜日回数（指定曜日,from,to）≡
  - /*前件：...*/　/*後件：...*/
  - variable sum :Int :=0
  - d := from;
  - /*不変条件：∃DW:曜日-set,∀d∈DW・from ≦d ∧ dayNumber(d) = 指定曜日 */
  - /*限度関数 t :: to - d */
  - do
  - if dayNumber(d)=指定曜日 then
  - sum := sum + 1
  - end
  - d := d + 1;
  - until d > to end;
  - sum

最終的なプログラム

```
曜日回数（指定曜日,from,to）≡
d := from;
while dayNumber(d) ≠ 指定曜日 do
  d := d + 1
end
(to - d) / 7＋1
```


効率の検討

- 実用的なプログラムの効率推測は非常に難しい（Knuth）
- インスタンスの数(最大･最小･平均･分散）を調べる
  - オブジェクトへアクセスするパスの最適化
- 実行・空間など各種の効率に考慮しながら、アルゴリズムを選択する
- 活動図などで実行効率を検討していく
  - シミュレーションツールなどで、モデルを評価する

再利用性・保守性分析

- 勘所（Hotspot）を発見する
  - 変更の少ない場所と、多い場所(Hotspot)を見極める
- 汎用性のある抽象データ型を作る
  - 抽象クラスを見つける
  - 多相を実現する
  - パラメータ化クラスを作る
    - 特定の型に依存しない操作を実現する
- 構造化されたモジュールを作る
  - 小さなモジュールを作る
  - 強結合のモジュールを作る
  - 少なく小さいインタフェースで実現する

Ⅷ.今後の展望と導入の課題

今後の展望
導入の課題
形式仕様とデザインパターン
参考文献

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

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, AMN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
- 算法パターン
  - 島内剛一他編、アルゴリズム辞典、共立出版、1994年
  - R.セジウィック著、野下浩平他訳、アルゴリズム、近代科学社、1992年

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 Bjørner, 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