第17章　GRASP: 責任駆動のオブジェクト設計

責任を理解することがオブジェクト指向設計を成功させる鍵である

17.1　UML と設計の原則

ソフトウェア開発にとって不可欠の設計ツールは、設計原則についての十分な知識を備えて取り組む姿勢です。

17.2　オブジェクト設計：例の入力、アクティビティ、出力

17.2.1　オブジェクト設計の入力となるのは何か

「プロセス」の入力となるものから考えてみましょう。 NextGen POS プロジェクトであれば、以下の項目が考えられます。

最初の、2日間の要件ワークショップが終了している。

主任アーキテクトと経営サイドが、タイムボックスが3週間に設定された最初の反復において、 Process Sale のシナリオをいくつか実装しテストすることに同意している。

20個のユースケースのうち3個—— Process Sale ユースケースを含め、最もアーキテクチャ的に重要でビジネスの価値が高いこれらのユースケースが、詳細に分析された。（UP は、反復型手法の典型として、プログラミングを始める前に、要件の10%〜20%を分析するように勧めている。）

補助仕様書、用語集、問題領域モデルという、ユースケース以外の成果物の作成が着手された。

プログラミングによって、例えば Java Swing の UI がタッチスクリーンでどうさするかどうかのような、非常に重要な技術上の問題が解決された。

主任アーキテクトが、 UML パッケージ図を使って、大まかな論理的アーキテクチャについてのアイディアをいくつか描いた。

成果物の入力にはほかに、変更を加えようとしている既存システムの設計文書も含める場合があります。既存のコードから UML パッケージ図へリバースエンジニアリングを実行することも、大局的な論理構造やいくつかのクラス図とシーケンス図を確認できて便利です。

成果物の入力となるもの、および、それらとオブジェクト設計との関係はどのようなものでしょうか。

ユースケーステキストは、ソフトウェアオブジェクトが最終的にサポートしなければならない目に見える振る舞いを定義する。

補助仕様書は、国際化のような、オブジェクトが満足しなければならない非機能的な目標を定義する。

システムシーケンス図は、システム操作メッセージを明らかにする。

用語集は、UI レイヤから送られてくるパラメタまたはデータ、データベースへ渡されるデータ、および、製品の UPC （米国統一商品コード）の正しい書式や検証など、アイテム固有の細かいロジックまたは検証に関する要件の詳細を明確にする。

操作契約はユースケーステキストを補完し、ソフトウェアオブジェクトがシステム操作で達成しなければならないことを明確にする。

UP 問題領域モデルは、ソフトウェアアーキテクチャの問題領域レイヤにおけるソフトウェアの問題領域オブジェクトの名前と属性について一部を提案する。

これらすべての成果物が必要とは限りません。UP ではすべての要素がオプションであって、リスクの低減につながるものを作成することを忘れないでください。

17.2.2　オブジェクト設計のアクティビティは何か

1つ以上の入力を得て、開発者は 1）ただちにコーディングを始める（理想的には、テストファースト開発もいっしょに行うことが望ましい）か、 2）オブジェクト設計のための UML モデリングを始めるか、または 3）CRC カードなど、他のモデリング技法を始めます。

UML モデリングを行うケースで重要なことは、UML そのものではなく、ただのテキストで記述するよりも視覚的により多くの表現を行えるような言語を用いて、視覚的にモデリングすることです。この場合はたとえば、モデリングの同じ1日のうちに、相互作用図と補完的なクラス図の両方（動的なモデリングと静的なモデリング）を描画したりします。

特に重要なのは、アクティビティの作図中（およびコーディング中）に、GRASP や GoF （Gang-of-Four）デザインパターンなど、さまざまなオブジェクト指向設計原則を適用することです。オブジェクト指向設計モデリングへの全体的なアプローチは、責任駆動設計（responsibility-driven design: RDD）のメタファに基づいており、協調するオブジェクトに責任をどのように割り当てるかを考察することが中心です。

モデリングの日には、おそらく開発チームは、むずかしく創造性の高い設計部部を小さなグループに分かれて分担し、壁のボード化ソフトウェアのモデリングツールで、2〜6時間ほど作業することになるでしょう。

火曜日には、この日も3週間のタイムボックスの反復ではまだ早い時期ですが、開発チームはモデリングをやめ、プログラマの帽子をかぶり、プログラミング前に過剰にモデリングするウォーターフォール型の精神構造に陥らないようにします。

17.2.3　出力は何か

このような分析の入力を設計中にも参照する場合のあることに注意してください。たとえば、ユースケーステキストや操作契約を読み直したり、問題領域モデルに目を通したり、UP 補助仕様書を考察し直したりすることがあります。

モデリングの日には何が作成されているでしょうか。

- オブジェクト設計に関する部分としては、コーディング前に深く考察しておきたい、設計のむずかしい部分の、UML 相互作用図、クラス図、パッケージ図

- UI のスケッチとプロトタイプ
- データベースモデル（UML データモデリングプロファイル表記法を使って。）
- レポートのスケッチとプロトタイプ

17.3　責任と責任駆動設計

ソフトウェアオブジェクトの設計や、大規模なコンポーネントの設計について考察する場合、よく行われている方法は、**責任、役割、協調**の観点に立つことです。これは、より大きなアプローチである責任駆動設計（responsibility-driven design: RDD）の一部です。

責任駆動設計においては、ソフトウェアオブジェクトを、責任をもつ存在としてとらえます。

責任は行うことの抽象化です。

UML は、責任（reponsibility）を「クラシファイアの契約または義務」と定義しています。

基本的には、責任は2つの種類に分けられます。実行（doing）と情報把握（knowing）です。

オブジェクトの実行責任には以下のものが含まれます。

- オブジェクトの生成や計算の実行など、何かをそれ自体で行う
- 他のオブジェクトのアクションを始動する
- 他のオブジェクトのアクティビティを制御し調整する

オブジェクトの情報把握責任には以下のものが含まれます。

- カプセル化されたプライベートなデータを把握する
- 関係するオブジェクトを把握する
- 導出または計算可能なものを把握する

ソフトウェアの問題領域オブジェクトでは、問題領域モデルから「情報把握」に関係のある重要な責任のインスピレーションが得られることがよくあります。問題領域モデルに属性と関連が示されているからです。たとえば、問題領域モデルの Sale クラスが time 属性をもつ場合は、表現上のギャップを低減するという目標に照らせば、ソフトウェアの Sale クラスがその時刻を把握するのが自然です。

クラスおよびメソッドへ責任を変換する際は、責任の細かさによって影響を受けます。大きな責任は数百個のクラスやメソッドに関わります。小さな責任であれば１

つだけのメソッドですむものもあります。「リレーショナルデータベースへのアクセスを提供する」ための責任であれば、サブシステムにパッケージ化された２百個のクラスや数千個のメソッドが関係するかもしれません。一方、「Sale を生成する」ための責任に関係するのは１つのクラスと１つのメソッド程度のはずです。

責任は抽象化であり、メソッドと同一ではありませんが、メソッドによって責任は満足されます。

責任駆動設計には協調という考え方も組み込まれています。責任は、単独で動作するメソッド、または他のメソッドやオブジェクトと強調して動作するメソッドを通して実装されます。たとえば、Sale クラスはその合計金額を把握する getTotal という名前のメソッドなどを何個か定義します。責任を果たすために、Sale は個々の SalesLineItem オブジェクトへ小計を問い合わせるメッセージを送るなどして、他のオブジェクトと強調します。

責任駆動設計は、オブジェクト指向ソフトウェア設計について考察するために一般的なメタファです。ソフトウェアオブジェクトを、ほかの人たちと協力して仕事を成し遂げる、責任をもった人間に似た存在として考えてください。責任駆動設計によって、オブジェクト指向設計を「責任をもったオブジェクトが協調するコミュニティ」としてとらえやすくなります。

17.4　GRASP: 基本的なオブジェクト指向設計への系統的なアプローチ

オブジェクト設計の基本を理解するために必要となる詳細な原則と推論には、名前をつけて説明することができます。GRASP の原則またはパターンは、オブジェクト設計の本質を理解しやすくし、設計にあたって行う推論を、系統立てて、合理的に、すっきりと説明のつく方法で行えるように支援します。設計原則を理解し利用するこの手法は、「責任割り当てパターン」を基礎としています。

GRASP は重要ですが、一方でこれは原則を組み立て、名前をつけるという学習支援にすぎません。いったん基本を理解したならば、特定の GRASP の用語（情報エキスパート、生成者、など）は重要でなくなります。

17.6　パターンとは

オブジェクト指向設計において、パターンとは、新しいコンテキストに適用できるように問題と解決策を記述し名前をつけたものです。多様な環境においてその解決策をどのように適用すればよいかの助言がなされていたり、影響や得失（trade-off）についての考察がなされているといっそう理想的です。

17.6.1　パターンには名前があるー重要

パターンには、情報エキスパートや抽象ファクトリなど、名前をもたせます。パターンや、設計のアイディア、原則に適切な名前をつけておくと以下のような利点があります。

- パターンの考えを分類し、理解し、記憶することが容易になる
- コミュニケーションが促進される

パターンに名前をつけ、それが広まり、しかも皆がその名前を使用することに合意すると、設計に関する込み入った話を短いセンテンスで伝えられるようになります。抽象化の利点です。

17.6.2　「新しいパターン」という矛盾

デザインパターンは、設計についての新しいアイディアを表現するものではありません。これとはまったく逆で、パターンは有効性がわかっている既存の知識、慣用句、原則を体系化しようとします。

したがって、GRASP パターンも新しい考えについては言及せず、広く使用されている基本原則の命名と体系化です。オブジェクト指向設計の専門家にとって GRASP パターンは、名称は知らなくても基礎として慣れ親しんだものであるはずです。これが重要です。

17.6.3　四人組によるデザインパターンの本

ソフトウェアの世界でパターンに名前をつけて共有するというアイディアは、1980年代半ばに、Kent Beck （XP でも高名な方です）によってもたらされました。

ただし Design Patterns は入門用の本ではありません。オブジェクト指向設計の経験やプログラミングの知識がかなりあることを前提にしており、コード例の大半は C++ で書かれています。

17.8　GRASP を使ったオブジェクト設計の簡単な例

17.8.1　生成者パターン

「オブジェクト X を生成するのは何か」。これは「実行」責任です。

わたしたちが AB324 とか ZC17 ではなく、 Square や Board という名前でソフトウェアクラスを定義するのはなぜでしょうか。その答えは、むろん表現上のギャップ

を低減するためです。これによって、UP 問題領域モデルが UP 設計モデルにつながり、問題領域の人間のメンタルモデルが、ソフトウェアアーキテクチャの問題領域レイヤにおける実現化とつながります。

名前：生成者（Creator）

問題：A を生成するのは何か

解決策（一例としてとらえるとよい）：

以下の１つ以上が真であるならば（真が多いほど、より確信が強まる）、クラス B にクラス A のインスタンスを生成する責任を割り当てる

- B が A を「含む」（contain）もしくは、合成物として A を集約する（aggregate）
- B が A を記録する（record）
- B が A を密接に使用する（closely use）
- B が A の初期化データをもっている（has the initializing data）

B および A はソフトウェアをオブジェクトを指すのであって、問題領域モデルのオブジェクトを指すのではありません。

しかし、オブジェクト指向設計に着手したばかりで、ソフトウェアクラスをまったく定義していない場合はどうすればよいでしょうか。その場合は、表現上のギャップの低減を図るために、問題領域モデルからインスピレーションを得ます。

表現上のギャップの低減にも沿った上で生成者パターンに照らすと、Board が Square を生成すると考えられます。また、Square は常に１つの Board の一部であり、Board は Square の生成と破棄を管理します。したがって、Square は Board とコンポジット集約の関連にあります。

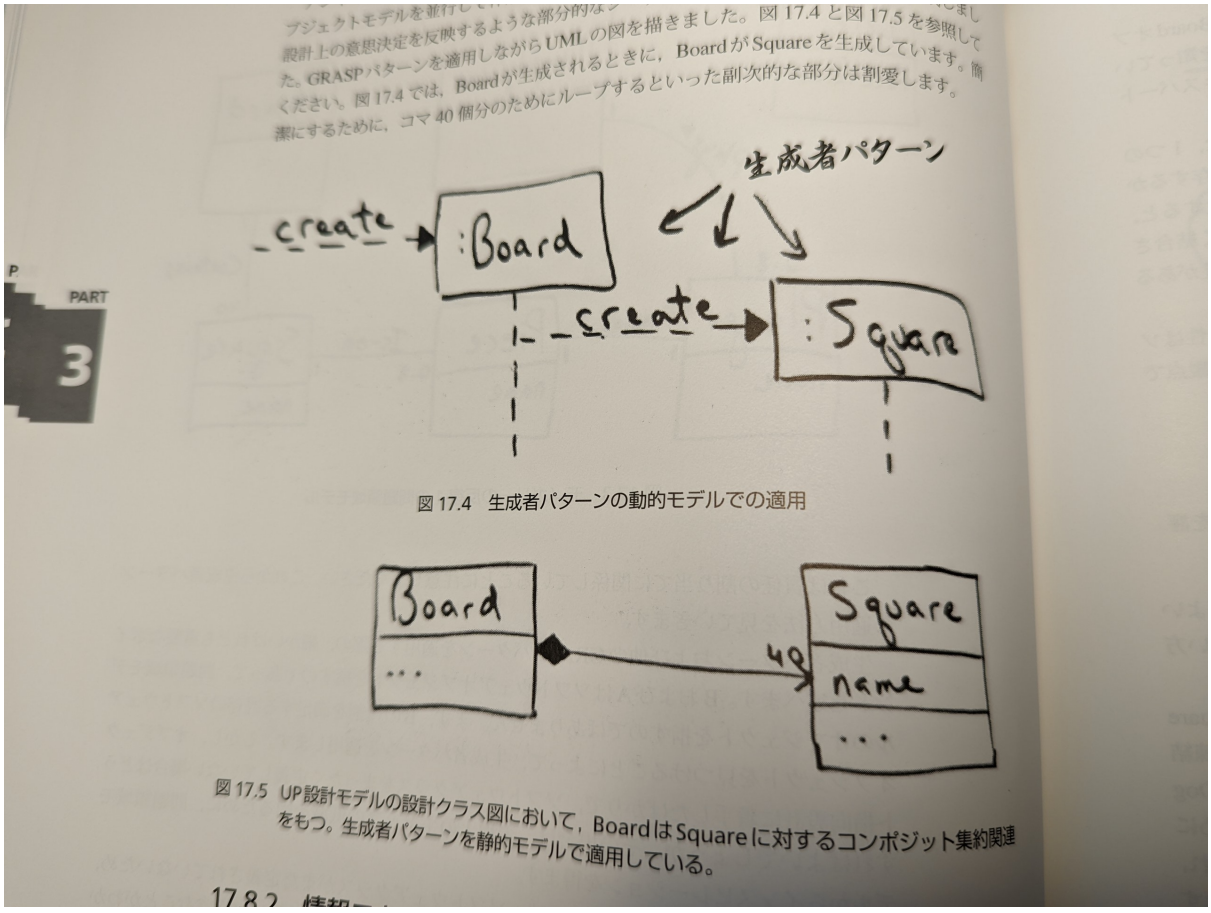


図 17.4　生成者パターンの動的モデルでの適用 / 図 17.5　UP 設計モデルの設計クラス図において、Board は Square に対するコンポジット集約関連をもつ。生成者パターンを静的モデルで適用している。

17.8.2　情報エキスパートパターン

情報エキスパート（Information Expert）パターンは、オブジェクト設計の責任割り当て原則の中でも特に基本的なものの１つです。

むろん、これは情報把握の責任ですが、エキスパートは実行の責任にも適用できます。

> **名前：**情報エキスパート（Information Expert）
>
> **問題：**責任をオブジェクトに割り当てるときの基本原則は何か
>
> **解決策（一例）：**責任の推敲に必要な情報をもっているクラスに責任を割り当てる

責任はその遂行のために情報を必要とします。情報とは、他のオブジェクトに関するものであったり、オブジェクト自体の状態であったり、オブジェクトを取り巻く環境であったり、オブジェクトが導き出せる情報であったり、さまざまです。

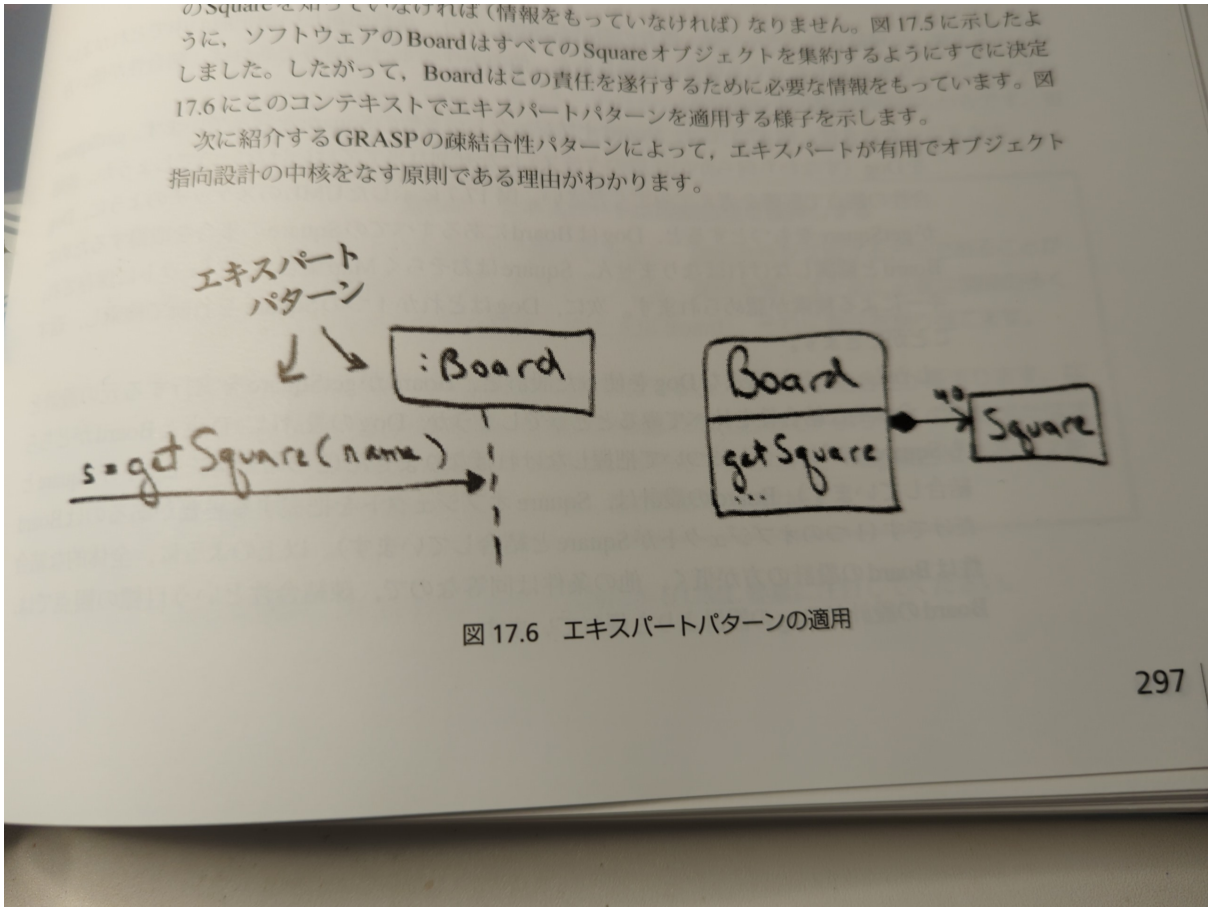


図 17.6　エキスパートパターンの適用

GRASP の疎結合性パターンによって、エキスパートが有用でオブジェクト指向設計の中核をなす原則である理由がわかります。

17.8.3　疎結合性パターン

結合（coupling）は、１つの要素が他の要素に対し、どの程度の強さで接続するか、把握するか、あるいは依存するかを表す尺度です。結合または依存が存在する場合は、依存される側の要素が変化すると、依存する側に影響が及びます。たとえば、サブクラスはそのスーパークラスに強く結合されます。オブジェクト B の操作を呼び出すオブジェクト A は、B のサービスへ結合性があるということです。

疎結合性の原則は、ソフトウェア開発のさまざまな次元で適用できます。疎結合性はソフトウェア構築における主要な目標の１つです。

名前：疎結合性（Low Coupling）

問題：変更の影響をいかに低減するか

解決策（一例）：（不必要な）結合性が低くなるように責任を割り当てる。複数の選択肢を評価する場合にもこの原則を用いる。

疎結合性パターンは、既存の設計を評価したり、複数の新しい選択肢の中でどれがよいかを評価したりする場合に利用します。他の条件がすべて同等であれば、結合性が低い方の設計を選択すべきです。

疎結合性が望ましいのはなぜでしょうか？言い換えると、わたしたちは変化の影響をなぜ低減したいと望むのでしょうか？疎結合性によって、ソフトウェアの修正に関する時間も、労力も、不具合の発生も現象する傾向があるからです。短い答えですが、ソフトウェアの構築および保守においてこれは大きな意味を持ちます。

情報エキスパートの利点に立ち戻ってみると、疎結合性を支援する方法であることがわかります。情報エキスパートパターンでは、責任を果たすために必要な情報の多くをもつオブジェクトが何かを調べ（たとえば Board）、それに責任を割り当てます。

責任を別の何か（たとえば Dog）に割り当てると、全体的な結合性が高まります。なぜならば、元の情報源あるいは起点から離れたところで、より多くの情報またはオブジェクトを共有しなければならないからです。つまり、Map 集合にあるコマを、Board の起点から離れた Dog と共有しなければなりません。

17.8.4　コントローラパターン

単純なレイヤ化アーキテクチャにも、UI レイヤと問題領域レイヤはあります。モノポリーゲームの人間のオブザーバのようなアクタが、ゲーム開始ボタンをマウスでクリックするなどの UI イベントを生成します。UI ソフトウェアオブジェクト（たとえば Java では、JFrame ウィンドウや JButton ボタンなど）は、マウスクリックのイベントに反応し、最終的にゲームを開始させなければなりません。

モデル／ビュー分離の原則に照らし、UI オブジェクトが、プレイヤの動きを計算するようなアプリケーションまたは「ビジネス」のロジックをもつべきでないことはすでに述べました。したがって、UI オブジェクトがいったんマウスイベントを捕捉した後は、要求を問題領域レイヤの問題領域オブジェクトへ委譲する（タスクを他のオブジェクトへ転送する）必要があります。

コントローラパターンは、「UI レイヤから送られるメッセージを、UI レイヤの次にあるいは UI レイヤを超えて受け取る最初のオブジェクトは何か」という問いに答えてくれます。

コントローラは、「UI レイヤとアプリケーションロジックレイヤをどのように接続するか」というオブジェクト試行設計の基本的な問いに対応します。

オブジェクト指向分析／設計手法の中には、「コントローラ」という名前を、要求の処理を受け取り、「コントロールする」アプリケーションロジックオブジェクトに与えているものがありました。

名前：コントローラ（Controller）

問題：UI レイヤからシステム操作を最初に受け取り調整する（コントロールする）オブジェクトは何か

解決策（一例）：以下のどれかに該当するオブジェクトに責任を割り当てる

- 「システム」全体、「ルートオブジェクト」、中でソフトウェアが動作するデバイス、あるいは、主要なサブシステムを表すオブジェクト（これらはどれもファサードコントローラのバリエーションである）
- 中でシステム操作が発生するユースケースシナリオを表すオブジェクト（ユースケースまたはセッションコントローラ）

オプション１：「システム」全体あるいは「ルートオブジェクト」を表す場合——MonopolyGame のようなオブジェクト

オプション１：中でソフトウェアが動作するデバイスを表す場合——このオプションは、電話や銀行の現金預け払い機（ソフトウェアクラスでいうと Phone や BankCachMachine）など、特別なハードウェアデバイスに属します。いまの例ではこれに当てはまるものはありません。

オプション２：ユースケースまたはセッションを表す場合——playGame システム操作が中で発生するユースケースは、Play Monopoly Game です。このため PlayMoopolyGameHandler のようなソフトウェアクラスが該当します（後ろに “…Handler” や “…Session” がつくのは、このオプションに該当する場合にオブジェクト指向設計で使われる慣用句です）。

オプション１の MonopolyGame クラスは、システム操作の個数が少ない場合には合理的です（高凝集性のところで得失について詳述します）。

17.8.5　高凝集性パターン

ソフトウェア設計において凝集性として知られる基本的な品質は、大まかにいうとソフトウェア要素の操作が機能的にどの程度関係しているかの度合いであり、また、ソフトウェア要素が行う仕事量の度合いでもあります。

100 個のメソッドと 2000 行のソースコードをもつオブジェクト Big は、10 個のメソッドと 200 行のソースコードしかないオブジェクト Small に比べてはるかに多くの仕事を行います。さらに、Big の 100 個のメソッドがさまざまな分野にわたる責任を負うとすると、Big は Small に比べて仕事の焦点がぼやけ、機能の凝集性が低下します。つまり、コードの量とコードの関係性がオブジェクトの凝集性です。

凝集性が悪い（低い）ということは、常にオブジェクトがそれ自体でだけ仕事をすることを意味するわけではありません。2000 行のソースコードがあれば、凝集性の

低いオブジェクトであっても、おそらくほかの多くのオブジェクトと協調しているはずです。ここで重要なことは、相互作用を進めると、結合性も密になってしまう傾向にあることです。悪い凝集性と悪い結合性は連動することがよくあります。

この原則は設計上の異なる意思決定を評価するためにも使われます。他の条件がすべて同等だとすると、凝集性の高い設計のほうが優れています。

> **名前：**高凝集性（High Cohesion）
>
> **問題：**オブジェクトの焦点を明確にし、わかりやすく、管理しやすくし、さらには副次効果として疎結合性も促進するにはどのようにすればよいか
>
> **解決策（一例）：**凝集性が高くなるように責任を割り当てる。複数の選択肢を評価する場合にもこの原則を用いる

17.9　オブジェクト設計への GRASP の適用

GRASP は、General（汎用） Responsibility（責任） Assignment（割り当て） Software（ソフトウェア） Patterns（パターン）を表す頭字語です。オブジェクト指向ソフトウェアの設計を成功に導くにはこれらの原則の「理解」（grasping）が重要であることを表すために、この GRASP という名前が選ばれました。

GRASP パターンは以下に示す 9 個です。

- 生成者
- 情報エキスパート
- 疎結合性
- コントローラ
- 高凝集性
- 多相性
- 人工品
- 間接化
- 防護壁（バリエーション防護壁）

17.10　生成者パターン

オブジェクトの生成は、オブジェクト指向システムで最もよく行われるアクティビティの１つです。したがって、生成の責任の割り当てに一般的な原則があると有用

です。この割り当てがうまく行われた設計は、結合性が疎に保たれ、明確になり、カプセル化と再利用性が促進されます。

生成者パターンをもとに、SalesLineItem インスタンスを「集約する」、「含む」などのクラスを見つけます。図 17.12 の部分的な問題領域モデルを見てください。

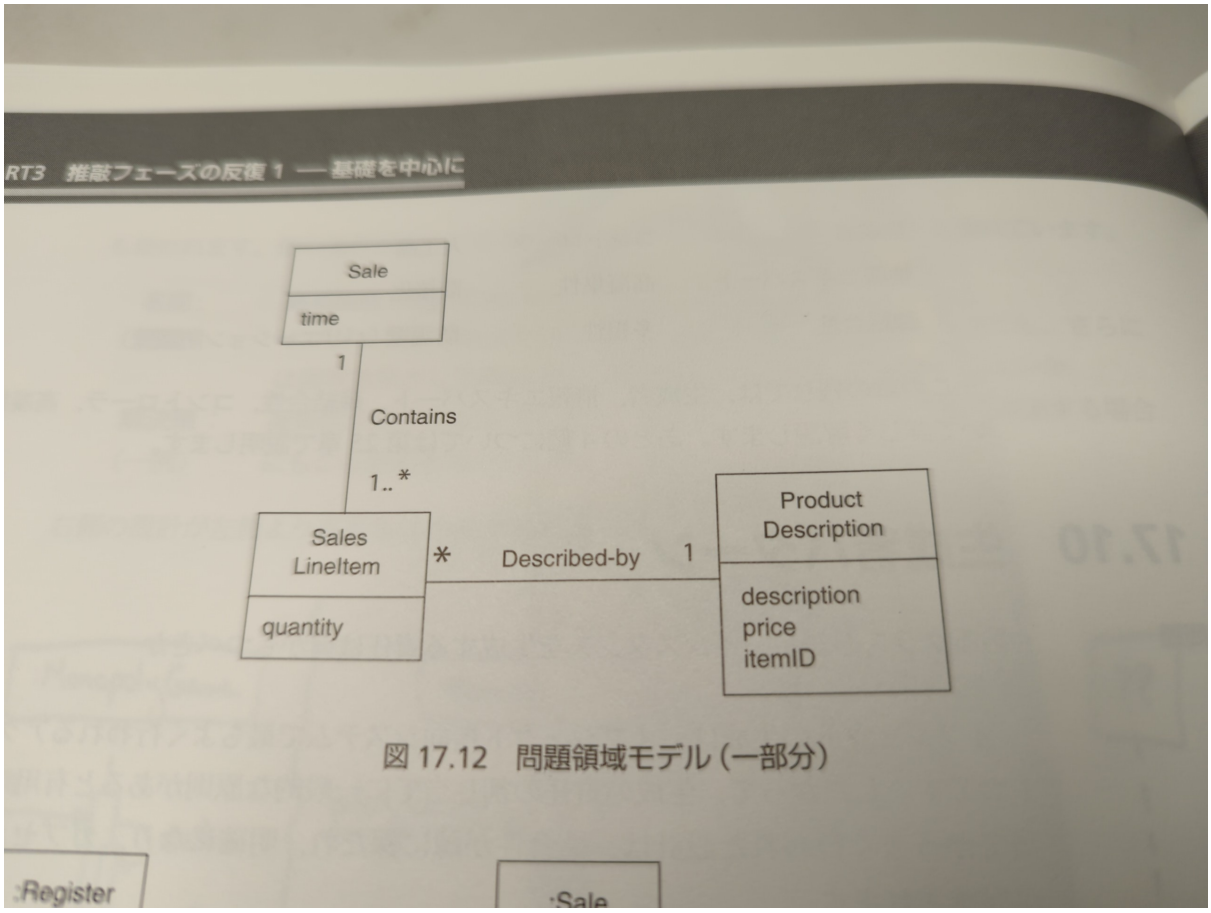


図 17.12　問題領域モデル（一部分）

Sale は多くの SalesLineItem オブジェクトを含む（実際には集約する）ため、生成者パターンによって、Sale が SalesLineItem インスタンス生成の責任をもつ候補として挙げられます。このことから、オブジェクトの相互作用の設計が図 17.13 のように導かれます。

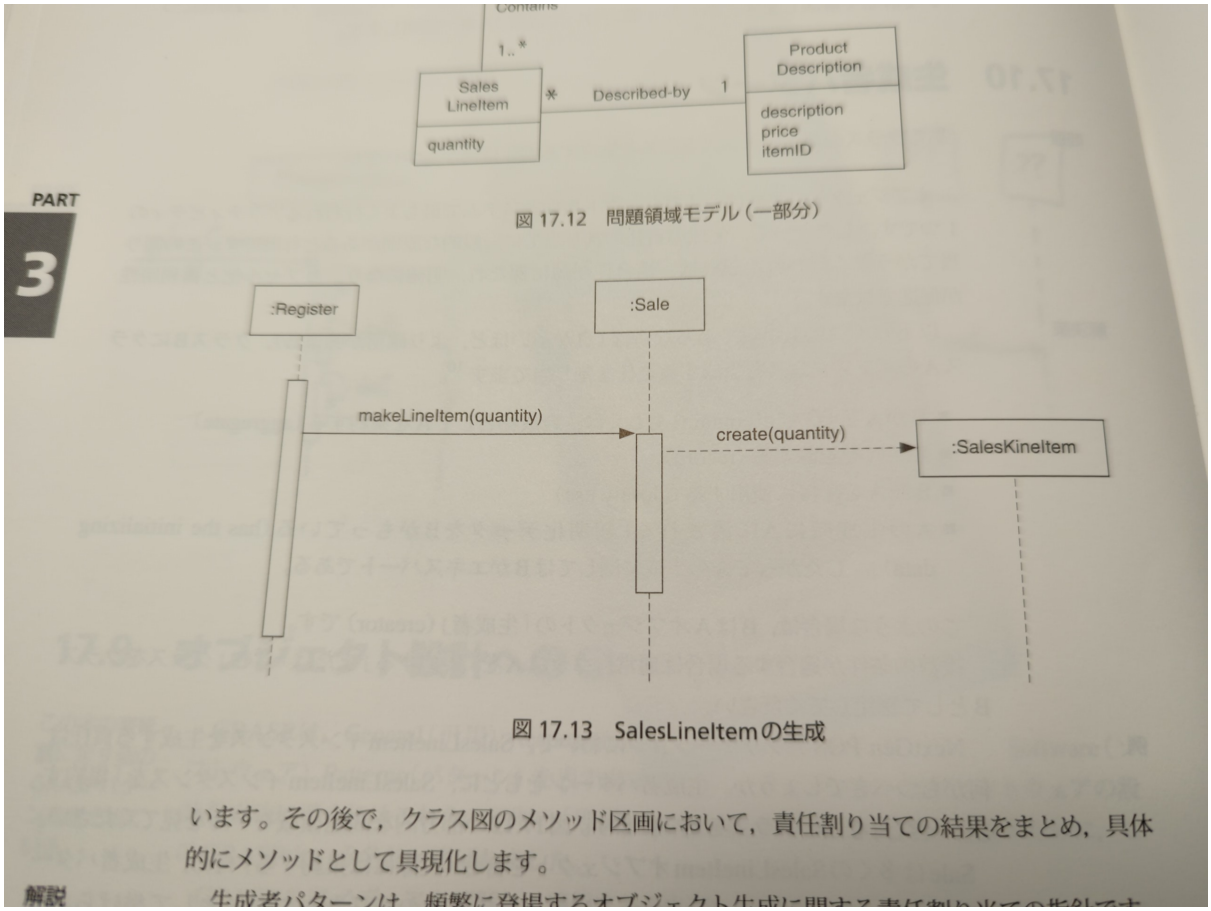


図 17.13　SalesLineItem の生成

このように責任を割り当てるには、makeLineItem メソッドが Sale で定義される必要があります。繰り返しますが、このような責任の考察と決定は、相互作用図の作成中に行っています。その後で、クラス図のメソッド区画において、責任割り当ての結果をまとめ、具体的にメソッドとして具現化します。

生成者パターンは、頻繁に登場するオブジェクト生成に関する責任割り当ての指針です。生成者パターンの基本的な目的は、何らかのイベントで生成されるオブジェクトと接続する必要のある生成者を見つけることです。これを生成者として選定することで疎結合性が支援されます。

「合成物が部分を集約する（aggregate）」、「コンテナが内容を含む（contain）」、「記録者が記録対象を記録する（record）」はどれも、クラス図のクラス間でよく見られる関係です。生成者パターンでは、囲い込む側であるコンテナクラスまたは記録者クラスが、含まれる、あるいは記録されるものを生成する責任を持つ候補することが提案されます。むろん、これは１つの指針にすぎません。

生成中に受け渡される初期化データをもっているクラスを探すことで、生成者が見つかる場合もあります。これは実際にはエキスパートパターンの例です。

多くの場合、生成はかなり複雑なタスクです。パフォーマンス上の理由からリサイクルインスタンスを使用したり、外部のプロパティ値に基づく条件処理を通して、似たようなクラス群の中から１つのインスタンスを生成したりすることがあります。このような場合は、生成者パターンによって提案されるクラスではなく、具象ファクトリ（Concrete Factory）または抽象ファクトリ（Abstract Factory）と呼ばれるヘルパクラスに生成を委譲する方法をお勧めします。

生成者を選定する要因となった既存の関連があるために、生成されたクラスはそれを生成したクラスにとってすでに可視であることが多いからです。

17.11　情報エキスパート（またはエキスパート）パターン

オブジェクトへの責任割り当てにおける一般原則は何か

オブジェクト設計においては、オブジェクト間の相互作用を定義する際に、ソフトウェアクラスに割り当てる責任を選択します。これが良好に行われていると、システムの理解、保守、拡張が行いやすくなり、将来のアプリケーションでもコンポーネントを再利用しやすくなります。

責任の遂行に必要な情報をもっているクラス、すなわち「情報エキスパート」に責任を割り当てます。

責任の割り当ては、責任を明確にすることから始めます。

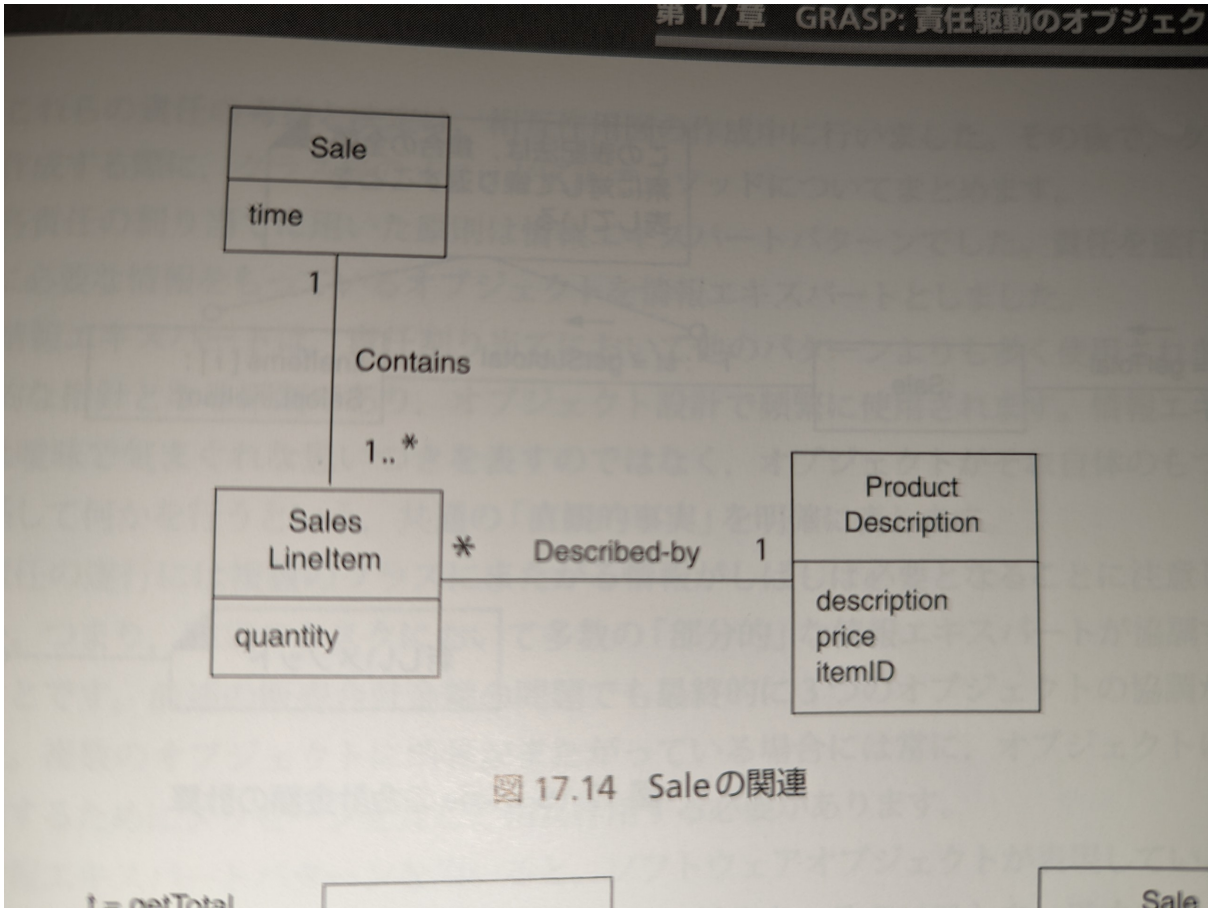
必要な情報を持つクラスを分析するには、UP 問題領域モデルを調べるべきでしょうか、UP 設計モデルを調べるべきでしょうか。UP 問題領域モデルは現実世界の問題領域における概念クラスを説明し、設計モデルはソフトウェアのクラスを説明するものでした。

1. UP 設計モデルに重要そうなクラスがある場合には、まずそれを調べます。
2. さもなければ UP 問題領域モデルを調べ、その表現を使用して（または拡張して）、対応する設計クラスの作成の参考にします。

たとえば、設計作業を初めたばかりで、ほとんど UP 設計モデルができていないと想定してください。このような場合は、UP 問題領域モデルから情報エキスパートを探します。

次に、UP 設計モデルに、たとえば Sale というソフトウェアクラスを追加し、合計金額を把握する責任を `getTotal` という名前のメソッドで表現し、このメソッドを Sale ソフトウェアクラスにもたせます。この方法は「表現上のギャップが小さい」

ため、現実の問題領域がどのようにこうせいされているかについての人間の思考になじみます。



合計金額の決定に必要な情報は何でしょうか。販売の SalesLineItem インスタンスのすべてとそれらの小計を知る必要があります。Sale インスタンスはこれらを知っています。したがって、情報エキスパートパターンの適用により、Sale がこの責任の割り当て先に適したクラスです。つまり Sale がこの場合の「情報エキスパート」です。

販売明細（SalesLineItem）の小計を決定するために必要な情報は何でしょうか。SalesLineItem.quantity と ProductDescription.price が必要です。SalesLineItem は、販売の数量（quantity）と、販売に関する ProductDescription を把握しています。したがって情報エキスパートパターンの適用により、SalesLineItem が小計を決定すべきであり、SalesLineItem が情報エキスパートです。

ProductDescription は価格の問い合わせに答える情報エキスパートです。したがって、SalesLineItem は ProductDescription に商品価格を問い合わせるメッセージを送ります。

設計クラス	責任

設計クラス	責任
Sale	販売の合計金額を把握する
SalesLineItem	販売明細の小計を把握する
ProductDescription	商品の価格を把握する

情報エキスパートは、責任の割り当てにおいて他のパターンよりも多く使用されます。基本的な指針となる原則であり、オブジェクト設計で頻繁に使用されます。情報エキスパートは曖昧で気まぐれな思いつきを表すのではなく、オブジェクトがそれ自体のもつ情報に関係して何かを行うという、共通の「直感的事実」を明確に表します。

責任の遂行には複数のクラスにまたがる情報がしばしば必要となることに注意してください。つまり、特定のタスクにおいて多数の「部分的」な情報エキスパートが協調するということです。前述の販売合計金額の問題でも最終的に３つのオブジェクトの強調が必要でした。複数のオブジェクトに情報がまたがっている場合には常に、オブジェクトは仕事を分担するためにメッセージを介して相互作用する必要があります。

オブジェクト指向ソフトウェアの世界では、あらゆるソフトウェアオブジェクトが「生きている」あるいは「生命を吹き込まれている」ため、責任を持ち、何かを行うことができます。基本的には、ソフトウェアオブジェクトは自分の知っている情報に関係のある何かを行います。筆者はこれをオブジェクト設計における「生物化」（animation）原則と呼んでいます。

情報エキスパートの適用が望ましくない場合があります。これは通常、結合性および凝集性との兼ね合いのためです。

主要な関心事の分割を行うと、設計の疎結合性（low coupling）と高凝集性（high cohesion）が向上します。このように、情報エキスパートパターンに照らせば、データベースサービスに関する責任を Sale クラスにもたせることが正当であるように思えても、その設計では通常は疎結合性と高凝集性の観点から品質が低くなります。

17.12　疎結合性パターン

依存性を弱め、変更による影響を小さくし、再利用性を高めるにはどのようにすればよいか

結合（coupling）は、１つの要素が他の要素に対し、どの程度の強さで接続するか、把握するか、あるいは依存するかを表す尺度です。

結合性が密な（あるいは強い）クラスは、多くのクラスに依存してしまいます。このようなクラスは以下のような問題があるため、望ましくありません。

- 関係するクラスになされた変更によってローカルの変更も強いられる
- 単独では理解しにくい
- そのクラスを使用するにはそのクラスが依存するクラスも存在しなければならないため、再利用しにくい

疎結合性（Low Coupling）パターンは、設計のあらゆす意思決定において考慮する必要のある原則です。常に考慮すべき根本的な目標なのです。疎結合性は、設計者があらゆる意思決定を評価する際に適用する評価用の原則でもあります。

疎結合性は、変更の影響の少ない、より高い独立性をもつくらすの設計を支援します。疎結合性パターンを情報エキスパートや高凝集性など他のパターンと切り離しては考察できません。責任割り当ての選択肢に影響する複数の設計原則の１つです。

結合が強すぎるかどうかの絶対的な測定基準はありません。重要なことは、現在の結合の度合いを判断し、結合の強さが問題を引き起こすかどうかを判定することです。ほとんどの場合は、もともと総称的な性質をもっていて再利用の可能性が高いクラスは、特に結合性を低くしておくべきです。

疎結合性の極端なケースはクラス間に結合がまったくない状態ですが、これは望ましくありません。オブジェクトテクノロジの中核は、メッセージを介してコミュニケーションし合う、接続されたオブジェクトでシステムを構築することだからです。疎結合性が過度に適用されると、設計の品質が低下します。

接続されたオブジェクト間の協調によってタスクが遂行されるオブジェクト指向システムを構築するには、クラス間にある程度の結合のあることが普通であり、これは必要でもあります。

密な結合自体が問題なのではなく、インタフェースや、実装、あるいは存在自体など、何らかの不安定要因を抱えた要素と密に結合することが問題なのです。

現実には必要でない場所で、「将来の保証」のためや結合性を弱くすること自体を目的として労力を費やすのは、時間の浪費です。

ものごとの疎結合化とカプセル化において注力する箇所を選定しなければなりません。実際に不安定な箇所や進化の速い個所に集中するのです。

17.13　コントローラパターン

UI レイヤからシステム操作を最初に受け取り調整する（「コントロールする」）オブジェクトは何か

コントローラ（controller）は、UI レイヤを超えてシステム操作メッセージを受け取る責任または処理する責任をもつ、最初のオブジェクトです。

以下のどれかに該当するクラスに責任を割り当てます。

- 「システム」全体、「ルートオブジェクト」、中でソフトウェアが動作するデバイス、あるいは、主要なサブシステムを表すクラス（これらはどれもファサードコントローラのバリエーションである）
- システムイベントが起きるユースケースシナリオを表すクラス。このようなクラスにはしばしば、＜ユースケース名＞Handler、＜ユースケース名＞Coodinator、あるいは＜ユースケース名＞Session のような名前がつけられる
    - 同じユースケースシナリオで起きるシステムイベントにはすべて、同じコントローラクラスを使用する
    - 大まかにいうと、セッションはアクタとの対話のインスタンスである。セッションの長さは任意であるが、多くの場合はユースケースの観点で編成される（ユースケースセッション）

NextGen アプリケーションのいくつかのシステム操作を図 17.20 に示します。このモデルはシステム自体をクラスとして表現しています（適法ですし、モデリング時に便利な表現でもあります。）

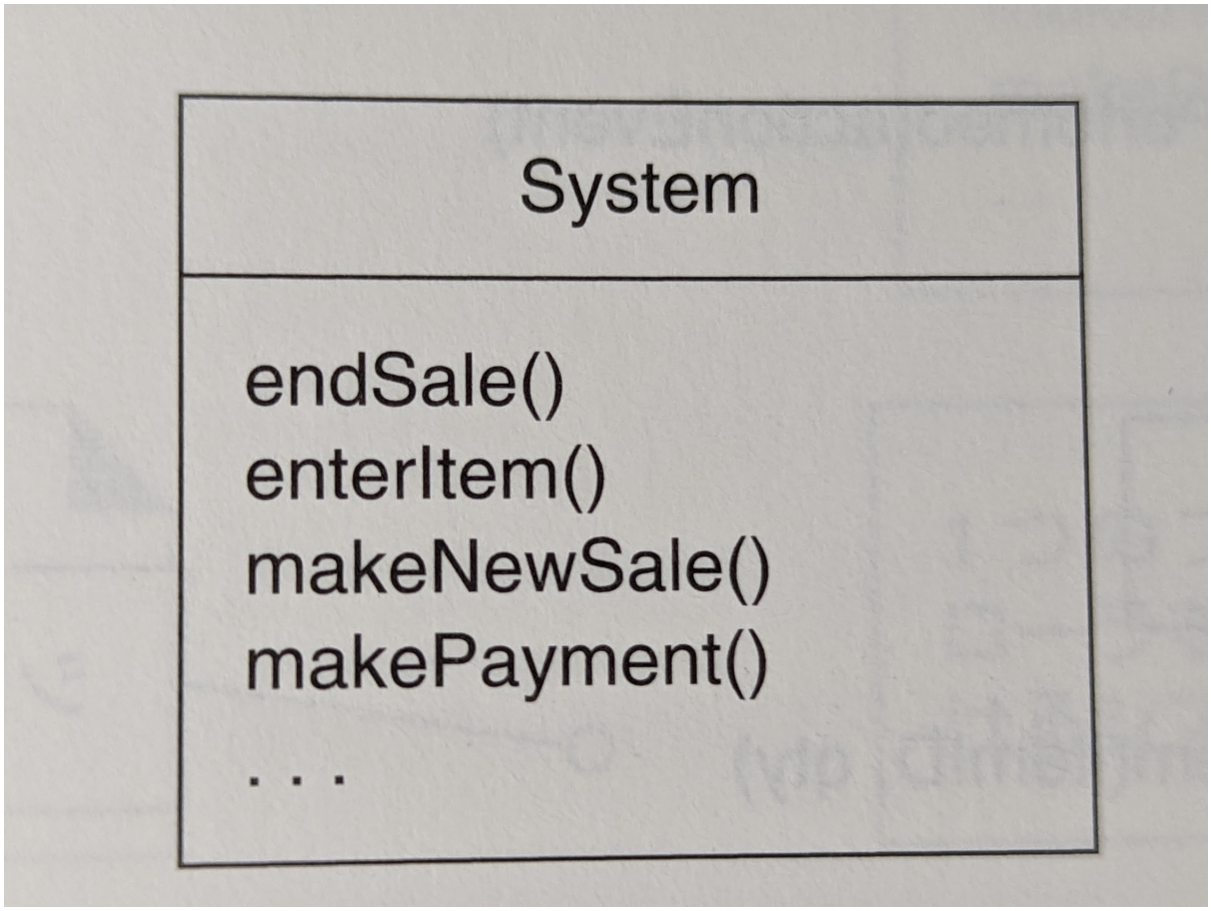


図 17.20　NextGen POS アプリケーションのシステム操作