POS 端末へのデザインパターン適用

大分県情報サービス産業協会
デザインパターン部会

「メンバ紹介」

部会長	佐藤　博治	大銀コンピュータサービス（株） 業務課
副部会長	進藤　朝春	九州東芝エンジニアリング（株） 半導体＆液晶システムグループ
	河野　政士	九州東芝エンジニアリング（株） 情報・技術システムグループ
	石崎　憲生	大分シーイーシー（株） 第１システム部
	山口　真吾	太平工業（株）大分支店 電気計装部　IT ソリューショングループ
	小林　大悟	新日鉄ソリューションズ（株） システム第２部
	廣田　朋寛	（株）富士通大分ソフトウェアラボラトリ マルチメディアシステム開発部
	副島　正行	（株）富士通大分ソフトウェアラボラトリ マルチメディアシステム開発部
	平川　公儀	（株）オーイーシー システムサービス部
技術委員	重光　貞彦	大分シーイーシー（株）
	築城　久敏	システムトレンド（有）

「もくじ」

第１章　「はじめに」

今日、プロジェクトの大規模化、複雑化、社会情勢や適用技術の急激な変更に伴う仕様変更、それとは相反する短期納入の要請等、ソフトウェア開発の現場を取り巻く環境は厳しさを増しつつある。

こうした中、オブジェクト指向設計（OOD）やアジャイルな開発スタイルを使用したプロジェクトを数多く見かけるようになってきた。オブジェクト指向設計は、差分プログラム的性格が強く、適切な設計を行うことで、変更に柔軟に対応でき、さらに急速に進展したフレームワーク等のオブジェクト指向技術の利用は、上記問題解決の手助けとなっている。

しかし、このようなプロジェクトでは、メンバ間のコミュニケーションを高め、変更に柔軟に対応できる開発体制が以前に増して重要になっており、オブジェクト指向的な考え方に基づいて、各メンバが“設計思想”を共有し、表現することが、ソフトウェアの品質や開発の効率化に大きく影響してくる。

ＵＭＬが広まり、プロジェクト内外で、統一した表記の使用が可能となったが、このことで、共通の“設計思想”を広め、共有することが、より重要になってきているのではないだろうか。

そして、現在、コミュニケーションを円滑に行う手段として、“デザインパターン”に注目が集まっている。デザインパターンは、今まで先人が培ってきた、オブジェクト指向的な設計の定石とその適用場面をテンプレート形式でまとめた実例集である。テンプレートに従った同一観点からの議論、適切なサンプルコードは、これらの実例の理解にも役立つのではないだろうか。

我々は、この“デザインパターン”に焦点をあて、実際のソフトウェア開発の現場で、どうした状況での使用が有功か、どれほど効力を発揮するのか、問題点やトレードオフは発生しないのか、を GoF のデザインパターンを具体的な開発事例に適用する中で考察していくこととした。

本論文の概要を以下に示す。第２章では、GoF のデザインパターンについて、概説する。第３章では、一般的なソフトウェアアーキテクチャの要素（入出力、ビジネスモデル、データベースアクセス等）を含む事例として、販売時点管理システム（以降 POS システムと呼ぶ）における POS 端末を開発事例に選定し、そのシステム概要について述べる。第４章では、POS 端末の各機能へのパターンの適用とその考察に言及する。そして、第５章を、まとめとする。

第２章「デザインパターンとは」

2.1 概要

デザインパターンは、元々、建築家 Christopher.Alexander とその仲間が、建築の設計についてまとめた考え方であった。建築はそこに実際に住む人の要求を満たした形で作られなければならない。しかし、実際の作業を行っていくためには、建築に対する詳細かつ専門的な知識が必要になり、そのままでは単に理想を述べただけのものとなってしまう。そこで Alexander が提唱したのは、「パターン」の考えであった。建築の過程で生じてくる様々なトレードオフや問題、そしてそれに対する典型的な解決の方法を、誰もがわかりやすいパターンとし、カタログ化して示すことにより、設計者が自由にパターンを選択して、より要求にかなった建築の手がかりとしていくという方式を提唱した。彼らは、253 のパターンを使って都市や町や建物を設計すると、かなり快適な生活空間が得られることを発見した。

ソフトウェアも、要求をかなえるための成果物を構築しなければならないという点において、建築の世界と非常に似た部分がある。その後、ソフトウエア技術にも、この考えを取り入れようという動きが起こった。よく使われる良い手法、良い設計のパターンを記録に残しておくことで再利用を促進するというものである。その中でも、「デザインパターン」という言葉をオブジェクト指向の技術者に広く知らしめたのは、Erich Gamma, Richard Helm, Ralph Johnson, John Vilissides (**通称 Gang of Four：GoF**)らによる"Design Patterns"という著書であった。この本がオブジェクト指向設計者の間で一種のベストセラーとなり、パターン運動を活性化するきっかけとなる。出版は 1995 年だが、現在でもその実用性、有効性は不変である。GoF のデザインパターンはミクロアーキテクチャレベルでの「生成」、「構造」、「振る舞い」という分類で 23 のパターンが理路整然とカタログ化されている。

以下に、GoF のデザインパターン（23 パターン）の概要を記述する。（詳細は、付録 A を参照）

表 2.1.1 GoF デザインパターン一覧表

	分類	パターン名称	概要
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	状態に合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

第３章「システム説明」

3.1　概説

コンビニやスーパーで買い物をすると、レジでバーコードを読み込む。すると自動的に商品名や金額がレジに表示される。このタイプのレジを POS レジという。POS とは「Point of Sales」の略で販売時点管理と訳される。POS システムとは、「販売時点」の情報を活用するシステムである。ちなみにその情報とは、商品名・金額だけでなく、購買者の年齢・性別、購買日時、天気・気温・湿度から、その店舗のある地域で開催されているイベント情報までが含まれる。それら様々な情報を分析・活用することで、効率的な品揃えや、戦略的な販売促進、売上げ・在庫管理が実現できるのである。

POS システムの役割として、以下のようなものがあげられる。

・商品管理・・・・・・売上げ商品情報の分析による適切な品揃えと適正価格による商品提供の実現。また新商品の購買マーケティングシュミレーション。

・顧客情報管理・・・・顧客の購買傾向の分析および購買予測等。

・省力化・・・・・・・商品自体のハンドラベラーによる値付けをなくし、レジのデータ入力を省き、チェッカー（レジを担当する人）教育の短縮化等。

・取引情報管理・・・・現金・各種クレジットカード、プリペイドカードのトータル管理

3.2　システム構成

本 POS システムの構成は、図 3.2.1 のとおりである。

顧客１人１人の購入したもののデータが支店のレジを通して本部に送られ、それにより欠品の防止や小回りのきく品揃えが可能になる。

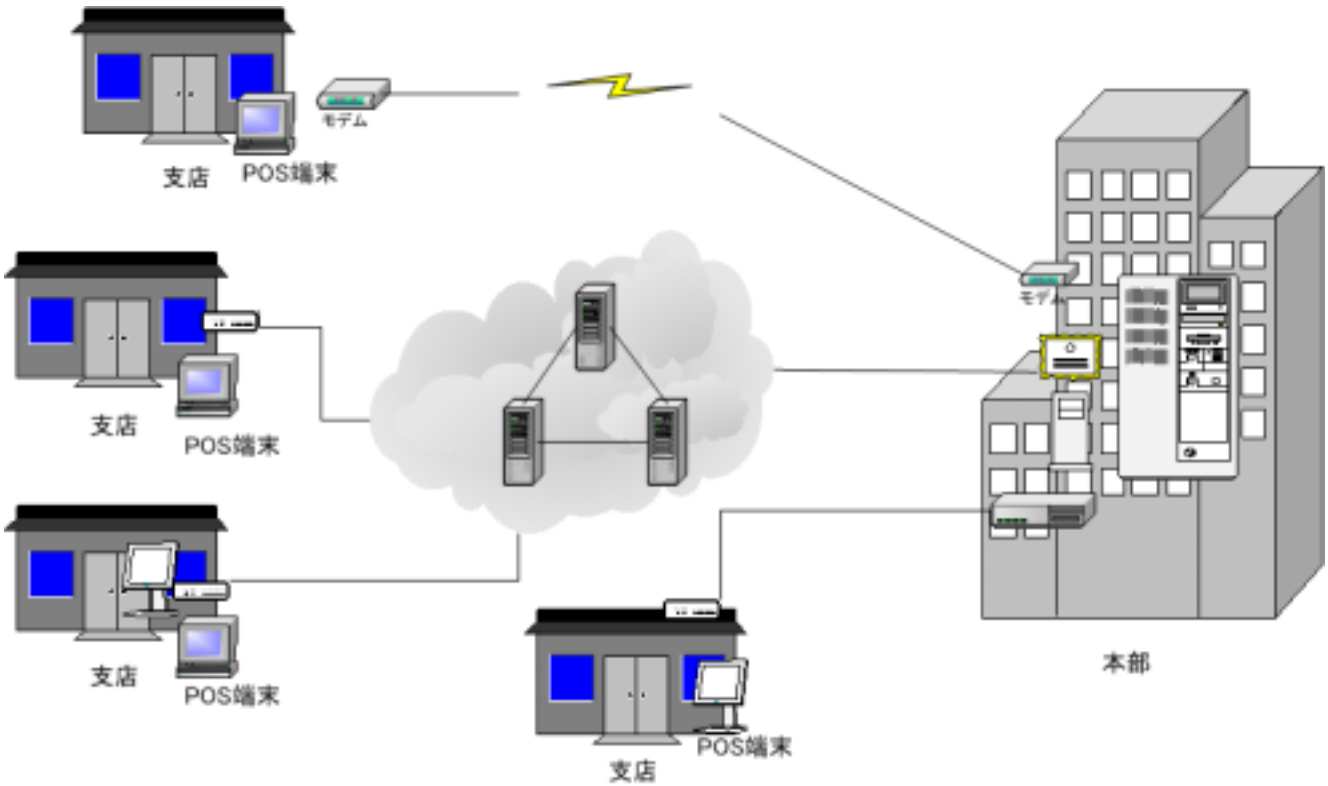

図 3.2.1 POS システム構成図

3.2.1 ユースケース

本論文の開発対象は、POS システム内に設置する POS 端末の機能である（図 3.2.2 及び表 3.2.1 参照)。

POS 端末の一般的な構成は、売上げ登録機能に加えてクレジットカード、プリペイドカードなど､顧客取引情報の読取機能(カードリーダ)､バーコードなど商品情報を読み取るスキャナ、商品情報や顧客情報をファイルする記憶装置とストアコントローラなどで構成される。

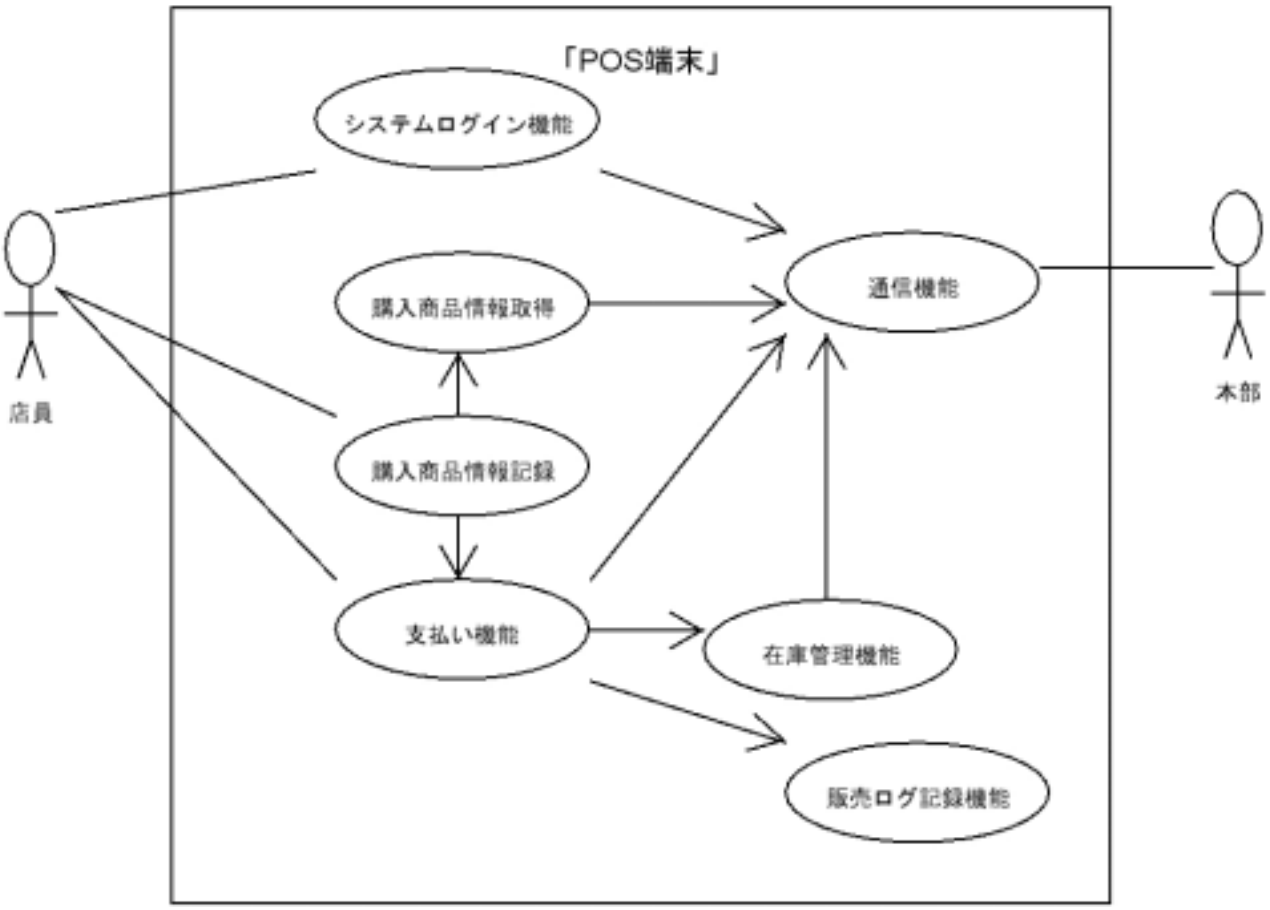

図 3.2.2 ユース関係にあるユースケース図

表 3.2.1 機能一覧表

機能名	説明
システムログイン機能	店員はシステムの使用に際して、ID とパスワードでログイン処理を行う。
購入商品情報取得機能	バーコードリーダを使ってバーコードから購入商品の情報を獲得するか、UPC(Universal Product Code:統一商品コード)のような商品コードを手作業で入力する。
購入商品情報記録機能	進行中の(現在の)販売、つまり購入された商品を記録する。
支払い機能	現金支払いおよびクレジットカード支払いの処理を行い､レシートの印刷を行う。
通信機能	プロセス間通信およびシステム間通信のメカニズムを用意する。
在庫管理機能	販売がコミットされた時点で在庫量を減らす。
販売ログ記録機能	完了した販売ログを保存する。

3.2.2　画面イメージ

本アプリケーションで利用する画面は以下のとおりである。

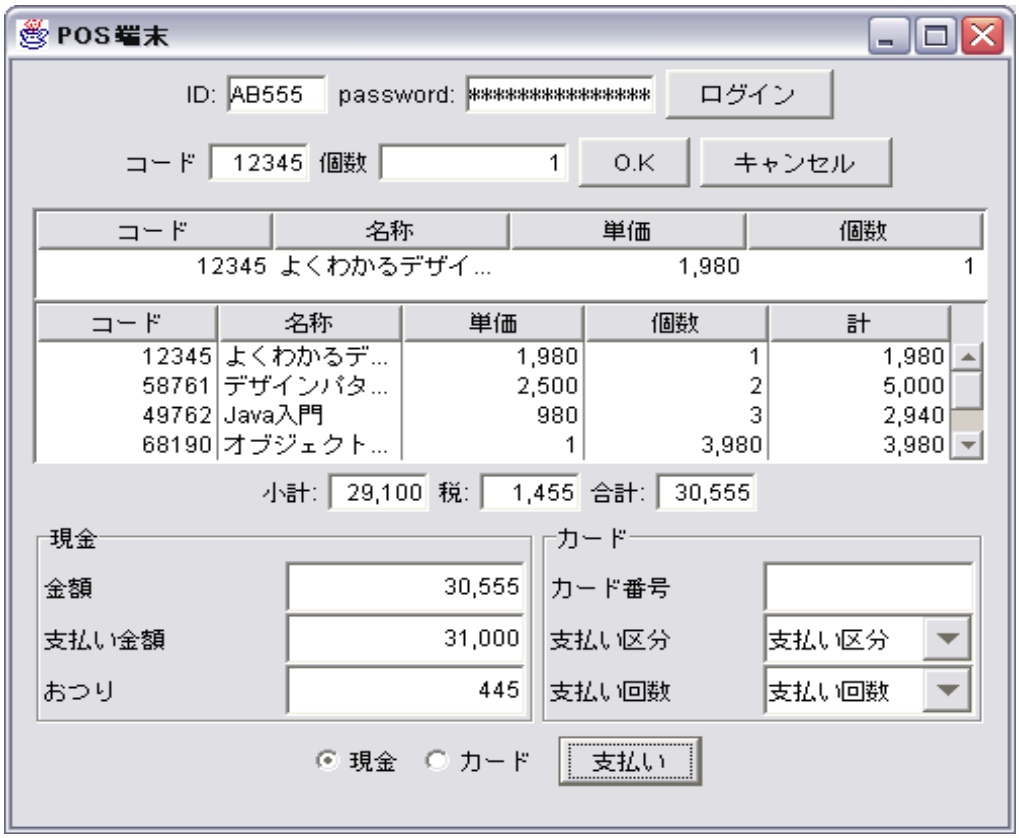

図 3.2.3 画面イメージ

3.2.3 処理の流れ

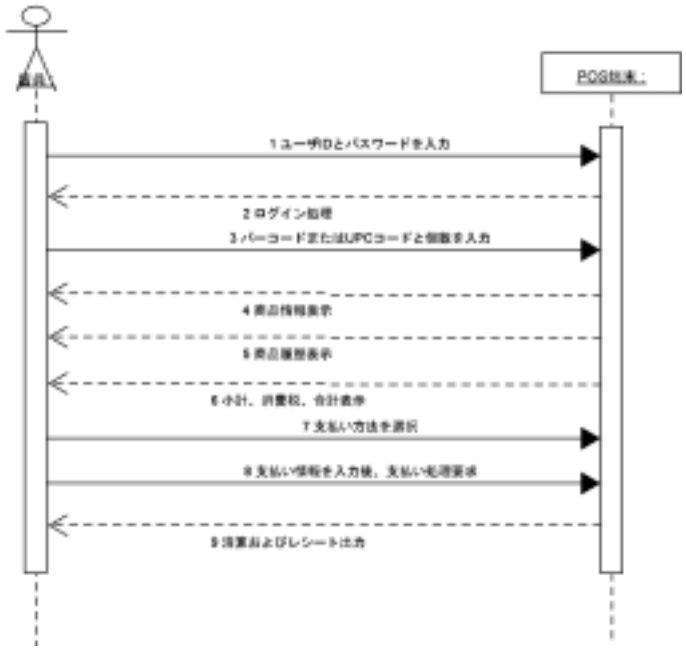

図 3.2.4　シーケンス図

第４章「デザインパターンを適用したクラス設計」

本論文における POS 端末は図 4.1 に示すように、7 つの機能から構成される。

・システムログイン機能
　店員が POS 端末を利用する際にユーザ認証を行う機能。本 POS 端末が行う一連の販売処理の最初に起動する。
・購入商品情報取得機能
　入力された商品のコードより、購入した商品の単価、商品名等の詳細情報を本部サーバより取得する。
・購入商品情報記録機能
　購入商品情報取得機能により取得した購入商品情報を、支払い、在庫管理、ログ出力の各機能で利用できるように記録する。過去の記録情報を履歴管理しているため処理のキャンセルが可能。
・支払い機能
　現金、カードでの支払いに対応し、支払方法に応じた処理とレシートの印刷をする。
・通信機能
　ユーザ認証要求、商品情報取得要求、及び在庫データ更新要求に応じて本部との通信を制御し、本部サーバデータの検索または更新を行う。通信方法の違いに対する制御は通信機能の内部で隠蔽することで共通のインタフェースを提供する。
・在庫管理機能
　販売処理が完了した時点で本部サーバ内の在庫情報を更新する。
・販売ログ記録機能
　販売処理を行った商品の販売ログを POS 端末内部にファイル保存する。

本章では、これら 7 つの機能のクラス設計及び適用パターンについて、詳細な説明をする。

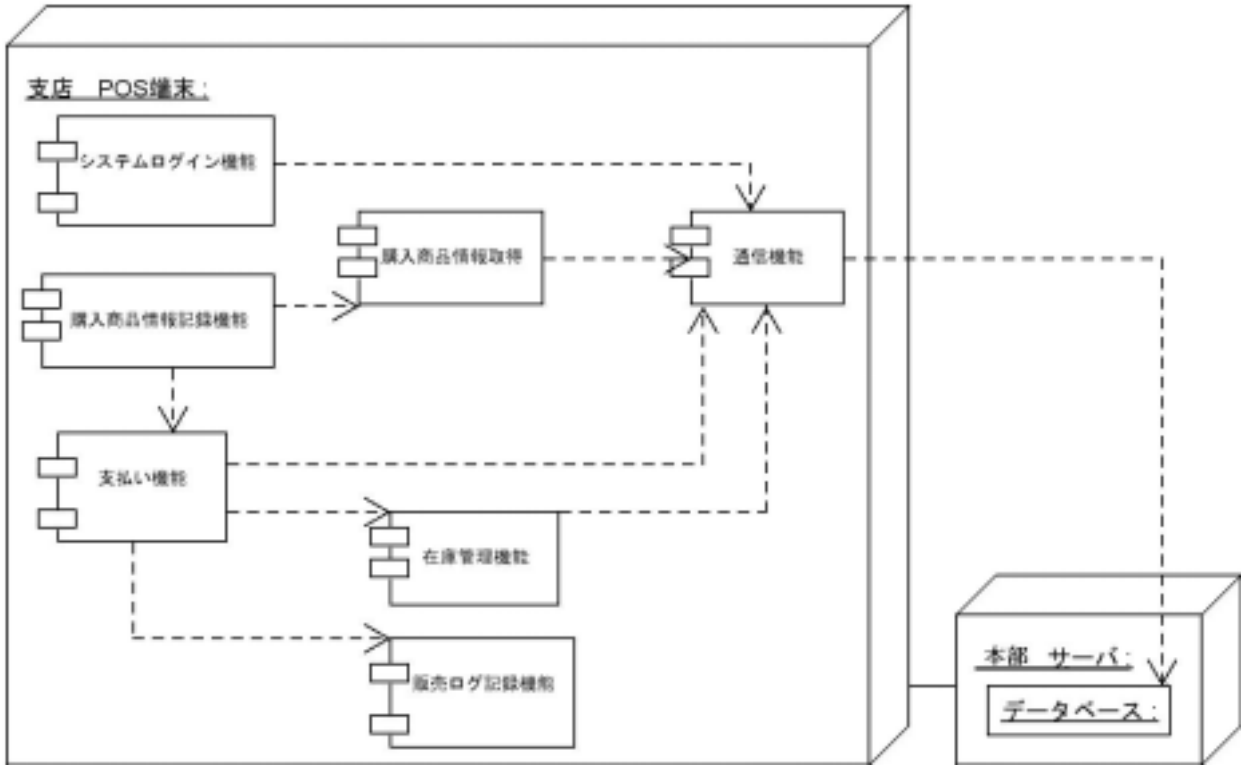

図 4.1　POS 端末システム配置図

4.1 「システムログイン機能」

4.1.1 機能説明

POS 端末を利用するために端末にログインする機能を提供する。

4.1.2 パッケージ図

図 4.1.1 にシステムログイン機能のパッケージ図を示す。システムログイン機能は 5 のクラスで構成しており、POS 端末にログインする機能を提供する。

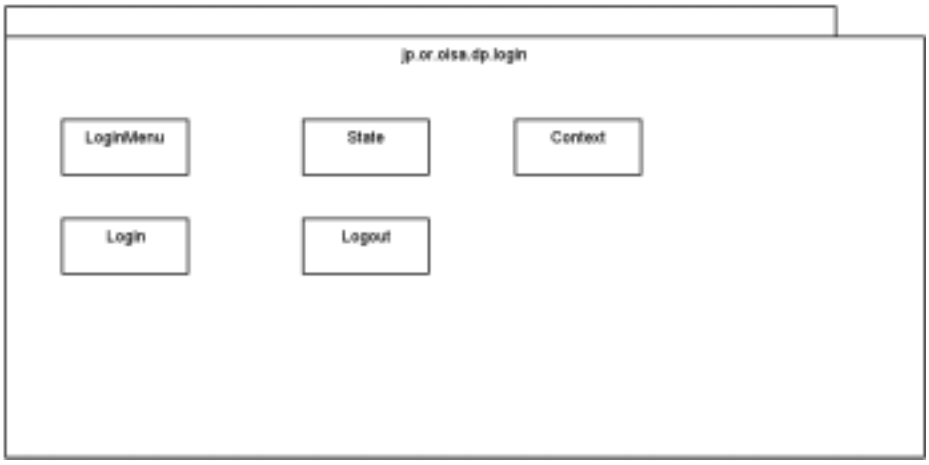

図 4.1.1 システムログイン機能パッケージ図

4.1.3　クラス図

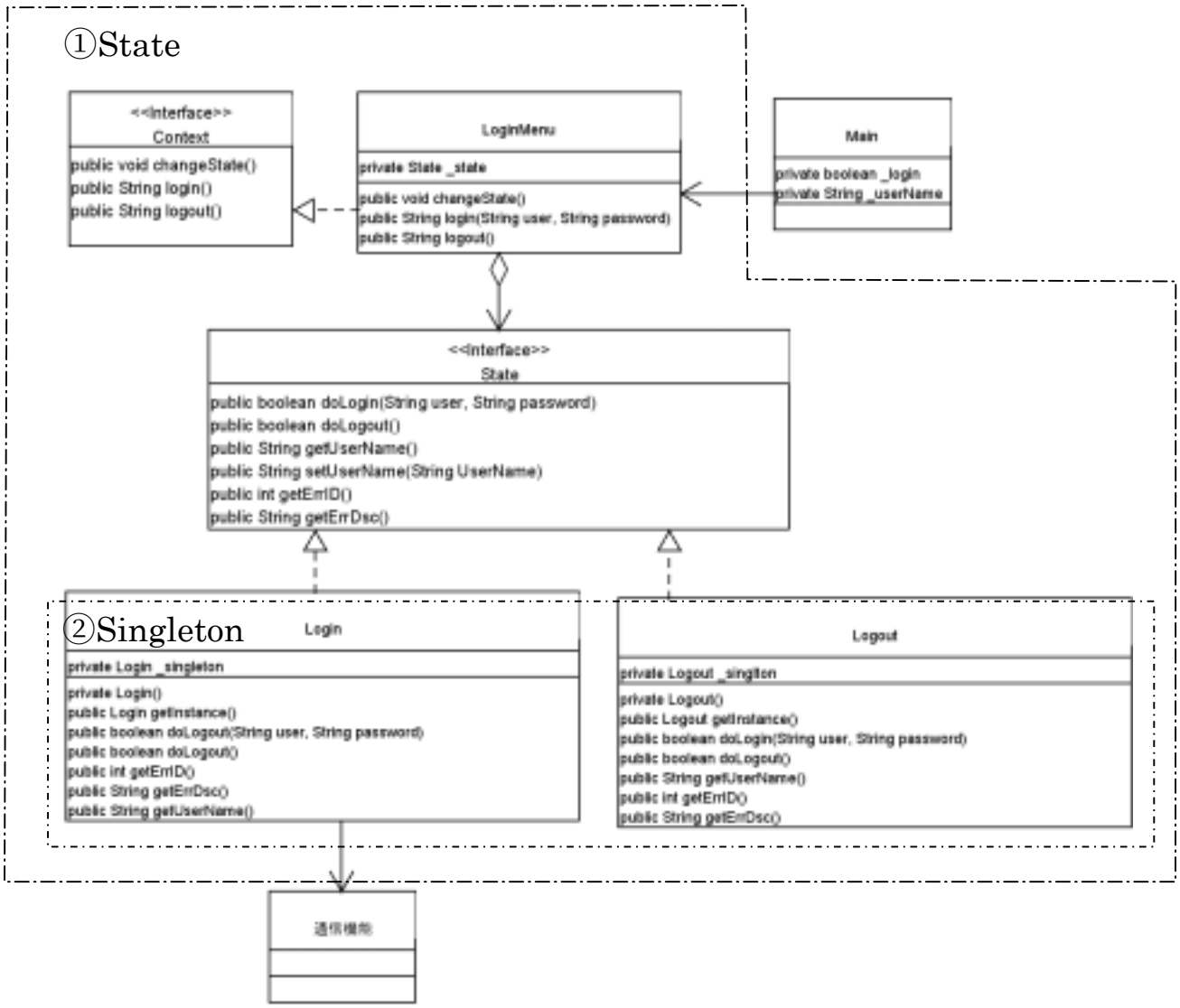

図 4.1.2 システムログイン機能クラス図

表 4.1.1 モジュール一覧表

モジュール名称	物理ファイル名称	機能
ログインメニュークラス	LoginMenu.class	Context を実装し、画面からのイベントを管理取り扱いを行う。
コンテキストインタフェース	Context.class	画面に必要な動作をあらかじめ規定しておく。
ステートインタフェース	State.class	ログインなどの状態に必要なメソッドを規定しておく。
ログインクラス	Login.class	ログインの具体的な動作を記述する。
ログアウトクラス	Logout.class	ログアウトの具体的な動作を記述する。

4.1.4 適応パターン

1) State

ログインログアウトの動作を定義する。

(1)適応範囲

図 4.1.2①部に適用。

(2)選定理由

State パターンを採用した理由は、状態をクラスとして表現できるためである。

(3)利点

State パターンを採用することにより、クラスを状態として表現することができる。このため、現在の状態がどうなっているかということにとらわれずに、コードを記述することができる。また、他の状態を表現する必要性が生じた時にも柔軟に対応することができる。

さらに、状態を表す変数はひとつであるので、状態が曖昧で不整合が生じるという矛盾を防ぐことができる。

ひとつのクラスでひとつの状態を記述することができるため、状態によりコードを分離できコードの可読性も向上する。

(4)注意点

State パターンでは状態をいかに管理するのか。また状態遷移のコーディングに気をつけなければならない。今回の場合は状態管理を LoginMenu にて行った。ただしこの場合実際には状態が変化したことを通知しているのは Login と Logout クラスであり、Login クラスと Logout クラスはお互いに知っていなければ状態の交換ができないというクラスの依存関係が生じてしまう。

2) Singleton

重複ログインを制限する。

(1)適応範囲

図 4.1.2②部の Login Logout に適用。

(2)選定理由

Login Logout に Singleton パターンを適用したのは、ひとつしかインスタンスが必要ないためである。

(3)利点

Singlton パターンでは、インスタンスをひとつしか生成しないということが保証できる。Login Logout など他にインスタンスが必要ない状態であるためにこのパターンを

採用した。

無駄なインスタンスを生成しないことにより、メモリの節約、またインスタンスがひとつであるので、DB 接続などの整合性を取る部分でのコードが書きやすくなる。

(4)注意点

マルチスレッドアプリケーションで使用する場合は、十分注意する必要がある。

4.2 「購入商品情報取得機能」

4.2.1 機能説明

バーコードリーダを使って入力されたバーコード、もしくは手作業で入力された UPC(Universal Product Code：統一商品コード)のバーコード種別を判別、通信機能を使用して購入商品に関する情報を獲得する。

4.2.2 パッケージ図

図 4.2.1 に購入商品情報取得機能のパッケージ図を示す。購入商品購入機能は 4 のクラスで構成しており、入力された商品コードより通信機能を使用して商品情報を取得する機能を提供する。

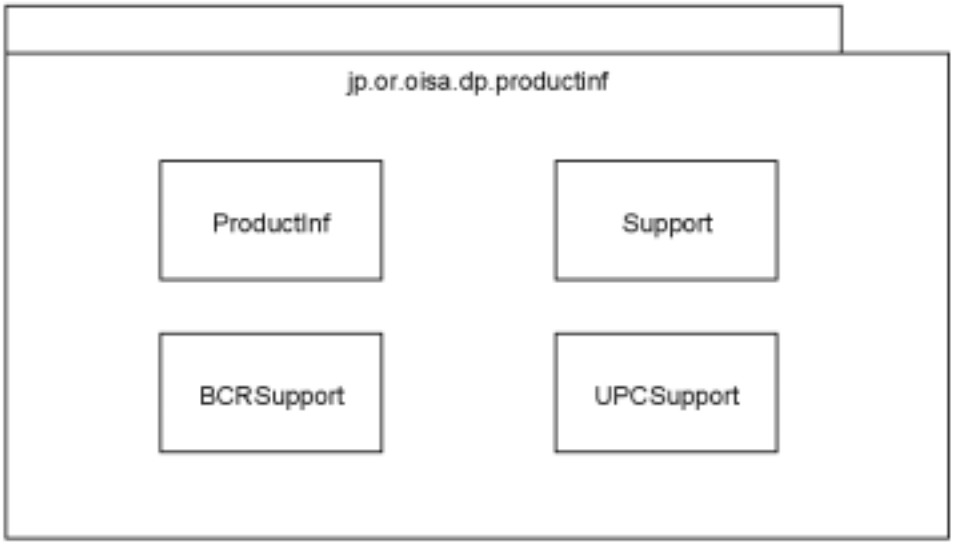

図 4.2.1 購入商品情報取得機能パッケージ図

4.2.3 クラス図

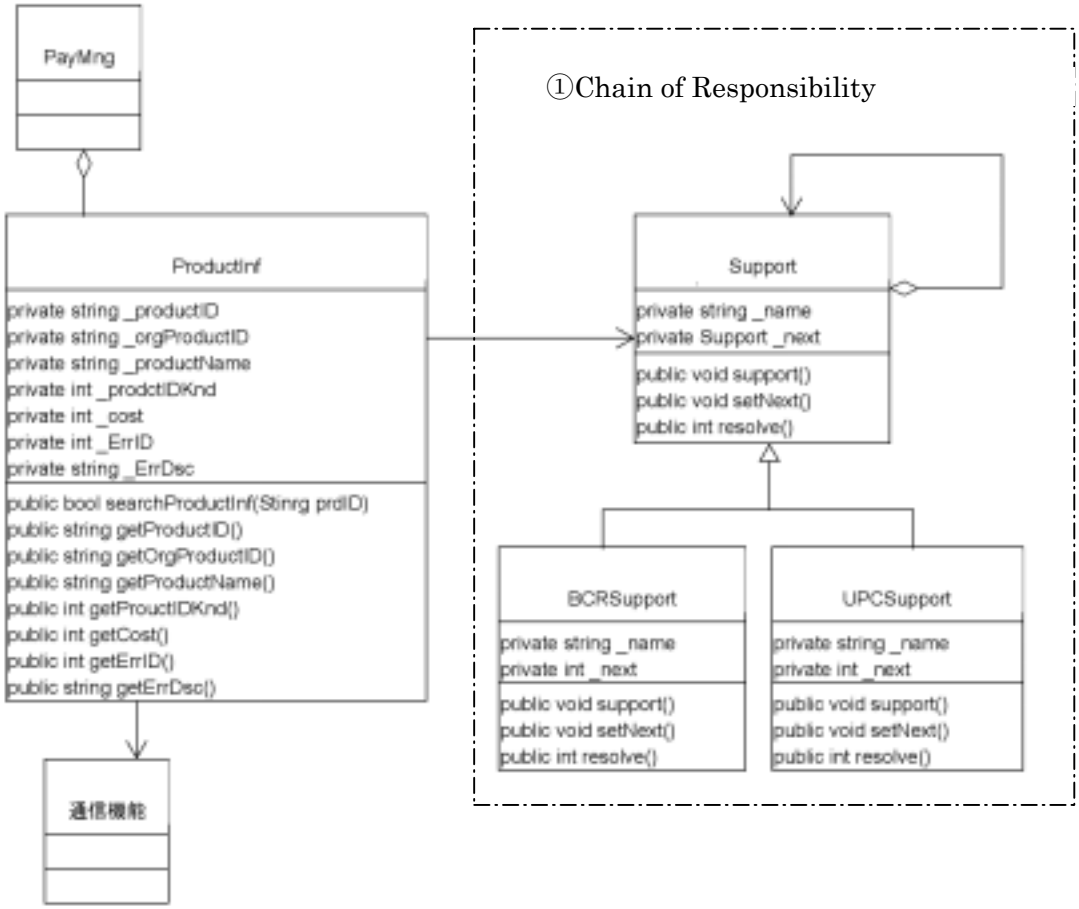

図 4.2.2 購入商品情報取得機能クラス図

表 4.2.1 モジュール一覧表

モジュール名称	物理ファイル名称	機能
購入商品情報クラス	ProductInf.class	外部クラスからの要求に対し情報を取得するクラス。
ハンドラインターフェース	Support.class	必要な動作をあらかじめ規定しておくインターフェース。
バーコードクラス	BCRSupport.class	バーコード判定処理の具体的な動作を記述する。
UPC コードクラス	UPCSupport.class	UPC コード判定処理の具体的な動作を記述する。

4.2.4 適用パターン

1) Chain of Responsibility

バーコード／UPC コードを判定する。

(1)適用範囲

図 4.2.2①部に適用。

(2)選定理由

バーコード／UPC コードのどちらが入力されるか不明であるためコード判断クラスをチェーン状にした。

(3)利点

Chain of Responsibility を使用すると、どのオブジェクトが要求を処理するのかを知っている必要がないためクラス間の結合度を低くすることが可能となる。また、コードの種類の追加はサブクラスを定義するのみでよいため、メンテナンス性が向上する。

(4)注意点

コードが追加されるとチェーンの再形成のために ProductInf クラスを変更する必要がある。

4.3 「購入商品情報記録機能」

4.3.1 機能説明

購入商品記録機能は、取得した購入商品の情報を記録し、支払い、在庫管理、ログ出力機能に対して
必要な情報を提供する機能である。

＜記録する情報＞

・商品名
・商品コード(バーコード)
・単価
・個数(正：購入、負：返品)
・小計
・税金
・合計
・保存件数

4.3.2 パッケージ図

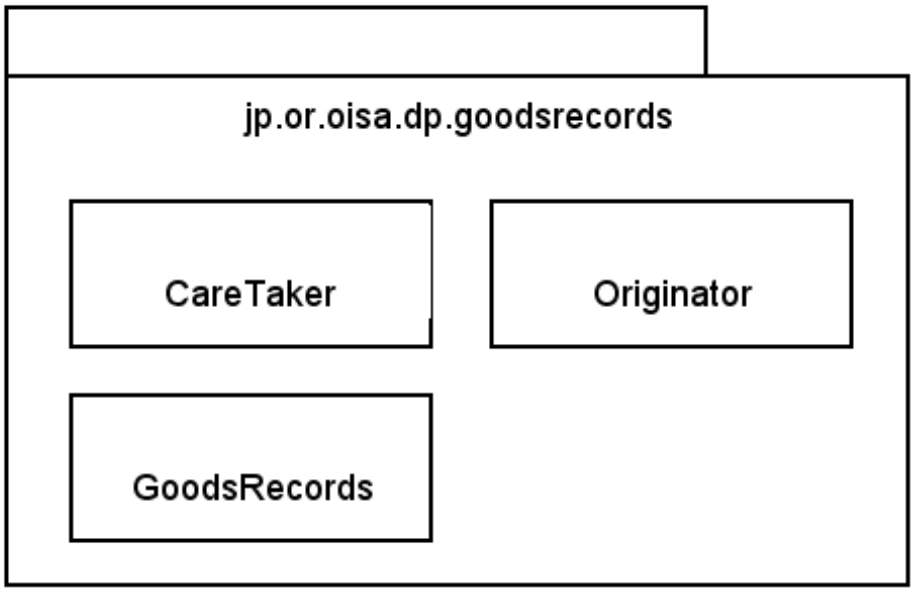

図 4.3.1　購入商品情報記録機能パッケージ図

4.3.3　クラス図

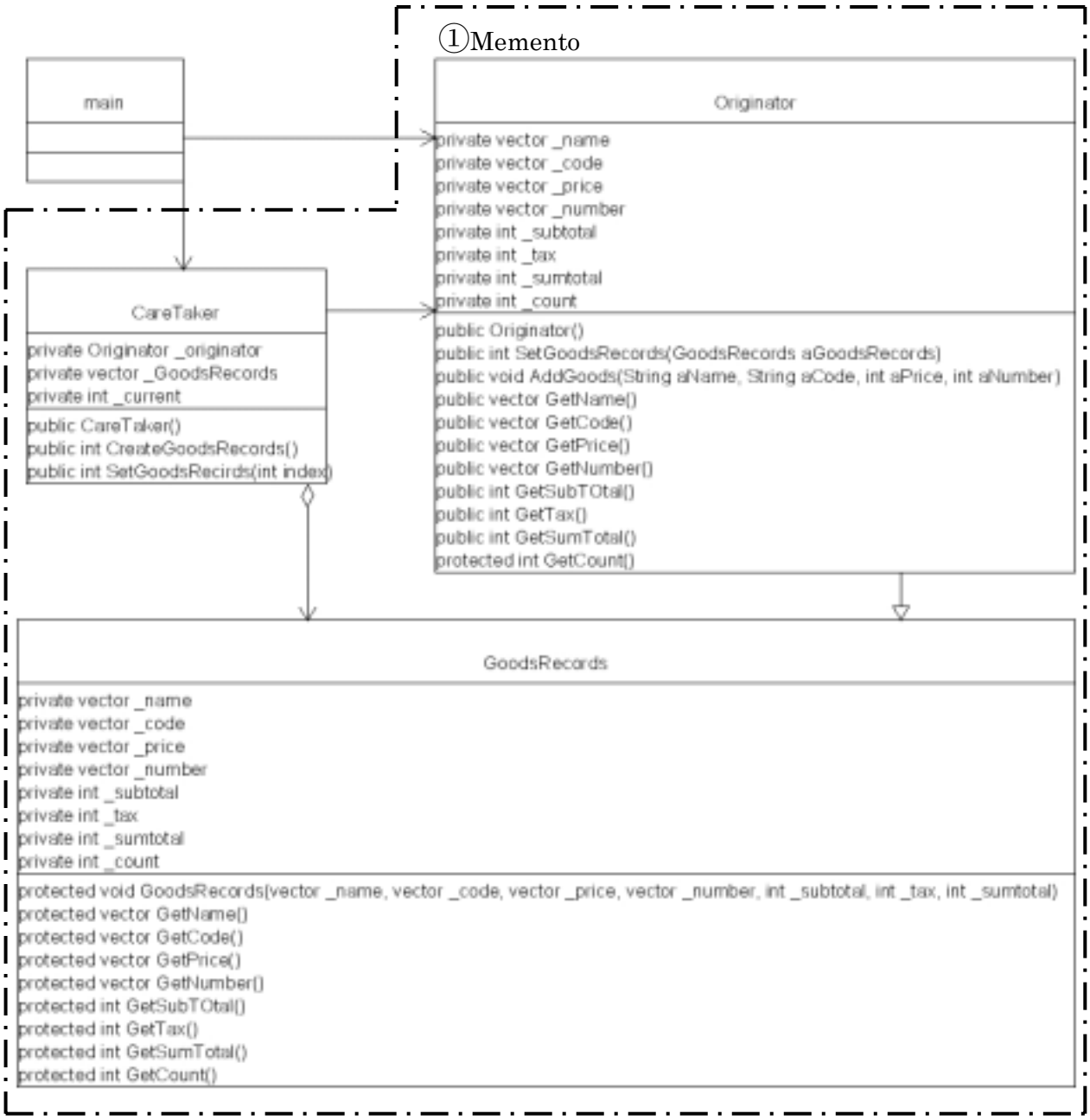

図 4.3.2　購入商品情報記録機能クラス図

表 4.3.1　モジュール一覧表

モジュール名称	物理ファイル名称	機能
記録状態管理クラス	CareTaker.class	記録状態の保管や過去の記録状態への復旧
記録状態クラス	Originator.class	最新の記録状態
記録状態保管クラス	GoodsRecords.class	複数の記録状態を保管

4.3.4 適用パターン

1) Memento

購入商品情報の状態を記録して保存しておき、あとで購入商品情報の状態を記録した状態に戻す

(1)適用範囲

図 4.3.2 の①部に適用

(2)選定理由

状態の履歴管理が出来る事により過去の購入商品情報の状態に戻せる。

例）Memento を利用した状態管理

状態①：Originator の状態(A)を保存する。

状態②：Originator の状態(A,B)を保存する。

状態③：Originator の状態(A,B)から状態①へ戻し、している Originator の状態（A,B）が削除される。

→Originator が状態(A)になる。

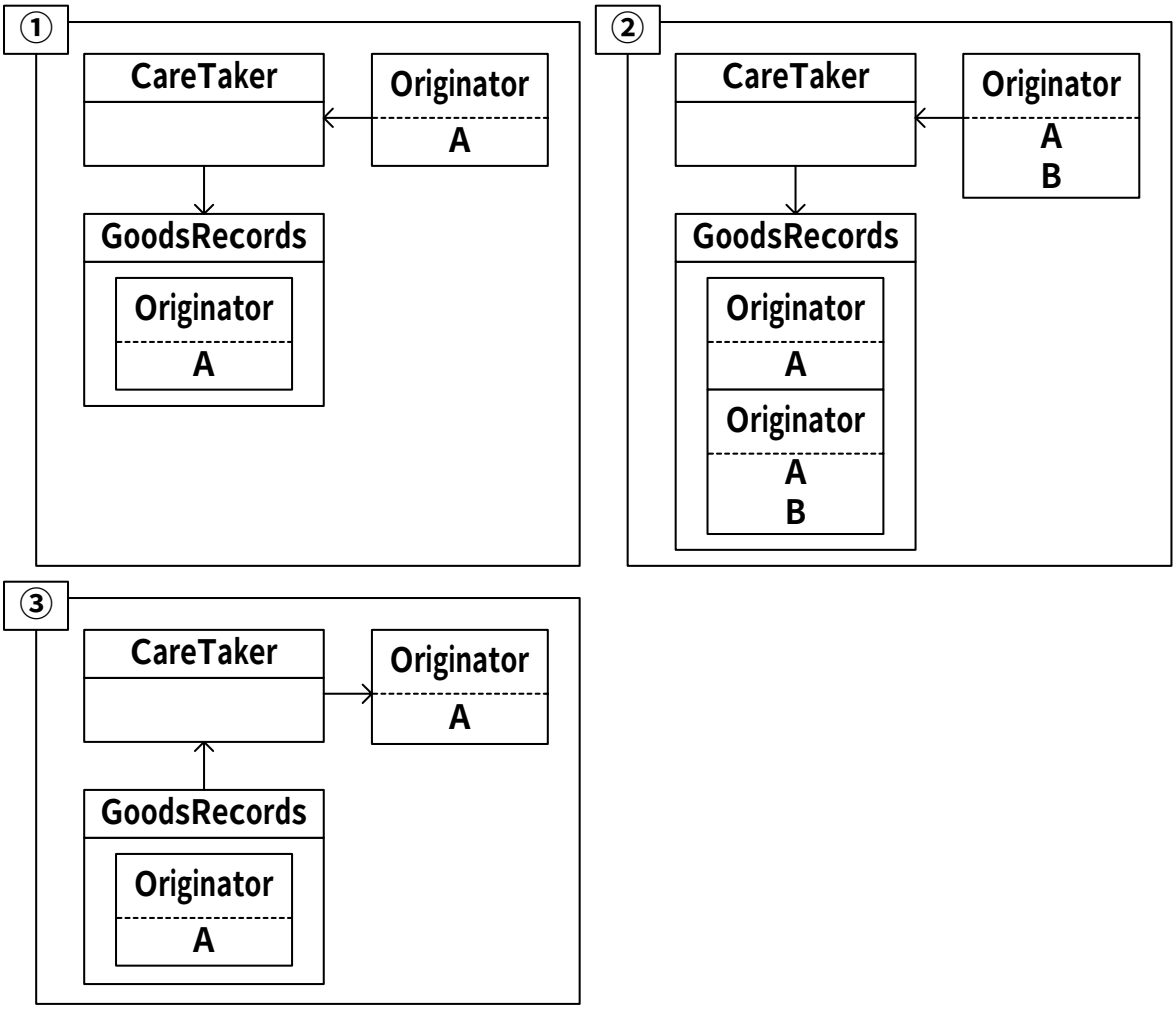

図 4.3.3　状態管理遷移イメージ

(3)利点

当初、Memento パターンではレコード単位の状態保存と思っていたが、実際はオブジェクト単位で状態の保存を行っている。このため、記録状態管理クラスは記録する状態の内部構造に依存せず、記録対象の内部構造が変更されても、修正の必要がない。

(4)注意点

状態を記録保管するたびにオブジェクト単位で状態を保存するため、多くの情報を保管するとメモリ消費量が膨大になる。

4.4　「支払い機能」

4.4.1　機能説明

POS 端末の支払い機能は、購入商品が確定した後の処理全体(支払い → レシート印刷 → 在庫管理 → 販売ログ管理)を制御するものである。支払いには「現金支払い」、「カード支払い」の２つの方法がある。顧客が指定する支払い方法により、出力するレシートの内容は違ってくる。レシートの明細を図 4.4.1 に示す。

【現金支払いの場合】

①商品コード	_org._code
②名称	_org._name
③単価	_org._price
④個数	_org._number
⑤小計	_org._subTotal
⑥消費税	_org._tax
⑦合計	_org._sumTotal
⑧お預かり金額	_money
⑨おつり	_otsuri
⑩支払い時刻	_date
⑪レジ係ユーザ名	_userName

【カード支払いの場合】

①商品コード	_org._code
②名称	_org._name
③単価	_org._price
④個数	_org._number
⑤小計	_org._subTotal
⑥消費税	_org._tax
⑦合計	_org._sumTotal
⑧カード No	_cardNum
⑨支払い区分	_payKubun
⑩支払い回数	_payNum
⑪支払い時刻	_date
⑫レジ係ユーザ名	_userName

図 4.4.1 レシートの明細

「支払い」、「レシート印刷」の処理が終了した後、「在庫管理」、「販売ログ管理」のそれぞれの機能に対し、実行の契機を与える。

4.4.2　パッケージ図

図 4.4.2 に支払い機能のパッケージ図を示す。

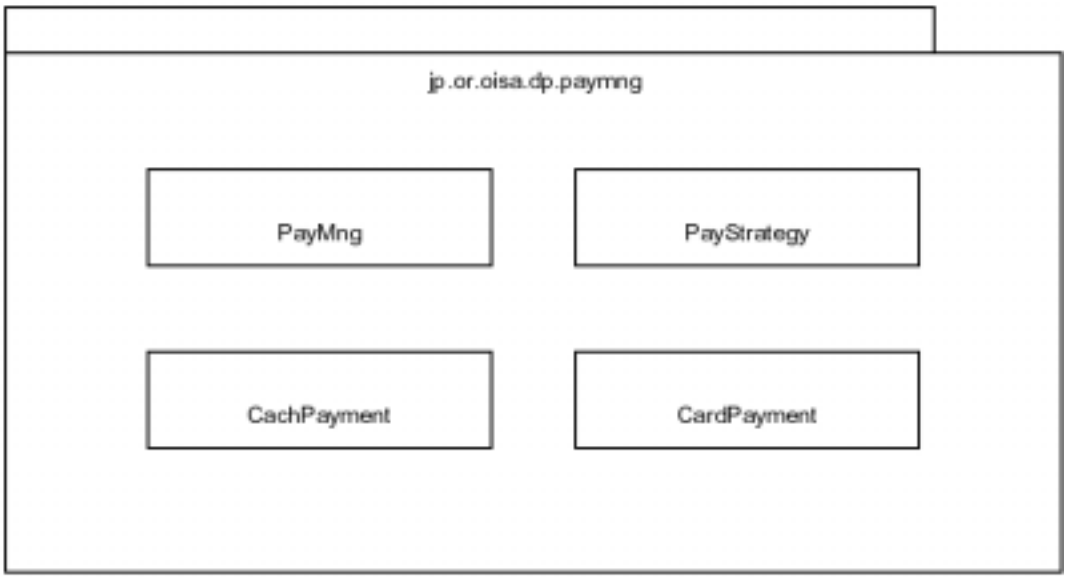

図 4.4.2 支払い機能パッケージ図

4.4.3　クラス図

図 4.4.3 に支払い機能のクラス図を示す。

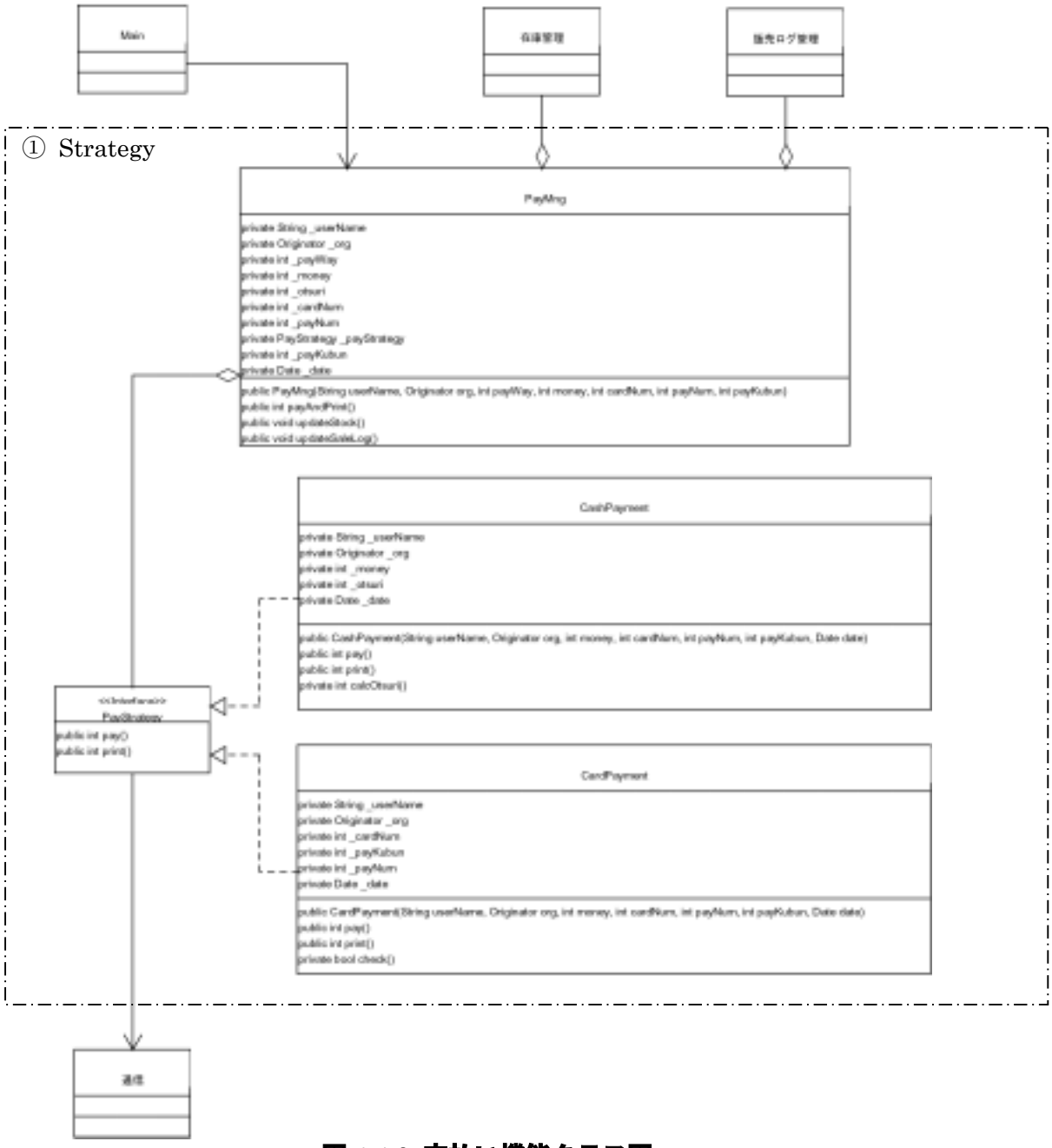

図 4.4.3 支払い機能クラス図

支払い機能は 4 つのモジュール(3 つのクラス＋1 つのインタフェース)で構成される。そのモジュール一覧を表 4.4.1 に示す。

表 4.4.1 モジュール一覧

モジュール名称	物理ファイル名称	機能
支払い管理	PayMng.class	購入商品確定後の全処理を管理するクラス
支払い方法	PayStrategy.class	支払い方法インタフェース
現金による支払い	CashPayment.class	現金による支払い処理を行うクラス
カードによる支払い	CardPayment.class	カードによる支払い処理を行うクラス

4.4.4 適用パターン

1) Strategy

支払いのアルゴリズムをカプセル化し交換可能にする。

(1)適用範囲

図 4.4.3 ①部に適用。

(2)選定理由

支払いの方法が現金かカードかによって、「支払い」、「レシート印刷」の処理内容が違ってくる。このように状況に応じてアルゴリズムの変更が必要となるため Strategy パターンを選定した。

(3)利点

Strategy パターンを使用した設計を行っておくと、もし将来新たに支払い方法が追加されたとしても Concrete Strategy クラスを１つ増やすだけで簡単に対応できる。機能追加に対し柔軟に対応できることは、工数削減や品質向上を容易にするという利点を持つ。

(4)注意点

「現金支払い」「カード支払い」共に対応が可能という設計であるので、PayMng クラスのコンストラクタには「現金支払い」「カード支払い」で用いる全ての情報を含める必要がある。そのため、支払い方法によって入出力内容が大きく変わってしまうような場合には、このパターンの適用はあまり効果がない。

4.5　「通信機能」

4.5.1　機能説明

ＰＯＳ端末の通信機能は、接続方法、対象データベースの種類に関わらず、在庫データの更新及びユーザ認証や商品情報取得に関するデータ取得要求に対して、規定のデータ構成にて必要なデータを返す機能である。

4.5.2　パッケージ図

図 4.5.1 に通信機能のパッケージ図を示す。通信機能は１１のクラスで構成しており、通信の確立やデータベース操作を行う共通クラスと差分クラスがある。通信機能の全てのクラスはＰＯＳ端末であるクライアント端末に配置し、クライアント端末上で動作する。

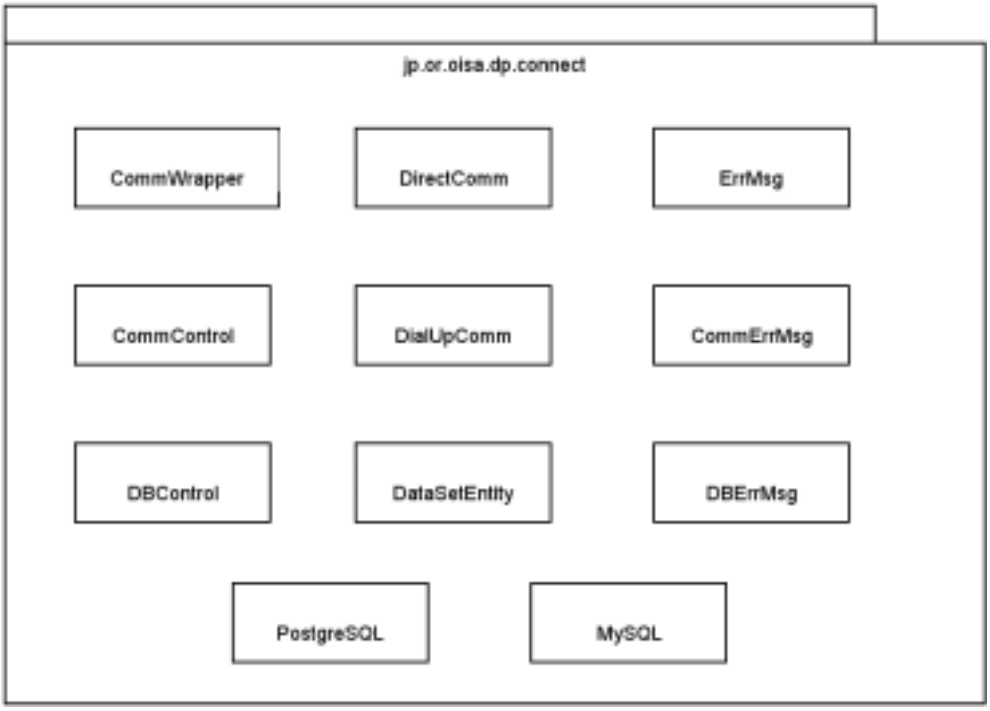

図 4.5.1 通信機能パッケージ図

4.5.3 クラス図

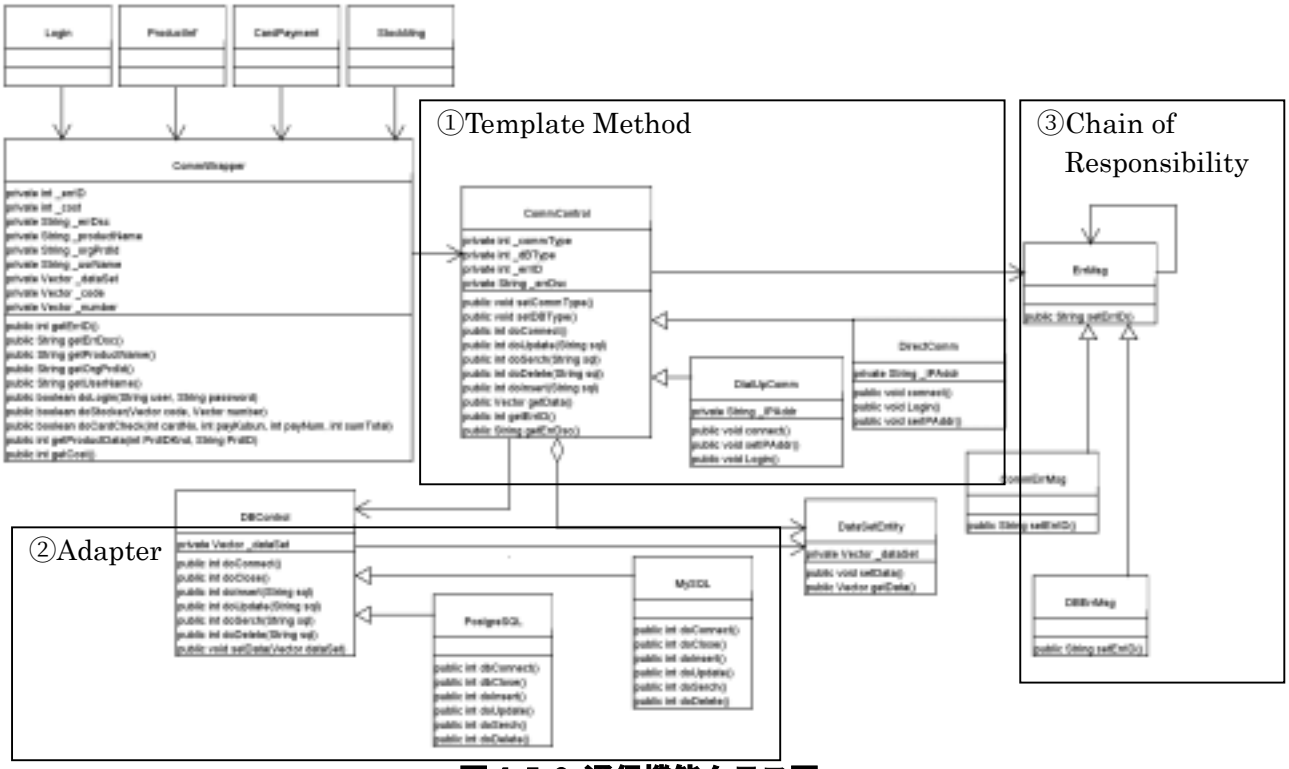

図 4.5.2 通信機能クラス図

表 4.5.1 モジュール一覧表

モジュール名称	物理ファイル名称	機能
通信ラッパークラス	CommWrapper.class	外部クラスの個別の要求に対し、必要なデータを取得する。
通信コントロールクラス	CommControl.class	通信種別コードにより、接続方法を選択する。
ＤＢ接続コントロールクラス	DBControl.class	データベースの種類に伴わず共通の処理で操作する。
専用線接続クラス	DirectComm.class	専用線接続に必要な認証、VLAN 設定を行う。
ダイアルアップ接続クラス	DialUpComm.class	ダイアルアップ接続に必要な接続、認証を行う。
データ格納クラス	DataSetEntity.class	データベース操作に必要なデータの格納と取得データ保持を行う。
PostgreSQL 操作クラス	PostgreSQL.class	PostgreSQL の接続、切断、追加、更新、検索、削除を行う。
MySQL 操作クラス	MySQL.class	MySQL の接続、切断、追加、更新、検索、削除を行う。
エラーメッセージ走査クラス	ErrMsg.class	エラーコードに従い、対応するエラーメッセージを走査する。
接続エラーメッセージクラス	CommErrMsg.class	入力されたエラーコードに対応するエラーメッセージを返す。
ＤＢエラーメッセージクラス	DBErrMsg.class	入力されたエラーコードに対応するエラーメッセージを返す。

4.5.4 適用パターン

1) Template Method

通信機能の個別の処理を再定義する。

(1)適用範囲

図 4.5.2①部の通信操作機能に適用。

(2)選定理由

通信確立の手順は通信方法によりことなるが、接続開始、接続終了、及び一部通信に必要な情報は共通化できるため、処理ステップ単位で共通化または差分化を可能とするTemplate Method パターンの採用が有効である。

(3)利点

汎用処理部分をある程度共通化するため、新たな通信手順を採用した場合の機能追加が容易となる。また、機能拡張に対する制限も少なく、一般的な抽象クラス設計に用いられている。

(4)注意点

Template Method を実装する場合には抽象化クラスのメソッドを最小化する必要がある。今後の機能拡張として確実に共通化されるメソッドではないメソッドを抽象クラスに宣言している場合は、サブクラスで不要なメソッドのオーバーライドが必要となり、返って理解しづらいアルゴリズムとなる。

2) Adapter

インタフェースに互換性のないＤＢ操作を共通インタフェースに変換する。

(1)適用範囲

図 4.5.2②部のＤＢ操作機能に適用。

(2)選定理由

一般的なリレーショナルデータベースの操作ステップはある程度統一されているが、コマンドレベルやデータベース情報の処理形態に違いがある。そこでデータベースを処理するクライアント側からは違いを意識せず、インタフェースの入れ替えにより、共通的な呼出し方法でありながら処理を変更できる Adapter パターンの採用が有効である。

(3)利点

選定理由にもあげているように内部処理の違いを意識する必要がないため、クライアントでは利用するデータベースに関わらずに実装することが可能となる。また、新たなデータベースの処理を追加する場合もこれまでの処理を変更することなく追加することが可能となる。

(4)注意点

インタフェースの多重継承の状態となるため、新たに機能を加えていく場合、階層構造に十分留意する必要がある。

3) Chain of Responsibility

エラーコードに対応するメッセージをチェーンに沿って検索する。

(1)適用範囲

図 4.5.2③部のエラー出力処理機能に適用。

(2)選定理由

エラーメッセージ取得処理の統一化することと機能単位にエラーメッセージをまとめる必要があるため、全エラーコードクラスの走査をシンプルに行える Chain of Responsibility パターンの採用が有効である。

(3)利点

機能単位でエラーメッセージを返すクラスを作成した場合であっても、Chain of Responsibility パターンを採用することで、エラーメッセージを取得する場合にどのクラスに対象のエラーコードが存在するのか意識する必要がない。

(4)注意点

入力したエラーコードに対応するエラーメッセージを正しく受け取るため、存在しない場合の処理やチェーン構造を正確に作成する必要がある。

4.6　「在庫管理機能」

4.6.1 機能説明

在庫管理機能は、販売が完了した（支払い処理終了）後に商品情報（商品コード、個数）を受け取り、同一商品コードごとに個数を計上したのち、通信機能を使用し、在庫情報を更新する機能である。

4.6.2 パッケージ図

図 4.6.1 在庫管理機能パッケージ図

4.6.3 クラス図

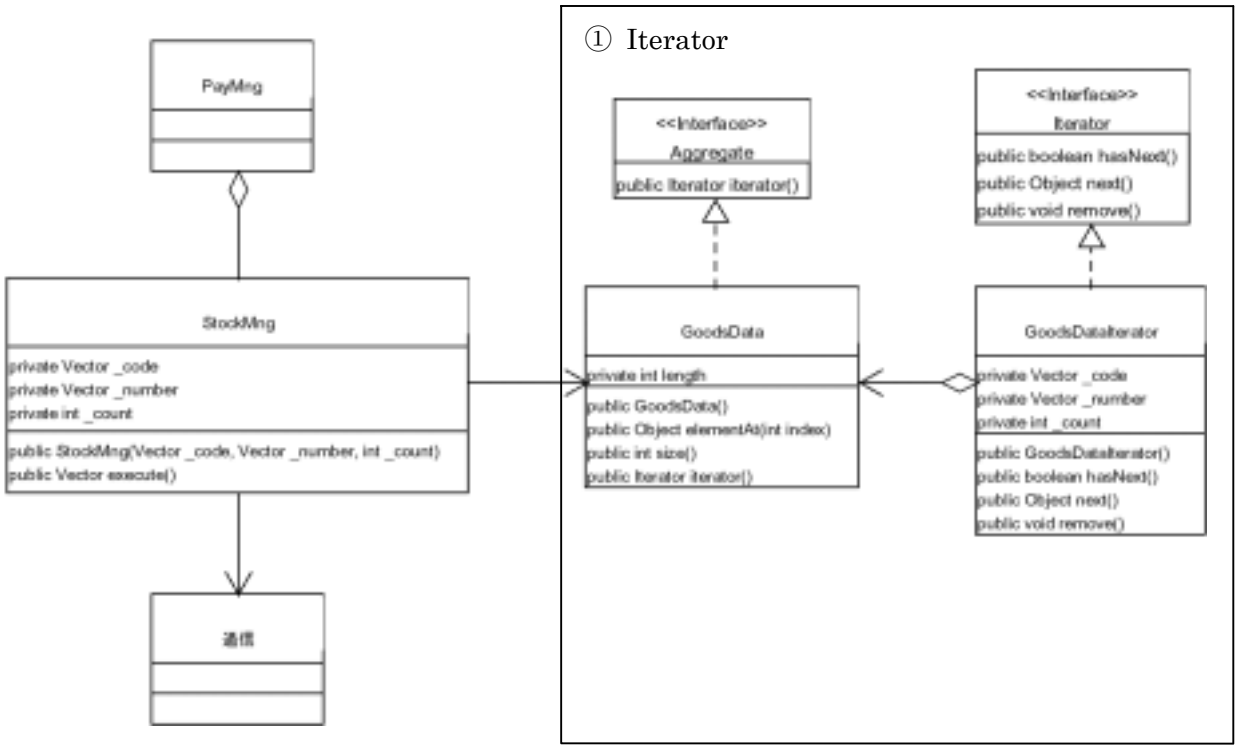

図 4.6.2 在庫管理機能クラス図

表 4.6.1 モジュール一覧表

モジュール名称	物理ファイル名	機能
在庫管理クラス	StockMng.class	商品コードごとに個数を計上するクラス
集合体インタフェース	Aggregate.class	集合体に対する Iterator を作成する機能を定義したインタフェース
数え上げインタフェース	Iterator.class	集合体の内部情報にアクセスするための機能を定義したインタフェース
集合体クラス	GoodsData.class	具体的な集合体(商品コード、個数)を表すクラス
数え上げクラス	GoodsDataIterator.class	集合体の内部情報を走査するクラス

4.6.4 適用パターン

1） Iterator

集合体（商品コード、個数）の走査処理を定義する。

（1）適用範囲

図 4.1.2 ①部に適用。

（2）適用理由

この機能の役割は PayMng クラスから商品情報を受け取り同一商品の個数を計上し、商品データ（商品コードと個数）を通信機能に渡す処理を行う。同一商品の個数を計上するには、商品コードを１つ１つ見比べていく必要がある。そこで Iterator パターンを用いて上位クラスから受け取った商品コードを1つ1つ読み上げていく処理を別クラスに実装すれば、在庫クラス（StockMng クラス）の実装と切り離して走査を行うことができるため、Iterator パターンの適用が有効である。

（3）利点

このように走査処理を別クラス（Iterator クラス）に分けることで、複数の集合体に対して同時に走査処理を行うことができる。また在庫管理クラスの処理内容を変更しても、走査処理部分は Iterator クラスに実装されているので、走査のアルゴリズム自体を変更し易くなる。

（4）注意点

Iterator インタフェースには集合体の数え上げを行うためのメソッドが定義されている。
Iterator クラスの next メソッドは繰り返し処理の次要素を返すメソッドだが、次要素がない場合は例外をスローしてしまう。そのため次の要素が存在するかどうか調べてから next メソッドを呼ばなければならないことに注意が必要である。

4.7 「販売ログ記録」

4.7.1 機能説明

販売ログ記録機能は、完了した販売ログをファイルに保存する機能である。

4.7.2 パッケージ図

図 4.7.1 に販売ログ記録機能のパッケージ図を示す。

図 4.7.1 販売ログ記録機能パッケージ図

4.7.3　クラス図

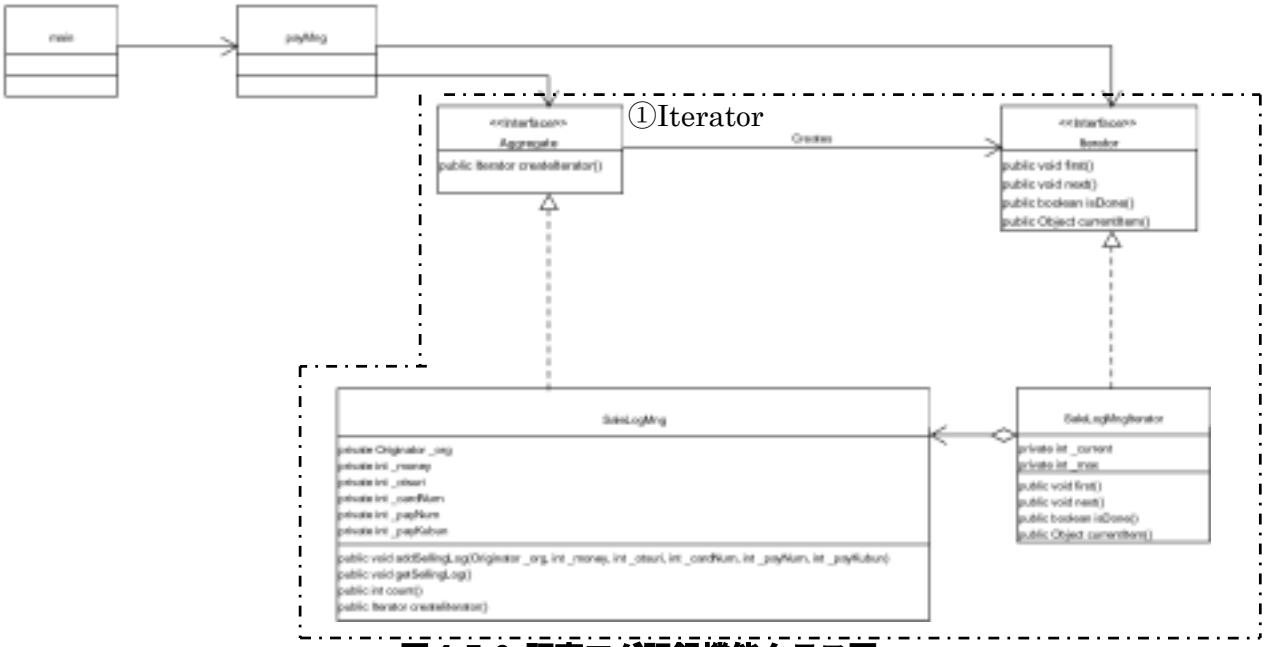

図 4.7.2 販売ログ記録機能クラス図

表 4.7.1 モジュール一覧表

モジュール名称	物理ファイル名	機能
集合体インタフェース	Aggregate.class	集合体を表すインタフェース
走査インタフェース	Iterator.class	要素の走査、スキャンを行うインタフェース
集合体クラス	SaleLogMng.class	具体的な集合体（購入商品情報）を表すクラス
走査クラス	SaleLogMngIterator.class	具体的な反復子を表すクラス

4.7.4 適用パターン

1) Iterator

販売ログ情報に順次アクセスする

(1) 適用範囲

図 4.7.2①部に適用。

(2) 選定理由

この機能の役割は、PayMng クラスから販売ログ記録に必要な情報をファイルに記録する機能である。PayMng クラスで格納された商品情報は Vector 配列で格納されているので、Vector 配列の中の要素を、一つつ取り出しながら、集合体(SaleLogMng)を走査するパターンである Iterator パターンの適用が有効である。

(3) 利点

この機能についての利点は、集合体(SaleLogMng)に走査の為のインタフェースを含めなくてよい(複数の走査アルゴリズムのサポート、走査のアルゴリズムの変更がし易くなる) などの点である。また、今回この機能の適用パターンを考える上で、このパターンを適用することにより複数の集合体に対して走査を可能にするといった利点もあることが分かった。

(4) 注意点

Iterator は検索時や並べ替え時にはパフォーマンスを悪くする場合があるため、適用には注意が必要である。

第５章「まとめ」

5.1　結果

実際のシステム設計でのデザインパターン適用の結果、以下のパターンを適用できた。

・ログイン機能　・・・State, Singleton
・購入商品情報取得機能　・・・Chain Of Responsibility
・購入商品情報記録機能　・・・Memento
・支払い機能　・・・Strategy
・在庫管理機能　・・・Iterator
・販売ログ記録機能　・・・Iterator
・通信機能　・・・Template, Chain Of Responsibility, Adapter

システム全体より分割した機能それぞれにデザインパターンを適用することができ、機能設計を担当者毎に分担したにもかかわらず、一貫した思想の設計が実現できた。このことから、実際のシステム設計を行う上でデザインパターンを利用することは有効であり、しかも多くの場合、デザインパターンを元にプログラム設計を行うことは可能である。

5.2　評価

今回の部会を通して、プログラム設計の段階でデザインパターンを用いることは可能であるということが実証できた。実際にデザインパターンを用いることで最も有益であった点は、デザインパターンを理解していれば、皆が同じコーディングイメージをすることができ、メンバ間のコミュニケーションが高められるということである。例えば、ある機能を設計する段階で、デザインパターンをサンプルコードとして参考にするだけで設計イメージをコードレベルまで掘り下げることが可能になる。そして設計者がイメージした機能を皆に説明する際に、パターン名称を元に説明することで皆が共通したコーディングイメージを持つことが可能になる。

デザインパターンを理解する際に注意が必要なのは、パターンを一つ一つ学習して理解していくという方法では時間がかかるし、実際のシステム設計でパターンをどのように活用すればよいのかがほとんど理解できないということである。デザインパターンを効果的に理解したいのであれば、実際のシステム設計を通してデザインパターンを活用していく方法が最も有効だと考えられる。デザインパターンの概要をざっくりと理解し、使えそうなパターンがあれば詳しく内容を確認することで、より実践的な理解が可能となる。ただし、デザインパターンを活用する前に、まずは構築すべきシステムの機能をしっかりと把握していることが大前提である。

これからデザインパターンの採用を検討している方に助言できることは、まずパターンの機能概要を把握し、実際のシステム設計の場で採用を検討してみて、使えそうであればパターンの詳細を確認してみることで、効率よくパターンを把握することができるということである。デザインパターンはシステム設計の際に設計者にコーディングイメージのサンプルを提供するもので、決して完成されたプログラムではない、ということを念頭においておけば失敗することはないので、臆することなく活用していけば徐々に理解も深まり、上手に活用することができるのではないかと考えられる。

付録A「ＧｏＦデザインパターン」

A.1　「ABSTRACT FACTORY」

A.1.1　目的

互いに関連したり依存し合うオブジェクト群を、その具象クラスを明確にせずに生成するためのインタフェースを提供する。

A.1.2　適用可能性

- システムを部品の生成、組み合わせ、表現の方法から独立にすべき場合。
- 部品の集合が複数存在して、その中の1つを選んでシステムを構築する場合。
- 一群の関連する部品を常に使用しなければならないように設計する場合。
- 部品のクラスライブラリを提供する際に、インタフェースだけを公開して、実装は非公開にしたい場合。

A.1.3　パターン構造

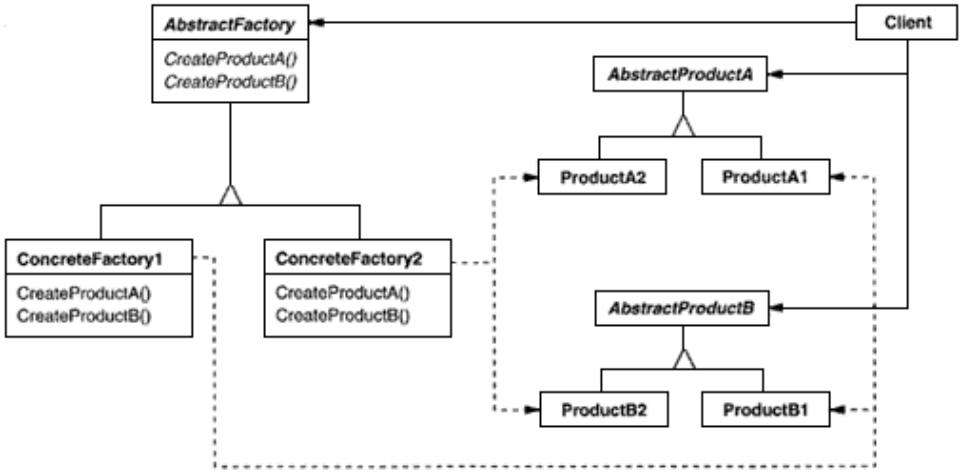

図 A.1.1 ABSTRACT FACTORY 構造

表 A.1.1 モジュール一覧表

モジュール名称	機能
AbstractFactory クラス	・ AbstractProduct のオブジェクトを生成するオペレーションのインタフェースを宣言する。
ConcreteFactory クラス	・ ConcreteProduct オブジェクトを生成するオペレーションを実装する。
AbstractProduct クラス	・ 部品ごとにインタフェースを宣言する。
ConcreteProduct クラス	・ 対応する ConcreteFactory オブジェクトで生成される部品オブジェクトを定義する。 ・ AbstractProduct クラスのインタフェースを実装する
Client クラス	・ AbstractFactory クラスと AbstractProduct クラスで宣言されたインタフェースのみを利用する。

A.1.4 関連するパターン

1) Factory Method パターン、Prototype パターン

AbstractFactory クラスは、しばしば factory method を使って実装されるが、Prototype パターンを使って実装することも可能である。

2) Singleton パターン

ConcreteFactory オブジェクトは、しばしば、Singleton オブジェクトである。

A.2 「ADAPTER」

A.2.1 目的

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。Adapter パターンは、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ことができるようにする。

A.2.2 適用可能性

- 既存のクラスを利用したいが、そのインタフェースが必要なインタフェースと一致していない場合。
- まったく無関係で予想もつかないようなクラス（必ずしも互換性のあるインタフェースを持つとは限らない）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。
- （オブジェクトに適用する Adapter パターンのみ）既存のサブクラスを複数利用したいが、それらすべてのサブクラスをさらにサブクラス化することで、そのインタフェースを適合させることが現実的でない場合。オブジェクトに適用する Adapter パターンでは、その親クラスのインタフェースを適合させればよい。

A.2.3　パターン構造

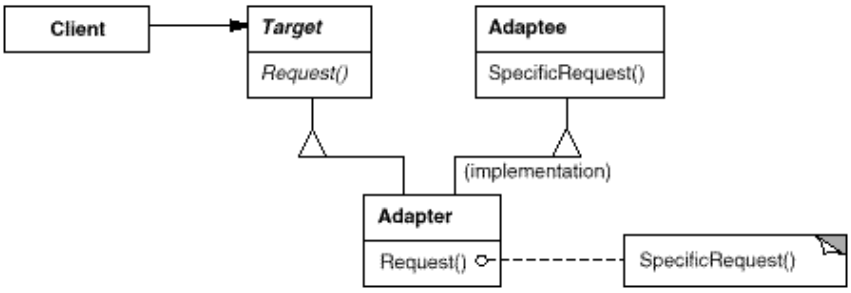

図 A.2.1 Adapter 構造（クラス適用）

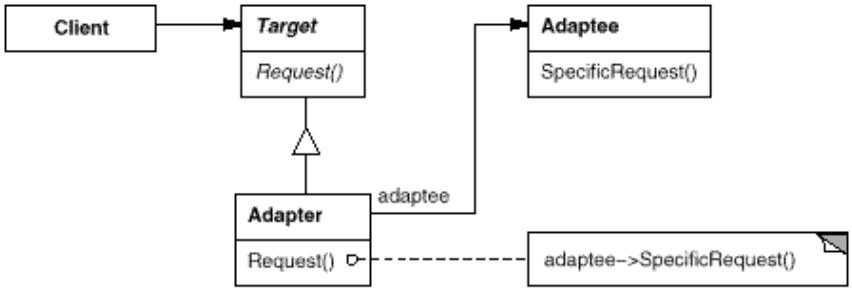

図 A.2.1 Adapter 構造（オブジェクト適用）

表 A.2.1 モジュール一覧表

モジュール名称	機能
Target クラス	・ Client クラスが利用する、ドメインに特化したインタフェースを定義する。
Client クラス	・ Target クラスのインタフェースに従ったオブジェクトと協力する。
Adaptee クラス	・ 部品ごとにインタフェースを宣言する。
Adapter クラス	・ 適合させる必要のある既存のインタフェースを持つ。

A.2.4 関連するパターン

1) Bridge パターン

このパターンは、オブジェクトを基にした adapter によく似た構造をしているが、目的が異なっている。Bridge パターンの目的は、インタフェースと実装を分離して、それらを容易にかつ独立して変更できるようにすることである。Adapter パターンの目的は、“既存の”オブジェクトのインタフェースを変換することである。

2) Decorator パターン

このパターンでは、インタフェースを変更することなく、オブジェクトに対して機能の追加を行う。アプリケーションにとっては、adapter よりも decorator の方が透過性が高い。
その結果 Decorator パターンは、純粋な adapter では不可能な、再帰的なオブジェクト構造をサポートする。

3) Proxy パターン

このパターンでは、他のオブジェクトに対する代表あるいは代理を定義するが、その際にインタフェースを変更することはない。

A.3 「BRIDGE」

A.3.1 目的

抽出されたクラスと実装を分離して、それらを独立に変更できるようにする。

A.3.2 適用可能性

- 抽出されたクラスとその実装を永続的に結合することを避けたい場合。たとえば、実装を実行時に選択したり交換したりしなければならないときに、このような場合が起こり得る。
- 抽出されたクラスとその実装の両方を、サブクラスの追加により拡張可能にすべき場合。この場合、Bridge パターンを用いることで、抽出されたクラスに異なる実装を結合したり、それぞれを独立に拡張することが可能になる。
- 抽出されたクラスの実装における変更が、クライアントに影響を与えるべきではない場合。すなわち、クライアントのコードを再コンパイルしなくても済むようにすべき場合。
- C++で抽出されたクラスの実装をクライアントから完全に隠ぺいしたい場合。C++では、クラスの内部表現はクラスのインタフェースで見ることができてしまう。
- 1つのオブジェクトを2つの部分に分割する必要があるような、サブクラスが増殖していくような継承を行う場合。
- 参照数を用いることにより複数のオブジェクト間で実装を共有したい場合、そして、そのことをクライアントから隠しておきたい場合。

A.3.3 パターン構造

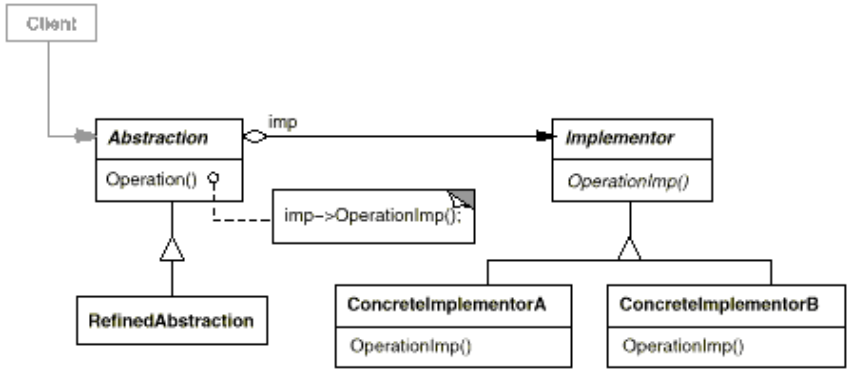

図 A.3.1 Bridge 構造

表 A.3.1 モジュール一覧表

モジュール名称	機能
Abstraction クラス	・抽出されたクラスのインタフェースを定義する。 ・Implementor 型のオブジェクトへの参照を保持する。
RefinedAbstraction クラス	・Abstraction クラスで定義されたインタフェースを拡張する。
Implementor クラス	・実装を行うクラスのインタフェースを定義する。このインタフェースは Abstraction クラスのインタフェースに正確に一致する必要はない。実際、この 2 つのインタフェースがまったく異なることもあり得る。典型的には、Implementor クラスのインタフェースはプリミティブなオペレーションのみを提供しており、Abstraction クラスは、これらのオペレーションを基にしてより高レベルのオペレーションを定義する。
ConcreteImplementor クラス	・Implementor クラスのインタフェースを実装する。具体的な実装について定義する。
Client クラス	・AbstractFactory クラスと AbstractProduct クラスで宣言されたインタフェースのみを利用する。

A.3.4 関連するパターン

1) Abstract Factory パターン

AbstractFactory クラスは、しばしば factory method を使って実装されるが、Prototype パターンを使って実装することも可能である。

2) Adapter パターン

このパターンは、関係のないクラス同士をつなぐことが目的である。このパターンは通常は設計が終わった後で適用される。それに対して、Bridge パターンは、抽出されたクラスと実装を独立に変更可能にするために設計の前段階で使われる。

A.4 「BUILDER」

A.4.1 目的

複合オブジェクトについて、その作成過程を表現形式に依存しないものにすることにより、同じ作成過程で異なる表現形式のオブジェクトを生成できるようにする。

A.4.2 適用可能性

- 多くの構成要素からなるオブジェクトを生成するアルゴリズムを、構成要素自体やそれらがどのように組み合わされるのかということから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。

A.4.3 パターン構造

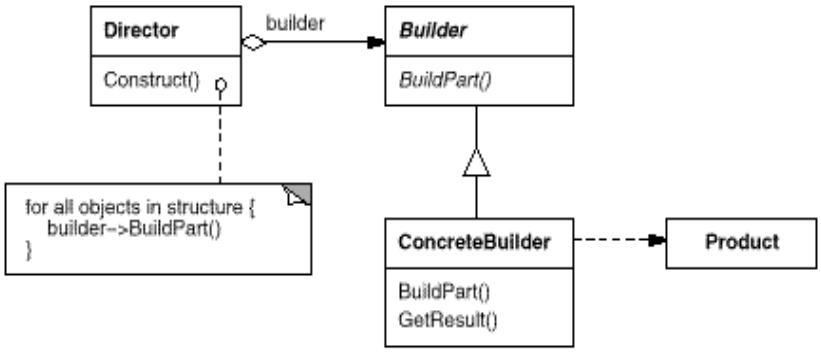

図 A.4.1 Builder **構造**

表 A.4.1 モジュール一覧表

モジュール名称	機能
Builder クラス	・ Product オブジェクトの構成要素を生成するための抽象化されたインタフェースを規定する。
ConcreteBuilder クラス	・ Builder クラスのインタフェースを実装することで、Product オブジェクトの構成要素の生成や組み合わせを行う。 ・ 自身が生成する表現を定義し、管理する。 ・ Product オブジェクトを取り出すためのインタフェースを提供する（たとえば、GetASCIIText オペレーション、GetTextWidget オペレーション）。
Director クラス	・ Builder クラスのインタフェースを使って、オブジェクトを生成する。
Product クラス	・ 作成中の、多くの構成要素からなる複合オブジェクトを表す。ConcreteBuilder クラスは、Product オブジェクトの内部表現を作成し、また、それを組み立てる過程を定義している。 ・ 構成要素を定義するクラス、および構成要素を最終的な Product オブジェクトに組み合わせていくためのインタフェースを含んでいる。

A.4.4 関連するパターン

1) Abstract Factory パターン

このパターンは、複合オブジェクトを作成するという点で Builder パターンに類似している。主な違いは、Builder パターンでは複合オブジェクトを段階的に作成していく過程に焦点をあてているのに対して、Abstract Factory パターンでは部品の集合を強調している（それが単純であっても複雑であっても）という点である。Builder パターンでは、Product オブジェクトを最終段階で返すことになるが、Abstract Factory パターンでは即座に返す。

2) Composite パターン

builder は、composite を作成することがある。

A.5 「CHAIN OF RESPONSIBILITY」

A.5.1 目的

１つ以上のオブジェクトに要求を処理する機会を与えることにより、要求を送信するオブジェクトと受信するオブジェクトの結合を避ける。受信する複数のオブジェクトをチェーン状につなぎ、あるオブジェクトがその要求を処理するまで、そのチェーンに沿って要求を渡していく。

A.5.2 適用可能性

- 要求を処理するオブジェクトの候補が複数存在し、最終的にどのオブジェクトが担当するのかは、前もってわからない場合。担当オブジェクトは自動的に決められる
- 受け手を明確にせずに、複数あるオブジェクトの1つに対して要求を発行したい場合
- 要求を処理することができるオブジェクトの集合が動的に明確化される場合

A.5.3 パターン構造

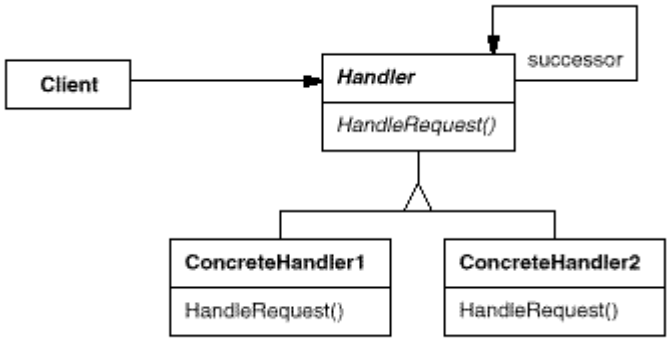

図 A.5.1 CHAIN OF RESPONSIBILITY 構造

表 A.5.1 モジュール一覧表

モジュール名称	機能
Handler クラス	・ 要求を処理するためのインタフェースを定義する。 ・ （オプション）successor へのリンクを実装する。
ConcreteHandler クラス	・ 要求の中で担当するものについて処理する。 ・ successor にアクセスできる。 ・ 要求を処理できるならば処理する。さもなければ、その要求を successor に転送する。
Client クラス	・ チェーンの中にある ConcreteHandler オブジェクトに要求を出す。

A.5.4 関連するパターン

1) Composite パターン

Chain Of Responsibility パターンは、しばしば Composite パターンとともに適用される。その場合、component の親オブジェクトを successor にすることができる。

A.6 「COMMAND」

A.6.1 目的

要求をオブジェクトとしてカプセル化することによって、異なる要求や、要求からなるキューやログにより、クライアントをパラメータ化する。また、取り消し可能なオペレーションをサポートする。

A.6.2 適用可能性

- 実行する動作によりオブジェクトをパラメータ化したい場合。手続き型言語では、そのようなパラメータ化をコールバック関数を使って表現する。すなわち、コールバック関数を、呼び出してほしいところに登録しておく、という形になる。Command パターンでは、そのようなコールバック関数の代わりにオブジェクトを使う。
- 要求を明確にし、順番に並べ、実行するのをそれぞれ別々に行いたい場合。command は、元の要求とは独立に存在し続ける。もし、要求を最終的に受信するオブジェクトをアドレス空間とは独立な方法で表現できるならば、command を別のプロセスに移してそこで要求を実行させることもできる。
- 要求の取り消しをサポートしたい場合。Command クラスの Execute オペレーションでは、command での処理の結果を再び元の状態に戻すことができるように、状態を保存するようにしておくことができる。その場合 Command クラスには、インタフェースとして、直前の Execute オペレーションの呼び出しの結果を元に戻す Unexecute オペレーションを追加しておかなければならない。実行される command は、履歴リストの中にたくわえられる。取り消しや再実行は、このリスト内を前後に移動しながら、Unexecute オペレーションや Execute オペレーションの呼び出しを行うことにより、何度でも実行できる。
- システムがクラッシュしたときにコマンドを再度実行できるように、ログの更新をサポートしたい場合。Command クラスのインタフェースにロードとセーブのオペレーションを備えておくことで、更新を永続的にログに記録しておくことができる。クラッシュからの復旧に際しては、ログに記録された command 群をディスクからロードして、それらに対して Execute オペレーションの呼び出しを再度行う。
- プリミティブなオペレーションを基に作られた高度なオペレーションによりシステムを構造化したい場合。そのような構造は、情報システムのトランザクションにとっては一般的である。一般に、1つのトランザクションは、データに対する更新手続きの集合をカプセル化している。Command パターンはこのようなトランザクションをモデル化する方法を与える。command は共通のインタフェースを持っているので、すべてのトランザクションを同じように呼び出すことができるようになる。このパターンにより、新しいトランザクションを追加してシステムを拡張することも容易になる。

A.6.3 パターン構造

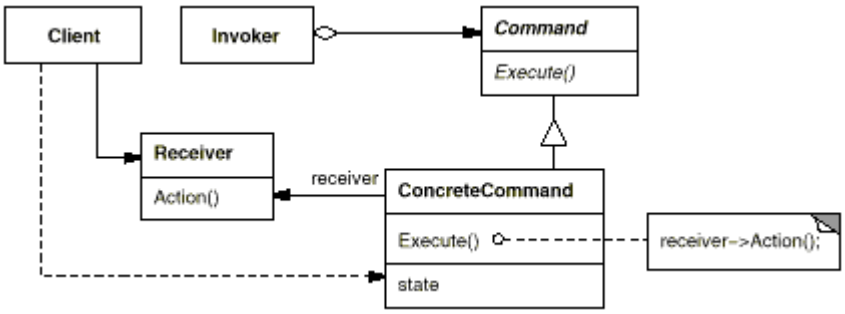

図 A.6.1 COMMAND 構造

表 A.6.1 モジュール一覧表

モジュール名称	機能
Command クラス	・ オペレーションを実行するためのインタフェースを宣言する。
ConcreteCommand クラス	・ Receive r オブジェクトとアクションの間のつながりを定義する。 ・ Execute オペレーションを、Receiver オブジェクトに対して該当するオペレーションの呼び出しを行うように実装する。
Client クラス	・ ConcreteCommand オブジェクトを生成して、それに対する Receiver オブジェクトを設定する。
Invoker クラス	・ command に要求を実行するように依頼する。
Receiver クラス	・ 要求を実現するためにオペレーションをいかに実行するのかを知っている。任意のクラスが Receiver になり得る。

A.6.4 関連するパターン

1) Composite パターン

MacroCommand クラスを実装するために Composite パターンを使うことができる。

2) Memento パターン

command がその実行結果を取り消すことができるように、状態を保存しておくことができる。

3) Prototype パターン

command を履歴リストに入れる前にコピーする場合、コピー元のオブジェクトは prototype として振る舞っている。

A.7 「COMPOSITE」

A.7.1 目的

部分－全体階層を表現するために、オブジェクトを木構造に組み立てる。Composite パターンにより、クライアントは、個々のオブジェクトとオブジェクトを合成したものを一様に扱うことができるようになる。

A.7.2 適用可能性

- オブジェクトの部分—全体階層を表現したい場合。
- クライアントが、オブジェクトを合成したものと個々のオブジェクトの違いを無視できるようにしたい場合。このパターンを用いることで、クライアントは、composite 構造内のすべてのオブジェクトを一様に扱うことができるようになる。

A.7.3 パターン構造

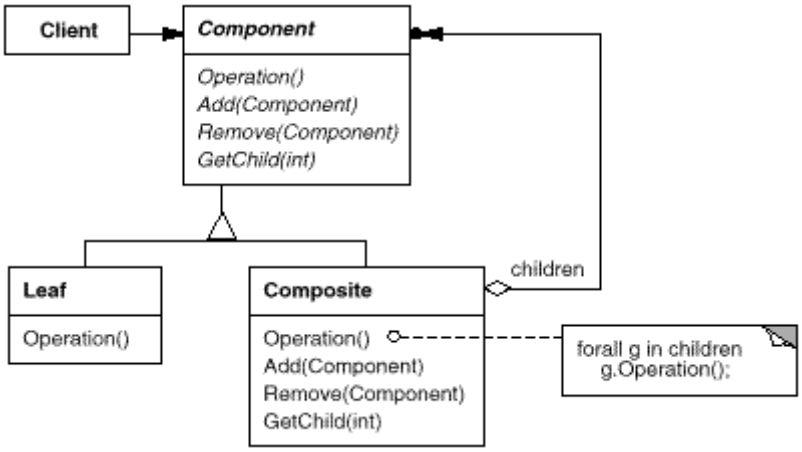

図 A.7.1 COMPOSITE 構造

表 A.7.1 モジュール一覧表

モジュール名称	機能
Component クラス	・ composite 内のオブジェクト（component と呼ぶ）のインタフェースを宣言する。 ・ すべてのクラスに共通なインタフェースのデフォルトの振る舞いを適宜実装する。 ・ 子にあたる Component オブジェクトにアクセスしたり、それを管理するためのインタフェースを宣言する。
Leaf クラス	・ composite 内の末端のオブジェクト（leaf と呼ぶ）を表す。つまり、leaf は子オブジェクトを持たない。 ・ composite 内のプリミティブなオブジェクトの振る舞いを定義する。
Composite クラス	・ 子オブジェクトを持つ component（すなわち、composite）の振る舞いを定義する。 ・ 子にあたる component を保持する。 ・ Component クラスのインタフェースで宣言された、子オブジェクトに関するオペレーションを実装する。
Client クラス	・ Component クラスのインタフェースを通して、composite 内のオブジェクトを操作する。

A.7.4 関連するパターン

1) Chain Of Responsibility パターン

親子関係にあるオブジェクト間のリンクは、Chain Of Responsibility パターンでしばしば使われる。

2) Decorator パターン

しばしば Composite パターンとともに使われる。decorator と composite を同時に使う場合、通常、これらは共通の親クラスを持つ。そのため decorator は、Add、Remove、GetChild のようなオペレーションで Component クラスのインタフェースをサポートしなければならなくなる。

3) Flyweight パターン

このパターンにより、component を共有できるようになる。しかし、共有されるオブジェクトは親オブジェクトを参照できなくなる。

4) Iterator パターン

composite を走査するために使われる。

5) Visitor パターン

Composite クラスや Leaf クラスに分散しているオペレーションや振る舞いを局所化する。

A.8 「DECORATOR」

A.8.1 目的

オブジェクトに責任を動的に追加する。Decorator パターンは、サブクラス化よりも柔軟な機能拡張方法を提供する。

A.8.2 適用可能性

- 個々のオブジェクトに責任を動的、かつ透明に（すなわち、他のオブジェクトには影響を与えないように）追加する場合。
- 責任を取りはずす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非実用的な場合。非常に多くの独立した拡張が起こり得ることがある。このような場合、サブクラス化によりすべての組み合わせの拡張に対応しようとすると、莫大な数のサブクラスが必要になるだろう。また、クラス定義が隠ぺいされている場合や入手できない場合にも、このパターンを利用できる。

A.8.3 パターン構造

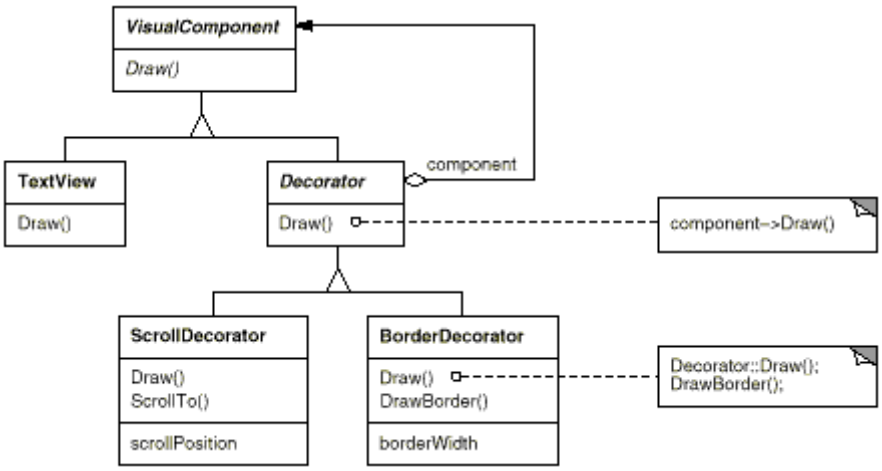

図 A.8.1 DECORATOR 構造

表 A.8.1 モジュール一覧表

モジュール名称	機能
Component クラス	・ 責任を動的に追加できるようになっているオブジェクトのためのインタフェースを定義する
ConcreteComponent クラス	・ 責任を追加できるようになっているオブジェクト（component）を定義する
Decorator クラス	・ component または decorator への参照を保持し、また Component クラスのインタフェースと一致したインタフェースを定義する
ConcreteDecorator クラス	・ component に責任を追加するオブジェクト（decorator）を定義する

A.8.4 関連するパターン

1) Adapter パターン

decorator は adapter とは異なる。decorator はそれが装飾しているオブジェクトの責任を変えるだけで、インタフェースまでは変えない。adapter はオブジェクトにまったく新しいインタフェースを与える。

2) Composite パターン

decorator は component を 1 つしか持たない退化した composite と見なすことができる。しかし、decorator は新たな責任を追加する。オブジェクトの集約が目的ではない。

3) Strategy パターン

decorator はオブジェクトの殻を変える。一方、strategy はオブジェクトの中身を変える。オブジェクトを変化させる方法には、この 2 通りが考えられる。

A.9 「FACADE」

A.9.1 目的

サブシステム内に存在する複数のインタフェースに 1 つの統一インタフェースを与える。Facade パターンはサブシステムの利用を容易にするための高レベルインタフェースを定義する。

A.9.2 適用可能性

- 複雑なサブシステムに単純なインタフェースを提供したい場合。サブシステムは発展するにつれて、より複雑になっていく。たいていのパターンは、適用するとたくさんの小さなクラスが導入されることになる。それにより、サブシステムの再利用性が増し、カスタマイズも容易になる。しかしその一方で、サブシステムをカスタマイズする必要のないクライアントにとっては、そのサブシステムの利用が難しくなる。このような場合に、facade はサブシステムの単純なデフォルトのビューを提供してくれる。ほとんどのクライアントにとってはこのデフォルトのビューだけで十分である。サブシステムをカスタマイズする必要のあるクライアントだけが、facade を越えてサブシステムの内部まで見ることになる。
- ある抽象を実装しているクラスとクライアントの間に多くの依存関係がある場合。あるサブシステムをクライアントや他のサブシステムから切り離して、独立性や移植性を高めるために facade を導入する。
- サブシステムを階層化したい場合。各階層の各サブシステムへの入り口を定義するために facade を使う。複数のサブシステムが依存し合っている場合、それらのサブシステムが互いに facade を通してのみやりとりを行うようにすれば、それらの依存関係を単純にすることができる。

A.9.3 パターン構造

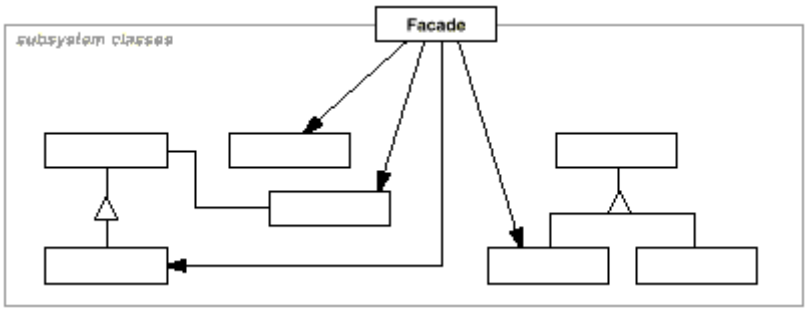

図 A.9.1 FACADE 構造

表 A.9.1 モジュール一覧表

モジュール名称	機能
Façade クラス	・ サブシステム内のどのクラスがクライアントからの要求に対して責任を負っているのかを知っている。 ・ クライアントからの要求をサブシステム内の適当なオブジェクトに委譲する。
サブシステム内のクラス	・ サブシステムの機能を実装している。 ・ Facade オブジェクトにより割り当てられた仕事を処理する。 ・ Facade オブジェクトについては何も知らない。つまり、サブシステム内のクラスは Facade オブジェクトへの参照を保持しない

A.9.4 関連するパターン

1) Abstract Factory パターン

サブシステムとは独立した方法でサブシステム内のオブジェクトを生成するインタフェースを提供するために、Facade パターンと一緒に利用することができる。また、プラットフォームに特化したクラスを隠ぺいするという点で、Facade パターンに対する代替案として利用することもできる。

2) Mediator パターン

既存のクラスの機能を抽出しているという点で Facade パターンと似ている。しかし、Mediator パターンの目的は Colleague オブジェクト間のやりとりを抽出することであり、それらの機能を 1 か所に集中させて、Colleague オブジェクトには機能を持たせないようにする。Colleague オブジェクトは mediator の存在を知っており、Colleague オブジェクト同士で直接やりとりをする代わりに mediator とやりとりをする。一方 facade は、サブシステム内のオブジェクトの利用を容易にするために、そのインタフェースを抽出するだけである。facade は新たな機能を定義しないし、サブシステム内のクラスは facade の存在を知らない。

3) Singleton パターン

通常、facade には唯一性が要求される。したがって、facade にはしばしば Singleton パターンが適用される。

A.10　「FACTORY METHOD」

A.10.1　目的

オブジェクトを生成するときのインタフェースだけを規定して、実際にどのクラスをインスタンス化するかはサブクラスが決めるようにする。Factory Method パターンは、インスタンス化をサブクラスに任せる。

A.10.2　適用可能性

- クラスが、生成しなければならないオブジェクトのクラスを事前に知ることができない場合。
- サブクラス化により、生成するオブジェクトを特定化する場合。
- クラスが責任をいくつかのサブクラスの中の1つに委譲するときに、どのサブクラスに委譲するのかに関する知識を局所化したい場合。

A.10.3　パターン構造

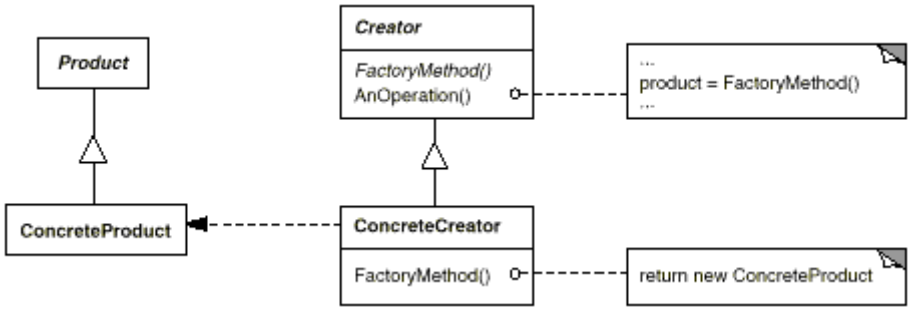

図 A.10.1 FACTORY METHOD **構造**

表 A.10.1 モジュール一覧表

モジュール名称	機能
Product クラス	・ factory method が生成するオブジェクトのインタフェースを定義する。
ConcreteProduct クラス	・ Product クラスのインタフェースを実装する。
Creator クラス	・ Product 型のオブジェクトを返す factory method を宣言する。また、ある Concrete Product オブジェクトを返すように factory method の実装をデフォルトで定義することもある。 ・ Product のオブジェクトを生成するために factory method を呼び出す。
ConcreteCreator クラス	・ ConcreteProduct クラスのインスタンスを返すように、factory method をオーバーライドする。

A.10.4 関連するパターン

1) Abstract Factory パターン

factory method を使って実装されることが多い。

2) Template Method パターン

factory method は通常、template method の中で呼ばれる。Document クラスの例では、NewDocument オペレーションが template method である。

3) Prototype パターン

Prototype パターンにより、Creator クラスをサブクラス化する必要はなくなるが、代わりに Product クラスにしばしば初期化オペレーションが必要になる。一方、Factory Method パターンでは初期化オペレーションは必要ない。

A.11「FLYWEIGHT」

A.11.1　目的

多数の細かいオブジェクトを効率よくサポートするために共有を利用する。

A.11.2　適用可能性

- アプリケーションが非常に多くのオブジェクトを利用する。
- 大量のオブジェクトのために、メモリ消費コストが高くつく。
- オブジェクトの状態を構成するほとんどの情報を extrinsic にできる。
- extrinsic 状態が取り除かれれば、オブジェクトのグループの多くを比較的少数の共有オブジェクトに置き換えることができる。
- アプリケーションがオブジェクトの同一性に依存しない。flyweight オブジェクトは共有されている可能性があるため、概念的には異なるオブジェクトなのだが、同一性テストの結果は真になってしまうことがあるだろう。

※Flyweight パターンの有効性は、それがどこで、どのように利用されるかに大きく依存する。上記のすべてがあてはまるときに Flyweight パターンを適用するとよい。

A.11.3 パターン構造

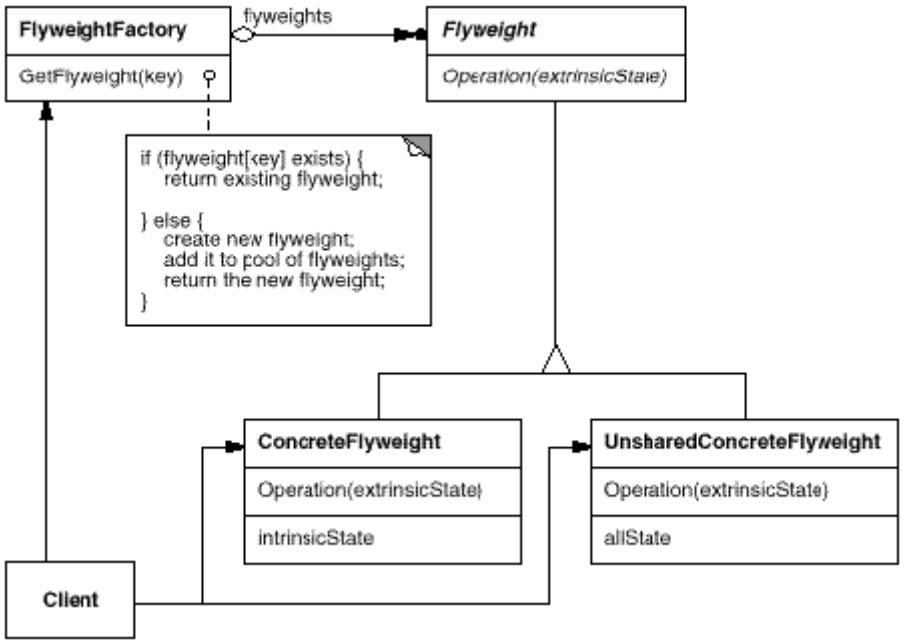

図 A.11.1 FLYWEIGHT 構造

表 A.11.1 モジュール一覧表

モジュール名称	機能
Flyweight クラス	・ flyweight が extrinsic 状態を受け取り、それに基づいて行動できるようにするためのインタフェースを定義する。
ConcreteFlyweight クラス	・ Flyweight クラスのインタフェースを実装し、intrinsic 状態があればその格納場所を追加する。ConcreteFlyweight オブジェクト（flyweight に該当する）は共有可能でなければならない。ConcreteFlyweight オブジェクトが格納する状態はすべて intrinsic でなければならない。すなわち、それは文脈に依存してはならない。
UnsharedConcreteFlyweight クラス	・ Flyweight のサブクラスのすべてが共有可能になっている必要はない。Flyweight クラスのインタフェースは共有を可能にしてはいるが、共有を強制するわけではない。UnsharedConcreteFlyweight オブジェクトは、flyweight からなるオブジェクト構造において階層を形成し、ConcreteFlyweight オブジェクトを子として管理する役目を持つのが一般的である。
FlyweightFactory クラス	・ lyweight を生成し、管理する。 ・ lyweight が正しく共有されることを保証する。Client オブジェクトが flyweight を要求したとき、FlyweightFactory オブジェクトはそのインスタンスが存在する場合にはそれを与え、存在しない場合には新たに生成する。
Client クラス	・ 1 つ、あるいは複数の flyweight への参照を保持する。 ・ lyweight の extrinsic 状態を計算するか、または格納する。

A.11.4 関連するパターン

1) Composite パターン

Flyweight パターンは、しばしば Composite パターンと組み合わされて、共有 Leaf ノードを持つ有向非循環グラフとして論理的な階層構造を実装するために使われる。

2) State パターン、Strategy パターン

これらのパターンを flyweight として実装するのは、しばしば最良の実装となる。

A.12 「INTERPRETER」

A.12.1 目的

言語に対して、文法表現と、それを使用して文を解釈するインタプリタを一緒に定義する。

A.12.2 適用可能性

- 文法が単純な場合。文法が複雑な場合には文法を表現するクラス階層が大きくなり、管理できなくなる。そのような場合には、パーザジェネレータのようなツールの方が適している。パーザジェネレータでは、アブストラクト・シンタックスツリーを構築せずに表現を解釈するので、メモリと処理時間を節約することができる。
- 効率が重要な関心事ではない場合。もっとも効率的なインタプリタは、通常は、構文解析木を直接解釈するのではなく、最初に別の形に変換するように実装されている。たとえば、正規表現はしばしば状態機械に変換される。しかし、そのような場合でも、Interpreter パターンを適用して変換を実装することができる。

A.12.3 パターン構造

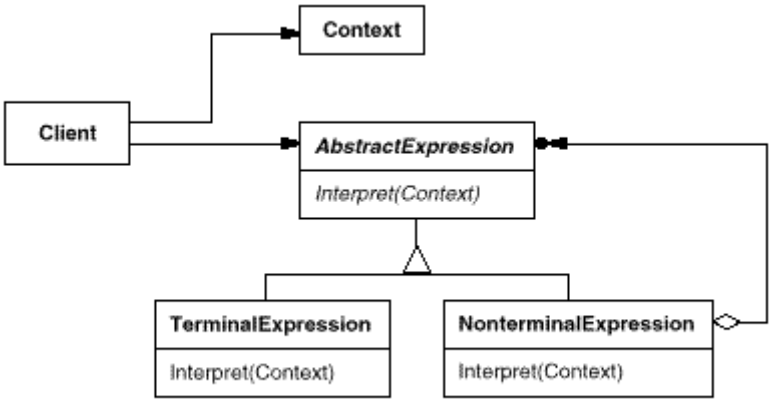

図 A.12.1 INTERPRETER 構造

表 A.12.1 モジュール一覧表

モジュール名称	機能
AbstractExpression クラス	・ アブストラクト・シンタックスツリーのすべてのノードに共通な抽象化された Interpret オペレーションを宣言する。
TerminalExpression クラス	・ 文法中の終端記号に関する Interpret オペレーションを実装する。 ・ 文中の 1 つ 1 つの終端記号について、インスタンスが生成される。
NonterminalExpression クラス	・ 文法中の R::=R1R2. . . Rn の形で表された規則 1 つ 1 つについて、このクラスが定義される。 ・ R1 から Rn の各記号について、AbstractExpression の型のインスタンス変数を保持する。 ・ 文法中の非終端記号について Interpret オペレーションを実装する。Interpret オペレーションは、典型的には、R1 から Rn で表された変数上で、自身を再帰的に呼び出していく。
Context クラス	・ インタプリタにとって、グローバルな情報を含んでいる。
Client クラス	・ 文法により定められた言語において、文を表現するアブストラクト・シンタックスツリーを作る（または、与えられる）。アブストラクト・シンタックスツリーは、NonterminalExpression クラスや TerminalExpression クラスのインスタンスから組み立てられる。 ・ Interpret オペレーションの呼び出しを行う。

A.12.4 関連するパターン

1) Composite パターン

アブストラクト・シンタックスツリーは、Composite パターンのインスタンスである。

2) Flyweight パターン

アブストラクト・シンタックスツリー内で終端記号を共有する方法を示している。

3) Iterator パターン

インタプリタが構造内を走査する際に Iterator オブジェクトを使うことができる。

4) Visitor パターン

アブストラクト･シンタックスツリー内の各ノードの振る舞いを 1 つのクラスにまとめて持たせるために、使うことができる。

A.13 「ITERATOR」

A.13.1 目的

集約オブジェクトが基にある内部表現を公開せずに、その要素に順にアクセスする方法を提供する。

A.13.2 適用可能性

- 集約オブジェクトの内部表現を公開せずに、その中にあるオブジェクトにアクセスしたい場合。
- 集約オブジェクトに対して、複数の走査をサポートしたい場合。
- 異なる集約構造の走査に対して、単一のインタフェースを提供したい（すなわち、ポリモルフィックな iteration をサポートしたい）場合。

A.13.3 パターン構造

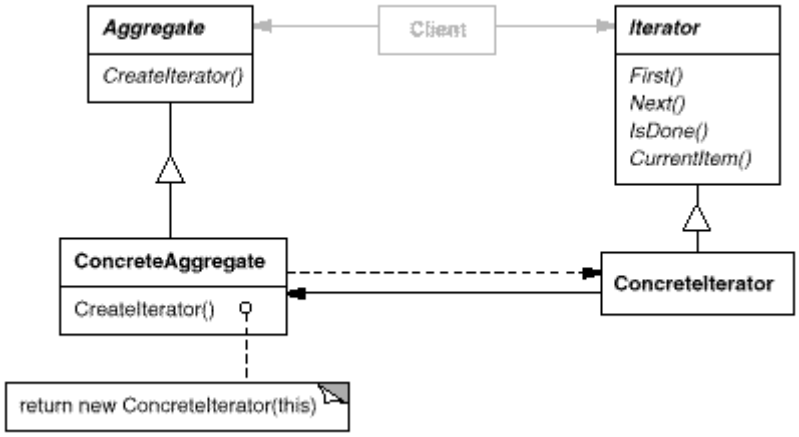

図 A.13.1 ITERATOR 構造

表 A.13.1 モジュール一覧表

モジュール名称	機能
Iterator クラス	・ 要素にアクセスしたり走査したりするためのインタフェースを定義する。
ConcreteIterator クラス	・ Iterator クラスで定義したインタフェースを実装する。 ・ Aggregate オブジェクトの走査の際に、カレント要素を記録する。
Aggregate クラス	・ Iterator オブジェクトを生成するためのインタフェースを定義する。
ConcreteAggregate クラス	・ Aggregate クラスで定義したインタフェースに対して、適切な ConcreteIterator クラスのインスタンスを生成して返すように実装する。 （ConcreteAggregate クラスのインスタンスを、総称して aggregate と呼ぶ）。

A.13.4 関連するパターン

1) Composite パターン

iterator は、Composite のような再帰的な構造に対して適用されることがある。

2) Factory Method パターン

ポリモルフィックな iterator では、Iterator のサブクラスの中から適切なクラスをインスタンス化するために、factory method を使う。

3) Memento パターン

Iterator パターンとともに使われることがある。iterator は、iteration の状態を把握するために memento を使うことができる。この場合、iterator は memento を内部に保持する。

A.14　「MEDIATOR」

A.14.1　目的

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。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する。それにより、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る。

A.14.2　適用可能性

- しっかりと定義されているが複雑な方法で、オブジェクトの集まりが通信する場合。その結果、オブジェクト間の依存関係が構造化できず、理解が難しい。
- あるオブジェクトが他の多くのオブジェクトに対して参照を持ち、それらと通信するので、それを再利用するのが難しい場合。
- いくつかのクラス間に分配された振る舞いを、できるだけサブクラス化を行わずにカスタマイズしたい場合。

2.14.3　パターン構造

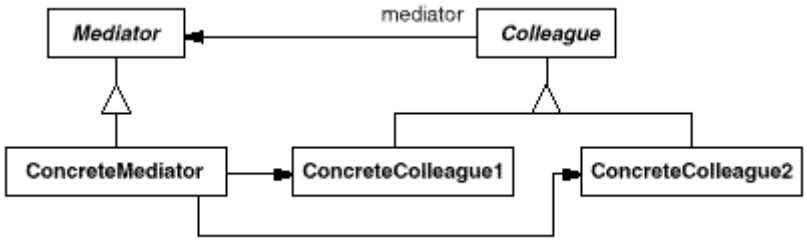

図 A.14.1 MEDIATOR 構造

表 A.14.1 モジュール一覧表

モジュール名称	機能
Mediator クラス	・ Colleague オブジェクトと通信するためのインタフェースを定義する。
ConcreteMediator クラス	・ Colleague オブジェクト間の調整を図ることにより、協調的な振る舞いを実装する。 ・ Colleague オブジェクトを保持している。
Colleague クラス	・ 各 Colleague クラスが Mediator オブジェクトを知っている。 ・ 各 Colleague オブジェクトは、他の Colleague オブジェクトと通信しなければならないときには、常に Mediator オブジェクトを介して行う。

A.14.4 関連するパターン

1) Facade パターン

Facade パターンは、その目的がより便利なインタフェースを提供するためにサブシステムを構成しているオブジェクトを抽象化するという点で、Mediator パターンとは異なっている。そのプロトコルは一方向である。すなわち、Facade オブジェクトはサブシステムを構成しているオブジェクトに対して要求を出すが、その逆はない。それとは対照的に、Mediator パターンは、Colleague オブジェクトが提供していない、または提供できない協調的な振る舞いを可能にし、そのプロトコルは双方向である。

2) Observer パターン

Colleague オブジェクトは、Observer パターンを使って mediator と通信することができる。

A.15 「MEMENTO」

A.15.1 目的

カプセル化を破壊せずに、オブジェクトの内部状態を捉えて外面化しておき、オブジェクトを後にこの状態に戻すことができるようにする。

A.15.2 適用可能性

- オブジェクトの状態（の一部）のスナップショットを、後にオブジェクトをその状態に戻すことができるように、セーブしておかなければならない場合。
- 状態を得るための直接的なインタフェースが、実装の詳細を公開し、オブジェクトのカプセル化を破壊する場合。

A.15.3 パターン構造

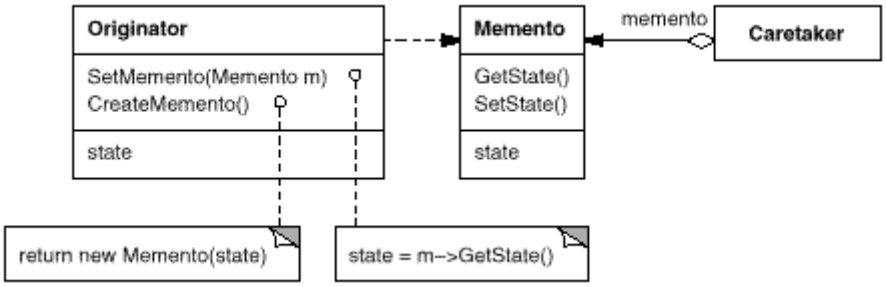

図 A.15.1 MEMENTO 構造

表 A.15.1 モジュール一覧表

モジュール名称	機能
Memento クラス	・ Originator オブジェクトの内部状態を保存する。Memento オブジェクトは、Originator オブジェクトの内部状態のうち必要な部分だけを Originator オブジェクトの判断により保存する。 ・ Originator 以外のオブジェクトによるアクセスから保護する。Memento クラスには 2 つのインタフェースが効果的に備わっている。Caretaker クラスには、Memento クラスの narrow インタフェースが見えるようになっている（それは Memento オブジェクトを他のオブジェクトに渡すことができるだけである)。それとは対照的に、Originator クラスには wide インタフェースが見えるようになっている。すなわち、そのインタフェースにより、Originator オブジェクトを前の状態に戻すために必要なすべてのデータにアクセスできるようになっている。理想的には、Memento オブジェクトを生成した Originator オブジェクトだけが Memento オブジェクトの内部構造にアクセスすることを許されるようにする。
Originator クラス	・ 内部状態のスナップショットを入れておくために Memento オブジェクトを生成する。 ・ 内部状態を元に戻すために Memento オブジェクトを使う。
Caretaker クラス	・ Memento オブジェクトを保管する責任がある。 ・ Memento オブジェクトの内容を操作したり調べたりすることはない。

A.15.4 関連するパターン

1) Command パターン

command は、取り消し可能なオペレーションのために状態を保存しておくのに memento を使うことができる。

2) Iterator パターン

前に述べたように、memento を iteration のために使うことができる。

A.16　「OBSERVER」

A.16.1　目的

あるオブジェクトが状態を変えたときに、それに依存するすべてのオブジェクトに自動的にそのことが知らされ、また、それらが更新されるように、オブジェクト間に一対多の依存関係を定義する。

A.16.2　適用可能性

- 抽象化により、2つの面が、一方が他方に依存しているという形で現れる場合。これらの面をそれぞれ別々のオブジェクトにカプセル化することにより、それらを独立に変更したり、再利用することが可能になる。
- 1つのオブジェクトを変化させるときに、それに伴いその他のオブジェクトも変化させる必要があり、しかも変化させる必要があるオブジェクトを固定的に決められない場合。
- オブジェクトが、他のオブジェクトに対して、それがどのようなものなのかを仮定せずに通知できるようにする場合。別の言い方をすると、これらのオブジェクトを密に結合したくない場合。

A.16.3　パターン構造

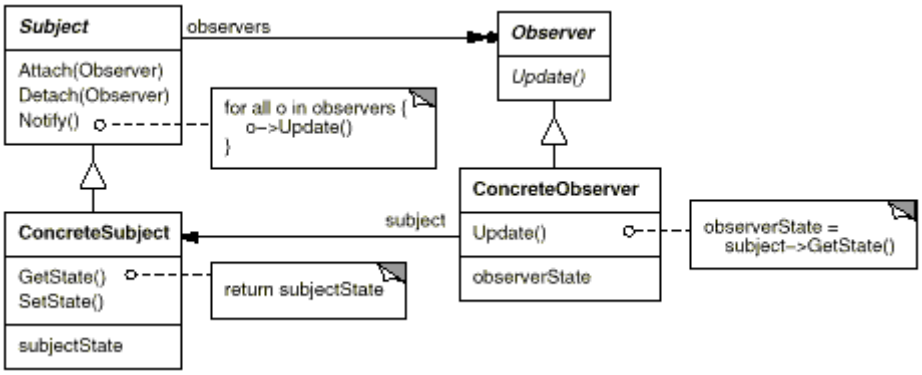

図 A.16.1 OBSERVER 構造

表 A.16.1 モジュール一覧表

モジュール名称	機能
Subject クラス	・ observer を知っている。任意の数の observer が subject の変化に対応している。
Observer クラス	・ subject 内の変化が通知されたときのために、更新のインタフェースを定義する。
ConcreteSubject クラス	・ ConcreteObserver オブジェクトに影響する状態を保存している。 ・ 状態が変わったときに ConcreteObserver オブジェクトに通知を送る。
ConcreteObserver クラス	・ ConcreteSubject オブジェクトへの参照を保持している。 ・ その状態を ConcreteSubject オブジェクトの状態と矛盾しないようにして保存している。 ・ その状態を ConcreteSubject オブジェクトの状態と矛盾しないようにしておくために、Observer クラスで宣言した更新のインタフェースを実装する。

A.16.4 関連するパターン

1) Mediator パターン

複雑な更新のセマンティクスをカプセル化することにより、ChangeManager オブジェクトは、subject と observer の間で mediator として働く。

2) Singleton パターン

ChangeManager オブジェクトは、自身を唯一の存在とし、グローバルにアクセスできるようにするために、Singleton パターンを使う。

A.17　「PROTOTYPE」

A.17.1　目的

生成すべきオブジェクトの種類を原型となるインスタンスを使って明確にし、それをコピーすることで新たなオブジェクトの生成を行う。

A.17.2　適用可能性

- インスタンス化されるクラスが、たとえば、ダイナミックローディングにより、実行時に明らかになる場合。
- 生成されるオブジェクトのクラス階層とパラレルな関係になる factory のクラス階層を作ることを避けたい場合。
- クラスのインスタンスが、状態の数少ない組み合わせの中の1つを取る場合。この可能な組み合わせ1つ1つに相当するインスタンスを prototype としてあらかじめ用意しておき、その複製を行う方が、毎回クラスを適当な状態でインスタンス化するよりも便利である。

A.17.3　パターン構造

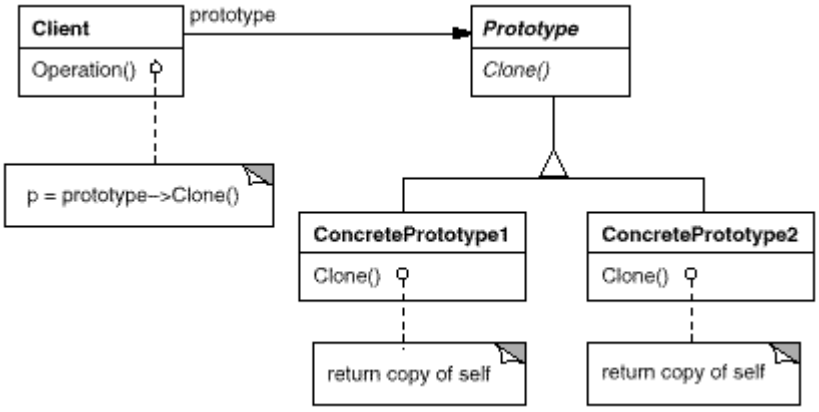

図 A.17.1 PROTOTYPE 構造

表 A.17.1 モジュール一覧表

モジュール名称	機能
Prototype クラス	・ 複製のためのインタフェースを宣言する。
ConcretePrototype クラス	・ 自身の複製を行うためのオペレーションを実装する。
Client クラス	・ rototype に複製を依頼することで、新たなオブジェクトを生成する。

A.17.4 関連するパターン

1) Abstract Factory パターン

Prototype パターンと Abstract Factory パターンはある面で競合している。しかし、これらは一緒に使うこともできる。すなわち、Abstract Factory パターンに prototype の集合を保存しておき、その中からオブジェクトの複製を行ってそれを返すようにすることができる。

2) Composite パターン、Decorator パターン

これらのパターンを駆使している設計では、Prototype パターンも有効に使える場合が多い。

A.18　「PROXY」

A.18.1　目的

あるオブジェクトへのアクセスを制御するために、そのオブジェクトの代理、または入れ物を提供する。

A.18.2　適用可能性

- remote proxy は、別のアドレス空間にあるオブジェクトのローカルな代理を提供する。
- virtual proxy は、コストの高いオブジェクトを要求があり次第生成する。
- protection proxy は、実オブジェクトへのアクセスを制御する。オブジェクトごとに異なるアクセス権が必要なときには、この protection proxy は有用である。

A.18.3 パターン構造

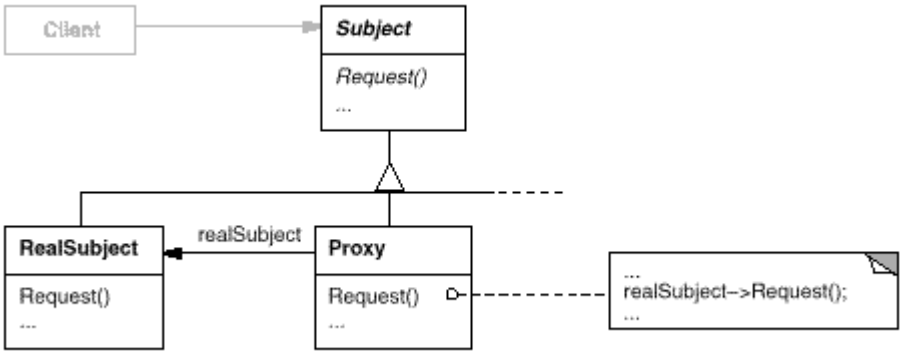

図 A.18.1 PROXY 構造

表 A.18.1 モジュール一覧表

モジュール名称	機能
Proxy クラス	・ RealSubject オブジェクトにアクセスするための参照を保持する。RealSubject クラスのインタフェースと Subject クラスのインタフェースが等しい場合には、Proxy クラスは Subject のオブジェクトを参照するようにしておくこともできる。 ・ Proxy オブジェクトを RealSubject オブジェクトと置き換えられるように、Subject クラスと同一のインタフェースを提供する。 ・ RealSubject オブジェクトへのアクセスを制御する。また、RealSubject オブジェクトの生成や消去に責任を持つこともある。
Subject クラス	・ RealSubject オブジェクトを利用できるところならばどこでも Proxy オブジェクトを利用できるように、RealSubject クラスと Proxy クラスに共通のインタフェースを定義する。
RealSubject クラス	・ Proxy オブジェクトがその代理を務めることになる実オブジェクトを定義する。

A.18.4 関連するパターン

1) Adapter パターン

adapter は、オブジェクトに異なるインタフェースを提供する。それとは対照的に、proxy は RealSubject オブジェクトと同じインタフェースを提供する。しかし、アクセス保護のために使われる proxy は、RealSubject オブジェクトならば実行するであろうオペレーションの実行を拒否するかもしれない。したがって、実際上は、proxy のインタフェースは Subject クラスのインタフェースの一部かもしれない。

2) Decorator パターン

decorator は proxy と似た形態で実装されるが、両者の目的は異なる。decorator はあるオブジェクトに 1 つ、または複数の責任を追加する。一方 proxy は、あるオブジェクトへのアクセスを制御する。proxy が decorator のように実装される程度は、proxy の種類により異なる。protection proxy は decorator とまったく同じように実装されるかもしれない。一方、remote proxy は RealSubject オブジェクトへの直接参照を持たず、“ホスト ID とそのホスト上でのローカルアドレス”などの間接参照だけを持つであろう。virtual proxy は、生成時にはファイル名などの間接参照のみを持つが、最後には直接参照を手に入れて使うようになる。

A.19 「SINGLETON」

A.19.1 目的

あるクラスに対してインスタンスが 1 つしか存在しないことを保証し、それにアクセスするためのグローバルな方法を提供する。

A.19.2 適用可能性

- クラスに対してインスタンスが1つしか存在してはならず、また、クライアントが、そのインスタンスを公開されたアクセスポイントを通してアクセスできるようにしなければならない場合。
- 唯一のインスタンスがサブクラス化により拡張可能で、また、クライアントが、拡張されたインスタンスをコードの修正なしに利用できるようにしたい場合。

A.19.3 パターン構造

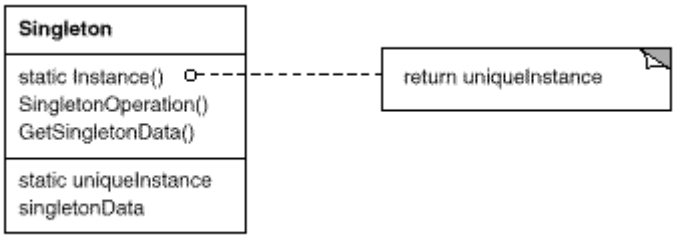

図 A.19.1 SINGLETON 構造

表 A.19.1 モジュール一覧表

モジュール名称	機能
Singleton クラス	・ Instance オペレーションを定義し、クライアントが唯一のインスタンスにアクセスできるようにする。Instance オペレーションは、Smalltalk におけるクラスメソッドや C++ における静的メンバ関数のようなクラスオペレーションである。 ・ インスタンスが 1 つしか生成されないようにする。

A.19.4 関連するパターン

1) Abstract Factory パターン、Builder パターン、Prototype パターン

これらのパターンは Singleton パターンを使って実装することができる。

A.20 「STATE」

A.20.1 目的

オブジェクトの内部状態が変化したときに、オブジェクトが振る舞いを変えるようにする。クラス内では、振る舞いの変化を記述せず、状態を表すオブジェクトを導入することでこれを実現する。

A.20.2 適用可能性

- オブジェクトの振る舞いが状態に依存し、実行時にはオブジェクトがその状態により振る舞いを変えなければならない場合。
- オペレーションが、オブジェクトの状態に依存した多岐にわたる条件文を持っている場合。この状態はたいてい1つ以上の列挙型の定数で表されており、たびたび複数のオペレーションに同じ条件構造が現れる。State パターンでは、1つ1つの条件分岐を別々のクラスに受け持たせる。これにより、オブジェクトの各状態を1つのオブジェクトとして扱うことができるようになる。

A.20.3 パターン構造

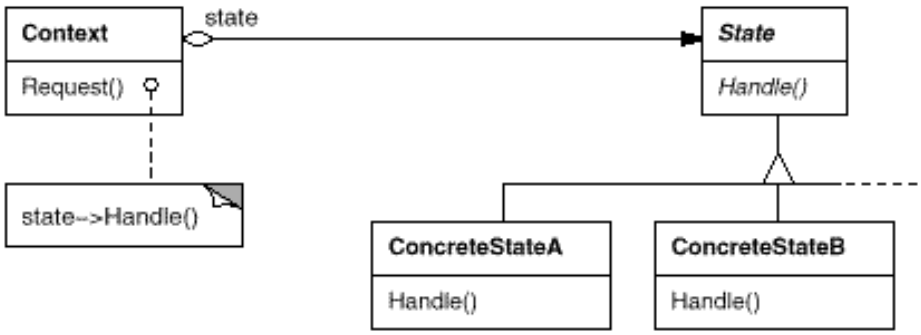

図 A.20.1 STATE 構造

表 A.20.1 モジュール一覧表

モジュール名称	機能
Context クラス	・ クライアントに必要なインタフェースを定義する。 ・ 状態を表す ConcreteState クラスのインスタンスを保持する。
State クラス	・ Context クラスの個々の状態に関する振る舞いをカプセル化するためのインタフェースを定義する。
ConcreteState クラス	・ Context クラスの 1 つの状態に関する振る舞いが実装される。

A.20.4 関連するパターン

1) Flyweight パターン

いつどのように ConcreteState オブジェクトが共有されるのかを説明している。

2) Singleton パターン

ConcreteState オブジェクトは、Singleton パターンにあてはまることがある。

A.21 「STRATEGY」

A.21.1 目的

アルゴリズムの集合を定義し、各アルゴリズムをカプセル化して、それらを交換可能にする。Strategy パターンを利用することで、アルゴリズムを、それを利用するクライアントからは独立に変更することができるようになる。

A.21.2 適用可能性

- 関連する多くのクラスが振る舞いのみ異なっている場合。Strategy パターンは、多くの振る舞いの中の1つでクラスを構成する方法を提供する。
- 複数の異なるアルゴリズムを必要とする場合。たとえば、空間と時間のトレードオフを反映する複数のアルゴリズムを定義する場合が考えられる。このとき、複数のアルゴリズムをクラス階層として実装していく際に、Strategy パターンを利用することができる。
- アルゴリズムが、クライアントが知るべきではないデータを利用している場合。Strategy パターンを利用することにより、複雑でアルゴリズムに特有なデータ構造を公開するのを避けることができる。
- クラスが多くの振る舞いを定義しており、これらがオペレーション内で複数の条件文として現れている場合。このとき、多くの条件文を利用する代わりに、条件分岐後の処理を Strategy クラスに移し換える。

A.21.3 パターン構造

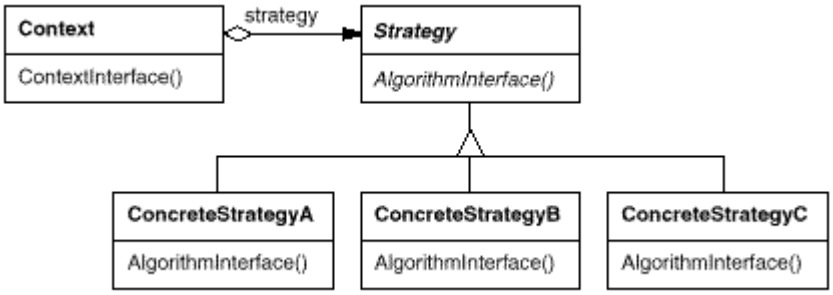

図 A.21.1 STRATEGY 構造

表 A.21.1 モジュール一覧表

モジュール名称	機能
Strategy クラス	・ サポートするすべてのアルゴリズムに共通のインタフェースを宣言する。Context クラスは、ConcreteStrategy クラスにより定義されるアルゴリズムを呼び出すためにこのインタフェースを利用する。
ConcreteStrategy クラス	・ Strategy クラスのインタフェースを利用して、アルゴリズムを実装する。
Context クラス	・ ConcreteStrategy オブジェクトを備えている。 ・ Strategy のオブジェクトに対する参照を保持する。

A.21.4 関連するパターン

1) Flyweight パターン

ConcreteStrategy オブジェクトは、有効な flyweight になることがある。

A.22 「TEMPLATE METHOD」

A.22.1 目的

１つのオペレーションにアルゴリズムのスケルトンを定義しておき、その中のいくつかのステップについては、サブクラスでの定義に任せることにする。Template Method パターンでは、アルゴリズムの構造を変えずに、アルゴリズム中のあるステップをサブクラスで再定義する。

A.22.2 適用可能性

- アルゴリズムの不変な部分をまず実装し、振る舞いが変わり得る部分の実装はサブクラスに残しておく場合。
- 同じコードがいたるところに現れることがないように、サブクラス間で共通の振る舞いをする部分は抜き出して、これを共通のクラスに局所化する場合。まず、既存のコードにおける相違点を識別し、次にその相違点を新しいオペレーションに分離する。最後に、既存のコードを、その相違点については新しいオペレーションを呼び出すようにした template method で置き換える。
- サブクラスの拡張を制御する場合。特定の時点で“hook”operationを呼び出すテンプレートメソッドを定義することができる。それにより、このポイントでのみ拡張が許されることになる。

A.22.3 パターン構造

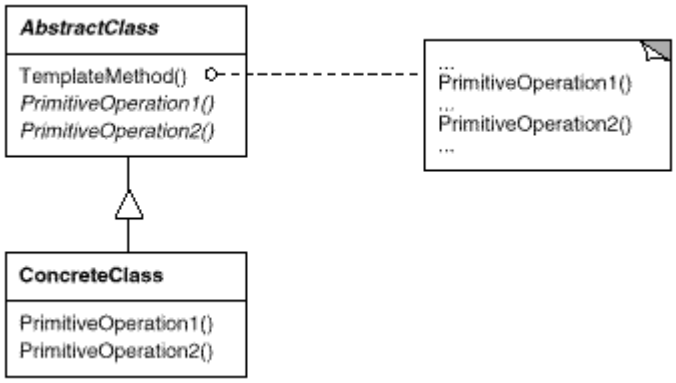

図 A.22.1 TEMPLATE METHOD 構造

表 A.22.1 モジュール一覧表

モジュール名称	機能
AbstractClass クラス	・ 抽象化された primitive operation を定義する。このオペレーションは具象サブクラスで定義され、アルゴリズムの各ステップを実装することになる。 ・ アルゴリズムのスケルトンを定義する template method を実装する。template method は、AbstractClass クラスで定義されるオペレーションやその他のオブジェクトのオペレーションと同様に、primitive operation を呼び出す。
ConcreteClass クラス	・ アルゴリズムのステップを特有の形で実行するために、primitive operation を実装する。

A.22.4 関連するパターン

1) Factory Method パターン

template method により呼び出されることがある。

2) Strategy パターン

template method では、アルゴリズムの一部を変更するために継承を利用している。それに対して Strategy パターンでは、アルゴリズム全体を変更するために委譲を利用している。

A.23 「VISITOR」

A.23.1 目的

あるオブジェクト構造上の要素で実行されるオペレーションを表現する。Visitor パターンにより、オペレーションを加えるオブジェクトのクラスに変更を加えずに、新しいオペレーションを定義することができるようになる。

A.23.2 適用可能性

- オブジェクト構造にインタフェースが異なる多くのクラスのオブジェクトが存在し、これらのオブジェクトに対して、各クラスで別々に定義されているオペレーションを実行したい場合。
- 関連のない異なるオペレーションをオブジェクト構造の中のオブジェクトに対して実行する必要があり、さらに、これらのオペレーションをクラスに持たせることで、クラスを“汚くする”ことを避けたい場合。visitor を利用すれば関連するオペレーションを1つのクラスの中に定義するので、それらをまとめておくことができるようになる。オブジェクト構造が多くのアプリケーションで共有されるときには、オペレーションをそれらのアプリケーションで共通に使うことができるように Visitor パターンを使用する。
- オブジェクト構造を定義するクラスはほとんど変わらないが、その構造に新しいオペレーションを定義することがしばしば起こる場合。オブジェクト構造のクラスを変更する場合には、すべての visitor のインタフェースを再定義する必要があり、潜在的にコストは高くつく。もしオブジェクト構造のクラスを変更することがしばしばあるならば、それらのクラスにオペレーションを定義しておく方がよい。

A.23.3 パターン構造

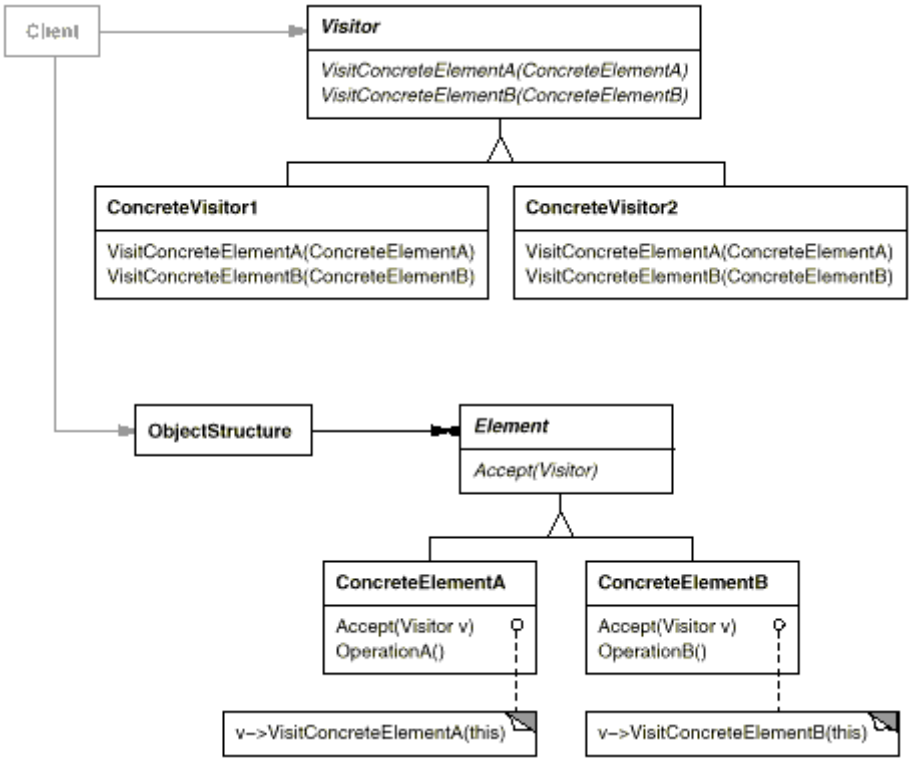

図 A.23.1 VISITOR 構造

表 A.23.1 モジュール一覧表

モジュール名称	機能
Visitor クラス	・ オブジェクト構造内の各 ConcreteElement クラスのために、Visit オペレーションを宣言する。オペレーションの名前とシグニチャによって、Visit 要求を visitor に対して送ったクラスを識別する。これにより、操作対象となる要素がどの具象クラスのオブジェクトなのかを、visitor がはっきり定めることができるようになる。そして、visitor は特定のインタフェースを通じて直接その要素にアクセスできるようになる。
ConcreteVisitor クラス	・ Visitor クラスで宣言された各オペレーションを実装する。各オペレーションは、オブジェクト構造内の対応するクラスに対して定義されたアルゴリズムを実装する。ConcreteVisitor クラスは、アルゴリズムに対してコンテキストオブジェクトを与え、そこでのローカルな状態を格納しておく。構造を走査していく過程で、状態に結果が蓄積されていくことがしばしばある。
Element クラス	・ 要素の抽象クラス。 ・ 引数として visitor をとる Accept オペレーションを定義する。
ConcreteElement クラス	・ 要素の具象クラス。 ・ 引数として visitor をとる Accept オペレーションを実装する。
ObjectStructure クラス	・ 要素を列挙することができる。 ・ 要素に対するオペレーションを visitor に任せるためのハイレベルなインタフェースを提供することもある。

A.23.4 関連するパターン

1) Composite パターン

Visitor パターンは、Composite パターンで定義されるオブジェクト構造上にオペレーションを適用するために使うことができる。

2) Interpreter パターン

Visitor パターンを言語の解釈のために適用してもよい。

参考文献

[1]「Java 言語で学ぶデザインパターン」
著者　：結城 浩
出版社：ソフトバンクパブリッシング　２００１年

[2]「オブジェクト指向における再利用のためのデザインパターン　改訂版」
著者　：Erich Gamma, Richard Helm,Ralph Johnson, John Vilissides
本位田　真一　吉田　和樹　監訳
出版社：ソフトバンクパブリッシング　１９９９年

[3]「ＵＭＬプレス　Vol.1」
出版社：技術評論社

[4]「実践ＵＭＬ」
著者　：Craig Larman
今野 睦　　依田 智夫　　監訳
出版社：ピアソン・エデュケーション　１９９８年

参考ＵＲＬ

[5]「Java プログラマーのための UML」
http://objectclub.esm.co.jp/UMLforJavaProgrammers/UMLforJavaProgrammers.html

[6]「デザインパターンの骸骨たち」
http://www002.upp.so-net.ne.jp/ys_oota/mdp/