

## ぼくにもわかるデザインパターン 第2章

### GoF パターン大カタログ ~パターンがみるみる頭にしみこむ~

(UML PRESS Vol.1 2001年12月号に特集「身近な例でGoF総ざらい&サーバサイドJava

ぼくにもわかるデザインパターン」として連載された)

[ 児高 慎治郎 中島 麻衣 坂口 直大 ]

#### GoF パターンカタログ

本章では、GoF 本に紹介されている 23 パターン (表 1) すべてをカタログにして紹介します。クラス図と適用例に示した JDK のソースなどを参考にして、GoF のデザインパターンの基本概念を確認していきましょう。

それでは、次ページからさっそく始まりです。

表 1 GoF 本のデザインパターンと本特集のたとえ話

#	分類	パターン名	目的
1	生成に関するパターン	Abstract Factory	オブジェクト群を明確にせず生成するためのインタフェースを提供する
2		Builder	オブジェクトを複合的に組み合わせる
3		Factory Method	インスタンス化をサブクラスに任せる
4		Prototype	コピーして新しいオブジェクトを生成する
5		Singleton	インスタンスが1つしか存在しないことを保証する
6	構造に関するパターン	Adapter	インタフェースに互換性のないクラス同士を組み合わせる
7		Bridge	機能と実装を別々の階層で拡張する
8		Composite	オブジェクトを木構造に組み立てる
9		Decorator	動的にオブジェクトに責任を追加できるようにする
10		Facade	複数のインタフェースに高レベルの統一インタフェースを与える
11		Flyweight	インスタンスを共有しコストを節約する
12		Proxy	オブジェクトへのアクセスを制御するために処理を代理人に任せる
13	振る舞いに関するパターン	Chain of Responsibility	要求に応じる役割をチェーン状につなぐ
14		Command	命令をカプセル化して再利用する
15		Interpreter	文法規則を表現する
16		Iterator	構造に順にアクセスする方法を提供する
17		Mediator	オブジェクト同士の結合度を低める
18		Memento	インスタンスの状態を戻すことができるようにする
19		Observer	状態の変化が自動的に通知され、更新される
20		State	状態にあわせて動作を変える
21		Strategy	アルゴリズムをカプセル化し交換可能にする
22		Template Method	特定の処理をサブクラスで行う
23		Visitor	構造と処理を分離する

Abstract Factory

ハイグレード車用部品とローグレード車用部品を作る下請け工場

たとえば

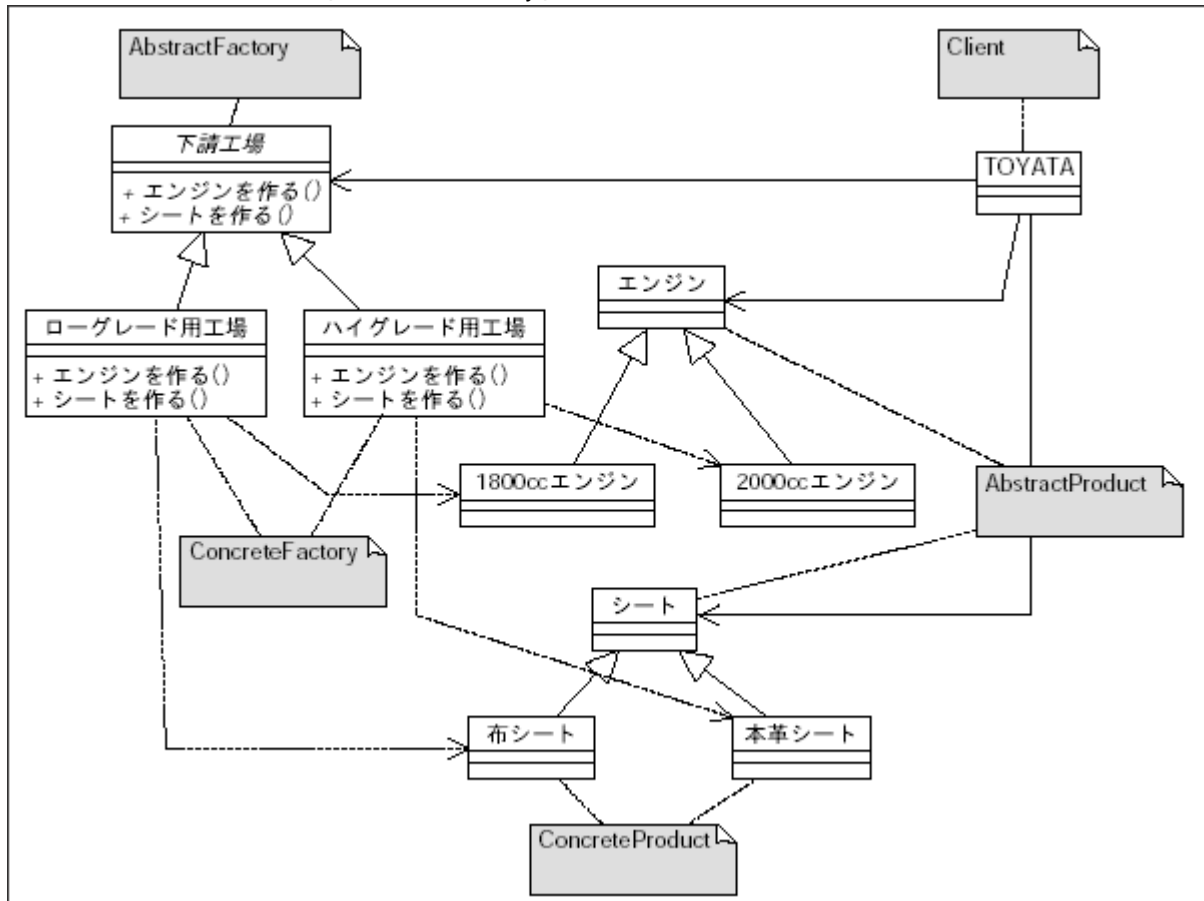
自動車メーカー TOYOTA は、人気車種カラーを生産しています。カラーには 2,000cc エンジン、本皮シート使用のハイグレード仕様と、1,800cc エンジン、布シートのローグレード仕様の 2 車種あります。いずれの部品も、製造はすべて下請けの勝どき機械に任せており、TOYOTA は勝どき機械が作った部品を組み立てて製品にし、販売しています。勝どき機械には、TOYOTA から「ローグレード車種何台分の部品」のように発注がくるため、管理のしやすさを考えて部品の製造工場をグレード別に分けています。一方 TOYOTA では勝どき機械が製造した部品を 1 箇所に集め、同じ工場ですべて同時に 2 車種の組み立てを行っています。それができるのは、それぞれの部品がグレードが異なっても取り付け方は同じになるように設計されているからです。

パターンの解説

表 1 と図 1 のように、ConcreteFactory は AbstractFactory インタフェースを実装しているので、Client はどの Factory に対しても、一様に Product 群の生成を依頼することができます。Client から Product 群の生成の依頼を受けると、ConcreteFactory は自分の役割の Product を生成し、Client に返します。また、各 Product は AbstractProduct インタフェースを実装していますので、Client はどの Factory で生成されたものか意識せずに各 Product を使用することができます。つまり AbstractFactory パターンとは、互いに関連する Product の集合を Factory を変更することによって生成し、かつそのインタフェースを提供することによって、Client から具体的な Product クラスを隠蔽するためのパターンです。

先の例でいえば、TOYOTA は下請けに対しどの車種の部品が欲しいのかを伝えれば下請けからその部品が届き、また TOYOTA はどの車種の部品か意識せずに一様にそれらの部品を使ってカラーを組み立てることができるのです。

図 1 TOYOTA と下請工場 (Abstract Factory) のクラス図



**適用例**

GoF 本では例としてユーザインタフェースツールキットを挙げています。たとえば Java はどのプラットフォームでもそのまま動くことを売りにしています。Windows と Mac ではユーザインタフェースが異なりますが、java.awt.Toolkit が AbstractFactory として動作することで、そのインタフェースの差異を吸収し、WriteOnce, RunAnywhere を実現しているのです。

表 1 例と GoF 本の対応 (Abstract Factory)

登場人物	GoF 名称
下請け工場	AbstractFactory (抽象的な工場)
ローグレード用工場	ConcreteFactory (具体的な工場)
ハイグレード用工場	
エンジン, シート	AbstractProduct (抽象的な部品)
1800cc/2000cc エンジン	Product (具体的な部品)
布/本皮シート	
TOYATA	Client

**Adapter****どこでも充電できる携帯電話用変換コネクタを作ります****たとえば**

友人の家に泊まったとき、ふと気づくと携帯電話の電池がもう残りわずか。しかし明日の朝 9 時に、バイトの面接結果が携帯電話に掛かってくるのです。

こんな経験したことはありますか?そして、その後の行動はこんな感じではないでしょうか。

**➤ 友人の充電器を試してみる**

期待はしてなかったものの、やはり友人の充電器のプラグとあなたの携帯電話の差込口がぜんぜん合わない。だって、友人の携帯は J 社で、あなたの D 社ですものね。

**➤ コンビニにダッシュ**

なんと「携帯充電サービスやってます」と入り口に書いてあります。しかし、どれを試しても全然ダメ。メーカー、機種ごとに微妙に規格が異なっていて接続できません。

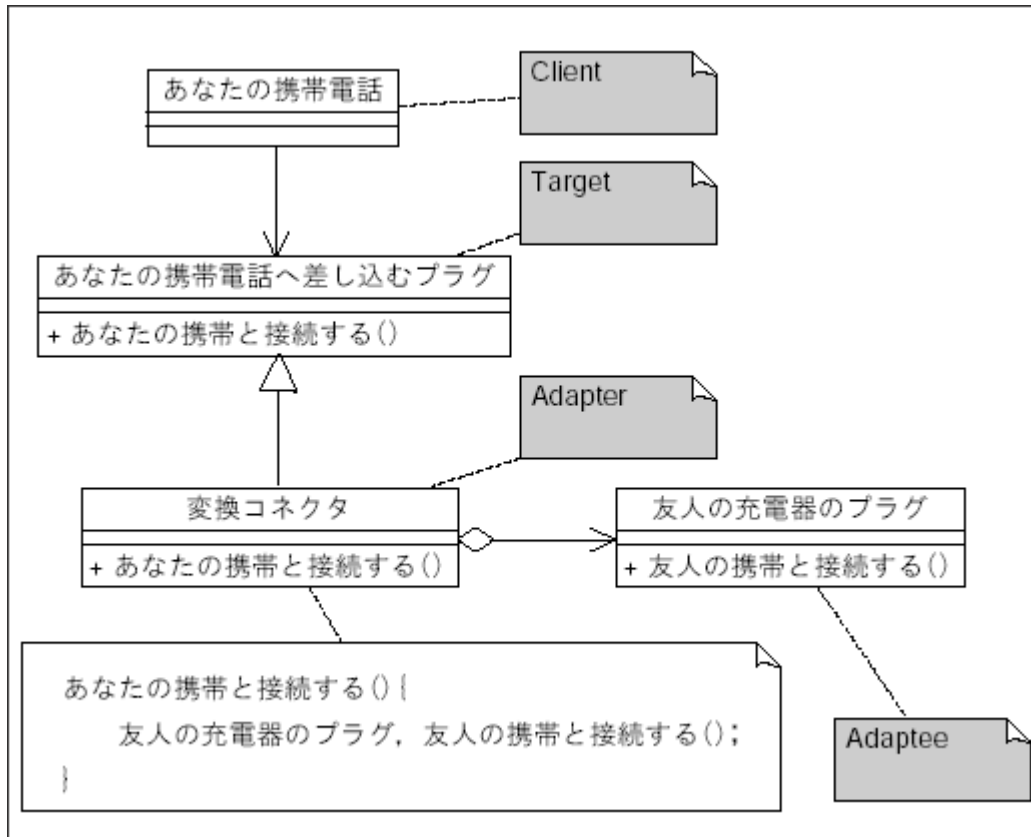
さて、携帯の充電器の「目的」はどのメーカー、機種でも変わらないのに、プラグの形が携帯の差込口と少しでも合わなければ充電することはできません。それなら変換コネクタを自分で作ってしまおうというのが、Adapter パターンの考え方です。

**パターンの解説**

Adapter パターンは、あるクラスのインタフェースをクライアントが求める他のインタフェースへ変換するときに用いられるパターンです。今回の例では、友人の充電器のプラグ (Adaptee) を変換コネクタ (Adapter) であなたの携帯電話 (Client) が求めるインタフェース (あなたの携帯と接続する) に変換します。

このパターンの利点としては、クライアントが要求するインタフェースを持たない既存のクラスを利用したい場合に、クライアントと既存のクラスのどちらも修正しなくて済むということです。クライアントや既存のクラスを修正するということは、友人の充電器のプラグをあなたの携帯へ差し込めるように、携帯電話の差込口や充電器のプラグをゴリゴリと削ることを意味します。

図2 変換コネクタ (Adapter) のクラス図



この変換コネクタ (Adapter) があれば、友人の充電器を使ってあなたの携帯電話を充電することができますね。

**適用例**

J2SE では java.awt.event.KeyAdapter など使われています。

表2 例と GoF 本の対応 (Adapter)

登場人物	GoF 名称
あなたの携帯電話へ差し込むプラグ	Target (対象)
あなたの携帯電話	Client (依頼者)
友人の充電器のプラグ	Adaptee (適合される側)
変換コネクタ	Adapter (適合する側)

**Bridge**

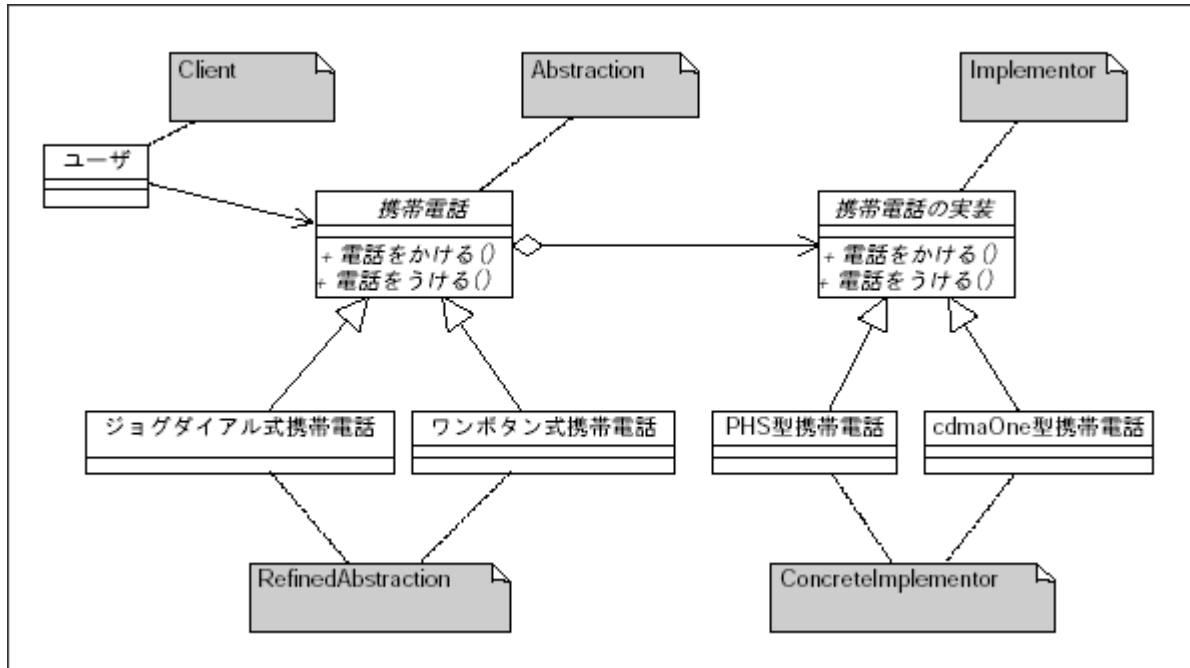
**携帯電話の機能と実装を別々の階層で拡張する**

**たとえば**

A君は携帯電話の開発を行うX社に入社した新入社員ですが、なぜか最近顔色がすぐれません。X社の携帯電話の開発工程が芳しくないことを知ったからです。

なんとX社では、新しい機種を開発するたびに「電話をかける」「電話を受ける」などの基本機能の設計を、1から作り直しています。さらに、それらの基本機能を利用するインターフェースは機種ごとに少しずつ異なっているため「ワンボタン機能」といった機能追加でも、個別のインターフェースに合わせて設計をやり直さざるを得ない状況です。これでは、時間とコストがかかりすぎます。この事態を回避するためには、携帯電話にとっての基本機能を共通化し、それをベースに拡張機能を追加するという考え方が有効になります。

図3 携帯電話の機能分離 (Bridge) のクラス図



**パターンの解説**

Bridge パターンは、共通機能を実現する「実装」のクラス階層とそれを使った拡張機能を実現する「機能」のクラス階層を、明示的に分離するためのパターンです。携帯電話では、まずその共通機能を Implementor インタフェース (携帯電話の実装) として定義しておきます。そして、実現方法の違いに合わせて ConcreteImpl クラス (PHS 型や cdmaOne 型携帯電話) を実装していきます。一方、Implementor で定義した共通機能を使って最低限の基本機能を持った携帯電話を実現するのが、Abstraction クラス (携帯電話) です。なお Abstraction クラスは、その処理を Implementor のインスタンスに委譲しています。そして、Abstraction クラスを拡張して、RefinedAbstraction クラス (ワンボタン対応携帯電話) を実装していくことになります。以上の利点をまとめると以下になります。

- **「機能」の追加に対して、既存のすべての 実装 が対応できる**

携帯電話クラスを拡張して「音声認識ダイヤル機能」を追加した場合、「PHS 型携帯電話」と「cdmaOne 型携帯電話」は同時にその機能に対応可能です。

- **「実装」の追加に対して、既存のすべての 機能 が対応できる**

新たに「FOMA 型携帯電話」を追加した場合、この携帯電話は自動的に「ジョグダイヤル機能」や「ワンボタン機能」に対応することになります。

このように「実装」と「機能」のクラス階層を分けると、機能の組み合わせによるクラス数の不要な増加を抑えることができます。

**適用例**

J2SE では、java.awt パッケージと java.awt.peer パッケージの間で Bridge パターンが適用されています。ボタンクラスでは、Implementor 役である java.awt.peer.Button が OS などによる違いを吸収し、Abstraction 役の java.awt.Button がその機能を拡張する形でボタンの機能を実現しています。

表3 例と GoF 本の対応 (Bridge)

登場人物	GoF 名称
携帯電話の実装	Implementor (実装者)
PHS 型携帯電話, cdmaOne 型携帯電話	ConcreteImplementor (具体的な実装者)
携帯電話	Abstraction (抽象化)
ジョグダイヤル式携帯電話, ワンボタン式携帯電話	RefinedAbstraction (改善した抽象化)
ユーザ	Client (クライアント)

Builder

1つの料理手順で多国籍料理を作る

たとえば

ある多国籍料理屋では、料理長の指示のもと、韓国料理人、アメリカ料理人といった各国料理人が腕を振るっています。韓国料理人が作る肉料理は焼肉ビビンバで、アメリカ料理人が作る肉料理はステーキです。

韓国料理人やアメリカ料理人が行う料理手順は、下ごしらえ / 調理 / もりつけと共通しています。料理長は料理人に対して、素材となる肉を提供し、基本的な手順を指示します。指示があると各料理人が調理を始めます。

お客さんから焼肉ビビンバの注文がありました。料理長はおいしい肉を吟味し、韓国料理人に下ごしらえ / 調理 / もりつけの指示を出します。韓国料理人は、料理長の指示に従って焼肉ビビンバを作ります。「下ごしらえ」の指示で焼肉用のタレを作り、「調理」の指示で肉を焼き、具とご飯をいためます。「もりつけ」で、どんぶりに盛ります。こうして焼肉ビビンバのできあがりです。同じようにステーキの注文があれば、料理長は同じ指示をアメリカ料理人に出します。このように、同じ料理手順で異なる

肉料理を作るといった目的を実現するために Builder パターンは使われます。

パターンの解説

「料理人」は、肉料理を作る料理手順のみをインタフェースとして定義している Builder (建築者) です。ConcreteBuilder (具体的な建築者) である「韓国料理人」や「アメリカ料理人」は「料理人」のインタフェースを継承し、肉料理の料理方法を個別に実装しています。

Director である料理長は料理の注文を受けると、料理人 (Builder) に「下ごしらえをしろ」といった指示を出します。たとえば注文された料理が焼肉ビビンバのとき、韓国料理人 (ConcreteBuilder) が料理長の指示に従って焼肉ビビンバ (Product) を料理していきます。その結果、できあがった料理がお客さんのもとへ届くことになります。

Builder パターンを用いることにより、新しい ConcreteBuilder クラス (たとえばメキシコ料理人) が追加されても、Director クラスが影響を受けないで済む (「料理人」への指示ができれば料理は作れる) というメリットがあります。このように1つの作成過程で異なる製品を作りたいときに、Builder パターンは役立ちます。

図4 料理長と料理人 (Builder) のクラス図

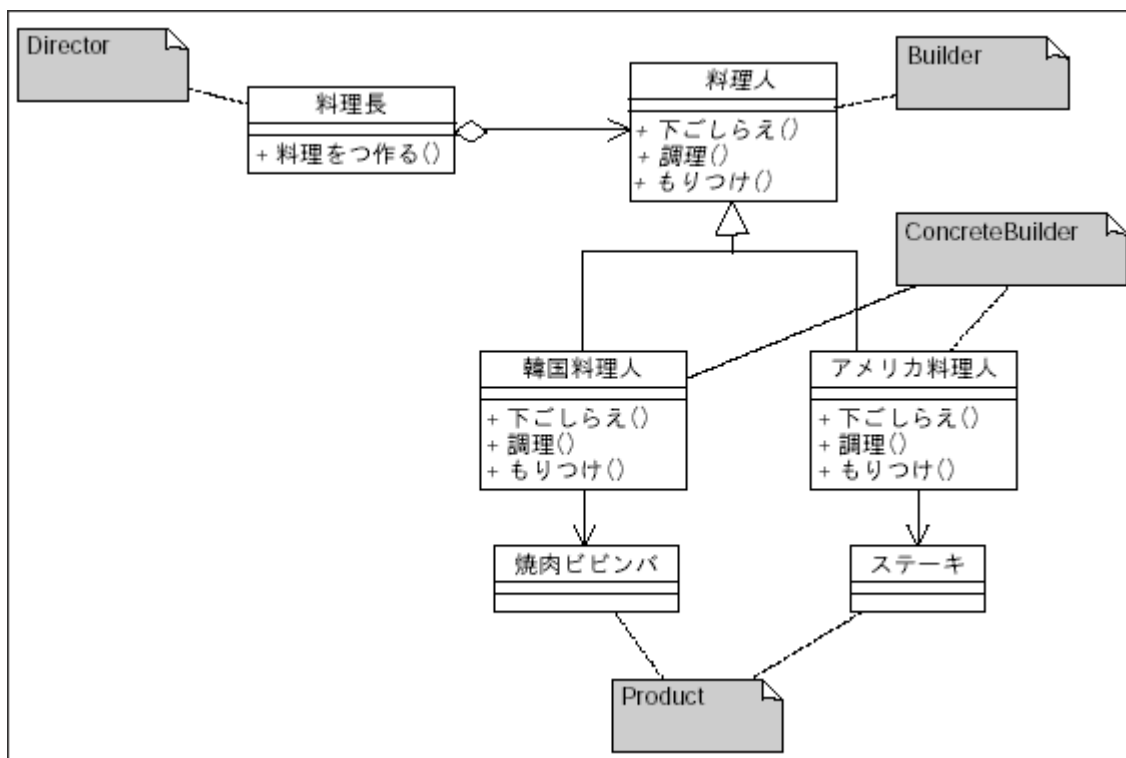


表 4 例と GoF 本の対応 (Builder)

登場人物	GoF 名称
料理長	Director (監督)
料理人	Builder (建築者)
韓国料理人, アメリカ料理人	ConcreteBuilder (具体的な建築者)
焼肉ピビンバ, ステーキ	Product (製品)

### 適用例

オブジェクトを CSV 形式などのファイルに保存 / 復元するしくみを作る場合に、読み込んだファイルの種類に応じて対応する元のオブジェクトを復元する部分で、Builder パターンの適用が考えられます。

### Chain of Responsibility

#### 業務仕様の質問に応じる人をチェーン状につなぐ

##### たとえば

新人 SE の A さんは金融系システムの開発に携わっていますが、まだまだわからない業務仕様が多いのです。そこで A さんは不明な業務仕様があると、プロジェクトリーダーの B さんに質問します。

B さんは業務仕様について知っていれば回答しますが、わからないものや、まだ決まっていない仕様の場合は発注元である Y 銀行の C さんに問い合わせます。C さんは問い合わせのあった業務仕様について回答してくれます。しかし C さんでは判断がつかない場合もあり、質問の種類に応じて営業担当の D さんや融資支援担当の E さんに業務仕様を問い合わせます。

このように、要求を処理する人が複数いて、誰が処理するのか決まっていない場合に Chain of Responsibility は有効です。

プロジェクトリーダーの B さんを通さずに、お客さんである C さんに直接聞いたほうが早いかもしれませんが。しかし毎回 B さんに質問していれば、A さんは誰が回答をくれるかを知らなくても済みます。担当が C さんから Y さんに変更になったとしても、プロジェクトリーダーの B さんだけがそのことを知っていれば済むのです。

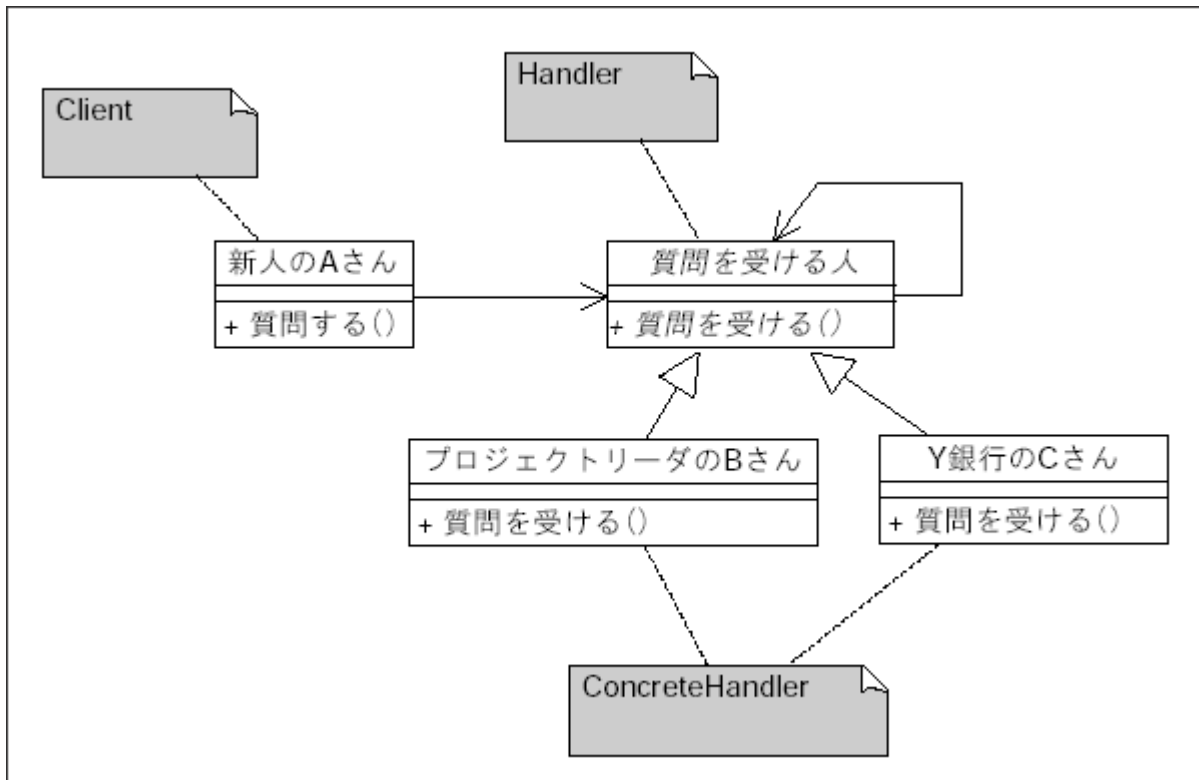
### パターンの解説

新人 SE である A さんの質問を受けつける B さんや C さんを、総称して Handler と呼びます。Handler では、A さんの質問を受けるといったインタフェースを定義しています。Handler の役割は「要求を処理するためのインタフェースを定義する」ことなのです。

プロジェクトリーダーの B さんや Y 銀行の C さんといった、A さんの質問を受けける各個人は ConcreteHandler です。この人たちが A さんの質問を実際に受けつけます。

また、質問を受けつける各個人は皆、次に質問を受けつけてくれる人のことを知っています。もしも B さんが A さんの質問に答えられなければ C さんへ、C さんが答えられなければ D さんへと質問に答えてくれる人をたらいまわしにしていくのです。

図5 新人SE と業務質問 (Chain of Responsibility) のクラス図



**適用例**

J2EE パターンの InterceptingFilter は ChainofResponsibility を適用しています<sup>1</sup>。

また、サーブレットエンジンの Tomcat では、要求の送信オブジェクト (Tomcat コア,サーブレット) と実際の受信オブジェクト (要求されたオペレーションを実行するモジュールの1つ) の結合を避けるために、インタセプタと呼ばれるメカニズムを採用しています。これに ChainofResponsibility が適用されています。

表5 例と GoF 本の対応 (Chain of Responsibility)

登場人物	GoF 名称
新人のAさん	Client (依頼者)
質問を受ける人	Handler (処理者)
プロジェクトリーダーのBさん, Y銀行のCさん	ConcreteHandler (具体的処理者)

<sup>1</sup> 特集 4第 3章も参照ください

**Command**

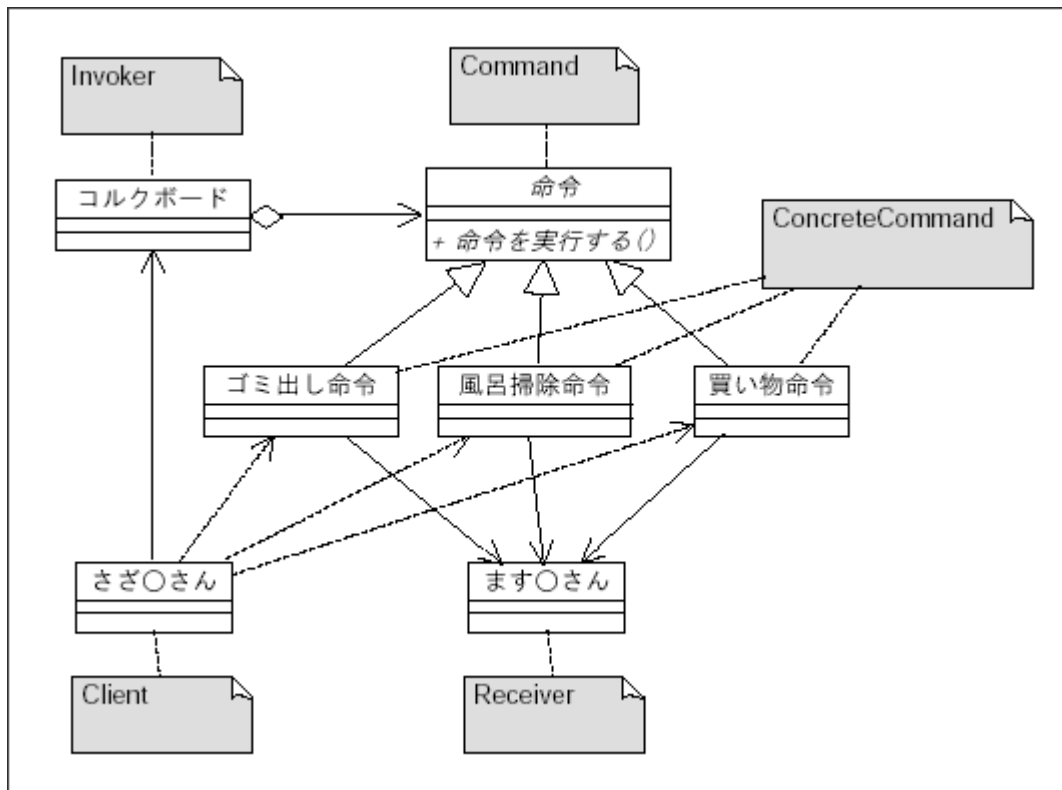
**さまざまな命令をマス さんに実行させることができます**

**たとえば**

マス さんの趣味は昼寝。3度の飯より昼寝好きです。よく晴れた休日の午後など、縁側で愛猫のドラと仲良くゴロゴロしています。そうなるともうだめです。テコでも動きません。サザ さんはそんなマス さんが不満です。こっちは食事の準備 / 掃除 / 洗濯 / 寝たきりになった波 じいさんの世話まですないとはいけません。少くく手伝わってもらえないとやっていけません。それなのに、マス さんは命令されるけど聞こえなかった、証拠がない、覚えられないと言い逃れをし、いつも縁側でゴロゴロ。

とうとう頭にきたサザ さんは考えました。お願いしたいことを紙に書いておき、やってほしいときに縁側のコルクボードに貼るのです。そうしておけば証拠も残るし、その紙も何度も使いまわすことができます。ある休日の朝、サザ さんは「風呂掃除」命令をコルクボードに張りました。はたして、マス さんはサザ さんの思惑通りやってくれるでしょうか。

図6 ますさんとさざさん (Command) のクラス図



**パターンの解説**

Client (サザ さん) は Receiver (マス さん) への命令 (ConcreteCommand) を作成します。Client はその命令を実行したいときに Invoker (コルクボード) に命令を渡すと、Invoker はその ConcreteCommand を実行します。この命令を取り消し可能にする場合は、Invoker は実行前の状態をセーブしておきます。また、何の命令を行ったかもここで管理します<sup>2</sup>。

ConcreteCommand は Receiver (マス さん) に対して行動の呼び出しを行い、Receiver はその呼び出しどおりに行動します。

このように命令をする人 (Client) と命令を実行する人 (Receiver) が Command によって分離され、それによって Command の再利用と履歴管理を可能にしています。これが Command パターンです。

応用として「夕飯を作る」命令の後に必ず「洗い物をする」命令を行わせる場合、Command パターンに Composite パターンを組み合わせるということも

<sup>2</sup> この例の場合、Invoker に命令を書いた紙が貼り付けられているので、履歴の管理もされています

できます。たとえば Excel のマクロはこのやり方で実装できそうです。

**適用例**

Command パターンの実装例については、第 3 章の事例を参照ください。

表 6 例と GoF 本の対応 (Command)

登場人物	GoF 名称
命令	Command (命令)
ゴミ出し命令	ConcreteCommand (具体的な命令)
風呂掃除命令	
買い物命令	
ます○さん	Receiver (受信者)
コルクボード	Invoker (起動者)
さざ○さん	Client (依頼者)

Composite

営業部の階層構造を簡潔に表現します

たとえば

某メーカーの営業部には、管理職と営業職の区別がありません。営業部に所属している社員は部長であろうが課長であろうが、平社員であろうが外回りの営業を行います。また社員の中にも、正社員と契約社員がいます。正社員と契約社員の違いは、正社員には部下がつきますが、契約社員には部下がつかないことです。また社員の評価は、すべて売上成績に基づいて行われます。その営業成績は、部下の営業成績が合算された形で計算されます。これには正社員も契約社員も、区別はありません。ある日人事部からAさんに問い合わせがありました。「君のこのころの営業成績は？」

Aさんには直属の部下が3人います(図7-1)。2人は契約社員で、1人は正社員です。正社員にも部下があり、部下の合計は全員で5人になります。全員の成績を計算するのは面倒そうですが、Aさんは直接営業成績を尋ねるのは直属の部下3人で済みます。なぜなら彼らの部下の成績は、彼らが集計してくれるからです。Aさんにとっては部下が正社員か契約社員なのかということとは関係ありません。

図7-1 営業成績の計算

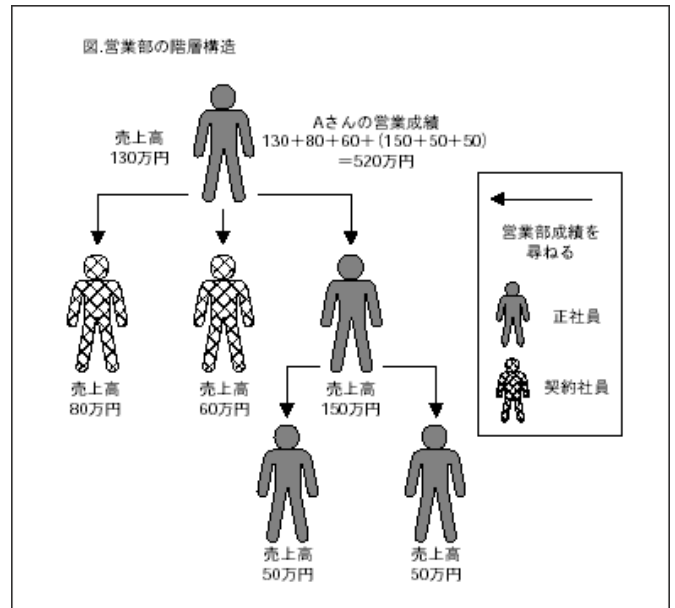
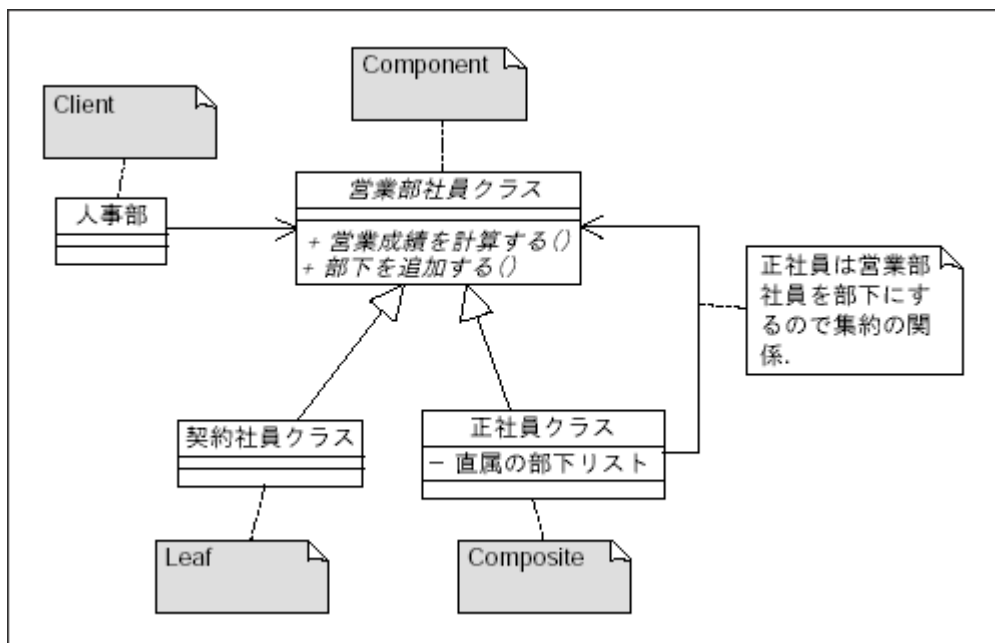


図7-2 営業部階層構造 (Composite) のクラス図



### パターンの解説

「営業部社員」Component は、Composite である正社員と Leaf である契約社員の共通の性質をまとめたものです。

Composite も Leaf も Component を継承しますので、営業成績を計算するメソッドと、部下を追加するメソッドを実装しなければなりません。これにより、人事部はその人に聞くだけで、その人の部下も含めた営業成績を得ることができます。ただ、Leaf は部下に対して追加できないので、部下を追加するメソッドは空にするか、追加しようとしたときに例外をスローするなどの必要があるでしょう。

このように Composite パターンは、Client が Leaf なのか Composite なのかを意識しなくても統一した操作を可能にし、それが Composite の場合は再帰的に操作を繰り返してくれます。

### 適用例

Windows などのファイルシステムで使われています。Leaf がファイル、Composite がフォルダです。あるフォルダを別の場所に移動すると配下のファイル/フォルダとも同じように移動します。これはまさに Composite パターンと言えるでしょう。

J2SE では java.awt.Component と Container や Button などの関係が Composite パターンになります。

表 7 例と GoF 本の対応 (Composite)

登場人物	GoF 名称
営業部社員	Component (構成要素)
正社員	Composite (合成物)
契約社員	Leaf (枝)
人事部	Client (依頼者)

### Decorator

エフェクタを使えば、チープなギター音がホットなサウンドに早変わり

### たとえば

ミュージシャン志望のハンス君は、近くの商店街でエレキギターを衝動買いしました。部屋に戻っておもむろに「ドレミファソラシド」の練習をはじめましたが、なんだかピンときません。

「俺が求めていたサウンドはこれじゃない！」思わずギターを叩き壊しそうになるハンス君。しかし「エフェクタ」という機械をギターとアンプの間につなげばいろいろな音色が得られることを知りました。さらに、いくつも重ねて通すことでいろいろな複合効果生まれるということも覚えしました

「これなら俺だけの音がつくれるぜ！」ハンス君はさっそく「エフェクタ」なるものを購入しに商店街の楽器屋へと走り出しました。ハンス君のチョイスしたエフェクタはリバーブ(空間的な残響効果)、コーラス(音を重複させる)、ディストーション(ヘビメタ風)、コンプレッサ(一定音量を保つ)など4つです。

気がはよいハンス君は、とりあえず買ってきたすべてのエフェクタを重ねて通して、フルボリュームでギターをかき鳴らしました。

「ギョイイイイインーーーー」

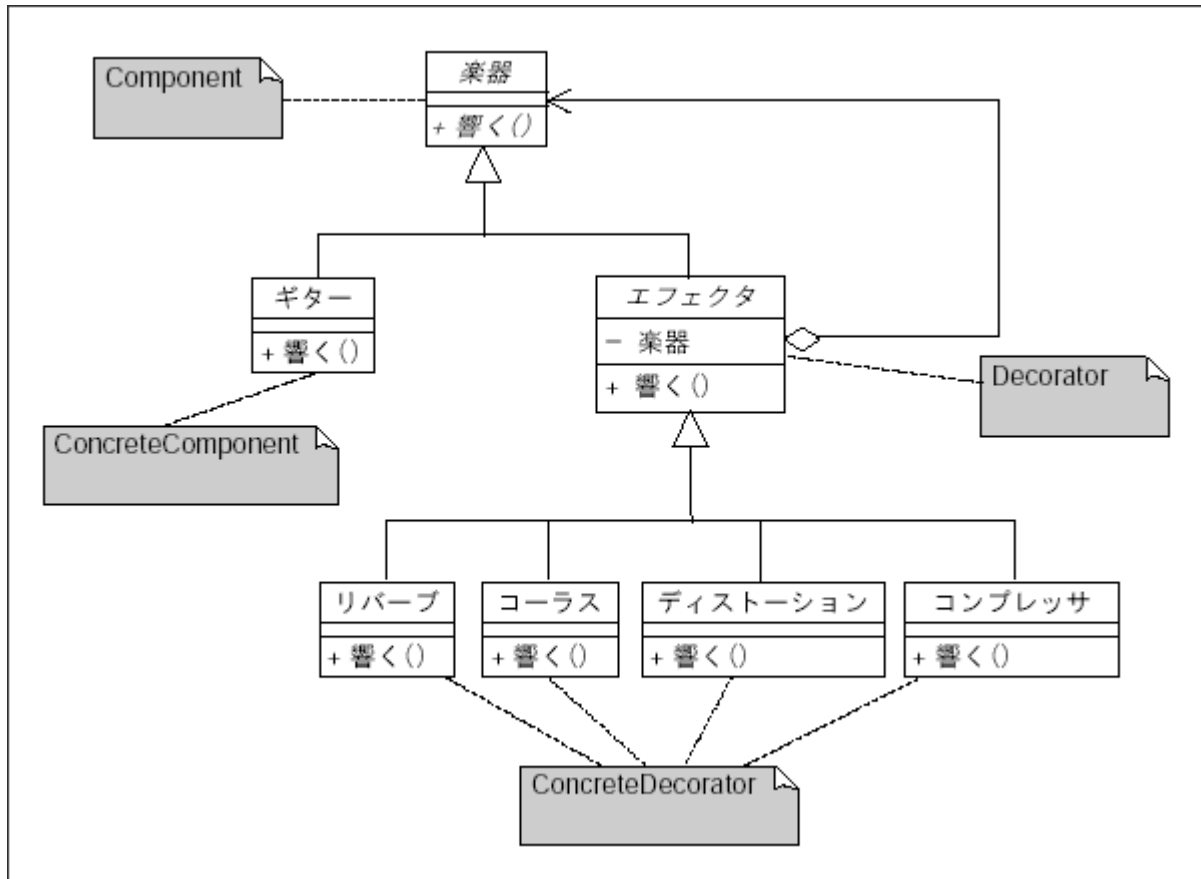
表 8 例と GoF 本の対応 (Decorator)

登場人物	GoF 名称
楽器	Component (コンポーネント)
ギター	ConcreteComponent (具体的なコンポーネント)
エフェクタ	Decorator (装飾者)
リバーブ ディストーション コンプレッサ コーラス	ConcreteDecorator (具体的な装飾者)

### パターンの解説

Decorator パターンは、継承を用いずに委譲を使って機能拡張をします。その拡張する様子が、さも機能を装飾していくように見えるので Decorator (装飾者) と呼ばれます。

図8 エフェクターとギター (Decorator) のクラス図



委譲を使った機能拡張のメリットは、装飾する芯となる Component を継承した ConcreteComponent に Decorator を継承した ConcreteDecorator を付けたりはずしたりして、実行時に簡単に機能の追加や削除を行えることです。また、重ねて使うこともできます。

今回の例では、最初は痩せた物足りない音を返すだけのギタークラスの「響く」を ConcreteDecorator である各エフェクタを通すことにより、いろいろな音響効果をもった音を返す「響く」に拡張することができましたね。

**適用例**

J2SE では、java.io.OutputStream が Decorator パターンを適用しています。

**Facade**

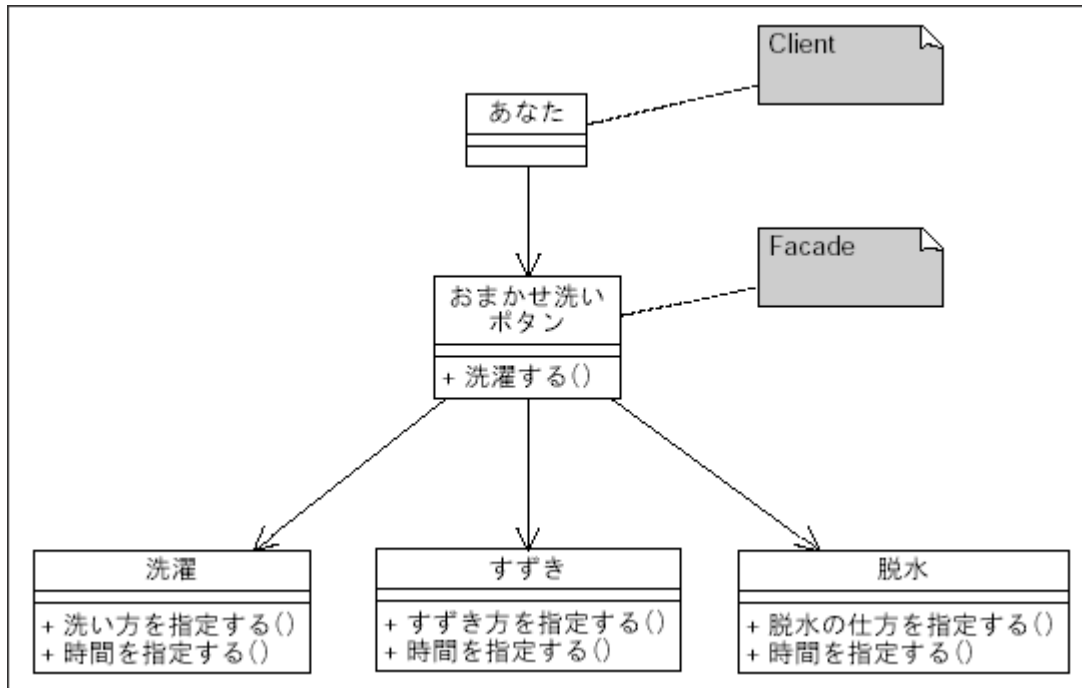
「おまかせ洗いボタン」一発だとらくちんです

**たとえば**

最近の洗濯機はすっかり便利になりました。単に全自動であるに留まらず、衣料の素材別に洗い方を変えたりでき上がり時間を指定したりと、数多くの便利な洗濯機が作り出されています。しかしいくら便利になっているとはいえ、洗濯機の持つ代表的な機能は、「洗う」「すすぐ」「脱水する」の3つです。

ここで「洗う」機能を考えてみると、「30分かけて洗う」「ひどい汚れは強めに洗う」「化繊類は丁寧に洗う」など、さまざまな洗い方をあなたは選ぶことができます。同様にすすぎや脱水についても、時間や処理方法を事細かに指定することができます。

図9 洗濯機の機能 (Facade) のクラス図



ところで、あなたは普段洗濯をするときにいちいち洗い方や仕上げの内容を指定したりするでしょうか？そう、たいていの全自動洗濯機なら付いている「おまかせ洗いボタン」を押して、あとはほったらかしですよね。でも大切なセーターを洗うときなどは、あなた自身が「洗濯」「すすぎ」の指示を細かく指定して使うこともできます。

このように、状況に応じて、自分で洗濯機の細かい機能を指定したり、ボタン1つでらくちんに済ませたりと使い分けができるのです。デザインパターンでも、Facade と呼ばれるパターンを使うことにより、簡単な操作であなたに代わって洗濯してくれる、この便利な「おまかせ洗いボタン」を実現することができます。

### パターンの解説

Facade とは、フランス語で「正面窓口」という意味です。Facade パターンは、Client (依頼人) に対して単純化された高レベルのインターフェースを提供します。当然あなた (Client) 自身は「洗濯」クラスや「すすぎ」クラスといった低レベルのインターフェースを使用することもできますが、時間がなくて適当に洗濯したいだけの Client にとっては、これらは多少荷の重いインターフェースだといえます。

そこで、あなたに代わって「洗濯」クラスや「すすぎ」クラスのメソッドを呼び出して適当に洗濯を行う「おまかせ洗いボタン」クラス (Facade) を用意するのです。もちろん便利な反面細かい操作はできなくなりますが、Client の要求のレベルによってはよりシンプルで扱いやすいインターフェースとなります。

このように Facade パターンを用意すれば、Client は巨大で複雑なクラス群を利用する際に、低レベルなインターフェースを意識せずに、必要な機能だけを備えた高レベルなインターフェースを使用することができます。

### 適用例

J2SE では、java.net.URL クラスで Facade パターンが適用されています。

表9 例と GoF 本の対応 (Facade)

登場人物	GoF 名称
「おまかせ洗いボタン」	Facade (正面)
あなた	Client (依頼人)
洗濯, すすぎ, 脱水	—

Factory Method

実際の衣料品の製造は各々の工場で行います

たとえば

(株)UNIQUE は国内有数の衣料品メーカーです。主力商品にはフリース/チノパン/ジャケットがありますが、生産性を重視するため、A工場ではフリース、B工場はチノパン、C工場はジャケットというように各工場決められた商品だけを製造しています。

また工場から店舗への配送もコスト削減のため、一括して白猫運輸に委託しています。工場は白猫運輸に衣料品を渡し、白猫運輸はその衣料品がジャケットであるかフリースであるかは意識せず、決められた店舗へ配送します。

工場は、営業部からの依頼で衣料品を製造します。製造した衣料品はいったん工場在庫として保持しておき、営業部からの依頼で、各店舗に配送するため白猫運輸に渡します。

パターンの解説

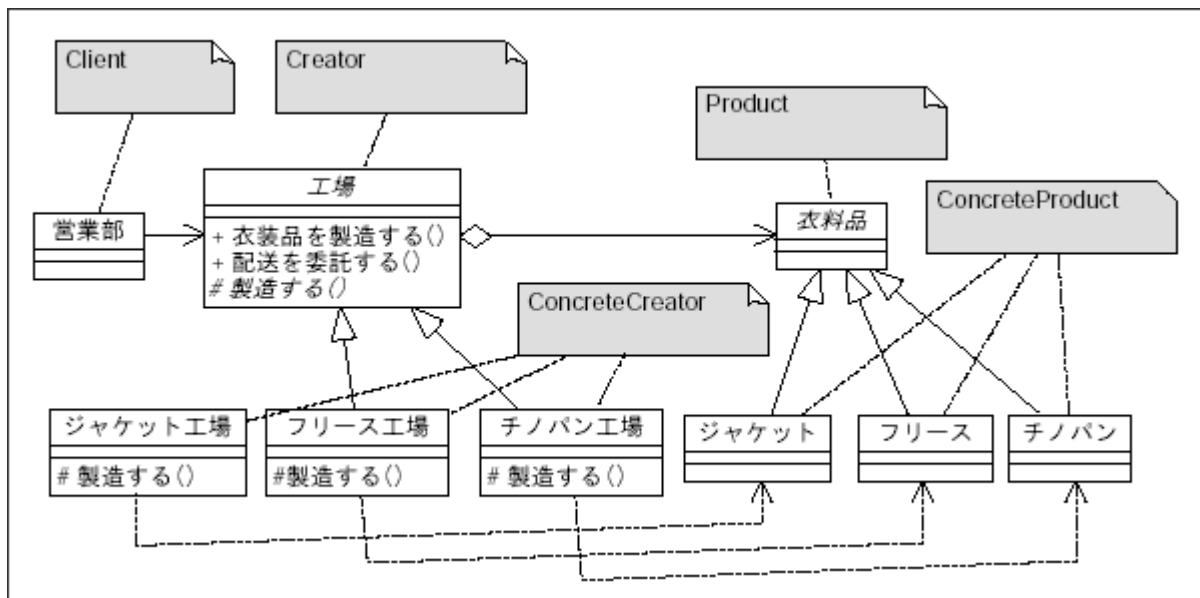
具体的に何を製造するかは各工場によってばらばらのため、製造する役割はサブクラスである

ConcreteCreatorで定義しなければなりません。ところがサブクラスで製造された衣料品をスーパークラスが知らなければ、配送を白猫運輸に依頼することができません。まさにこんなとき、FactoryMethodパターンが有効です。

ClientがCreatorに対し「衣料品を製造する」メソッドを呼ぶと、CreatorはサブクラスであるConcreteCreatorの「製造する」メソッドを呼びます。ConcreteCreatorはConcreteProductを生成し、Productとして返します。Creatorでは受け取ったProductを保持しておき、Clientからの要求でそのProductを操作します。

すなわちFactoryMethodパターンとは、スーパークラスでそのインスタンスを扱いたい(工場が衣料品の配送を白猫運輸へ依頼したい)が、スーパークラスではそのインスタンスを生成することができない(何を製造するかはサブクラスである各工場によって異なる)とき、スーパークラスではインスタンスを生成するためのインタフェースだけを規定し、具体的なインスタンスの生成は各サブクラスで行うというパターンです。

図 10 衣料品工場 (Factory Method) のクラス図



ちなみに、FactoryMethod パターンは TemplateMethod パターンの応用事例です。工場クラスの「衣料品を製造する」メソッドが呼ばれたとき、具体的な処理はサブクラスの「製造する」で実装します。この例では、FactoryMethod パターンはインスタンスの生成を TemplateMethod パターンで実装しているとも言えるでしょう。

### 適用例

java.net.URLConnection の getContent メソッドは FactoryMethod パターンを使用しています。

表 10 例と GoF 本の対応 (Factory Method)

登場人物	GoF 名称
工場	Creator (工場)
衣料品	Product (製品)
ジャケット工場	ConcreteCreator (具体的な工場)
フリース工場	
チノパン工場	
ジャケット	ConcreteProduct (具体的な製品)
フリース	
チノパン	
営業部	Client (依頼者)

## Flyweight

### 体操服を共有して出費を節約します

#### たとえば

12人兄弟のお母さんは、それぞれの子供たちが学校に入学するたびに必要なものを買ってあげなければなりません。文具、本、かばん、体操服...、必要なものはたくさんあるのに12人分なんて。考えただけでも大変な出費です。そこでお母さんは考えました。「中学校の体操服を兄弟たちで使いまわすようにして、かかるコストを抑えよう！」

お母さんは長男の入学時に中学校の体操服を1つだけ買い、あとは兄弟たちで使いまわすようにしました。このとき、12人は同じ中学校に通うので体操服のデザイン、中学校の校章マークはみんな同じです。ただ、名前とクラスがかかっている名刺だけはそれぞれ12人で違うので、体操服を使うときに付け替えるようにしました。そうすれば、うまく兄弟たちで使いまわすことができます。

...というように、毎年毎年12人の兄弟たちに体操服を買うのではなく兄弟で体操服を共有することにより、出費を抑えることができます。

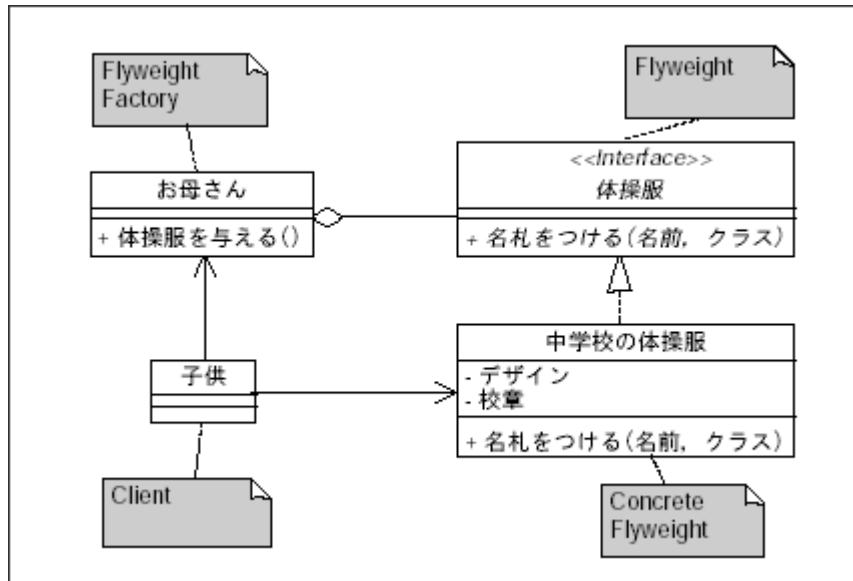
#### パターンの解説

Flyweight パターンは、類似のインスタンスを多数利用するときに「インスタンスを共有して使いまわす」パターンです。

この Flyweight を実装したものを ConcreteFlyweight といいます。たとえば話でいうと、Flyweight が体操服で、ConcreteFlyweight が実際に使いまわす中学校の体操服です。また、Flyweight の管理者を FlyweightFactory といいます。FlyweightFactory は、Client から「該当する Flyweight」の要求がきたときに、その Flyweight インスタンスを返す役目をします。たとえばではお母さんが FlyweightFactory、12人の兄弟たちが Client にあたります。兄弟たちがお母さんに中学校の体操服を要求すると、お母さんから共有する体操服をもらえるというわけです。

ここで、共有するインスタンスはまったく同じものでなくても OK です。共有できる情報は共有インスタンスに持たせ、共有できない情報はインスタンスを利用する際にそのインスタンスに渡すことにより、インスタンスの再利用を効率よく行うことができます。

図 11 兄弟と体操服 (Flyweight) のクラス図



体操服については、みんなで共有する情報として、体操服のデザインや中学校の校章マークなどがあります。

そして、共有せず体操服を利用する際に与える情報としては、名札(名前, クラス)があるでしょう。

このように共有できる情報と共有できない情報を切り出すことにより、インスタンス(中学校の体操服)を上手に共有し、効率よく利用することができます。これによりメモリ(出費)を節約することができるというわけです。

### 適用例

J2SE では、`java.lang.String` で Flyweight パターンが利用されています。JVM 内で、同じ定数を表す String インスタンスを使いまわすようにしています。

表 11 例と GoF 本の対応 (Flyweight)

登場人物	GoF 名称
体操服	Flyweight (フライ級)
中学校の体操服	ConcreteFlyweight (具体的なフライ級)
お母さん	Flyweight Factory (フライ級の工場)
子供	Client (依頼者)

## Interpreter

英文法を定義し、英文を解釈します

### たとえば

「文法」と聞いて誰もが真っ先に思い出すのは、やはり英文法ではないでしょうか。「This is a book.」という文は、基本文型の第 2 文型 (SVC) であると中学校で習ったはずですが。「this」が S、「is」が V、「a book」が C です。またさらに、「a book」は「a」という冠詞と「book」という名詞に分かれます。「this」は代名詞、「is」は自動詞に分類されます。

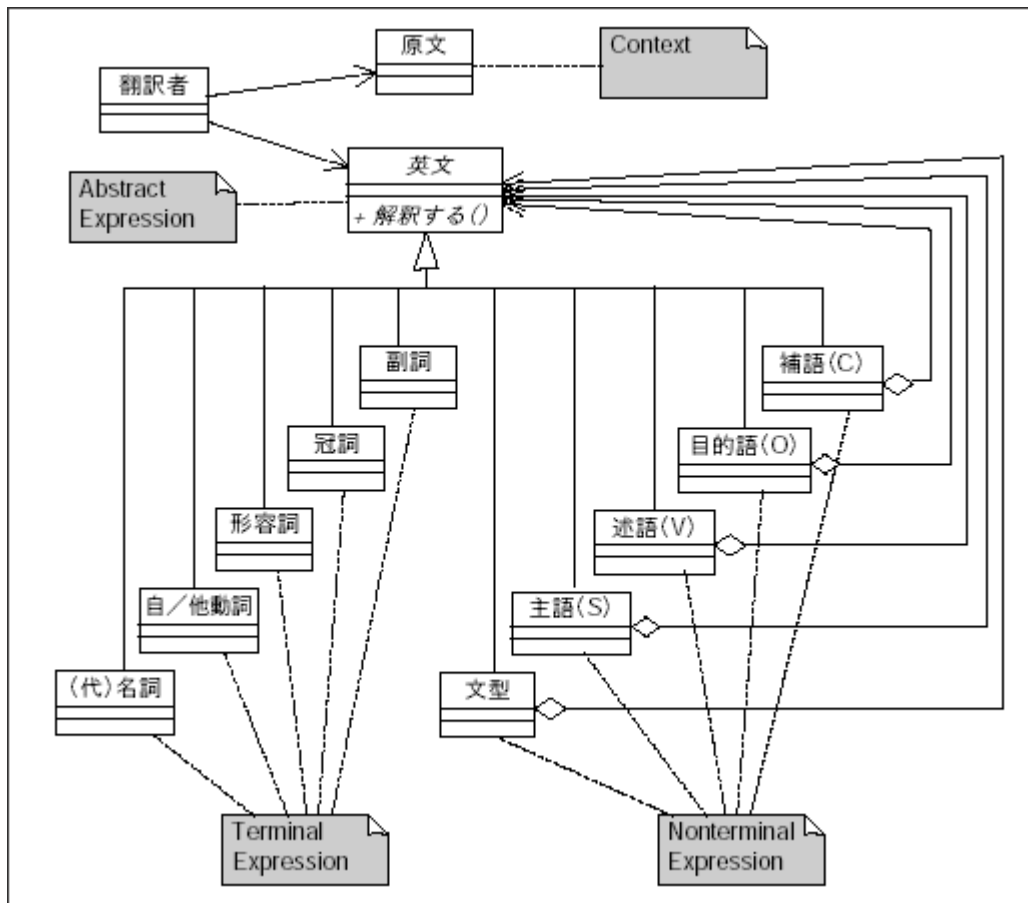
では、もしこのような文法の解釈を行うプログラムを作るとしたらどうしますか? 「正規言語」「有限オートマトン」「LR 法」といったコンパイラの原理を勉強しますか?

実はここで Interpreter パターンを利用すると便利です。確かに英文法のような複雑な文法規則を表現するには多少荷が重いのですが、ちょっとした文法規則をちょっとした時間で作る場合には、非常に有効です。

### パターンの解説

Interpreter パターンは、ある文法規則を定義し、与えられた言語を解釈するために利用されるパターンです。

図 12 英文法 (Interpreter) のクラス図



文法規則には大きく分けて 2 つの表現があります。1 つは TerminalExpression ( 終端となる表現 ) と呼ばれ、「代名詞 ::= this | that | ... 」といった他の規則を適用する表現です。2 つ目は NonterminalExpression ( 非終端となる表現 ) で、「英文 ::= SV | SVC | ... 」のようにさらに別の規則を内包します。NonterminalExpression はスーパークラスである AbstractExpression を集約しており、TerminalExpression もしくは NonterminalExpression のインスタンスを子として持ちます。つまり、NonterminalExpression は再帰的に ( 文法規則が許すかぎり ) いくつでも子孫となる Expression の具象クラスを持つことができます。逆に TerminalExpression は AbstractExpression を集約しないので、それ以上は子を持つことができません。このように、Interpreter パターンでは言語を解釈するロジック ( 文法規則 ) を個別の Expression 具象クラスに局在化できます。これにより個々のクラスの責任分担が明確になり、少ない労力でロジックの実装を行うことができるようになります。

**適用例**

J2SE では、java.text.Format のサブクラスで Interpreter パターンが適用されています。AbstractExpression の役割を担う Format のサブクラスとしては、数値を解析する NumberFormat や日付や時刻を解析する DateFormat などがあります。

表 12 例と GoF 本の対応 ( Interpreter )

登場人物	GoF 名称
英文	AbstractExpression ( 抽象的な表現 )
( 代 ) 名詞, 自 / 他動詞, 形容詞, 冠詞, 副詞	TerminalExpression ( 終端となる表現 )
文型, 主語 ( S ), 述語 ( V ), 目的語 ( O ), 補語 ( C )	NonterminalExpression ( 非終端となる表現 )
翻訳者	Client ( 依頼者 )
原文	Context ( 文脈, 前後関係 )

## Iterator

## 5年1組の生徒を名簿から順に取り出します

## たとえば

〇〇小学校の5年1組には現在40人の生徒がいます。生徒は生年月日の順番に出席番号が付けられていて、出席番号順にクラスの名簿が作られています。先生が生徒の出席を取るときなどは、この名簿を1番から40番まで読み上げ、生徒全員の出欠状況を確認します。

しかしある日いつものように先生が40人の生徒の出席を取り終わり教室を出ようとしたら、ある生徒が「先生！私、名前呼ばれていません！」先生は1人の生徒の名前を飛ばしてしまったようです…。

このようなミスをなくすために、先生は「自動出席読取機」を買いました。その機械に生徒の情報として5年1組クラス名簿を入れると、その名簿にある生徒の名前を順に表示してくれるという機械です。機械には「次」ボタンがついており、ボタンを押すたびに1番から順に生徒の名前が表示されていきます。そして先生は、表示された名前を読み上げます。

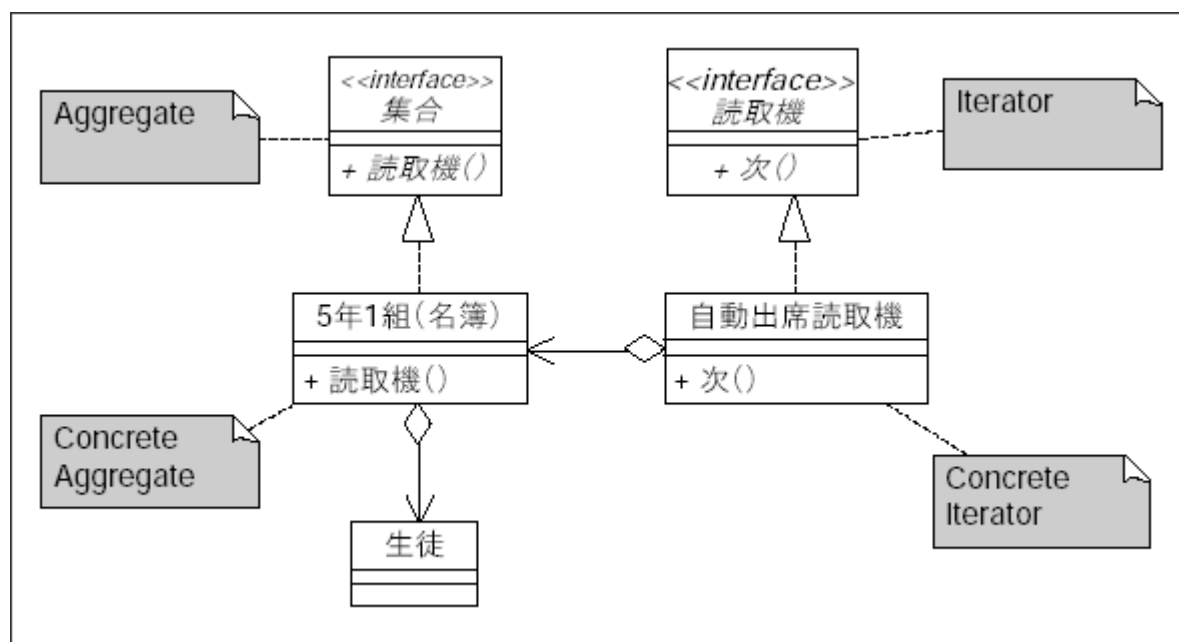
「自動出席読取機」があると、生徒の名前が全員分順番に表示されることが保証されているので、もう生徒の名前を呼び逃がすということも起きません。

## パターンの解説

Iterator パターンとは、集合の内部構造を利用者に見せずに、その集合に順番にアクセスする（走査する）方法を提供するパターンです。

実際に走査する方法を提供するものを ConcreteIterator といい、現在集合のどの部分を走査しているかという情報を保持しています。たとえば話の中では「自動出席読取機」がこの ConcreteIterator にあたります。またこの例には出てきませんでしたが、集合の要素を走査するためのインタフェースを定義したものが、Iterator（読取機）といいます。走査する集合を ConcreteAggregate といい、自分自身の集合の走査方法を提供する ConcreteIterator を生成して返します。たとえば話では、5年1組（名簿）がこの ConcreteAggregate にあたります。ConcreteIterator である「自動出席読取機」を生成して返すのですが、ここでは「先生に買ってもらう」というのがその方法です。そして ConcreteAggregate のインタフェースは、Aggregate（集合）と呼ばれます。

図 13 生徒名簿取得 (Iterator) のクラス図



このパターンを使用すると、先生は生徒がどの順番で並んでるかの構造を意識せずに、「自動出席読取機」から簡単な操作で生徒を順番に取り出すことができます。また、あいうえお順や席順というような異なる走査方法で順番に取り出したいという場合も、専用の読取機さえあれば、先生はその違いを意識せずに簡単な方法で取り出すことができますというわけです。

### 適用例

J2SE では `java.util.Iterator` が用意されています。Collection や List などの集合から Iterator を生成することにより、利用者は集合を走査することができます。

表 13 例と GoF 本の対応 (Iterator)

登場人物	GoF 名称
自動出席読取機	ConcreteIterator (具体的な反復子)
読取機	Iterator (反復子)
5年1組 (名簿)	ConcreteAggregate (具体的な集合体)
集合	Aggregate (集合体)
先生	Client (利用者)

## Mediator

### 孔明がすべての部隊の面倒を見ます

#### たとえば

蜀の名軍師、諸葛亮孔明は、他国との戦いである計略を考えました。その計略は以下のとおりです。

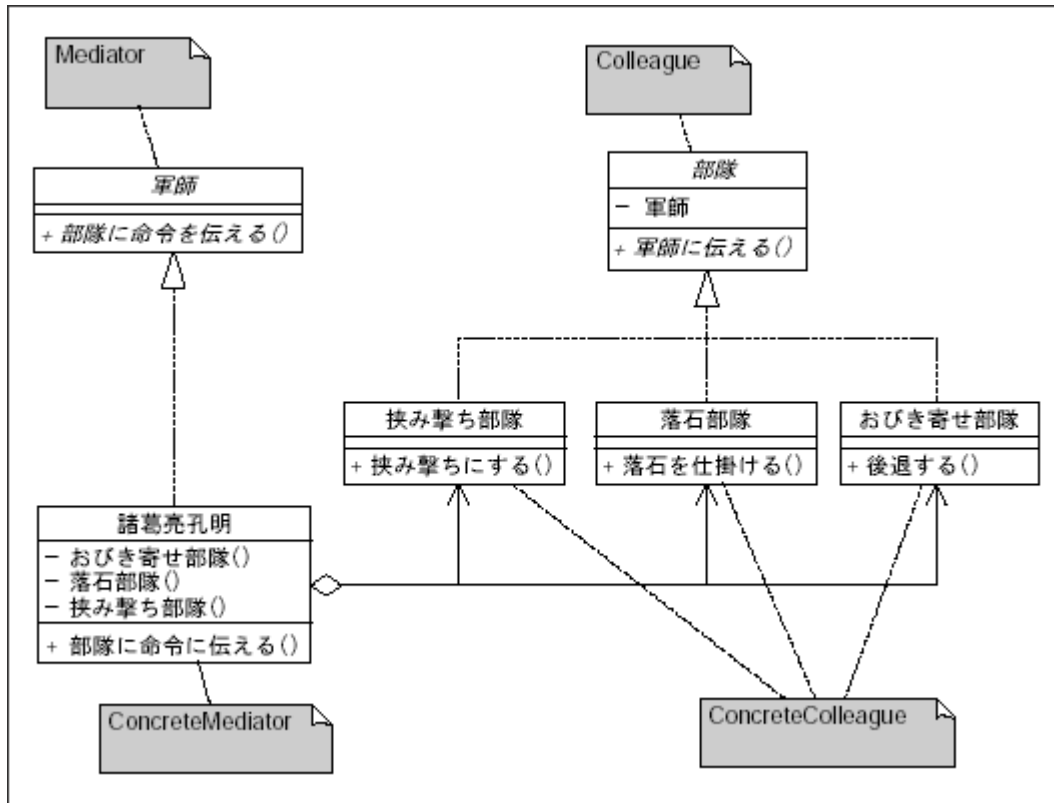
部隊を3つに分け、まず1つめの部隊が敵の本体と遭遇したら戦いを挑み、ある程度戦ったら負けたふりをして後退します。敵は、ここぞとばかりに追撃してくるでしょう。1つめの部隊はただひたすら後退し、敵部隊をあるポイントまでおびきよせます。敵がそのポイントまで来たら、2つめの部隊は崖の上から大石や大木を敵部隊めがけて落とします。敵はあわてて退却するでしょう。それを見計らって、後退していた1つめの部隊は引き返して再び攻撃をしかけます。そして、3つめの部隊が敵の退路から敵部隊に攻撃をかけ、挟み撃ちにします。この計略の成功を握るカギは、各部隊がいかに他部隊の動きを把握しているかです。もし、各部隊が他部隊の動きを把握していなかったとしたら、大石や大木を味方の部隊に落としてしまうことにもなりかねませんし、落石をしかける前に1つめの部隊が後退をやめて攻撃を開始してしまうかもしれません。

孔明はそういったミスを防ぐために、各部隊はすべての状況の変化を孔明に報告させることにしました。孔明は各々の状況を逐次判断して各部隊に命令を出すのです。そうすることにより、各部隊は他部隊が存在していることすら知らなくても確実に任務を遂行することが可能になります。なぜなら、部隊はすべての状況を把握している孔明の命令だけを聞けばよいからです。

#### パターンの解説

Mediator パターンは、オブジェクト同士がお互いを明示的に参照し合うことがないようにして、結合度を低めることを促進します。これにより、オブジェクトの相互作用を独立に変えることができるようになります。オブジェクト指向設計では、オブジェクト間に振る舞いを分配することを積極的に検討します。しかし、ときにはそれが災いして、オブジェクト間に多くの関連を生み出すことになり、各オブジェクトが他の大部分のオブジェクトを知らなければならなくなることもあります。

図 14 孔明と各部隊 (Mediator) のクラス図



この例では、Colleague を継承した ConcreteColleague である各部隊は、Mediator を継承した ConcreteMediator (孔明) だけを知っていればよいので他部隊のことを認識する必要がなく、たとえ異なる任務を遂行する部隊が追加されたとしても、彼らは頭を悩ませなくともいいのです。

**適用例**

J2SE では javax.swing.FocusManager など使われています。

表 14 例と GoF 本の対応 (Mediator)

登場人物	GoF 名称
軍師インタフェース	Mediator (調停者, 仲介者)
諸葛孔明	ConcreteMediator (具体的な調停者, 仲介者)
部隊インタフェース	Colleague (同僚)
おびき寄せ部隊	ConcreteColleague (具体的な同僚)
落石部隊	
挟み撃ち部隊	

**Memento**

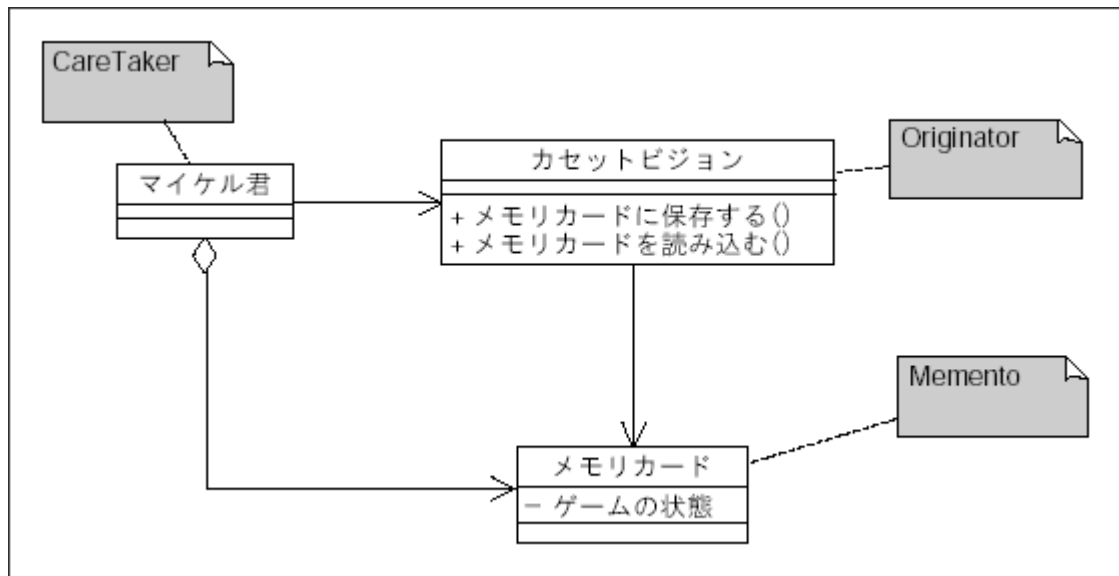
メモカードに状態を保存しておけば、途中からゲームができます

**たとえば**

テレビゲームが大好きな小学生マイケル君は、クリスマスに買ってもらった最新ゲーム機「カセットビジョン」で今日も楽しくゲームで遊んでいます。でも、そんなマイケル君には悩みがありました。ゲームを何時間もやっているとお母さんに「ゲームばかりしていないで勉強なさい！」と口うるさく怒られるし、なにより、マイケル君が心から尊敬しているゲーム名人の桜田名人が「ゲームは1日1時間、僕らの仕事はもちろん勉強、成績上がればゲームも楽しい、外で遊ぼう元気よく、僕らは未来の社会人」といつもテレビで言っているからです。

でも、きまって1番いいところで1時間経ってしまい、泣く泣く電源を切るはめになってしまいます。1度電源を切ると、再開するときはまた1番最初からになってしまいます。

図 15 メモリカードとテレビゲーム (Memento) のクラス図



マイケル君は、ゲームを前回電源を切ったところからまた始められたらいいなあと思いました。そうすれば、時間を気にすることなく、ゲームをすることができます。しかしそこは、さすがは最新機種カセットビジョン。外付けメモリカードにゲームの途中の状態を保存することができ、メモリカードに保存さえしておけば、電源を切ってしまうても、次は前回保存した状態からゲームを再開できるので...でも別売り。貯金していたおこずかいでメモリカードを買ったマイケル君。これで、お母さんにゲームのやりすぎで叱られることもないし、桜田名人との約束も守ることができますね。

### パターンの解説

Mementoパターンは、インスタンスの内部状態を外部に保持しておき、インスタンスをいつでもこの状態に戻すことができるようにします。Mementoには、「思い出させるもの」「記念品」「形見」「忘れ形見」といった直訳があり、Originatorの内部状態を保持する役割を持っています。(マイケル君)なのですが、基本的にはCareTakerはMementoの中身を知ることはできません。Mementoの中身を知り自由にアクセスできるのは、Originator(カセットビジョン)だけです。

この例では、「メモリカード」がMementoになりますね。このMementoを保持しているのがCareTaker。OriginatorはMementoを作ること、渡されたMementoの状態に自分を戻すことを役割としています。

### 適用例

このパターンは、アプリケーションにアンドゥ(やり直し)やリドゥ(再実行)の機能を持たせるときによく使われています。

J2SEのjavax.swing.undoパッケージではMementoパターンが用いられています。

表 15 例とGoF本の対応(Memento)

登場人物	GoF名称
メモリカード	Memento (記念品)
カセットビジョン	Originator (作成者)
マイケル君	CareTaker (世話をする人)

Observer

日経平均株価の変動を顧客全員へ知らせます

たとえば

A君は某弱小証券会社の新入社員です。デイリーワークとして、日経平均が前日終値の±200円を超えると、先輩から渡された顧客リストをもとに、各々の顧客に株価一覧をFAXするという仕事を与えられていました。顧客リストの追加/削除などのメンテナンスは先輩が随時行います。顧客には個人デイトレーダー、法人企業、新聞記者、大学生、大学教授もいます。A君は全員に同じようにFAXしますが、顧客がそのFAXを何に利用しているのかわかりません。A君はただその時刻の株価を調べ、リストをもとに全員にFAXするだけです。もし顧客への通知方法が顧客ごとに異なっていたらどうでしょうか。ある顧客には電話、ある顧客にはEメール、そして別の顧客には訪問して口頭で伝える...それだけでA君の1日は終わってしまいそうです。

ある日もう1人の新人B君が入社しました。当然彼にもA君と同じような仕事を与えられました。しかし、顧客リストをもとに顧客へFAXするというところまでは同じですが、日経平均株価ではなく円相場でした。顧客リストもA君とは異なるようです。

しかしよくみると、A君の仕事とB君の仕事には共通部分が多く、整理してマニュアル化しようということになりました。そのマニュアルには顧客への

FAX方法、顧客リストのメンテナンスが記載されています。

パターンの解説

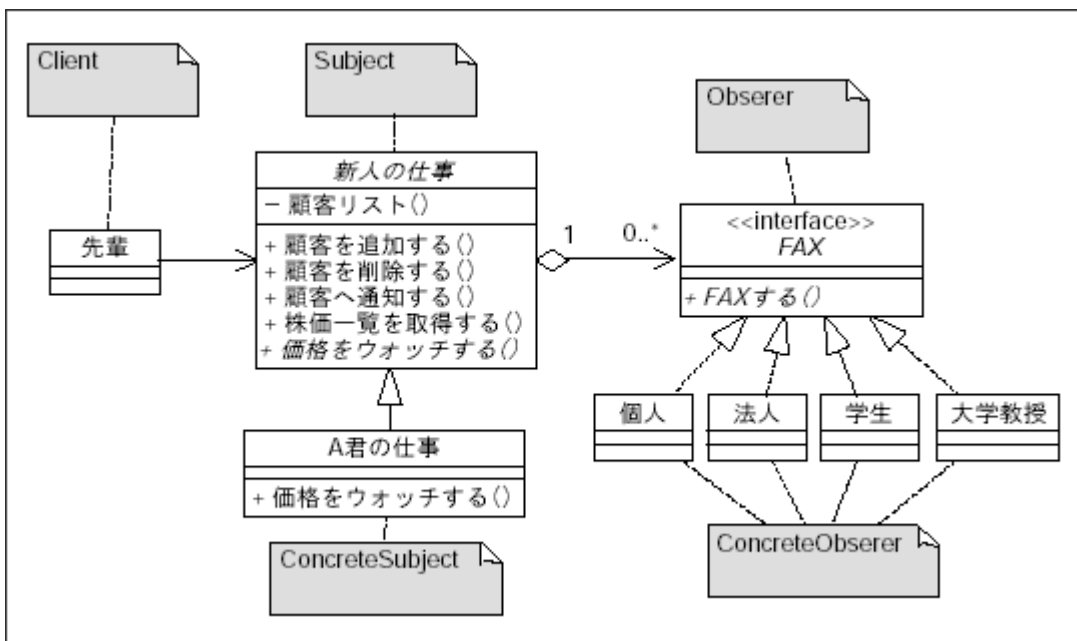
マニュアルとしてまとめた「新人の仕事」自体は仕事をしませんので、インスタンス化できない抽象クラス Subject となり、具体的に仕事をする人はそれを継承した具象クラスの「A君の仕事」 ConcreteSubject となります。

顧客リストは ConcreteSubject のインスタンス変数です。顧客が増えたときには顧客クラスのインスタンスを「顧客を追加するメソッド」で追加します。そして顧客へ通知するメソッドで顧客リストに登録されている顧客全員に株価一覧をFAXします。

ここで ConcreteSubject と ConcreteObserver との関係が1対多になっていることに注意してください。ConcreteObserver が持っている情報(日経平均株価)が変化するとき、ConcreteObserver は全員に対して同じように通知します。ConcreteSubject は通知を受け取り、何が行動をします。

ConcreteSubject は ConcreteObserver がどういう行動をとるのか感知しません。これが Observer パターンの特徴です。

図 16 株価情報伝達 (Observer) のクラス図



**適用例**

J2SE では Observer パターン用のクラスがすでに用意されています。Subject には `java.util.Observable` , Observer には `java.util.Observer` が対応しています。

表 16 例と GoF 本の対応 (Observer)

登場人物	GoF 名称
新人の仕事	Subject (被験者)
A 君の仕事	ConcreteSubject (具体的な被験者)
FAX	Observer (観察者)
顧客 (個人, 法人, 学生, 大学教授)	ConcreteObserver (具体的な観察者)
先輩	Client (依頼者)

**Prototype**

**好きな曲を集めた MD を、欲しい人にコピーしてあげる**

**たとえば**

No Music, No Life。俺の人生には音楽は欠かせない。今夜はアイツと湘南の海までドライブ。隣にアイツを乗せてハイウェイで風になる…。アイツのハートががちり GET するには俺の魂が入った My Best Selection, これしかないぜ。

車内にイカス BGM, 窓の外には湘南のさわやかな風と波の音。ロマンチックな雰囲気になるには最高の状況。

「この曲いい曲ね」

「ああ、これはスタン・ゲッツの BODY AND SOUL っていう曲さ。『身も心も捧げているのに、どうして君はわかってくれないの?』っていう意味なんだ。この MD は君をイメージして作ったんだ」

「この MD 今度貸して」

「いいよ。今日家に来なよ。コピーしてあげるから」

「うん」

湘南の夜は今日もアツイ。

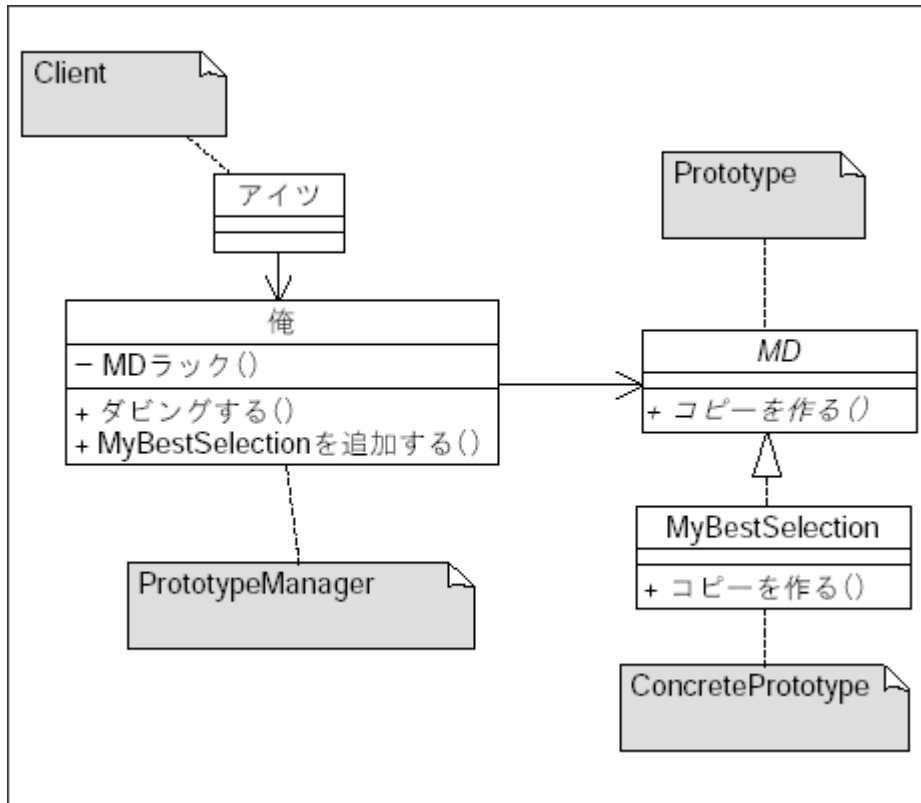
**パターンの解説**

世の中の無限と思えるほどの数の曲から、「俺」が作成する可能性のある MD をあらかじめクラスとして定義することは不可能です。もしクラスで定義できるのであれば、「アイツ」は「俺」にコピーの依頼などせず、クラスから自分でインスタンス化するでしょう (アイツは欲しい CD を CD ショップで購入するでしょう)。しかし今回の例では「アイツ」が欲しいのは「俺」がインスタンス化した「My Best Selection」であって、これは CD ショップでは購入できません。このようなとき Prototype パターンが有効になります。

Prototype インタフェース (MD) は Clone メソッド (コピーを作る) を定義していて、ConcreteProduct (MyBestSelection) はその Clone メソッドを実装しています。そのため、MyBestSelection のインスタンスはコピーを作成することができます。

PrototypeManager は ConcreteProduct を生成すると、それを自分自身のインスタンス変数 (MD ラック) に保持します。

図 17 オリジナル MD (Prototype) のクラス図



Client からの要求があるとそこから該当のインスタンスを取り出し、コピーをして返します。

**適用例**

JavaBeans の生成は Prototype パターンと言えます。また、グラフィックエディタのカットアンドペーストも Prototype パターンを使って実装できるかもしれません。ユーザが GUI で選択したオブジェクトを登録し (Ctrl+C でコピーし)、それを再利用する (Ctrl+V で貼り付け) とき、そのオブジェクトを 1 から生成するよりも、コピーして再利用するほうが簡単でしょう。

このように生成されたインスタンスを原型として保持しておき、Client からの要求があったときに随時コピーして渡す、これが Prototype パターンです。

表 17 例と GoF 本の対応 (Prototype)

登場人物	GoF 名称
MD	Prototype (雛型)
MyBestSelection	ConcretePrototype (具体的な雛型)
アイト	Client (依頼者)
俺	PrototypeManager (管理者)

Proxy

あいちゃんの影武者を立てます

たとえば

○×プロダクションの売れっ子モデルの「あいちゃん」には、毎日さまざまな雑誌、テレビ局、ラジオ局などから出演交渉が来ます。しかし、すべての仕事を受けていたら体がいくつあっても足りません。その対処法として、あいちゃんにそっくりな影武者を立てることにしました。

まず、影武者あいちゃんが依頼人から仕事の出演交渉を受けます。そして仕事についての詳細な情報を聞き、場所や時間などをスケジュールリングします。その後、仕事内容が「写真撮影」や「雑誌のインタビュー」など、本物のあいちゃんでも影武者あいちゃんでも十分受けられる仕事であったらなら、そのまま影武者あいちゃんが仕事をします。しかし、もし仕事内容が「TV出演」「ラジオ出演」「レコーディング」など、本物のあいちゃんでもいけないものであったら、あいちゃん登場！です。ここで本物のあいちゃんを呼び、それらの仕事をしてもらいます。

このような対策を取れば、受けた依頼のすべてを本物のあいちゃんがしなくても済み、いざというときにあいちゃんを呼んで仕事をしてもらえばよいので、大変効率的といえます。

パターンの解説

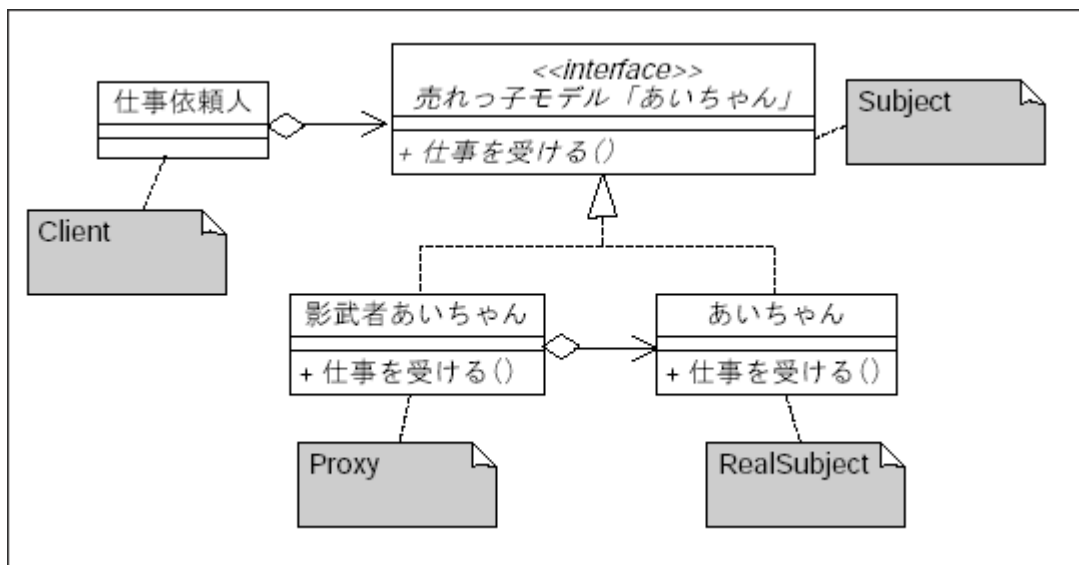
Proxy パターンとは、ある仕事を代理で受け、いざ必要というときに本人を呼び出して処理を行うというパターンです。

代理人として仕事を受けるのが Proxy です。代理人ではできない仕事を担当するのが実体である RealSubject です。たとえ話で言うと、代理人である Proxy が影武者あいちゃん、仕事を行う本人である RealSubject があいちゃんです。そして仕事を依頼するのが Client、仕事依頼人です。

また、Client (仕事依頼人) から見ると Proxy (影武者) も RealSubject (あいちゃん) も、仕事を依頼する人には変わりはありません。これを意識させないようにするのが Subject で、Proxy と RealSubject のインタフェースを定義したものです。たとえ話では、売れっ子モデルの「あいちゃん」です。

Proxy パターンを利用すると、要求があるたびに実体である RealSubject を生成しなくてもよくなります。一旦要求を Proxy が受け、いざ実際に必要となったときに RealSubject を生成すればいいのです。そうすることにより、生成のコストを抑えることができます。

図 18 あいちゃんの影武者 (Proxy) のクラス図



**適用例**

Java では RMI や EJB で Proxy パターンが適用されています。

表 18 例と GoF 本の対応 (Proxy)

登場人物	GoF 名称
影武者あいちゃん	Proxy (代理人)
あいちゃん	RealSubject (実際の主体)
売れっ子モデル「あいちゃん」	Subject (主体)
仕事依頼人	Client (依頼人)

**Singleton****アメリカ合衆国に大統領は 1 人****たとえば**

複数存在すると困るものって、世の中にたくさんあると思いませんか？

国王は 1 つの国に 2 人以上いたら争いの種となってしまうし、1 つの会社に社長が複数人いたら、どちらの経営方針に従ってよいかわからず、従業員は戸惑うことになります。彼氏や彼女も何人もいたら後で厄介なことになりますね。

アメリカ合衆国に大統領は、1 人しか存在しません。これはアメリカ合衆国の憲法や法律などの取り決めによります。

このような取り決めをクラス自体に持たせ、生成されるインスタンスの数を制限するのが Singleton パターンです。

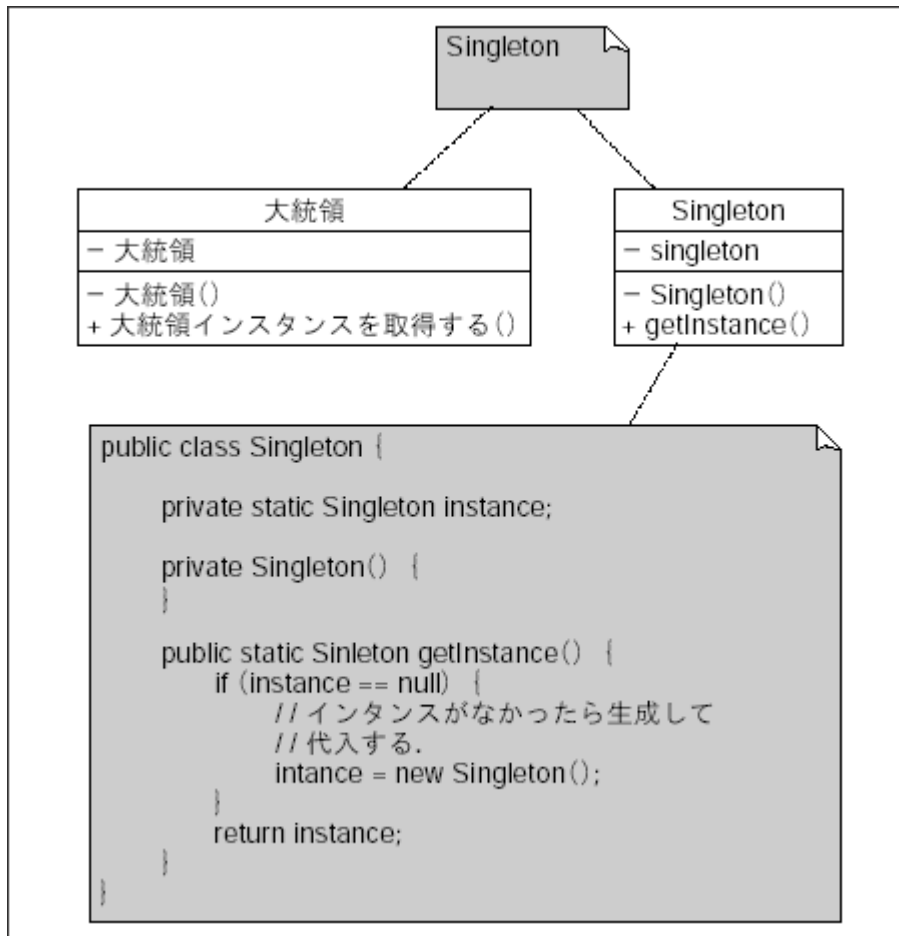
**パターンの解説**

Singleton クラスは、インスタンス化要求を制御してインスタンスが 1 つしか作られないことを保証し、またインスタンスにアクセスするための一般的な方法を提供します。このパターンのミソは、クラス自体が自分のインスタンスを管理する役割を持っていることです。クラス図の大統領クラスは大統領インスタンスを保持するための変数がありますね。

Singleton クラスのコンストラクタは private 宣言しておき、他のクラスからのインスタンス化要求を受け付けないようにします。大統領クラスのコンストラクタである「大統領()」も private 宣言されていて、他のクラスから「new 大統領()」なんてすることはできません。また、Singleton クラスのインスタンスを得るには public 宣言してある static メソッド getInstance を呼びます。これが Singleton クラスのインスタンスにアクセスするために提供された唯一の方法です。大統領クラスでは「大統領インスタンスを取得する」メソッドがそれにあたります。

getInstance メソッドが呼ばれると、Singleton クラスは自分自身のインスタンスを保持する static 変数をチェックし、自分のインスタンスがあればそれを返却し、そうでなければ生成してから static 変数に代入した上で返します。

図 19 大統領 (Singleton) のクラス図



大統領クラスでは大統領インスタンスを取得するメソッドが呼ばれると,static 変数である大統領をチェックし,大統領がいればそれを返し,大統領がいなければ「new 大統領()」で,大統領のインスタンスを作り,大統領変数に格納した後返します。

**適用例**

アプリケーションの世界では,印刷ジョブを管理するプリンタスプーラやファイルシステム,ウィンドウマネージャ,デバイスドライバなどが Singleton パターンで設計されることが多いようです。

J2SE では java.lang.Runtime クラスで,Runtime オブジェクトの取得に Singleton パターンを適用しています。

表 19 例と GoF 本の対応 (Singleton)

登場人物	GoF 名称
大統領, 国王, 社長, 彼氏 or 彼女	Singleton (一枚札)

**State**

**エスカレータの状態にあわせて動作を変えます**

**たとえば**

Xビルサービスのエスカレータは,そのときどきの状態によって動作を変えることができます。エスカレータの状態は管理センターで随時管理しており,状態としては「運転状態」(エスカレータが動いている)と「停止状態」(エスカレータが停止している)があります。

エスカレータの開始地点にはセンサーが付いていて,センサーの前を人が通過すると管理センターに信号が届くようになっています。また管理センターには停止スイッチがついていて,エスカレータを停止することができます。このエスカレータの状態を状態チャート図で表すと図 20-1 のようになります。A~Dの動作を繰り返してエスカレータは毎日動いています。

つまり、管理センターに「センサー反応」や「スイッチ切」という命令が届くと、「運転状態」「停止状態」という各状態にあわせてエスカレータの動作が変わるわけです。このように複雑な状態遷移を容易に表現するものとして State パターンがあります。

**パターンの解説**

State パターンとは、ときどきの状態にあわせて動作の内容を変えることができるパターンです。

状態を表すクラスを State といい、状態における動作のインタフェースを定義します。そして具体的な状態を表すクラスを ConcreteState といい、State で定義された動作の内容を実装します。先のたとえば、State がエスカレータで、ConcreteState が「運転状態」「停止状態」という各状態です。その

中で定義された動作が「センサー反応」や「スイッチ切」などのメソッドです。

また、状態に対して動作を呼ぶ役目が Context で、自分の動作である ConcreteState を保持しています。たとえば話でいうと、Context が管理センターで、そのときの状態を保持しています。

あるものの状態をクラスで表現し、各状態の動作をカプセル化するのが State パターンです。Context である管理センターに振る舞いを記述し、「運転状態」だったらこうで、「停止状態」だったらこうで...と条件を分岐させて複雑なコードを書かなくても、ConcreteState に動作を記述して Context で保持しておけば、管理も保守も簡単になります。

図 20-1 エスカレータ (State) のステート図

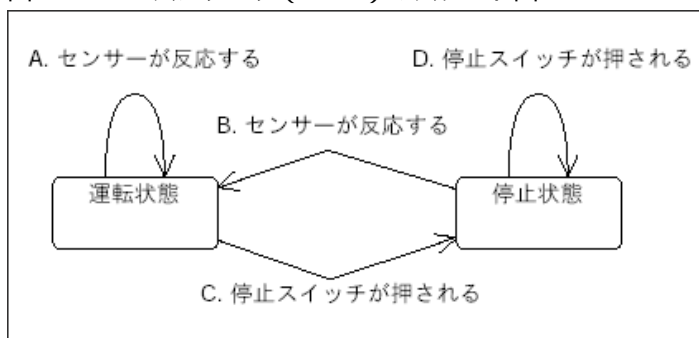
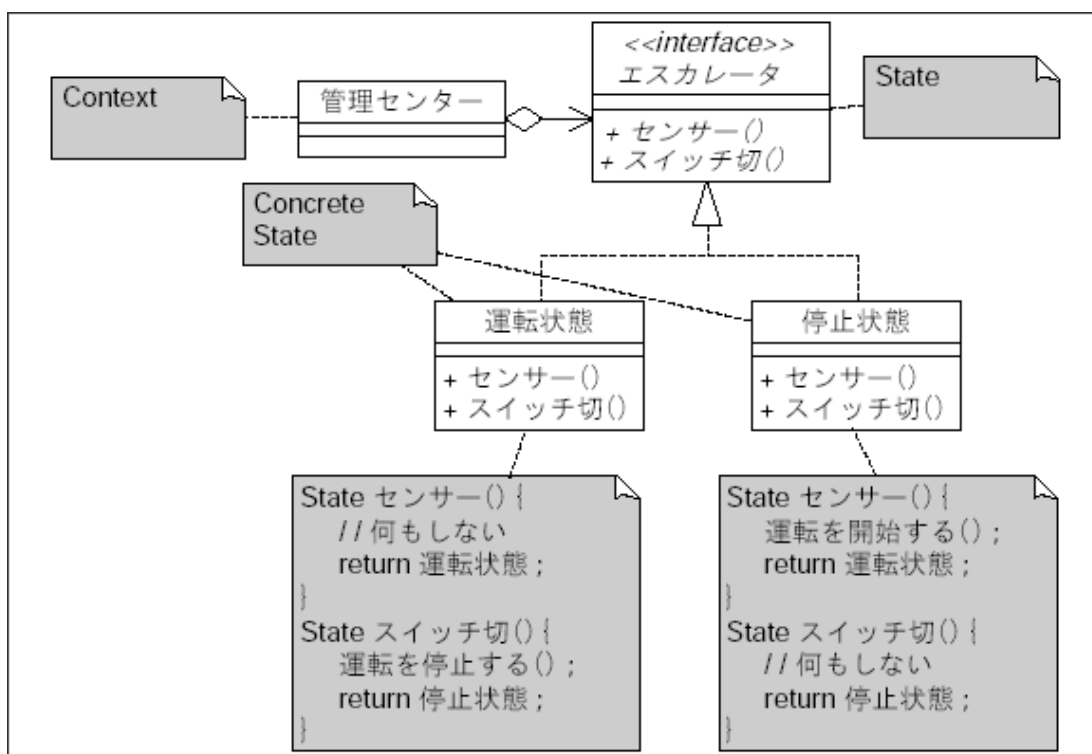


図 20-2 エスカレータ (State) のクラス図



**適用例**

JavaAPI では利用されていないようですが、さまざまな状態に合わせた処理などに応用できます。

表 20 例と GoF 本の対応 (State)

登場人物	GoF 名称
エスカレーター	State (状態)
運転状態, 停止状態	ConcreteState (具体的な状態)
管理センター	Context (状況)

**Strategy**

**状況に応じてお金を手に入れる戦略を選びます**

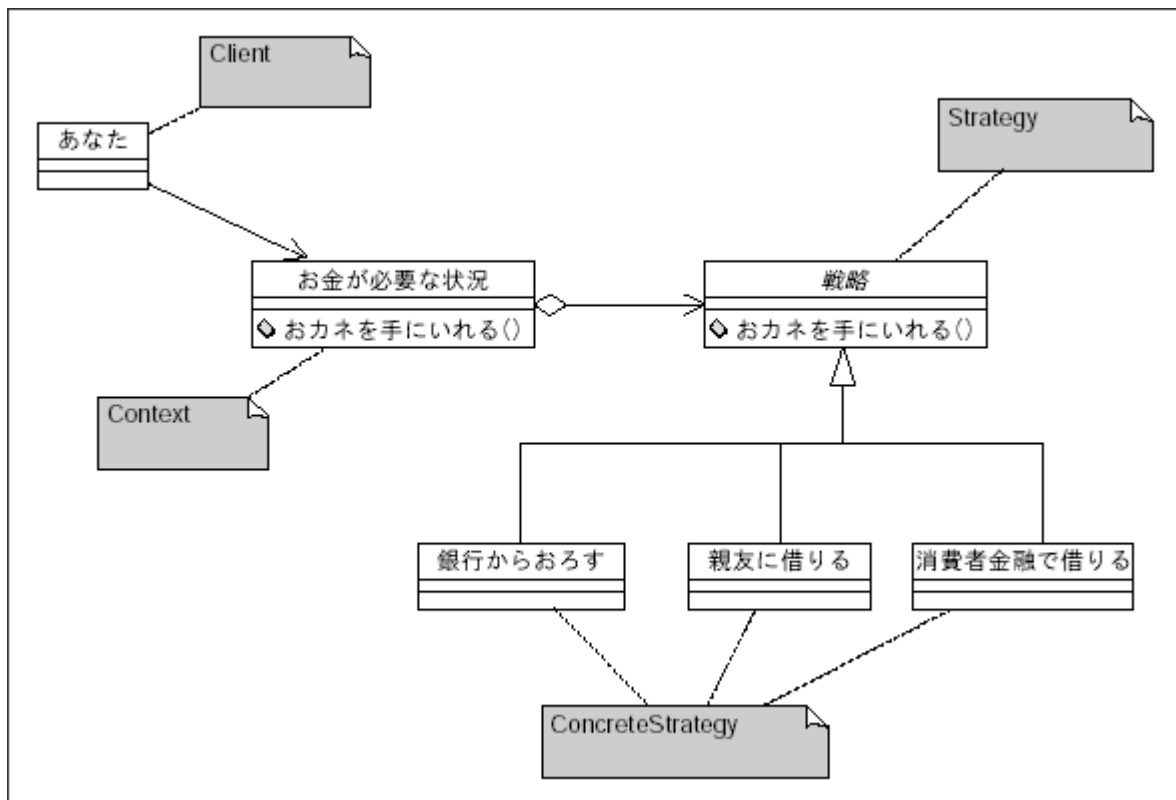
**たとえば**

「あっ！そう言えば、今日デートの約束だっけ。フランス料理を食べにいくんだよな…」とあなたは不安そうに自分の財布を開きます。しかし案の定、あなたの財布には千円札が数枚見えるだけ。今日は給料日の前日です。確か銀行の預金は数千円しかないはず。

「げげっ！そうだ、Aに借りよう…」と親友である(はずの)A君に相談します。「は？おめえ何言ってた。それより、これまでに貸した10万返してくれよ！」と、逆に詰めよられるはめに。

トボトボ駅前に向かいました。すると道端のお姉さんが「よろしく願います～」とポケットティッシュを1つくれました。で、何気なくティッシュの裏を見ると、そこには「アコモ」という文字が。「そうだぁ！この手があったか」とあなたは消費者金融のATMに向かって走り出しました…。

図 21 お金を手に入れる戦略 (Strategy) クラス図



このような共通の目的（お金を手にいれる）に対して、それを実現する戦略を複数用意しておき、その場の状況に応じて適切な戦略を選ぶというデザインパターンに、Strategy パターンがあります。

### パターンの解説

たとえ話では、あなたの目的（＝解決すべき問題）は「お金を手に入れる」ことでした。また、この目的のための具体的な戦略としては「銀行の預金からお金をおろす」「友達から借金する」「消費者金融の ATM でキャッシングする」の 3 つがありました。Strategy パターンではまず、この「お金を手に入れる」という目的を、Strategy インタフェースのメソッドとして定義します。その上で、上記の具体的な戦略を、Strategy インタフェースのサブクラスである ConcreteStrategy クラスに個別に実装します。Client（あなた）は、状況を判断して適切な ConcreteStrategy（「銀行から下ろす」や「親友に借りる」）をインスタンス化し、Context クラス（「お金が必要な状況」）を通してその戦略を実行します。ちなみに、Context クラスは ConcreteStrategy に処理を委譲する形で結びついています。

このようにすると、新たなお金を手に入れる戦略（「親から借りる」など）が追加されても、ConcreteStrategy クラスを 1 つ追加すれば対応できます。さらに、Client は Context が委譲している ConcreteStrategy のインスタンスを動的に入れ替えることで、たとえ話のようにその場の状況に応じてつぎつぎと戦略を切り替えるということも実現できます。

### 適用例

J2SE では、java.awt.LayoutManager クラスで Strategy パターンを適用しています。

表 21 例と GoF 本の対応（Strategy）

登場人物	GoF 名称
お金が必要な状況	Context（文脈）
戦略	Strategy（戦略）
銀行からおろす	ConcreteStrategy（具体的な戦略）
親友に借りる	
消費者金融で借りる	

### Template Method

#### ハンバーガに具をはさむ手順を個別に定義します

#### たとえば

ハンバーガショップ M ではハンバーガの種類ごとに調理手順書があり、チーズバーガと BLT バーガにはそれぞれ別の調理手順書が使用されていました。店長 A さんは、新商品が登場するたびにバーガ調理手順書を作成していました。しかし、毎回新しく作成するのは時間がかかって面倒です。そんなある日店長 A さんは、チーズバーガと BLT バーガの調理手順書を見比べて、ほとんど同じ手順だということがわかったのです。

ハンバーガを調理するにはいくつかの工程がありますが、その中でハンバーグをこねたり焼いたりという工程の手順はどんなハンバーガでも変わらなかったのです。ただ 1 点、パンに具をはさむ工程の手順だけがハンバーガごとに変わることがわかりました。

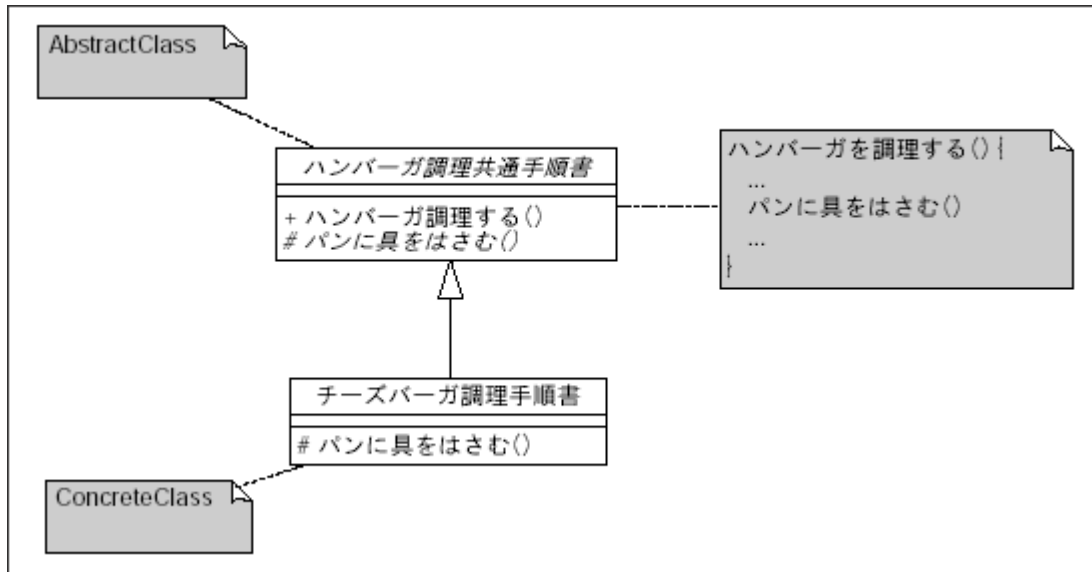
A さんはいろいろと考えた末に、ハンバーガの調理手順書を共通化しました。これがハンバーガ調理共通手順書です。この手順書を参照してすべてのハンバーガを調理します。しかしパンに具をはさむ工程の手順だけはハンバーガごとに異なるため、その箇所には「パンに具をはさむ工程は各ハンバーガの手順書を参照せよ」と注意書きしておきました。こうしてチーズバーガ調理手順書や BLT バーガ調理手順書には、パンに具をはさむ工程の手順だけを詳細に書いたのです。

### パターンの解説

ハンバーガ調理共通手順書は、AbstractClass です。AbstractClass ではテンプレートメソッドを実装します。たとえ話では「ハンバーガを調理する」があてはまります。また AbstractClass では、ConcreteClass で実装する必要がある抽象メソッドを定義します。これは「パンに具をはさむ」です。チーズバーガ調理手順書は、ConcreteClass です。AbstractClass で定義された抽象メソッド、つまり「パンに具をはさむ」を実装します。

TemplateMethod パターンはアルゴリズムの不変な部分をスーパークラスで実装し、変わりうる部分をサブクラスで実装するパターンです。

図 22 ハンバーガー調理 (Template Method) のクラス図



この例では、それぞれのサブクラス(チーズバーガー調理手順書や BLT バーガー調理手順書)で共通となる部分(ハンバーガーをこねる, ハンバーガーを焼くなど)を抜き出し、スーパークラス(ハンバーガー調理共通手順書)に実装しました。

その上で、個別な部分(パンに具をはさむ)をサブクラスに記述できるようにしました。

TemplateMethod パターンを使用することにより、ハンバーガー調理手順を効果的に再利用できました。

**適用例**

J2EE では、javax.servlet.http.HttpServlet クラスが TemplateMethod パターンを適用しています。また、第 3 章の事例でも取り上げていますので参照ください。

表 22 例と GoF 本の対応 (Template Method)

登場人物	GoF 名称
ハンバーガー調理共通手順書	AbstractClass (抽象クラス)
チーズバーガー調理手順書, BLT バーガー調理手順書	ConcreteClass (具象クラス)

**Visitor**

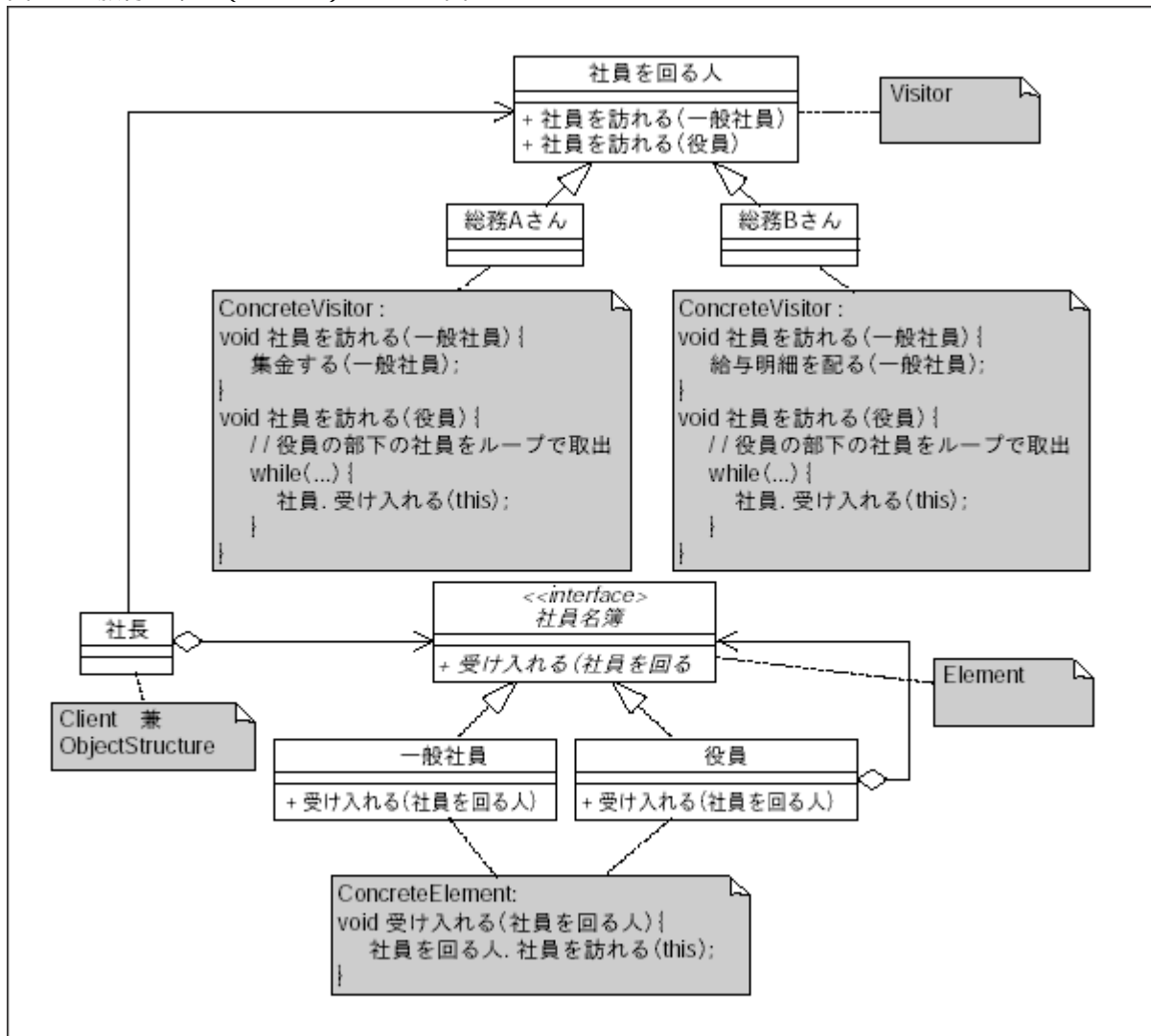
**会社の全社員を回って旅行の集金をします**

**たとえば**

ある会社の総務の A さんは社員旅行の集金をする係なので、毎月社員全員の席を回ってお金を集めています。社員とは、社長からツリー構造で成り立っている社員名簿の全員を指します。

一方 B さんは、給与明細を配って印鑑をもらう仕事をしています。しかし A さんも B さんも、どのような社員がいるかということは気にしていません。社長が持つツリー構造の社員名簿通りに、社員を回るだけです。社員の増減や人事異動があったとしても社員名簿が変わるだけで、A さんと B さんの動作は変わりません。また、保険証を全員に配って回る仕事をする C さんが加わっても、社員名簿の中身を理解しなくても OK です。

図 23 旅行の集金 (Visitor) のクラス図



**パターンの解説**

Visitor パターンとは、操作する対象の構造と実際の操作を分離するパターンです。ある操作をするために対象の構造の中を渡り歩くものを Visitor (社員を回る人) といい、その操作である「(社員を) 訪れる」を宣言します。

Visitor を実装したものを ConcreteVisitor (A さんや B さん) といい、操作 (社員を訪れる) の処理の中身を実装します。また、Visitor が訪れる構造の要素を Element (社員名簿) といい、Visitor を受け入れるためのインタフェースを定義します。Element を実装したものは ConcreteElement (一般社員と役員)、Element の構造を操作できるものが ObjectStructure (社長)、操作の依頼者が Client (社長) です。

Client (社長) は社員名簿のツリー構造を保持しており、社員名簿に対する操作を A さんや B さんに依頼します (Visitor の受け入れ)。

ConcreteElement の Visitor を受け入れるための操作は、引数に Visitor を受け取り、受け取った Visitor に対して操作 (訪問する) を呼び出します。そして、Visitor 中の操作 (訪問する) は引数に ConcreteElement を受け取り、受け取った ConcreteElement に対して実際の処理を行います。図 23 のように (Java などの) this で自分自身を引数で渡し、お互いがお互いを呼び出して処理を行います。

呼び出しの構造はとても複雑ですが、Element と Visitor がお互いを呼び出すことにより、構造の部分を Element へ、操作の部分を Visitor へ分離することができます。

こうすることにより、新しい操作 (Visitor) を追加したり、変更したりすることが容易になります。

### 適用例

JavaAPI では使用されていないようですが、さまざまな場面で利用できます。

表 23 例と GoF 本の対応 (Visitor)

登場人物	GoF 名称
社員を回る人	Visitor (訪問者)
総務の A さん, B さん	ConcreteVisitor (具体的訪問者)
社員	Element (受け入れ者)
一般社員, 役員	ConcreteElement (具体的受け入れ者)
社長	Client (依頼者)
(2つの役割を兼任)	ObjectStructure (オブジェクトの構造)