3. デザインパターン解説

この章では、デザインパターンの解説を行う。デザインパターンは急速に拡張している分野であり、ここですべてのデザインパターンを解説するわけにはいかないが、開発現場で使うことの多い重要なパターンおよびこの後の章で使用するパターンを解説しておく。

3.1 パターンの種類

この本では、デザインパターンだけでなく、オブジェクト指向分析・設計に役立つ他のパターンも説明する。デザインパターンの成功を見て出現した、分析段階で使用する分析パターンや、設計の上流工程で行うアーキテクチャ決定に役立つアーキテクチャパターンなどである。

ソフトウェア開発に役立つパターンは、今のところ、下記のような種類がある。

- **デザインパターン**

  2 章で説明したように、デザインパターンは、サブシステムやコンポーネントあるいはそれらの間の関係を洗練する構成を提供する。ある文脈での一般的な設計上の問題を解く、コンポーネント間に繰り返し現れる協調構造を記述する。主に設計工程で用いられるが、一部のデザインパターンは分析工程でも使える。

- **分析パターン**

  分析パターンは、対象問題領域 (ドメイン) に存在するドメインオブジェクトとその間の関係を提供する。設計上の問題を「解く」わけではなく、再利用できるモデルを提供する。分析工程で用いられる。

- **アーキテクチャパターン**

  ソフトウェアシステムの基本的で構造化された組織や構成を表す。既定義のサブシステムの集合とその責任、それらの間の関係や組織化のための規則・指標を提供する。設計の上流工程で用いられる。

- **プロセスパターン**

  ソフトウェアの開発に関わる定石である、開発チームの組織化や開発方法論・開発手順あるいはプロジェクト管理法などを提供する。4 章は、このプロセスパターンを使って、オブジェクト指向開発方法論に基づく開発手順を記述したが、本書ではプロセスパターン自体は解説しない。

  ソフトウェア開発の全工程にプロセスパターンは存在するが、「誰でも、ほとんど考えもしないで、ソフトウェアを構築できる」プロセスパターンなどは存在しない[*1]。

- **算法** (Algorithm) **パターン**

  問題を解くための具体的な算法を表す。空間効率と実行効率のトレードオフを提供する。デザインパターン中の戦略 (Strategy) パターンの中身の定義であるとも見なせる。設計の下流工程で使う。

- **イディオム**

  プログラミング言語に依存した詳細レベルの抽象パターン。特定の言語で、コンポーネントやその間の関係をどうやって実装するかを記述する。設計の下流工程および実装工程で使う。

  本書はプログラミング言語になるべく依存しない方針なので、イディオムに関する記述はしない。

---

1. 少なくとも、現時点のソフトウェア工学では存在しないし、将来にわたってもこのような「万能プロセスパターン」は作成できないだろう。まして、「要求定義を行えばソフトウェアが自動作成できる」などというのは、全くの夢物語である。このように喧伝している方法論なりツールは存在するが、それらは一種のプログラミング言語でしかない。

3.2 デザインパターンの分類

デザインパターンは、サブシステムやコンポーネントあるいはそれらの間の関係を洗練する構成を提供する。ある文脈での一般的な設計上の問題を解く、コンポーネント間に繰り返し現れる協調構造を記述する。複数のデザインパターンを組み合わせて使うことが多い。

主に設計工程で用いられるが、一部のデザインパターンは分析工程でも使える。

デザインパターンは、今のところ、以下の 3 つに分類できる。

- **生成パターン**
  オブジェクトの生成と初期化を行うことで、オブジェクトがどう生成され組み立てられ表現されるかの知識を、システムの他の部分と独立にするためのパターンである。
- **構造パターン**
  クラスとオブジェクトをどう組み立てていくかに関わるパターンである。
- **振る舞いパターン**
  オブジェクト間の責任と算法の割り当て、およびオブジェクト間のコミュニケーションを扱うパターンである。

以下に、それぞれの主要なパターンを説明する。なお、パターン名は日本語 (英語) の形にしているが、他のデザインパターン本との整合性を取るため、パターンの説明順序はパターンの英語名のアルファベット順にした。

3.3 生成パターン

オブジェクトの生成と初期化を行うことで、オブジェクトがどう生成され組み立てられ表現されるかの知識を、システムの他の部分と独立にするためのパターンである。

オブジェクト指向の設計では、普通、クラスがインスタンスを生成し、より特殊なクラスが必要なときはサブクラスを作るのだが、それでは保守性や再利用性に問題が出る場合があり、それを避けようとするのがこのパターンの主なねらいである。

本書で説明する生成パターンは以下の通りである。

- **抽象工場** (Abstract Factory)
- **工場操作** (Factory Method)
- **一枚札** (Singleton) **パターン**

3.3.1 **抽象工場** (Abstract Factory)

互いに関連するプロダクトとしてのオブジェクト群を、その具象クラスを明確にしないまま生成するインターフェースを提供することにより、顧客クラスが具象クラスを指定せずプロダクトを生成できるようにするパターンである。

プロダクトを個々の部品から一歩一歩組み立てる必要のあるとき、同じ部品ファミリーから１個のプロダクトに組み立てるとき、部品の生成・組合せ・実装方法を、システムの他の部分から独立にしたいとき、などに使う。

抽象工場 (Abstract Factory) パターンの主な役割は、アプリケーションプログラムのいろいろな場所で、下記のような、プロダクトによる分岐が発生する弊害を回避することである。

プログラム 3.1 プロダクトによる分岐の例

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

上のようなプログラムは、プロダクトや部品の構成が変わったときに大きな影響を受ける。

そこで、下図のようなクラス構成にする。

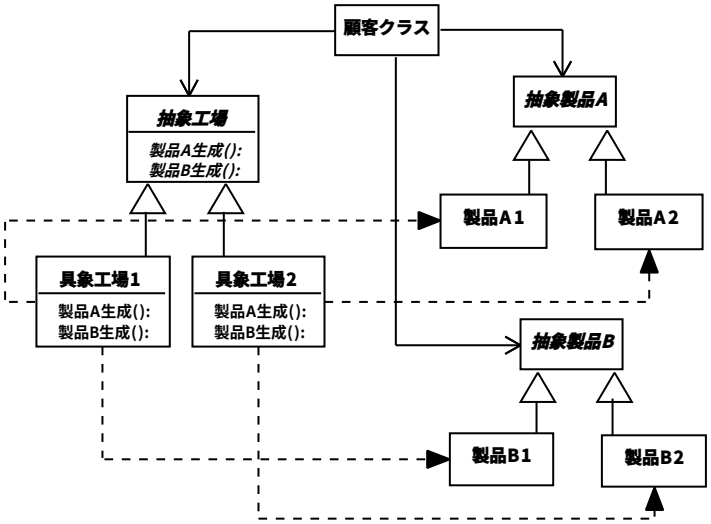

図 5 抽象工場 (Abstract Factory) パターンのクラス図

ここで、各クラスの責任は以下の通りである。

- **抽象工場** (AbstractFactory)

  抽象製品を生成する操作のインターフェースを宣言する。

- **具象工場** (ConcreteFactory)

  具象製品を生成する操作を実装する。

- **抽象製品** (AbstractProduct)

  製品オブジェクトのインターフェースを定義をする。

- **具象製品** (ConcreteProductr)

  対応する具象工場オブジェクトによって生成される製品オブジェクトを定義し、抽象製品のインターフェースを実装する。

- **顧客** (Client)

  抽象工場と抽象製品のインターフェースのみを用いて、それらの提供するサービスを受ける。

ここでは、あるプロダクトを作るのに関わる部品としての製品クラスを、種類ごとに抽象[*2] 製品 A あるいは抽象製品 B のサブクラス[*3] として定義する。それらを使ってプロダクトを生成する工場としてのクラスを抽象工場クラスのサブクラスとして定義する。抽象工場クラスとそのサブクラスの具象工場クラスは、共通のインタフェースを持った製品生成操作を定義する。

こうすることで、顧客クラスのプログラムは、具象クラスのどれを使うかをソースコード上に具体的に書かずに、実行時に与えられた具象工場のオブジェクトをパラメータとして、必要な具象工場クラスのオブジェクトを呼び出すことができる。

比喩として、PC を組み立てる作業に抽象工場パターンを使う例を考えることで、具体的に見てみよう。

顧客クラスは「PC 組立屋」、抽象工場は「PC 工場」でそのサブクラスとして「PowerPC 工場」と「PoorPC 工場」があるものとする。「PC」には「PowerPC」と「PoorPC」があり、部品として「CPU」「HD」「EthernetCard」とそれらのサブクラスの具象クラスがあるものとする。

クラス図と順序図は以下のようになる。

2. 図上では抽象クラスの字体を斜体とする。
3. 製品の種類だけサブクラスを作ることになる。

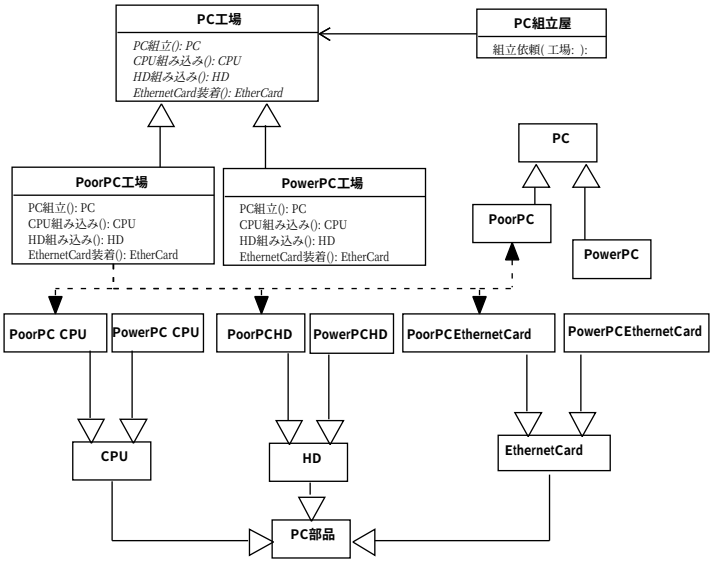

図 6 抽象工場を適用したクラス図の例

順序図からメッセージの動きを見ると分かりやすいので、そこから説明する。まず、実行時に、PC 組立屋オブジェクトはユーザーインタフェースを担当するオブジェクト（ここでは PC 組立屋 GUI とした）から PowerPC 工場オブジェクトをパラメータとした組立依頼メッセージを受け取る。

次に PC 組立屋オブジェクトは、パラメータとして受け取った PowerPC 工場オブジェクトへ PC 組立メッセージを送り、結果として「ある PowerPC」オブジェクトを受け取る[*4]。もちろん、この順序図では示していないが、PowerPC 工場オブジェクトは、その内部で PowerPC クラスに依頼して「ある PowerPC」オブジェクトを生成してもらっている。

4. 破線の矢印は、オブジェクトからオブジェクトへの制御の戻りを表す。

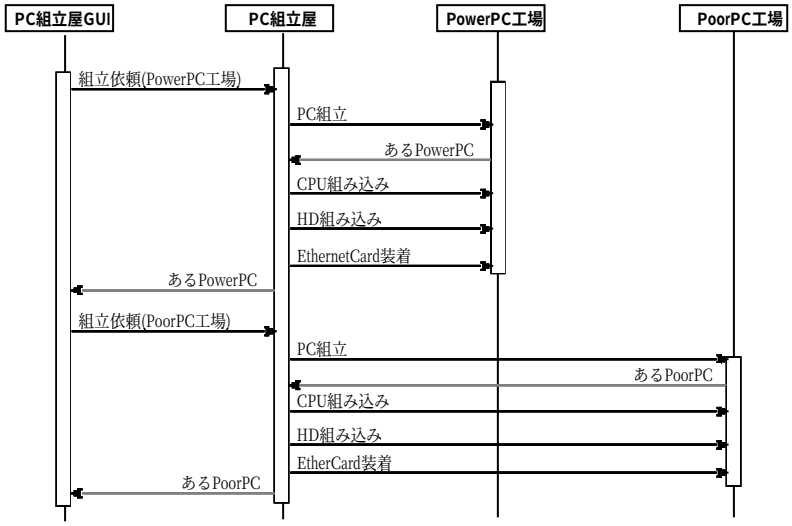

図 7 抽象工場を適用した順序図の例

この時点では、まだ「ある PowerPC」オブジェクトは箱だけで、CPU も HD も EthernetCard も装着していないので、PC 組立屋オブジェクトはこれらを組み込むメッセージを PowerPC 工場オブジェクトへ送る。最後に、「PC 組立屋 GUI」オブジェクトに完成した「ある PowerPC」オブジェクトを渡す。

PC 組立屋が、組立依頼 (PoorPC 工場）メッセージを受けたときは、今度はメッセージをパラメータとして受け取った PoorPC 工場オブジェクトへ送るだけで、基本的な流れは PowerPC の場合と変わらない。すなわち、PC 組立屋のソースコード上に、PC の種類による分岐は一切ないことになる。

この方法では、以下のプログラム例の断片で示すように、CPU を追加するときの修正を 1 カ所にできる利点もある。

プログラム 3.2 抽象工場の普通の実装方法

```
PC 工場 ::[*5] CPU 組み込み -- [*6] 既存のクラスと操作
    抽象操作 -- サブクラスで定義するので変更はない。
```

```
PowerPC 工場 ::CPU 組み込み -- 既存のクラスと操作
    return = PowerPC.*7 new -- PowerPC クラスのインスタンス生成
PoorPC 工場 ::CPU 組み込み -- 既存のクラスと操作
    return = PoorPC.new -- PoorPC クラスのインスタンス生成
新たな PC 工場 ::CPU 組み込み -- 追加したクラスと操作
    -- CPU のインスタンスを返す
    return = 新たな PC.new -- 新たな PC クラスのインスタンス生成
```

以下のような同一操作方式[*8] を使うと、さらに修正を局所化できる。

プログラム 3.3 同一操作方式

```
PC 工場 ::CPU 組み込み操作
    -- 修正する必要がない
    -- cpuClass 操作で返される CPU クラスのインスタンスを生成する
    return = self[*9] .cpuClass.new
PowerPC 工場 ::cpuClass
    return = PowerPC -- CPU のクラスすなわち PowerPC を返す
PoorPC 工場 ::cpuClass
    return = PoorPC -- CPU のクラスすなわち PoorPC を返す
```

3.3.2 工場操作 (Factory Method)

オブジェクトを作成するためのインターフェースを定義するが、実際にどのクラスのインスタンスを生成するかはサブクラスに任せるパターンである。

生成しなければならないオブジェクトのクラスを、生成する側のクラスが事前に知ることができない場合や、クラスの責任をいくつかのサブクラスに委譲するとき、その知識を局所化したい場合に使う。

---

5. OCL では、クラス名 :: 操作名で、あるクラスの操作の定義であることを示す。
6. -- は OCL で注釈を表す記号である。
7. オブジェクト名 . 操作名は、あるオブジェクトの操作を示す OCL の記号。
8. 後述する工場操作 (Factory Method) パターンを使っている。
9. self はオブジェクト自身を表す OCL の予約語。

このパターンを使用しないと、オブジェクトを生成する部分のソースコードが、製品 A か製品 B というクラス名を埋め込んだものになってしまい、製品を追加したり、修正する場合にソースコードを変更しなければならず、変更余波が大きくなってしまう。

工場操作 (Factory Method) パターンのクラス図は、以下のようになる。

ここで、「ある操作」は「工場操作」を使ったサービスを提供する操作である。1 つとは限らず、いろいろな操作があり得る。

各クラスの責任は以下のようになる。

- **製品** (Product)

  工場操作が生成するオブジェクトのインタフェースを定義する。

- **具象製品** (ConcreteProduct)

  製品のインターフェースを実装する。

- **作成者** (Creator)

  製品オブジェクトを返す工場操作を宣言する。標準の製品オブジェクトを返す工場操作の標準的実装を定義する場合もある。工場操作以外の操作の中で、製品オブジェクトを生成するためには工場操作を呼び出す。

- **具象作成者** (ConcreteCreato)

  具象製品オブジェクトを返す工場操作を実装または再定義する。

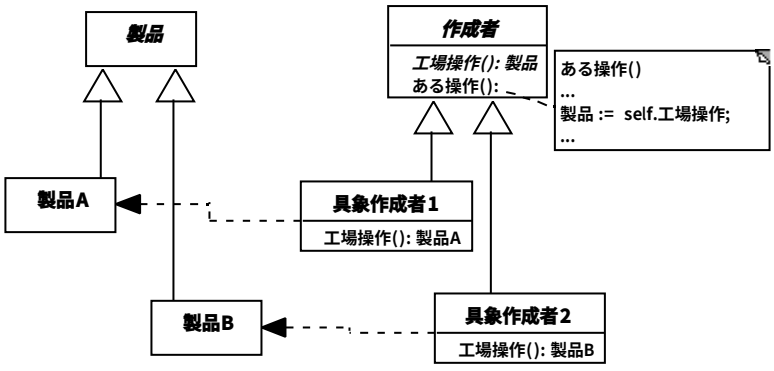

図 8 工場操作 (Factory Method) パターンのクラス図

このパターンでは、特定の製品のオブジェクトを作るためだけに、対応する具象作成者クラスを作らなければならないが、パラメータ化クラス (テンプレート) を使う[*10] か、クラスをパラメータとして渡す[*11] ことによって、具象作成者クラスを不要とする実装も可能である。

工場操作 (Factory Method) パターンは、抽象工場 (Abstract Factory) パターンの対案になりやすい。抽象工場パターンは工場クラス階層が複雑になるが、工場操作は具象作成者（すなわちアプリケーション）クラス階層が複雑になる。どちらも一長一短であり、対象システムの特性によって使い分けなければならない。

3.3.3 一枚札 (Singleton) パターン

クラスにインスタンスが 1 つしかないことを保証するパターンである。

大域変数を使わず、代わりに一枚札 (Singleton) パターンを使うことで、保守性と再利用性が増す。また、後々インスタンスの数を増やしたり、サブクラスを作って操作や内部表現を詳細化することができる利点もある。

一枚札 (Singleton) パターンのクラス図は以下のようになる。

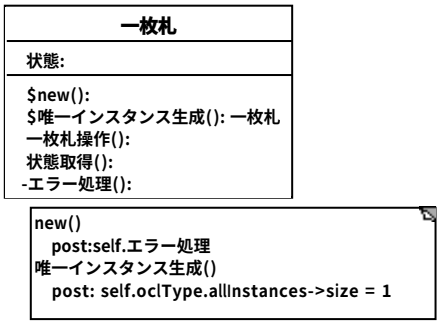

図 9 一枚札 (Singleton) パターンのクラス図

10.C++ の場合。
11.Smaltalk の場合。

一枚札 (Singleton) クラスの責任は、唯一のインスタンスを生成するクラスレベルの操作[*12] を定義することである。

一枚札パターンでは、普通インスタンスの生成に使われる new 操作はエラーにする。代わりに、1 つしかインスタンスがないことを保証するインスタンス生成操作「唯一インスタンス生成」を定義する。

上のクラス図の注釈中の「唯一インスタンス生成」の定義に出てくる

```
post: self.oclType.allInstances->size = 1
```

は、この操作の後件 (post) の記述で、自分自身 (self で表す) すなわち一枚札のインスタンスの型[*13](oclType 操作は型を返す) のすべてのインスタンス (allInstances 操作は、型の全インスタンスを返す) の個数 (size 操作はものの集まりの個数を返す) が 1 であること、すなわち一枚札クラスのインスタンスが 1 つであることを示している。

注釈中の new 操作の定義は、後件が自身にエラー処理操作をメッセージとして送ることを示している。一枚札クラスのエラー処理操作定義の頭にハイフンが付いているのは、エラー処理操作が外部に公開されていない私有 (プライベート) 操作であることを示す。

一枚札クラスの new と「唯一インスタンス生成」操作の前にドル印 $ が付いているのは、これらの操作がクラスレベルであることを示している。

3.4 構造パターン

クラスとオブジェクトの静的構造をどう組み立てていくかに関わるパターンである。

素直な設計だと 1 つのクラスが負うべき責任を、複数のクラスで分担して、保守性と再利用性を向上させようというのがこのパターンの主な役割である。

本書で説明する構造パターンは以下の通りである。

12.Smalltalk ではクラス操作で、C++ では static なメンバー関数である。
13.OCL は、型とクラスを同一のものと見なす。

- **変換器** (Adapter) **パターン**
- **混成** (Composite) **パターン**

3.4.1 **変換器** (Adapter[*14]) **パターン**

あるクラスのインターフェースを、他のインターフェースへ変換するパターンである。

既存のクラスを利用したいがインターフェースが一致していない場合や、将来作成されるクラスとも協調していける再利用可能なクラスを作成したい場合に使う。

クラス図は以下のようになる。

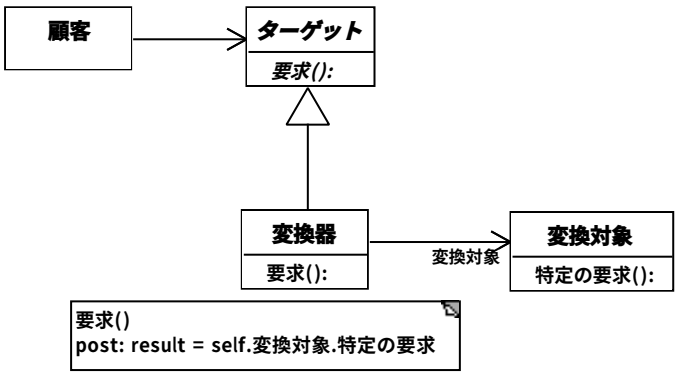

図 10 変換器 (Adapter) パターンのクラス図

ここで、各クラスの責任は以下の通りである。

- **ターゲット** (Target)

  顧客オブジェクトが使用する問題領域 (ドメイン) に特化したインターフェースを定義する。

14.Adaptor という綴りでもよい。

- **顧客** (Client)

  ターゲットのインタフェースを使って、変換対象オブジェクトと協調する。
- **変換対象** (Adaptee)

  顧客が利用したいサービスを定義している。
- **変換器** (Adapter)

  変換対象オブジェクトのインターフェースをターゲットオブジェクトのインターフェースに適合させる。

ここで、顧客クラスはターゲットオブジェクトしか知らないように作る。変換対象クラスは変換器クラスによってブラックボックスとなり、別の用途のために改良を続けることができる。

変換器には、変換対象クラスにないインターフェースや機能を追加することができる。

上のクラス図の変換器クラスは、設計時に変換対象クラスのインターフェースを知っていれば作成できるが、変換対象クラスが設計時に分かっていない場合に対処できない。このような問題を解決するのがインターフェース調整機能付きのプラガブル変換器 (Pluggable Adapter) である。クラス図は以下のようになる。

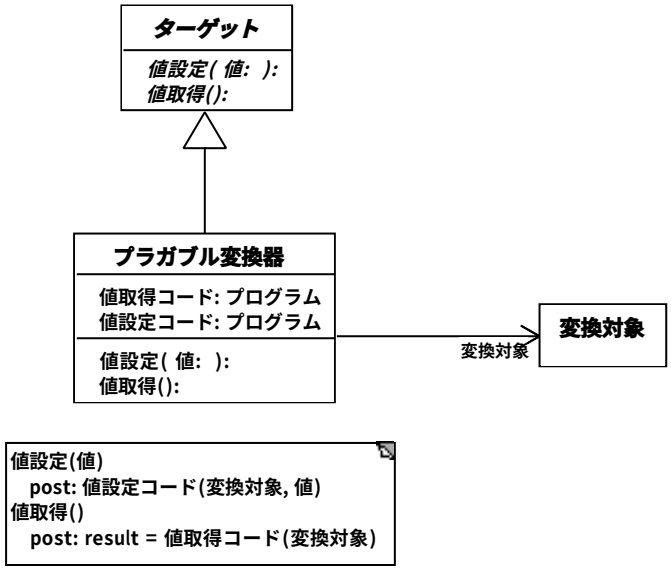

図 11 プラガブル変換器 (Pluggable Adapter) パターンのクラス図

顧客オブジェクトのプラガブル変換器オブジェクト初期設定部分のコードは下記のようなイメージになる[*15]。

プログラム 3.1 プラガブル変換器オブジェクトの初期設定

ある変換器 := プラガブル変換器 . 生成 (ある変換対象）;
ある変換器 . 値取得コード := A;
ある変換器 . 値設定コード := B;

ここで、A は変換対象をパラメータとして、変換対象の値を取得するプログラムのソースコードである。B は変換対象と値をパラメータとして、変換対象の値を設定するプログラムのソースコードである。

値設定操作は B を実行することにより変換対象オブジェクトの「値」を設定し、値取得操作は A を実行することにより変換対象オブジェクトの「値」を取得する。

15. 実装言語によって、実装方法はかなり異なる。

3.4.2 混成 (Composite) パターン

部分全体構造を表現するために、オブジェクトを木構造で構成し、個々のオブジェクトとその入れ物とを統一的に扱うことができるパターンである。

クラス図は以下のようになる。

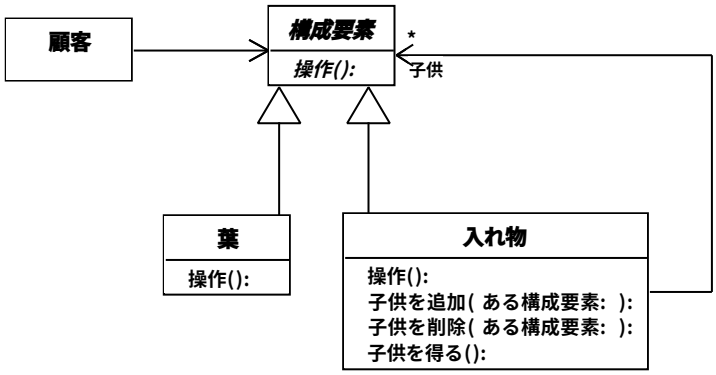

図 12 混成 (Composite) パターンのクラス図

各クラスの責任は以下のようになる。

- **構成要素** (Component)

  入れ物オブジェクトのインターフェースを宣言し、パターン内のすべてのクラスに共通な標準の振る舞いを実装し、子供の構成要素をアクセスし管理するインターフェースを宣言し、場合によっては、再帰構造の構成要素の親へアクセスするインタフェースを定義するか実装する。

- **葉**[*16] (Leaf)

  入れ物の中の、子供を持たない原子オブジェクトである葉オブジェクトを表現する。

16. 下の木構造の図を上下ひっくり返すと木に見え、そのとき葉に当たる部分に来るのでこのような名前付けをする。

- **入れ物** (Composite)

  子供を持つ構成要素の振る舞いを定義し、子供の構成要素を格納し、子供に関する操作を実装する。

- **顧客** (Client)

  構成要素のインターフェースを通して、入れ物の中のオブジェクトを扱う。

例えば、ファイルシステムでは構成要素オブジェクトがデスクトップ、入れ物オブジェクトがフォルダー、葉オブジェクトがファイルに対応する。

このクラス図で表される木構造データは下図のようになる。

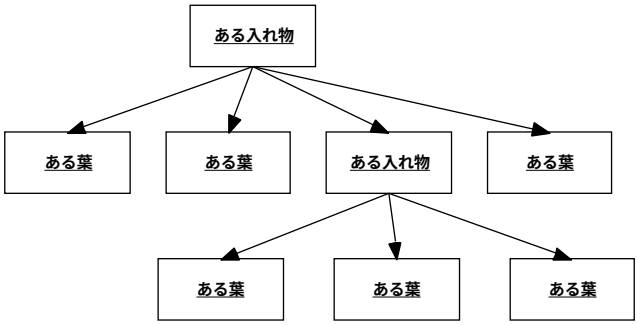

図 13 混成 (Composite) パターンが表す木構造の例

混成パターンには以下のような利点がある。

- **統一的なインターフェース**

  再帰的で複雑な構造に統一的なインターフェースを与える。

- **検索などが簡単**

  再帰的な呼び出しの行える言語[*17] では、簡単なコードで混成パターン内の全オブジェクトの挿入・検索・削除を行うことができる。

17. すべてのオブジェクト指向言語は、プログラム自らの複製を作ってそれを自ら呼び出すという再帰呼び出しが可能である。

- **検索などの効率がよい**

  挿入・検索・削除のいずれも、基本的に計算量 $O(\log_2 n)$ で行うことができるため、かなり効率的[*18] である。

- **顧客コードの単純化**

  顧客オブジェクトは、木構造を走査するとき、葉と入れ物のどちらを扱っているか知る必要がないため、コードが単純で分かりやすくなる。

- **新しい構成要素を追加しやすい**

  構成要素を追加しても、顧客クラスには影響がない。

- **性能改善のためのキャッシュを効かせやすい**

  木構造を走査するとき、着目しているオブジェクトのそばのオブジェクトを走査する可能性が高い。このため、ある大きさの部分木をキャッシュに入れれば、ヒットする確率が高くなる。

混成 (Composite) パターンのクラス図では、入れ物オブジェクトが構成要素オブジェクトを保持する「子供」というロール名が付いた関連しかなかったが、子供が親を頻繁に参照したいときは、親オブジェクトへのリンクも持つとよい。構成要素と入れ物クラスの関連だけを取り出すと、以下のようなクラス図になる。

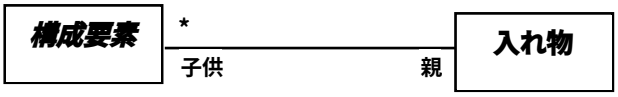

図 14 親へのリンクを持った構成要素

この図で表される親子関連は、普通リスト構造で保持されるが、子供が多い場合アクセス効率が悪くなる。このようなときは、入れ物オブジェクトが構成要素オブジェクトを持つ関連は 2 分木などの木構造で実装すると、効率が良くなることが分

18.$O(\log_2 n)$ は、データ量が n のとき $\log_2 n$ に比例した計算量を示し「$\log_2 n$ のオーダーの計算量」という呼び方をする。$O(\log_2 n)$ はかなり効率が良く、例えば 100 件のデータを 1 秒で処理できるとき、10000 件のデータに対しても 2 秒で計算が行える。ところが、例えば $O(n^2)$ だと 10000 件の場合 3 時間近く掛かり、$O(n^3)$ だと 11 日以上かかる！

かっている。つまり混成 (Composite) パターンの一部に混成 (Composite) パターンを使うことになる。

ガーベージコレクション機能[*19]を持たないオブジェクト指向言語の場合、誰が不要になった構成要素を削除するかが問題になることがある。葉オブジェクトが多くの他のオブジェクトに共有されている時などに問題が起こる。すなわち、参照されている葉オブジェクトを間違って削除してしまう危険性があり、それを避けようとして削除しないと、使われないオブジェクトがゴミとなってメモリー空間を徐々に少なくしていく。最近のパソコンのOSやアプリケーションに、この種の欠陥を持ったものが多くなっている。

3.5 振る舞いパターン

オブジェクト間の責任と算法の割り当て、およびオブジェクト間のコミュニケーションを扱うパターンである。

単純な設計では、2つのオブジェクトとその関連で動的構造を構成するところを、中間の媒介をするオブジェクト群を想定することによって、保守性と再利用性を向上させようというのがこのパターンの役割である。あるいは 1 つのオブジェクトでは動的構造を表現するには責任が大きくなりすぎるとき、それを分割する場合もある。

本書で説明する振る舞いパターンは以下の通りである。

19. 誰からも参照されず不要になったオブジェクトを削除し、オブジェクトが使用していたメモリー空間を、再利用できるメモリー空間に返す機構のこと。

- **命令** (Command) **パターン**
- **通訳** (Interpreter) **パターン**
- **観察者** (Observer) **パターン**
- **状態** (State) **パターン**
- **戦略** (Strategy) **パターン**

3.5.1 **命令** (Command) **パターン**

命令をカプセル化することによって、命令を受信するオブジェクトに関する知識を不要とし、命令に対する種々の操作を実現するパターンである

命令の明確化・順序化・実行をそれぞれ独立化したい場合や、命令の取り消しを行いたい場合、命令の再実行を必要とする場合、あるいは基本的な命令からなる高度な命令（トランザクションなど）によってシステムを構造化したい場合に使用する。

クラス図と順序図は以下のようになる。

ここで、顧客クラスと受け手のクラスは同一であることが多い。また、顧客クラスと命令クラスだけの単純な実装もあり得る。

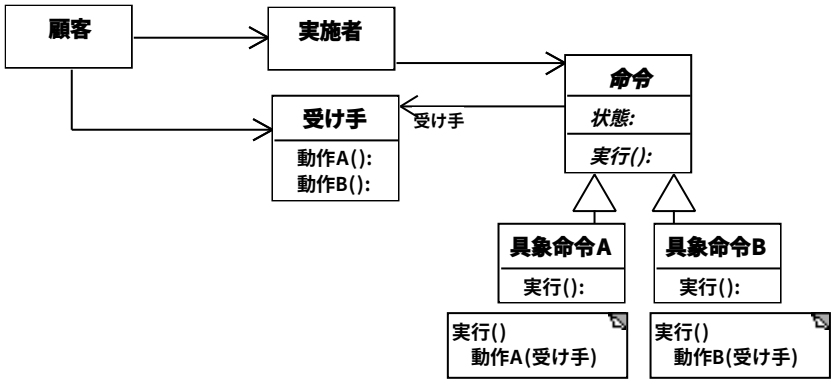

図 15 命令 (Command) パターンのクラス図

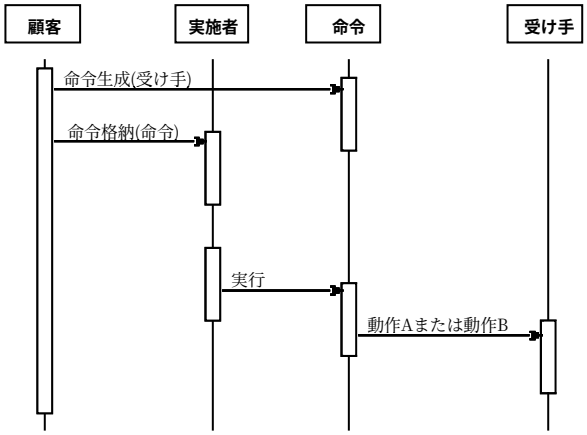

図 16 命令 (Command) パターンの順序図

ここで、各クラスの責任は以下のようになる。

- **命令** (Command)

  操作を実行するためのインターフェースを宣言する。

- **具象命令** (ConcreteCommand)

  受け手と動作の組を定義し、受け手の対応する操作を起動することで、命令を実行する。

- **顧客** (Client)

  具象命令オブジェクトを生成し、その受け手を設定する。

- **実施者** (Invoker)

  命令オブジェクトに要求実行を依頼する。

- **受け手** (Receiver)

  要求を実現するための具体的操作を知っている。任意のクラスが受け手になり得る。

順序図の流れに沿って説明すると、最初に顧客オブジェクトが命令オブジェクトを生成する。生成された命令オブジェクトは受け手を知っている。次に、顧客オブジェクトが実施者オブジェクトに命令を格納する。

実施者オブジェクトが、いつ命令を実行するかはアプリケーションによって異なるが、例えば、あるイベントが発生すると命令が実行されるように実装するのが普通である。実行メッセージを受けた命令オブジェクトは、受け手へ動作 A または動作 B のメッセージを送る。もちろん、コマンドが 10 個あれば 10 個の動作があることになる。

命令 (Command) パターンを使うと、結果として以下のような効果を得られる。

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

取り消しと再実行を行うには、命令の履歴が必要になる。履歴は、順序のあるものの集まり (OCL では Sequence）クラスで実装することが多い。

後述する通訳 (Interpreter) パターンを使って、コマンドを「言語」化するとより汎用性が増す。

3.5.2 通訳 (Interpreter) パターン

データを言語と考え、文法[*20] と意味を与え、それを通訳 (解釈・実行）する（要するに通訳系＝インタープリターを作る）パターンである。

ほとんどのデータはちょっとした工夫で「文脈自由文法に従う言語」と見なせるので、問題の複雑さを文法で表し、個々の要素や組み合わせ規則で表したくない時に使用する。個々の要素や組み合わせ規則に制限を設けたくない時にも使える。

---

20. データを文字列の流れ (Stream) と考え、文脈に依存しない文法 (文脈自由文法) を与えるのが一般的である。

もちろん、文脈自由文法に従う言語を処理したい時は、当然このパターンを使うことになる。

正規表現、図形表現、ドキュメント、プログラミング言語、通信文・プロトコルなど、固定幅以外のデータのほとんどが対象となり得る。

例えば、以下のような変数が使える簡易言語を考える。

プログラム 3.1 簡易言語の例

```
let
    x = 3,
    y = 7,
    z = 2
in
    x * (y + z / 4)
```

ここでは、let 以下の文の定義を、in の中の式で使う。

この簡易言語の文法定義[*21] は、以下で与えられる。

プログラム 3.2 簡易言語の文法[*22]

```
プログラム　→　Let 式
　| 式
Let 式　→　’let’　定義　’in’　式
定義　→　代入　定義 aux
定義 aux　→　’,’　定義
　|　ε　--　空である ( 何もないことが許される）ことを示す
代入　→　変数名　’=’　数
式　→　項　式 aux
式 aux　→　’+’　式
　|　’-’　式
```

21.BNF 記法（Buckus-Naur-Form）と呼ぶ、文脈自由文法の一般的な定義法である。

22.T-gen（『Justin O. Graver 著、T-gen User’s Guide、University of Florida』参照）の例題より引用。

| ε

項 → 因子 項 aux

項 aux → ’*’ 項

| ’/’ 項

| ε

因子 → 変数名

| 数

| ’(’ 式 ’)’

変数名 → < 変数 > -- < > 記法は終端記号を表す

数 → < 数 >

< 変数 > → [a-z] -- a-z のいずれかの文字であることを示す[*23]

< 数 > → [0-9]+ -- 0-9 のいずれかの文字が 1 個以上続くことを示す

通訳 (Interpreter) パターンのクラス図は以下のようになる。

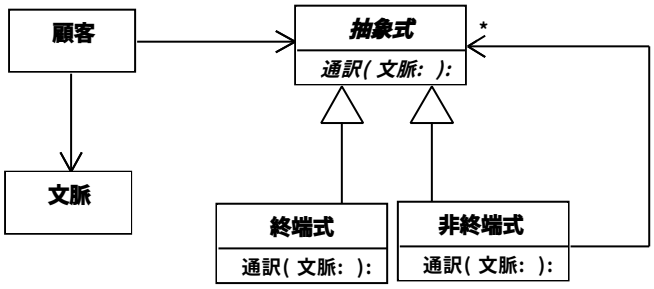

図 17 通訳 (Interpreter) パターンのクラス図

このようなクラス図で通訳できるのは、以下の条件を満たす場合である。

23. 正規表現と呼ばれる記法である。

- **文法が簡単であること**

  文法が複雑であると、抽象式のクラス階層が複雑になり、保守困難になる。この場合は、構文解析ツール[*24]を使い、変換器 (Adapter) パターンを使ってラップした方がよい。これらの構文解析ツールは抽象構文木[*25]を作らず通訳するものが多いので、空間効率・実行効率とも優れた通訳系を生成する。

- **実行効率がさほど重要でないこと**

  効率的な通訳系の多くは、状態遷移マシンに変換するなどして、抽象構文木を直接には使わない。しかし、抽象構文木を使った通訳パターンは、通訳をすべてオブジェクト指向パラダイムの元で行えるので、文法が簡単な言語の場合には、保守性や再利用性が優れている。

各クラスの責任は以下のようになる。

- **抽象式** (AbstractExpression)

  抽象構文木のすべての頂点に共通な抽象操作を定義する。

- **終端式** (TerminalExpression)

  文法中の終端記号に関する操作を具現する。終端記号ごとにインスタンスが生成される。

- **非終端式** (NonterminalExpression)

  非終端記号それぞれについてクラスが定義される。上の簡易言語の例だと「式」「項」「因子」などがクラスになる。非終端記号それぞれについて抽象式型のインスタンス変数を保持する。非終端記号それぞれについて通訳操作を具現する。

- **文脈** (Context)

  通訳系の実行時の環境、すなわちグローバルな情報を持つ。

---

24. GNU プロジェクトの GNU C と連動する Bison, ベル研究所の UNIX, C と連動する yacc, フロリダ大学の Smalltalk と連動する T-gen, Sun Microsystems社の Java と連動する JavaCC などがある。
25. 言語の構成要素を表現しやすいように構文木を圧縮したもの。

- **顧客** (Client)

  言語の特定の文について、非終端記号オブジェクトと終端記号オブジェクトから組み立てられる構文解析木を作り（または与えられる）、通訳操作を呼び出す。

式　－3＊（4＋5）を解析した抽象構文木の例を以下に示す。

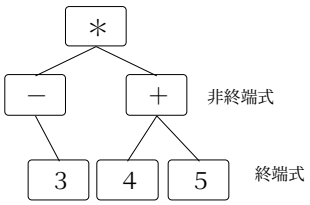

図 18 抽象構文木の例

通訳パターンを使うと以下のような効果がある。

- **文法の変更と拡張が容易である**
- **構文解析部を実装するのも容易である**
- **言語表現を新しい方法で評価することを容易にする**

  言語を処理するだけでなく、清書したりチェックをする操作を簡単に定義できる。

実装上の工夫としては、以下のような手があるが、ここでは概要の紹介に留める。詳細はコンパイラーの教科書を参照していただきたい[*26]。

- **終端記号を共有する**
- **部分実行**

  一度実行した時に、効率化のための情報を保存し、次にそれを使う。

26.『A.V.　エイホ他著、原田賢一訳、コンパイラ　原理・技法・ツール、サイエンス社、1990 年』参照。

- **並行実行**

  抽象構文木の部分木は、副作用がなければ理論的には並行実行できる[*27]。

- **遅延評価（Lazy Evaluation）**

  実際に必要になるまで式の評価を行わないことで、無限のリストなども扱えるようになり、問題の定義に近いプログラムが可能になり、プログラムの自由度が増す。

3.5.3 観察者 (Observer) パターン

あるオブジェクトが状態を変えたとき、そのオブジェクトに依存するすべてのオブジェクトに通知し、自動的に更新するパターンである。

関連あるオブジェクト同士を、密に結合せずに連動させたい場合や、あるオブジェクトに依存するオブジェクトを固定的に決められない場合に使う。

クラス図と順序図は 2 章に示した通りである。

各クラスの責任は以下の通りである。

- **主体** (Subject)

  観察者オブジェクトを知っている。任意の数の観察者オブジェクトが主体を観察できる。観察者オブジェクトを依存者として追加・削除するインターフェースを提供する。

- **観察者** (Observer)

  主体の変化を受け取ったときの更新操作インターフェースを定義する。

- **具象主体** (ConcreteSubject)

  具象観察者が興味を持つ「状態」を持っている。「状態」が変化したとき具象観察者オブジェクト群に通知する。

- **具象観察者** (ConcreteObserver)

  具象主体への参照を保持する。具象主体の「状態」と矛盾しないように、自身の「状態」を追従させる。観察者クラスで宣言した更新操作のインターフェースを、具象主体と整合性を保持するように具現する。

27.OS と言語によってはできないことがある。

観察者 (Observer) パターンを使うことによって、主体側のオブジェクトと観察者側のオブジェクトの間の抽象的結合が提供される。結果として、以下のような効果が得られる。

- **各観察者が異なるアーキテクチャ・GUI でも問題がない**
- **主体と観察者は独立していて、相手に影響を与えずに変更できる**
- **ソースコードでメッセージの受け手を明確にしておく必要がない**
- **観察者が多くなっても問題が少ない**

しかし、観察者が「主体の変化」に伴うコストを予測できないという欠点もある。

実装に関する注意点を以下に説明する。

(1) 変更管理 (ChangeManager) オブジェクト

複数の具象主体があるような場合、変更管理 (ChangeManager) オブジェクトを使うことがある。

変更管理オブジェクトの責任は以下の通りである。

- **主体を観察者にマップし、これを維持していく操作を提供する**

  個々の主体で観察者への参照を持つと空間効率が悪いので、共同の検索表を使う。ハッシュ表など使って実装することが多い。主体は観察者への参照を保持する必要がなくなる。更新操作の引数へ主体を追加し、どの主体からの更新メッセージか明らかにする。
- **更新のための戦略を定義する**

  主体からの要求により、依存するすべての観察者を更新する。例えば、操作が複数の依存し合う主体に影響するとき、観察者に対する通知が何度も繰り返されないように、すべての主体が修正された後に 1 回だけ通知を送るようにする。

(2)SASE(Self-Addressed Stamped Envelope) パターン

命令 (Command) パターンと観察者 (Observer) パターンの結合パターンであり、観察者が主体にプログラムを含んだ封筒を渡す。

封筒の中身を以下のようにする。

- **受け手＝観察者自身**
- **イベント＝観察者が興味のある主体の変化**
- **プログラム＝主体が変化したときに、主体が観察者に送るべきメッセージ**
- **引数＝プログラムの引数**

主体が変化すると、自身に「イベント発生」メッセージを送る。「イベント発生」操作は、関係する封筒を集め、中のプログラムを実行する。すなわち、観察者を受け手としたメッセージが実行される。

SASE の利点は以下の通りである。

- **不要な更新がない**
- **観察者に興味ある通知だけが来る**
- **観察者に更新操作を実装する必要がない**
- **主体と観察者の結合が弱くなり、保守性と再利用性が向上する**
- **観察者のプログラムが単純・簡単になる**
- **観察者のプログラムから、通知を吟味して対策を考える部分がなくなる**

(3) 更新プロトコル

観察者オブジェクトの状態を更新するプロトコルとして、push モデルと pull モデルが考えられる。

push モデルは、更新時に観察者にすべての情報を送るため、メッセージ量が減るが、主体と観察者の結合度が増大し、保守性と再利用性が低下する。

pull モデルは、更新時に観察者へ最小の情報を送りあとで観察者が主体に問い合わせるため、主体と観察者の結合度は低いが、効率が悪くなる恐れがある。

観察者側からのポーリングする手も考えられる。不要な更新が減り、効率は良くなるが、更新タイミングがずれる欠点が出てくる。

(4) ダングリング参照の回避

主体が削除されたとき、観察者オブジェクトに相手先のないダングリング参照が発生し、実行時エラーの原因となる。これを避けるには主体が削除されたことを観察者に通知する方法がある。ガーベージコレクションが行われている言語では問題が起きない。

3.5.4 状態 (State) パターン

オブジェクトは一種の状態遷移マシンすなわち独立に動く一個の実行プログラムと考えられる。この状態遷移マシンの実装方法の 1 つが状態 (State) パターンである。

オブジェクトの状態変化が複雑な場合、ソースコードの各所に以下のような条件分岐が発生する。

プログラム 3.1 状態による分岐

```
if  self.state = # 状態 A  then  状態 A の処理
elsif self.state = # 状態 B then 状態 B の処理
...
endif
```

このような条件分岐はあちこちに出現し、保守性や再利用性を損ねる。これを避け、オブジェクトの状態変化を別なオブジェクトにカプセル化するのが状態 (State) パターンの役割である。

クラス図は以下のようになる

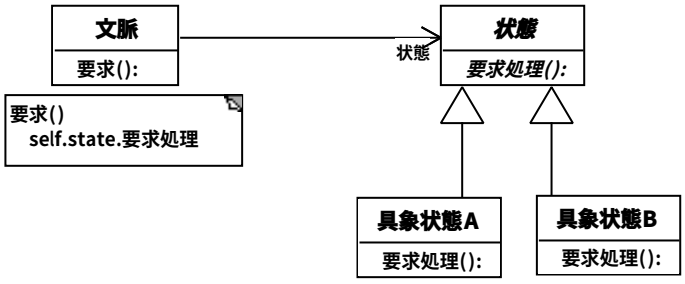

図 19 状態 (State) パターンのクラス図

各クラスの責任は以下の通りである。

- **文脈** (Context)

  顧客に必要なインターフェースを定義し、現在の状態を表す具象状態クラスの全てのインスタンスを保持する。

- **状態** (State)

  文脈オブジェクトのある状態の振る舞いをカプセル化するためのインターフェースを提供する。

- **具象状態** (ConcreteState)

  文脈オブジェクトのある 1 つの状態での振る舞いを具現する。

通常、オブジェクトの内部状態は変数やプログラムポインターに分散するが、状態パターンを使うと、状態に依存した振る舞いを局所化でき、状態遷移が明確になる。結果として、状態や遷移の追加や変更が容易になる。

実装上の問題点としては、以下が挙げられる。

- **どのオブジェクトで状態遷移を実装するか？**

文脈クラスで実装すると、状態に関する知識が文脈クラスと状態クラスに分散してしまい保守性に問題が出る。一方、状態クラスのサブクラスで実装すると、サブクラス間の依存性が発生し、特定の文脈に依存した状態遷移になるため、状態クラスを他の文脈で再利用することができなくなる。

- **具象状態オブジェクトの生成と破壊のタイミング**

具象状態オブジェクトを、必要なときだけ生成する場合、利用しないオブジェクトの生成を避けることができ、空間効率が良くなる。一方、状態遷移が頻繁に起こる場合は、最初に具象状態オブジェクトを生成しておくと、生成・破壊の回数が減り実行効率が良くなる。

状態 (Stte) パターンを使わない状態遷移マシンの実装方法は、状態遷移マシンを状態遷移図という一種の言語で定義されたプログラムと見て、それを通訳するやり方である。つまり、通訳 (Interpreter) パターンを使って、状態遷移図に対応するパラメータ言語[*28] を解釈・実行するわけである[*29]。

状態遷移マシンによる実装の方が、状態遷移を修正する場合にパラメータ言語を修正するだけでよいので柔軟性が高いが、実行速度で問題が出る場合がある。また、既存のツールがない場合、言語によっては状態遷移マシンを実装しづらい[*30]。

3.5.5 戦略 (Strategy) パターン

関連ある算法 (Algorithm) を定義し、それぞれをカプセル化し、交換可能にするパターンである。このパターンを使うことで、算法を使用する顧客オブジェクトから算法を独立にしておくことができる。

複数の異なる算法を使用する場合や、算法の変化が予測される場合、あるいは顧客オブジェクトが知るべきでないデータを算法が利用している場合に使用する。

状態 (State) パターンと同じように、戦略ごとの処理を切り替えるための条件分岐がソースコード中に現れるのを避けようとするのが、このパターンを使う主な動機である。

クラス図は以下のようになる。

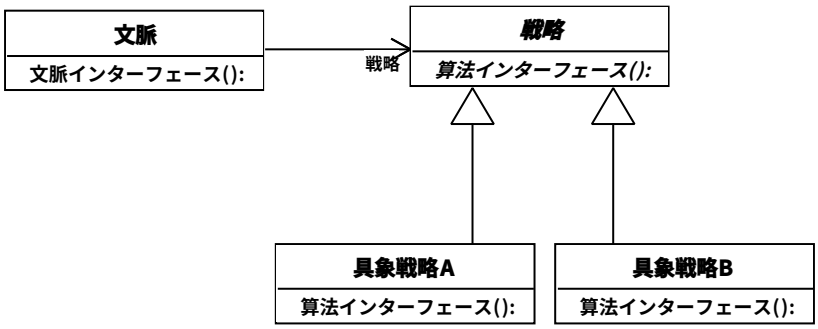

図 20 戦略 (strategy) パターンのクラス図

---

28. 基本的に、（前状態、イベント、動作、後状態）の 4 つ組からなる言語となる。
29. UML の状態遷移図の記法の元になった STATECHART は、本来、記法だけでなくこのような通訳系を持った CASE ツールである。
30. 状態遷移マシンの入力パラメータの内、動作は要するにプログラムなので、プログラムをパラメータとして渡せる言語でないと実装できない。

各クラスの責任は以下の通りである。

- **戦略** (Strategy)

  提供する算法すべてに共通のインターフェースを定義する。

- **具象戦略** (ConcreteStrategy)

  戦略で定義されたインターフェースの算法を具現する。

- **文脈** (Context)

  具象戦略オブジェクトを保持し、算法を具象戦略に隔離し、場合によっては戦略オブジェクトが使うデータへアクセスするインターフェースを定義する。

戦略パターンを使うことで、以下のような効果が得られる。

- **算法の共通化が図れる**

  関連する算法の共通機能を戦略クラスに定義できる。

- **算法の局所化が図れる**

  個々の算法を独立に保守したり再利用することが容易になる。ただし、ある具象戦略に関係ないインターフェースを定義しなければならない場合がある。

- **文脈クラス中の条件文を排除できる**

- **異なる実装を提供できる**

  時間と空間のトレードオフにより、適当な算法の実装を選択できる。

具象戦略オブジェクトから文脈オブジェクトのデータを参照できなければならないため、戦略クラスと文脈クラス間のインターフェースをどう取るかが実装上の問題になる。以下の 3 つの方法がある。

- **戦略クラス操作の引数として渡す**

  一番使われる方法である。毎回渡す方法と、最初に 1 回だけ渡す方法がある。

- **文脈オブジェクト自身を引数として渡す**

  便利だが、文脈オブジェクトと戦略オブジェクトの結合度が高くなり、保守性や再利用性が低下する。

- **戦略オブジェクトから文脈への参照を持つ**
  これも便利だが、戦略オブジェクトを第三者のオブジェクトのために再利用することができなくなる。

3.6 分析パターン

分析パターンは、対象問題領域 (ドメイン) に存在するドメインオブジェクトとその間の関係を提供する。設計上の問題を「解く」わけではなく、再利用できるモデルを提供する。分析工程で用いられる。

本書で説明する分析パターンは、以下の通りである。

- **当事者** (Party) **パターン**
- **組織構造** (Organization Structure) **パターン**
- **責任** (Accountability) **パターン**

3.6.1 当事者 (Party) パターン

人と組織の役割の上での同一性を提供するパターンである。住所録や顧客管理システムなどで、人と会社や組織を同列に扱わなければならない場合が多い。

クラス図は以下のようになる。

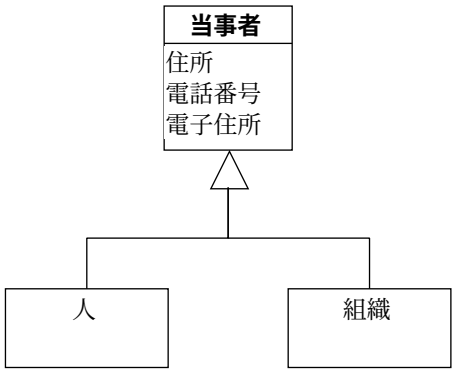

図 21 当事者 (Party) パターンのクラス図

3.6.2 組織構造 (Organization Structure) パターン

組織構造の変化にも影響を受けにくいモデルを提供するパターンである。

組織構造のクラス図は、普通に考えると以下のようになるであろう。

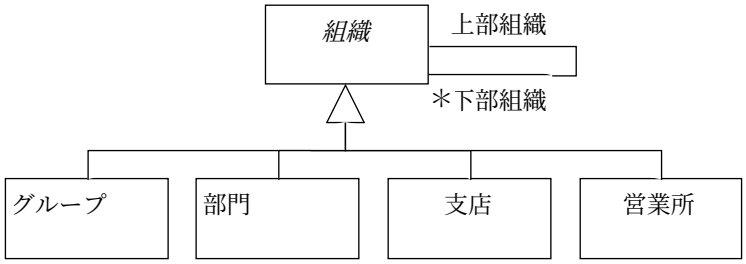

図 22 普通に考えた組織構造のクラス図

多くの組織では、組織構造は半年から 1 年ほどで常に代わり得る。このクラス図では、そのような変更が組織の上下関係だけに影響を及ぼす場合は、上部組織と下部組織のリンク[*31] を追加したり削除するだけで、クラス図に影響を与えず保守できる。また、新しい種類の組織が増えたときは、組織クラスのサブクラスとして新組織に対応するクラスを追加しなければならないのは、どんな構造でも必要なことだろう。

しかし、このクラス図では、組織の関係が上下の 2 項関係以外で変化したとき、例えば、人事上の組織と製品販売上の組織の 2 種類の組織構造が会社に導入されたときなどに対応できない。また、ある組織が 3 月末まで存続し、4 月からは新しい組織になるのだが、システムの上では両者を同時に管理したい場合などにも対応できない。クラス構造を変えざるを得なくなるのだ。

組織構造 (Organization Structure) パターンは、このような場合にもクラス構造を変更せずに、インスタンスの追加や削除で対応しようというパターンである。

クラス図は以下のようになる。

---

31. 関連のインスタンスのことをリンクと呼ぶ。

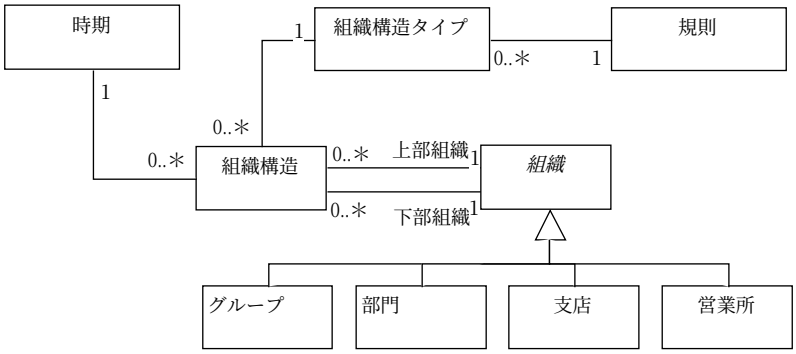

図 23 組織構造 (Organization Structure) パターンのクラス図

例えば、私が所属している Smalltalk グループは、SI システム 2 部という部門に所属しているが、そのオブジェクト間の関係を表すと下図のようになる。営業の所属関係はこの関係と少し異なるのだが、その関係も、上のクラス図に組織構造タイプが「営業所属」、組織構造が「Smalltalk グループ営業の所属」というようなインスタンスとそのリンクを付け加えれば、クラス図は変えなくとも対応できる。

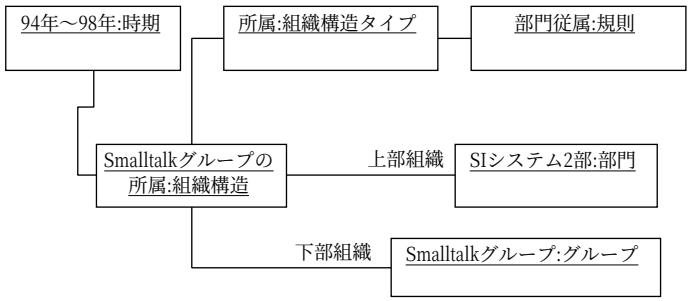

図 24 組織構造のオブジェクト図

3.6.3 責任 (Accountability) パターン

組織構造 (Organization Structure) パターンを拡大し、組織を責任から見た抽象モデルを提供するのが責任 (Accountability) パターンである。

責任 (Acountability) パターンのクラス図は以下のようになる。

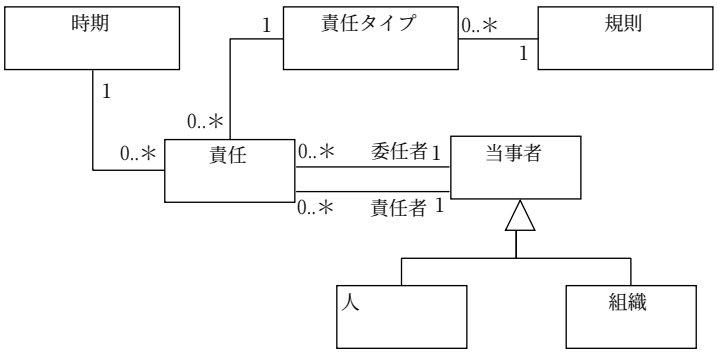

図 25 責任 (Acountability) パターンのクラス図

今度は、組織構造だけでなく、人と組織に関わる「責任」があり委任者オブジェクトや責任者オブジェクトがある構造は、すべてこのクラス図で表され、新たな責任の追加などもインスタンスの追加で済んでしまう。

例えば、佐原が SRA で働いているという関係は以下のように表される。

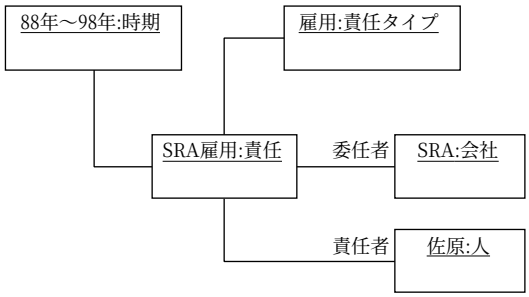

図 26 責任パターンで表す雇用関係

また、SRA が 98 年に A 社のコンサルティングを行うという関係は、以下のように表される。

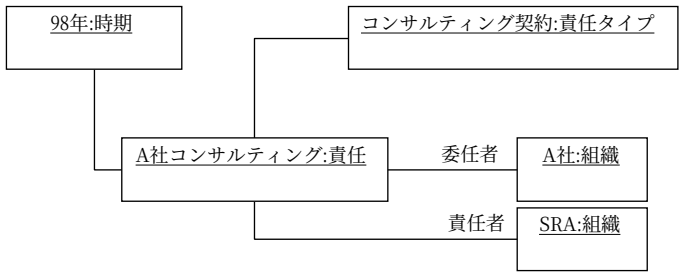

図 27 責任パターンで表すコンサルティング関係

3.7 アーキテクチャパターン

ソフトウェアシステムの基本的で構造化された組織や構成を表す。既定義のサブシステムの集合とその責任、それらの間の関係や組織化のための規則・指標を提供する。設計の上流工程で用いられる。

本書では、以下のアーキテクチャパターンを説明する。

- **階層** (Layer) **パターン**
- **クライアント / サーバーパターン**

3.7.1 階層 (Layer) パターン

システムを実現するとき、ある階層のモジュールは、直下の階層のモジュールの公開されたインタフェースしか使わないようにすることによって、各階層の独立性を高め、保守性や再利用性の向上を図るパターンである。

例えば、私のパソコン上で動いているソフトウェアは、下図のような階層パターンを構成している。

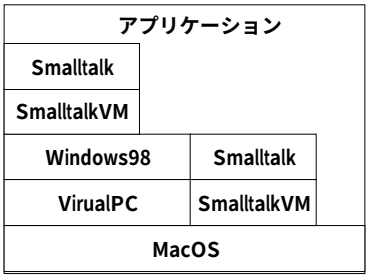

図 28 階層 (Layer) パターンの例

例えば、この原稿を書いている FrameMaker というドキュメントプロセッサーなどの Macintosh アプリケーションは MacOS の上で動いている。4 章などのモデルの確認のためのプロトタイプは、 MacOS の上の Smalltalk の VM(仮想マシン) 上の VisualWorks と呼ばれる Smalltalk で作った。一部の機能は、MacOS 上の VirtualPC という PC 互換機をシミュレートするソフトウェア上に実装した Windows98 で動く Smalltalk VM 上の VisualWorks 上で動く DistributedSmalltalk で確認した。

このような構成にしておけば、例えば MacOS を 8.1 から 8.5 にバージョンアップして差し替えても、基本的には何の問題も生じない[*32]。また、Smalltalk VMやVirtualPCも、この本を書いている間にバージョンアップして差し替えたが、他の部分に影響はない。

階層パターンは、一時的に若干の実行効率低下をもたらすが、各階層のモジュールが改良されることにより、最終的にはかなり効率良いシステムができる。

しかし、ソフトウェア開発の現場では、ミクロの効率化を追うあまり階層パターンを使わず、後で互換性や効率上の問題で廃棄されたソフトウェアは数知れない。

32. 実際にも何の問題も生じなかった。

3.7.2 クライアント / サーバー・パターン

分散処理システムの中で、外部スキーマに当たるアプリケーションと、概念スキーマに当たるドメイン知識（問題領域知識あるいはビジネスロジックとも言う）、および内部スキーマに当たるデータベースをそれぞれ独立に構成することにより、保守性と再利用性を高めるためのパターンである。観察者 (Observer) パターンを分散処理システム用に拡張したものとも見なせる。

このパターンのシステム構成は下図のようになる。

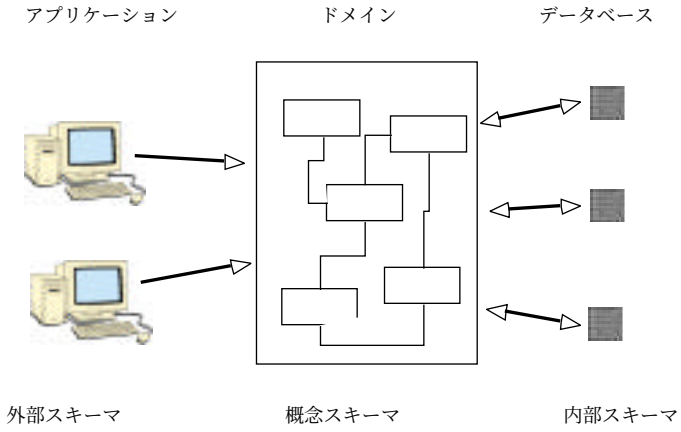

図 29 3-Tier 構成

外部スキーマは概念スキーマの公開されたインタフェースしか使わず、概念スキーマは内部スキーマの公開されたインタフェースしか使わないので、このパターンは階層 (Layer) パターンを内包している。

この図は、3-tier 構成のクライアント / サーバー・パターンの例であるが、概念スキーマと内部スキーマを統合した 2-tier 構成のシステムもある。このパターンの基本形は 3-tier 構成であり、2-tier 構成の特徴は、各階層の独立性がやや弱くなることを除いて 3-tier 構成と同じなので、以下は 3-tier 構成を念頭に置いて説明する。

このパターンおよび分散処理システムの実装上の主要な注意点として、以下が考えられる[*33]。

- **データパス**

  ネットワークを通してメッセージをやり取りするので、各スキーマのプロセス間でどうやってデータを交換するかが効率にかなり影響する。効率は基本的にメッセージ数とメッセージのサイズに依存する。

- **整合性**

  オブジェクトの交換に時差が生じ、途中でオブジェクトが破損することもあり得るため、各スキーマ間で矛盾が生じる可能性がある。

- **並行性**

  分散システムは並行に動くので、同期を取る仕掛けが必要になる。

- **保護**

  分散システムは、ネットワークのどこからでもシステム内部をアクセスすることができるため、特定のオブジェクトを権限のない人なりプロセスがアクセスできないように保護する必要がある。

- **時間**

  デッドロックとメッセージの再実行を避けるため、各スキーマのプロセス間で時計の同期が必要になる。

- **頑丈さ**

  各スキーマのプロセスあるいはネットワークに障害が発生し得るため、その障害からの回復が必要になる。

- **異種混合**

  異なる CPU や OS や通信方式を使う可能性があるため、互換性のあるソースコード作成[*34] やオブジェクトのやり取りが難しい。また、ファイルの管理やアプリケーションの版管理などにも困難が生じる。

以下に、主要なアーキテクチャ上の選択項目を説明する。

---

33. 詳細は『Edited by John A. McDermid, Software Engineer’s Reference Book, Butterworth-Heinemann Ltd., 1991』参照のこと。
34. ビットの並びや、整数を表すバイト数が異なったりする。

(1)RPC(Remote Procedure Call)

プロセス間のメッセージのやり取りは、プロセス間通信 (IPC=Inter-Process Communication) か共有メモリー (Shared Memory) を使う。この両者は原理的に同じ機能を提供することが分かっていて、どちらを選択するかは OS などの環境に依存するが、IPC を使うことが多い。

IPC は通信と同期の 2 面を持つ。同期は、通信の効率化には寄与しないが、リソースのアクセスを保証するためや順序のあるメッセージのために必要となる。２つのプロセス間の双方向通信は、柔軟性のある 1 方向の送受信を基本にしたものと、送信と応答の受信を 1 組にした 2 方向通信を基本にしたものがある。

後者の代表例が、RPC(Remote Procedure Call) であり、通信と同期の両方のサービスを提供し、分かりやすく素直な実装のため、データパスとして使用する例が多い。

RPC は、異なるコンピュータ間のデータ型変換を行うインターフェース変換機能や、呼び出しの失敗時の再呼び出し機能、あるいはサーバーのネットワークアドレスの動的束縛機能や、サーバの動的選択を行うサーバー管理機能などを提供する。

(2) 時間

分散処理システムでは、並行制御やデッドロックの検出あるいは安全のために、イベントで指令する必要がある。

このイベントに時刻印 (Time-stamp) を付け、イベントの順序を決定できるようにしなければならない。しかし、時刻印を付けただけでは同じ時刻印を持つイベント間で順序が不定となり、イベントの集合は半順序になってしまう。

全順序、すなわち任意の２つのイベント間にすべて順序が付くようにするためには、ローカル時刻にサイトの識別子を加えた組（ローカル時刻、サイト識別子）を時刻印とすればよいことが分かっている。

(3) トランザクション

分散システムでは、ある動作は一定時点まで整合性が取れない。例えば、索引オブジェクトは更新したが、参照されるオブジェクトはまだ更新していないという瞬間があり、この時不整合が生じている。

このため、ある命令や動作をまとめて原子トランザクションを構成する必要がある。原子トランザクションは、エラーが発生したらトランザクション開始時の状態に戻れるよう構成する。

原子トランザクションは、他の原子トランザクションを入れ子にして包含することができる。

原子トランザクションの終了時点を、コミット点 (commit point) と言う。引き渡されると原子トランザクションの取り消しはできなくなる。

トランザクションは、以下の性質を持つ。

- **原始性**

  トランザクションは完全に実行されるか、全く実行されないかのどちらかでなければならない。トランザクションが失敗したら、システムはトランザクション実行前の整合性がある状態に戻ることができなければならない。

- **永続性**

  トランザクションが一旦引き渡されたら、システムはそれが失われないことを保証しなければならない。

- **直列化可能性**

  並行に動くトランザクションは互いに干渉してはならない。平行に動く２つのトランザクションは、どのような順番で実行されようとも同じ結果をもたらさなければならない。

- **隔離性**

  トランザクションが引き渡される前に、中間状態を観察されてはならない。失敗 (abort) したトランザクションに依存したすべてのトランザクションは失敗させなければならない。

トランザクションのコミットログ (commit log) は、トランザクションの取り消しと再実行のための記録である。少なくとも以下の項目を含んでいる必要がある。

- **トランザクション ID**
- **命令タイプ (挿入・更新・削除など）**
- **古い値**
- **新しい値**

2 相コミット (two-phase commit) は、トランザクションの原子性を保証するための技法である。2 相コミットは、以下の規則に従って 2 段階 (2 相) でトランザクションのコミットを行う。

- **規則 1**

  各トランザクションはコミット点に達するまで、安定記憶[*35] へ書き込みを実行しない。

- **規則 2**

  各トランザクションの演算は作業領域で行い、その過程で生じるデータの変化をログに記録するまでコミットしない。

各相の処理は以下の通りである。

- **第 1 相の処理**

  更新準備のため、ログへデータを書き込む。

  この段階でトランザクションが失敗しても、安定記憶には影響を与えていないので、ログを破棄すればよい。

- **第 2 相の処理**

  ログ上のデータを、安定記憶へ書き込む。

  この段階の失敗は、ログを元に完全な形で安定記憶へ書き込むことができる。

(4) 並行性制御

並行性制御では、トランザクションの直列化可能性が満たされなければならないが、このための最も一般的な技法が 2 相ロックである。

---

35. データベースなど、記録がなくならないことを保証している格納領域のこと。

- **2 相ロック**

  2 相ロックは、第 1 相でトランザクションがロックを行い、第 2 相でロックが解除された後は、ロックが行われないことを保証する。これによってトランザクションの直列化可能性が保証される。

  しかし、これだけではトランザクションの隔離性が保証されないので、並行に動くすべてのトランザクションは、コミットするまで排他的ロックを掛けなければならない。従って、ロックの手順は以下のようになる。

  プログラム 3.1 ロック手順

  ```
  トランザクション開始
  read か write の前にロックする
  トランザクションの「仕事」をする
  コミットする
  ロックを解除する
  ```

  しかし、これだけでは、並行に動くトランザクションが同じオブジェクトをアクセスし、異なる順序でロックを行うとデッドロックが発生する。

  デッドロックを解消するには以下の手段がある。

  (a) デッドロックを発見し[*36]、トランザクションを失敗させ、再実行する。

  (b) ロックを一定時間後に解除する。

- **楽観的並行制御**

  楽観的並行制御は、資源の奪い合いはまずないと仮定する。この仮定は、コミット点でのみ成り立つ。その後、もし奪い合いが生じたらトランザクションを再実行する。

  この方法では、あるトランザクションの「仕事」結果が失われコミットできなくなる危険性があるが、プログラムが簡単になり実行効率も良くなるという利点がある。

---

36. デッドロックの発見方法は本書の範囲を越えるので『Edited by John A. McDermid, Software Engineer's Reference Book, Butterworth-Heinemann Ltd., 1991』などを参考にしていただきたい。

- **悲観的並行制御**

  前述のロック手順に従って、トランザクションのコミットの前にロックを掛ける。

  楽観的並行制御より確実であるが、やや複雑で実行効率が悪くなる。

- **時刻印トランザクション**

  トランザクションに時刻印 (Time-stamp) を付けることによってデッドロックを避ける手法である。

  トランザクションの直列化を保証するため、最新のトランザクションが更新したデータを読み書きしようとする、すべてのトランザクションを失敗させる。結果として、必要以上にトランザクションを失敗させるため効率上の問題は発生するが、デッドロックは発生しないことが保証される。

(5) 保護

分散システムの安全性は、以下の要因に分けられる。

- **安全への脅威**

  ネットワーク上を流れる情報を盗む、偽のシステムあるいはメッセージになりすます、ソフトウェアやハードウェアの欠陥などといった脅威への対処を行う。

- **アクセス制御**

  アクセス制御マトリクスを使い、誰がどのリソースにアクセスできるかを制御する、リソースにレベルを記したラベルを付け該当のレベルからしかアクセスできないといった制御を行う。

- **本物証明** (Authentication)

  人・メッセージ・システムなどが本物であることを証明する。暗号技術が必要になる。

(6) 複製 (Replication)

故障対策や実行効率向上のために、ファイルやオブジェクトの複製を作ることがある。この場合、複製されたファイルやオブジェクトの整合性を保持しなければならない。

主要なサイトを決めておき常にここから複製する方法[*37] や、重み付けをした投票権を使う方法[*38] がある。

3.8 算法 (Algorithm) パターン

問題を解くための具体的な算法を表す。空間効率と実行効率のトレードオフを提供する。デザインパターン中の戦略 (Strategy) パターンの中身の定義であるとも見なせる。設計の下流工程で使う。

本書では、混成 (Composite) パターンと関係の深い、辞書パターンのみ紹介する。

3.8.1 辞書 (Dictionary) パターン

集合の中から要素を探索する・挿入する・削除するの 3 つの操作を効率良く行うためのデータ構造と算法 (Algorithm) を辞書 (Dictionary) パターンと呼ぶ。辞書 (Dictionary) パターンを実現するための代表的なデータ構造が、線形リスト構造と木構造とハッシュ表である。

線形リスト構造は、探索・挿入・削除の計算量が O(n) であるので、要素数 n が大きいときには向いていないが、配列に比べ挿入・削除が簡単で、実装も木構造とハッシュ表に比べて簡単であるため、よく使われる。

線形リスト構造の例を以下に示す。

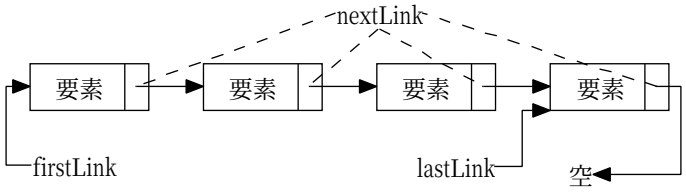

図 30 線形リスト構造

37. この場合は効率の向上にしか役立たない。
38. 方法の詳細は本書の範囲を越えるので『Edited by John A. McDermid, Software Engineer's Reference Book, Butterworth-Heinemann Ltd., 1991』を参考にしていただきたい。

上の図で、firstLink は線形リスト構造の先頭を指し、lastLink は線形リスト構造の最後の要素を指す。nextLink は「次の要素」を指す。

線形リスト構造を実現するクラス図の例を以下に示す。

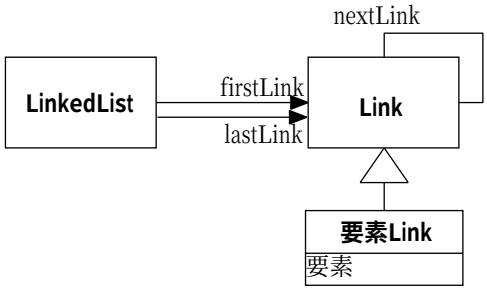

図 31 線形リスト構造のクラス図

上の図で、Link クラスは nextLink 関連を持っているだけなので「要素」を格納できないため、サブクラスの要素 Link クラスで線形リスト構造に格納したい要素を持つ。

木構造は、データの探索・挿入・探索がいずれもデータ個数を　n　個としたとき $O(\log_2 n)$ で行うことができる。しかも、挿入・削除を行っても「順序」が保たれるため、「順序」が重要なデータを格納するのに適している。

木構造のうちの 2 分木の例を以下に示す。

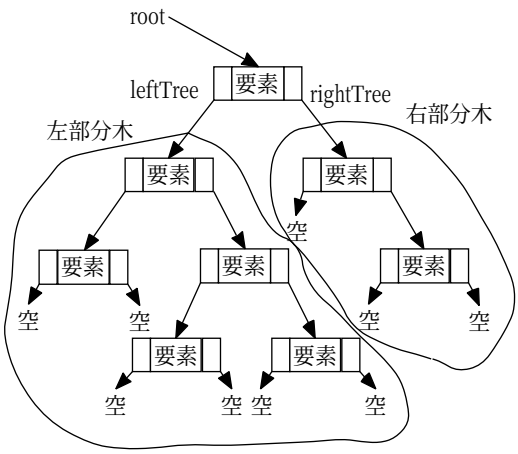

図 32 2 分木の構造

上の図で、root は木の根を示し、ここから探索などを始める。leftTree は、着目している要素の左側にある部分木構造を指し、rightTree は要素の右側の部分木構造を指す。

木構造を実現するクラス図は、混成 (Composite) パターンそのものである。2 分木の場合は、混成 (Composite) パターンが縮退した以下のようなクラス図で実現できる。

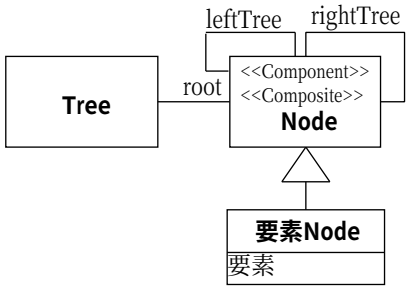

図 33 2 分木のクラス図

ハッシュ表は、O(1) すなわちデータ個数に関係なく一定の時間で探索・挿入できる。ただし、ハッシュ表は n が大きいときには向いていないし、「順序」は考慮されないので「順序」が意味を持つデータの格納には適していない。

ハッシュ表構造の例を以下の図に示す。ハッシュ関数[*39]でハッシュキーを計算し、その値が同じ要素をハッシュ表のハッシュアドレスを先頭にした線形リストにつなげる。探索する場合は、まずハッシュ関数を計算しハッシュアドレスを求め、以後は線形リスト構造中から要素を探し出す。

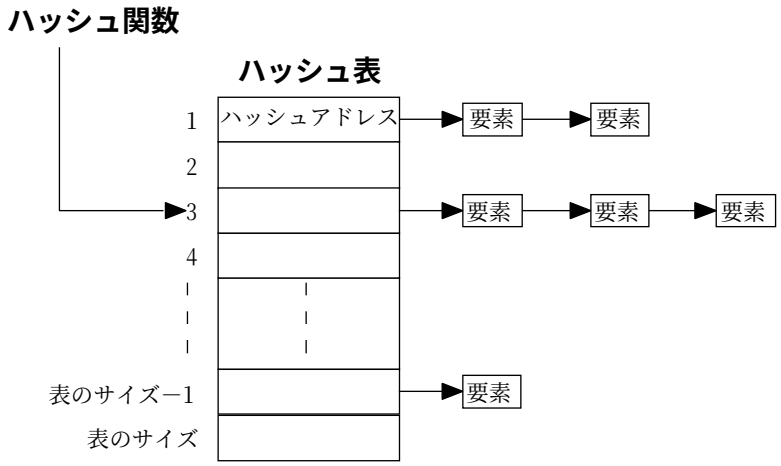

図 34 ハッシュ表の構造

ハッシュ表を実現するクラス図の例を以下に示す。

ここで hash 操作はハッシュ関数である。パラメータ key の型 oclAny は OCL の任意の型で、hash 操作は任意の型の key を受け取り Integer を返す。抽象クラス HashTable の hash 操作は抽象操作で、サブクラスの ConcreteHashTable クラスで実装する。key.oclType は、key の型を示す。

39. ハッシュ表を検索するためのキーを返す関数。ハッシュ表の大きさの範囲内になり、かつできるだけデータが重複しないように設計者が定義する。

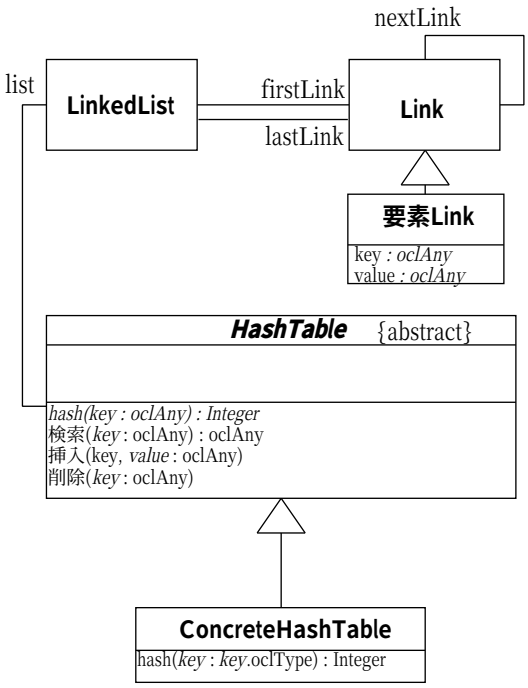

図 35 ハッシュ表のクラス図

HashTable の検索操作の仕様は以下のようになっている。

仕様 1 HashTable **クラスの検索操作の仕様**

(1) **検索** (key : oclAny) : oclAny

ハッシュ表を探索して、パラメータとして与えられた要素と等しいキーを持つオブジェクトを返す。

- **後件**

-- パラメータの key と結果のハッシュキーが等しく、かつ

-- パラメータの key と等しい key を持つ要素 Link オブジェクトがある。

-- あるいは、要素 Link オブジェクトに nextLink が空のものがあり、かつ

-- パラメータの key と同じ要素 Link オブジェクトがなければ、

-- 結果は false である。

(self.hash(result) = self.hash(key） and

要素 Link.allInstances->exists(each : Link | key = each.key)) or

((要素 Link.allInstances->exists(end : Link | end.nextLink->isEmpty) and

要素 Link.allInstances->forAll(each : Link | key <> each.key)) implies result = false)