Chapter 1

オブジェクト指向によるプログラミング

ソフトウェアの開発効率を高めるためには、他の工業製品と同じように、ソフトウェアの部品化が有効であると考えられる。同時に、それぞれの部品はソフトウェアの仕様に合わせて柔軟に拡張できなければならない。このように、開発効率の高いソフトウェアは、部品としてのモジュール性を維持しながら (閉じたプログラム)、同時に拡張性を持つ (開いたプログラム)、という性質を両立する必要がある [40]。このような命題に対して、従来から、さまざまなソフトフェア開発手法が試みられてきたが、十分な成果を挙げているとは言えない。

その中で、オブジェクト指向に基づいたソフトウェア開発では、モジュール性と拡張性を両立できる点が注目され、さまざまな開発フレームワーク、開発言語を生み出している。

本章では、現在のソフトウェア開発工程で広く利用されているオブジェクト指向プログラミングについて、その特徴をまとめた上で、従来のオブジェクト指向プログラミングを基盤とした様々な開発手法を取り上げ、それらの利点／問題点を議論する。

1.1 オブジェクト指向プログラミング

オブジェクト指向プログラミングは、ソフトウェアの生産性を高めるものとして、ソフトウェア開発におけるプログラミング手法の主流となりつつある。現在では、オブジェクト指向は、分析／設計の工程でも採用されており、実装の直前まで柔軟性を保持した設計が可能となっている [13][50]。また、実装工程においても、オブジェクト指向に基づいてプログラミングすることによって、設計の変更に対して柔軟に対処できる実装が可能となる。これらの理由から、多くのソフトウェア開発プロジェクトが、ソフトウェア開発の全工程にオブジェクト指向を導入するようになっており、実装工程でのオブジェクト指向プログラミングは、ソフトウェア開発に不可欠な手法となっている。

オブジェクト指向プログラミングは以下の4つの項目によって定義される。

オブジェクト オブジェクトとは、プログラムの概念的な基本構成要素である。オブジェクト指向プログラミングでは、この基本構成要素を “もの (オブジェクト)” として扱い、その性質をクラスとして定義する。オブジェクトとはクラスで定義された “もの” の

実体である。プログラムは、オブジェクト同士の結合として構築され、オブジェクト相互の連係によって動作する。

カプセル化 カプセル化とは、オブジェクトの内と外の境界を明確にすることである。オブジェクトのデータ構造と実装の詳細は、他のオブジェクトから隠蔽される。クラスではオブジェクトの状態を操作するための唯一のアクセス手段 (インタフェース) であるメソッドを定義する。このようにオブジェクトの内部情報を隠蔽することで、そのオブジェクトにどのようなメソッドが提供されているかさえわかっていれば、オブジェクトの提供するサービスを利用することができる。その際、オブジェクト内部の振舞や状態を知る必要がない。

継承 継承とは、クラスの性質の全部あるいは部分を他のクラス (サブクラス) に引き継ぐことである。あるクラスから、部分的に性質の追加や削除を行った新しいクラスを手にいれたい場合、既存のクラスの性質を引き継いだクラスを新たに定義する。この場合のもとのクラスをスーパークラス、スーパークラスの性質を引き継いだクラスをサブクラスと呼ぶ (この作業をクラスを「派生」すると言う)。クラス派生において、サブクラスがスーパークラスのメソッド (クラスにアクセスするための手続き) やメンバ変数 (クラスの状態を保持する変数) を継承する場合は、スーパークラスの性質がすべてサブクラスに引き継がれる (これを実装の継承と言う)。その上で、プログラマはサブクラスで必要な拡張／制約機能を再定義 (オーバーライド) する。つまり、プログラマはサブクラスで必要なスーパークラスとの差分のみを定義すればよい (差分プログラミング)。

実装の継承では、クラス派生によって共通の性質を持つクラスが階層状に組織化された構造ができる。しかし、クラス階層の異なる部分の性質を合わせ持ったクラスを新たに定義することは難しい。一つの解決策として多重継承 (複数のスーパークラスから派生すること) を用いる方法があるが、多重継承では、曖昧性 (どのスーパークラスのメソッドや変数を参照しているのか) を解決するために、クラス定義やインスタンスの生成、メソッドの呼び出しを行うコードが複雑になるという問題がある。

そこで、実装とは別にクラスのインタフェースだけを引き継ぐ方法が提案されている (サブタイピング)。サブタイピングでは、クラスのメソッド本体やメンバ変数は引き継がず、実装は各クラスで個別に行う。サブタイピングでは、スーパークラスの実装に影響されることなく、複数のインタフェースを持ったクラスを実現することができる。多重継承についても、サブタイピングではその都度実装を行うので、実装の多重継承に見られる諸問題が発生しない。一方で、サブタイピングでは、継承のたびに個別に実装を行わなければならず、スーパークラスの性質を利用した差分プログラミングを行えない。

多相性 多相性とは、使用するオブジェクトの選択を実行時まで遅らせることである。オブジェクトの利用者は、オブジェクトが提供するサービスをメソッド呼び出しの形で利用する。一つの呼び出しについて実際に実行されるメソッドのコードは実行時に決ま

る。すなわち、オブジェクト指向言語では、スーパークラスのオブジェクト変数にサブクラスのオブジェクト変数を代入できるので、実行時に、実際にオブジェクト変数に結びつけられているオブジェクトによって、どのクラスのメソッドが呼ばれるかが動的に定まる (動的束縛)。動的束縛を利用すれば、スーパークラスのメソッド呼び出しによってプログラムの骨組みを生成し、後から具体的にサブクラスのオブジェクトを割り当てることができる。つまり、オブジェクトを利用する側で、スーパークラスの型を用いれば、新たに生成したサブクラスのオブジェクトを扱う場合にも、それを利用する側のコードを変更する必要がない。

オブジェクト指向プログラミングで用いるクラスは、カプセル化によって内と外の境界が明確であるため、他の工業製品の部品のように独立した物として取り扱える。また、新しい部品クラスを作成する場合に、クラス派生を使うことで、これまで定義したクラスの実装を無駄にすること無く、新しい機能との差分を記述するだけで済まされる。従って、クラス階層の設計時に将来の拡張を十分考慮すれば、クラス派生による差分プログラミングによって、プログラムを効率的に拡張できる。さらに、多相性を用いることで、拡張された未知のクラスのオブジェクトを利用する場合に、それを利用する側のコードを変更する必要がなく、柔軟な拡張が可能である。

以下では、オブジェクト指向に基づくプログラム開発を、クラス階層の設計とオブジェクト間結合の設計の二側面に分けて、それぞれの問題点を整理する。その後、オブジェクト指向プログラミングをベースとした各種のアプリケーション開発の骨組みについて具体的に検討し、それらの特徴や問題点を列挙する。

1.2 オブジェクト指向プログラミングの開発側面

クラスをベースとしたオブジェクト指向プログラミングでは、図 1.1に示すように、クラスの設計を行う “a-kind-of” 関係構築とインスタンスの結合を行う “a-part-of” 関係構築の二側面があり [13]、プログラマはそれぞれの側面でプログラムを設計する。以下に各側面でのプログラム開発について整理する。

1.2.1 “a-kind-of” 開発側面

開発の “a-kind-of” 側面では、プログラマはクラス階層を構築する。スーパークラスの実装を継承したサブクラスを生成し、サブクラスでスーパークラスとの差分のみを記述することで (差分プログラミング)、効率のよいプログラム開発が可能である。Smalltalk-80 システム [25] や Java のクラスライブラリ [38] を用いた独自クラスの開発がこれに相当する。その反面、“a-kind-of” 側面での開発には、以下のような問題がある。

- 将来の拡張に備えたクラス階層の構築は困難
- 複数のスーパークラスの性質を合成するのは簡単でない (多重継承の問題)
- サブクラス化によりクラスのカプセル化が壊れる (クラスの脆弱性の問題)

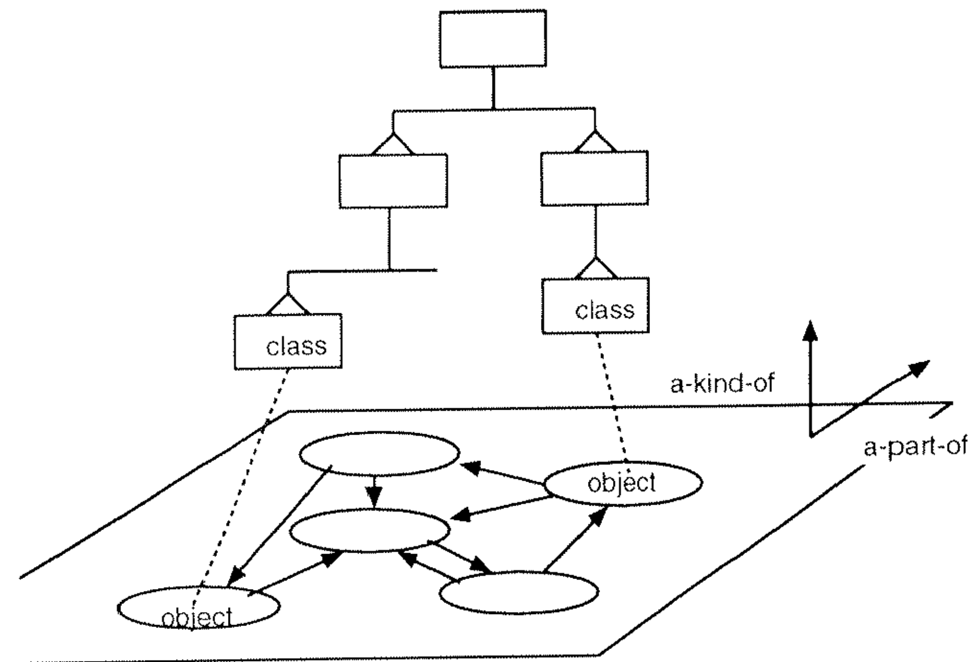

図 1.1: オブジェクトの開発側面

これに対して、Java などで用いられているクラスのインタフェースのみを継承する方法では、多重継承や脆弱性の問題は回避できるが、差分プログラミングのメリットを享受できない。一般に、"a-kind-of" 関係の設計はクラス階層を十分に理解し、実装とインタフェースの使い分けが可能な上級のプログラマでなければ行えない。

1.2.2 "a-part-of" 開発側面

開発の "a-part-of" 側面では、プログラマは、ある程度汎用的に作られた部品クラスのインスタンス (コンポーネント) を、フレームワーク上で相互に結合してプログラムを構築する。これにより、必要な機能をもったコンポーネントを収集し結合するだけで、容易にプログラム開発が可能である。Active-X[16] や JavaBeans[22] などのコンポーネント群を用いたプログラム開発がこれに相当する。しかし、コンポーネントに基づく開発には以下のような問題がある。

- 不特定多数のコンポーネント同士を相互に結合させるための結線が煩雑であり、かつ容易に結合状態が見通せない
- 一般に大規模ソフトウェアでは、コンポーネント相互の依存関係は複雑に絡み合っており、プログラムの拡張をコンポーネント間の依存関係を維持しながら、a-part-of の側面だけで行うのは困難である。

これに対し、コンポーネントの結合構造に対するフレームワーク (MVC フレームワークなどが代表的) やパターンを用意することで、構造の見通しを良くする方法があるが [16][6][24]、これらがカバーするのはプログラムの一部であり、プログラム全体を一つのフレームワークに基づいて構築することは難しい。

以上のように、プログラム開発を "a-part-of","a-kind-of" に分業したとしても、現状ではどちらにも問題があり、プログラマは両方に精通しなければならない。

なお、差分プログラミングの手段として、クラスとサブクラスではなく、コピーと委譲に基づくようなオブジェクト指向言語 [58][9] もあるが、あるオブジェクトを雛型として新しいオブジェクトを定義する部分は概念的に "a-kind-of" に相当するので、前述の問題点があてはまる。

以下では、オブジェクト指向をベースとした開発フレームワークを具体的に見ていく。

1.3 クラスライブラリを用いた開発

1.3.1 クラスライブラリとは

クラスライブラリとは、アプリケーション構築に必要な汎用的な機能 (文字処理、I/O 処理、ハッシュテーブル、GUI 部品等) を、クラス群にカプセル化したものである。クラスライブラリの利用により、プログラムを記述するたびに必要な定型的な処理をプログラム中に簡単に組み込むことができる。

クラスライブラリには、ANSI C++標準ライブラリ [46]、Booch コンポーネント [28]、Smalltalk-80 標準ライブラリ [25]、X-Window の C++ツールキットである InterViews [4]、Fresco [29]、Java のクラスライブラリ [38] などがある。

1.3.2 クラスライブラリの問題点

オブジェクト指向プログラミングの初期の成功例である Smalltalk-80 システム [25][26] では、その豊富なクラスライブラリを活用することで効率のよいソフトウェア開発が可能であると言われた。しかし実際には、クラスライブラリの利用だけでは、ソフトウェアの生産性は向上していない。それは、以下のような理由による。

利用性の問題 通常クラスライブラリは、数百ものクラスから構成され、その情報量は膨大なものとなっている。クラスライブラリに属するクラス間の複雑な関係は、各クラスライブラリ毎に独自の規約に基づいており、規約を理解しなければクラスライブラリを利用することは難しい。したがって、各種のクラス群を熟知し、それらをどのように組み立てていけば整合性のあるソフトウェア構造が成立するのかを見抜くことは、かなり経験のあるプログラマにとっても容易ではない。

粒度の問題 クラスライブラリはモジュール単位が小さいため、クラスライブラリの再利用を主体に大規模なソフトウェアを構築しようとすると、多くのオブジェクト間の結合

が必要になる。結果的に、オブジェクトの接続にかかるコストが膨大になり、クラスライブラリを用いる効果が現れない。また、複雑なオブジェクト間の結合関係が構築され、プログラムから部分的にオブジェクトを抜き出して別の用途に利用するといったことが困難になる。

以上のような理由から、クラスライブラリ単独での利用はあまり普及していない。クラスライブラリは、プログラミング上不可欠であるが、後述するフレームワークと組み合わせて利用する必要がある。そのような使い方として、MFC(Mircosoft Fundation Class) ライブラリ [16] の基本データ型 (CObject クラスから派生していないクラス) などが挙げられる。

1.4 デザインパターンを用いた開発

1.4.1 デザインパターンとは

1970 年代の建築設計において、Alexander [3] は、設計のノウハウをパターン化し、再利用できる事を示した。その後、Coad [14] がソフトウェアの設計においても、パターンの利用が有効であることを提唱した。

デザインパターンとは、パターンの再利用をソフトウェア開発の分野に流用し、オブジェクト指向プログラミングでたびたび利用される設計のパターンを抽出し、利用可能にしたものである。デザインパターンでは、クラス間の関係をいくつかの典型的な「パターン」に分類することにより、小さい部品であるオブジェクトの構造を組み立てる手がかりを提供している。

デザインパターンの利点としては、以下の点が挙げられる。

- プログラム設計者間での共通言語として役立つ。プログラマが他のプログラマに、あるフレームワークを説明する際に、デザインパターンを共通の知識として持っていれば、そのフレームワークがどのデザインパターンに当てはまるかを述べることで、容易にフレームワークの構造を説明できる。

- 典型的なオブジェクト指向設計の例題として、初心者によるオブジェクト指向設計に役立つ。

デザインパターンの実施例としては、ET++GUI アプリケーションフレームワーク [5] や、Choice オブジェクト指向 OS [30] 、HotDraw [18] ドキュメンティングフレームワークなどがある。これらのプロジェクトは、デザインパターンを利用したプログラミングを行い、その設計ドキュメントをデザインパターンを用いて記述している。

1.4.2 デザインパターンカタログ

Coplien らは、さまざまなソフトウェアの設計問題をパターン化する試みとして、デザインパターンによる解法を [45][44] に集約している。また、Gamma らは代表的なデザインパ

表 1.1: デザインパターンカタログの内容 ([24] より引用)

目的	パターン名	内容
生成に関するパターン	Abstract Factory	部品オブジェクトの集合
	Builder	複合オブジェクトの生成方法
	Factory Method	インスタンス化されるサブクラス
	Prototype	インスタンス化されるクラス
	Singleton	クラスの唯一のインスタンス
構造に関するパターン	Adapter	オブジェクトへのインタフェース
	Bridge	オブジェクトの実装
	Composite	オブジェクトの構造とコンポジション
	Decorator	サブクラス化を伴わないオブジェクトの責任
	Facade	サブシステムのインタフェース
	Flyweight	オブジェクトの格納コスト
	Proxy	オブジェクトへのアクセス方法
振舞に関するパターン	Chain of Responsibility	要求を満たすオブジェクト
	Command	要求を満たすタイミングと方法
	Interpreter	言語の文法と解釈
	Iterator	オブジェクトの走査方法
	Mediator	オブジェクト間相互作用
	Memento	オブジェクトの外部に保存される私的な情報
	Observer	依存するオブジェクトの監視
	State	オブジェクトの状態
	Strategy	アルゴリズムの交換
	Template Method	アルゴリズムのステップ
	Visitor	オブジェクトへの操作の適用

ターンを、デザインパターンカタログ [24] としてまとめた。デザインパターンカタログでは、オブジェクト指向プログラミングにおける典型的設計ノウハウを表 1.1のように分類している。この分類にもとづいて、プログラマは、それぞれに固有の設計問題に即したデザインパターンを見つけ、利用する。例えば、GUI アプリケーションを、複数の look-and-feel 規格に対応するようにしたい場合は、生成に関するパターンの中から Abstract Factory パターンを選択し適用する。

1.4.3 デザインパターンの問題点

デザインパターンそのものは、あくまでプログラム構造に対する部分的なヒントであり、デザインパターンを使ったプログラム実装手段が用意されているわけではない。従って、デザインパターンによる利用が直接、完全なアプリケーションの構築には結び付つかない。また実際のプログラミングでは、デザインパターンには当てはまらない、さらに細かいレベルの実装パターンが存在し、デザインパターンだけでは、プログラムの構造を完全に説

明するのに必ずしも十分ではない。

1.5 フレームワークを用いた開発

1.5.1 フレームワークとは

フレームワークとは、特定ドメインのプログラム群に共通する骨組みを抜き出し、雛型化したものである。

例えば、GUI 環境のアプリケーションソフトウェアには様々な種類があるが、ウインドウが現れてその中で作業して終るという骨組みは共通である。このような GUI アプリケーションにおける定型部分を、フレームワークとして構築しておく。フレームワークに基づくソフトウェアの開発では、既存のフレームワークに対して、各アプリケーション毎の独自部分のみを記述する。

このような特定のアプリケーションドメインに特化したソフトウェアの骨組みのことを、アプリケーションフレームワークと呼ぶ。

1.5.2 クラスライブラリとの違い

クラスライブラリを用いたプログラミングでは、プログラマはソフトウェアの骨組みを組み立て、足りない部分をライブラリの中から選び利用する (図 1.2(a))。

一方、フレームワークを用いたプログラミングはその逆になる。プログラマは、ソフトウェアの骨組みとしてフレームワークを利用し、フレームワークにアプリケーション固有の部分を付け加える (図 1.2(b))。

フレームワークの動作例として、ドキュメントを扱うソフトウェアを C++で記述する場合を説明する。ドキュメントの抽象クラス Document クラスでは、ファイルメニューの「Open」が選択された時に呼ばれるメソッドとして、`FileOpen()` を定義する (C++では、サブクラスでオーバーライドする関数を`virtual` と宣言する必要がある)。

```
class Document{
public:
   virtual void FileOpen();
};
```

フレームワークは、ファイルメニューの「Open」が選択された時に、

```
Document *doc -> FileOpen();
```

のように、Document クラスのオブジェクト変数に対して、`FileOpen()` メソッドを呼ぶように定義している。`void Document::FileOpen()` メソッドはデフォルトのファイル Open メニューに対する動作 (通常は何もしない) を定義している。

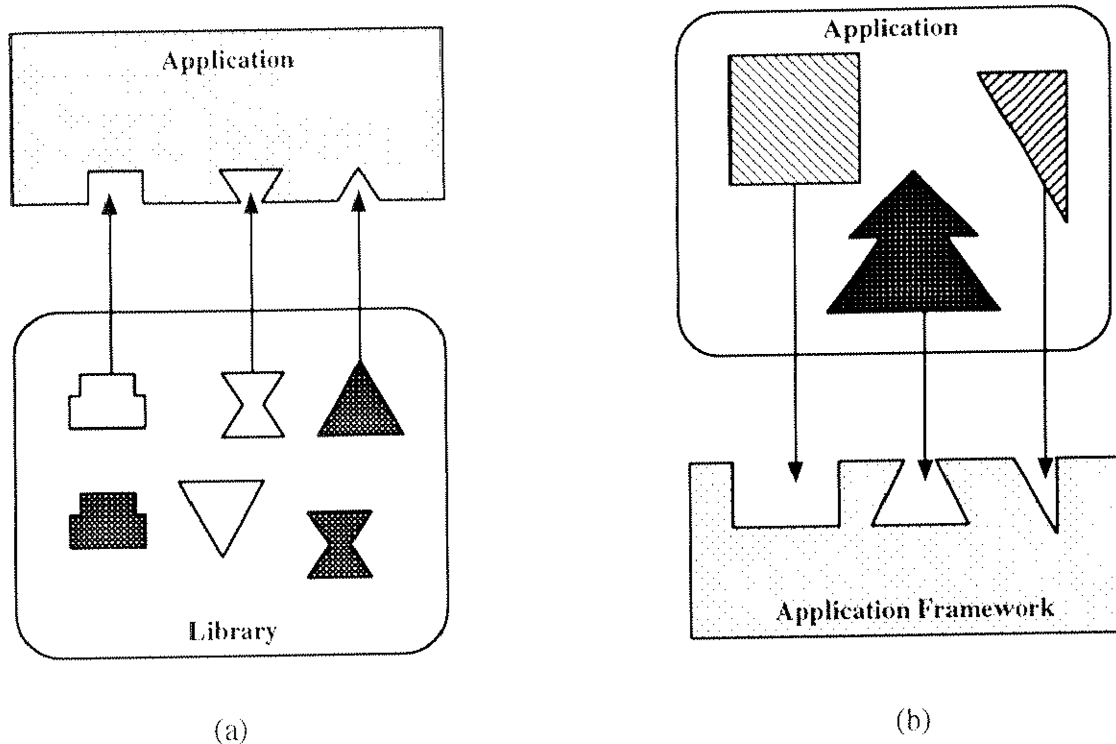

図 1.2: クラスライブラリ (a) とフレームワーク (b) の位置付け

これに対して、個別のアプリケーションで「Open」に対する固有の応答を実現したい場合は、Document クラスから MyDocument サブクラスを派生し、`FileOpen()` をオーバーライドする (図 1.3左)。

```
class MyDocument : public Documemt{
public:
  void FileOpen();
};

void MyDocument::FileOpen()
{
  //独自の「ファイル開く」処理の記述
}
```

次に、フレームーワークが利用する Document クラスのオブジェクトを MyDocument クラスのオブジェクトに置き換える (図 1.3右)。これは、例えばフレームワークの内部で、Document クラスを生成する関数`Document *CreateDocument()` を用意しておき、Document

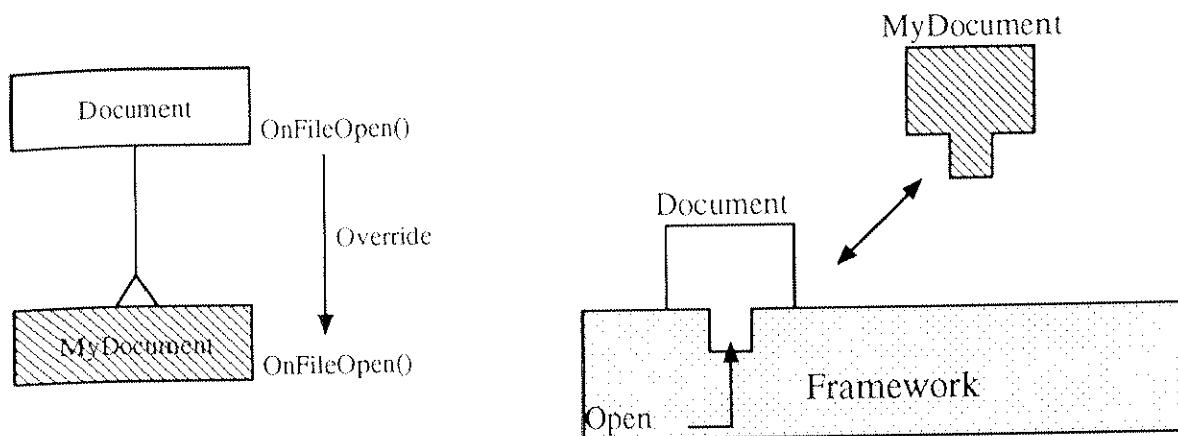

図 1.3: フレームワークの利用

オブジェクトの生成にはこの関数を呼ぶようにした上で、以下のようにする。

```
Document *App::CreateDocument()//App はフレームワークのクラスとする
{
    return new MyDocument();
}
```

これにより、`doc->FileOpen()` で実行されるのは、`MyDocument::FileOpen()` となり、アプリケーション固有の「Open」に対する動作が実行される。

このように、フレームワークの利用では、クラス派生によってアプリケーション固有の部分を記述する。オブジェクト指向の多相性により、プログラマがどのような派生クラスを作成したとしても、フレームワークは、それらをスーパークラスのインスタンスとして統一的に扱うため、フレームワーク側のコードを変更する必要が無い。この仕組みによりフレームワークの利用が可能になっている [64]。

1.5.3 アプリケーションフレームワークの例

GUI のアプリケーションフレームワークとして最初のものは、Smalltalk-80 の MVC フレームワーク [21] [25] である。商用で最初に MVC フレームワークを採用したものとして、Macintosh の開発環境である MacApp [6] があげられる。その他のアプリケーションフレームワークとしては表 1.2 のようなものが報告されている。その他にも、VLSI のデザイン [27]、構造化エディタ [43]、最適化コード設計 [20]、物理実験 [23]、などを挙げることができる。

また、統合開発環境で結果的にアプリケーションフレームワークが用いられる例として、Microsoft が Visual C++(VC++) などと共に提供する Microsoft Fundation Class(MFC) [66]

表 1.2: 主なアプリケーションフレームワーク

フレームワーク名	内容
ET++ [5]	UBS(Union Bank of Switzerland Information Laboratory) で Andre Weinand,Erich Gamma 氏らが開発した、C++を用いたグラフィカルユーザインタフェースのための MVC フレームワーク。その後、チューリヒ大学の Phillipp Achermant 氏,DominikEichelberg 氏によってマルチメディア向けに拡張して MET++となる。2 次元,3 次元グラフィックス、GUI コンポーネント、サウンドなどの可能コンポーネントを提供。
Andrew Toolkit [2]	IBM&カーネギーメロン大学のジョイントベンチャーの成果。AUIS(Andrew Userinterface System) として配布。ATK(Andrew Toolkit) オブジェクト指向プログラミング環境を持つ。開発は C 言語だが、C++に似たオブジェクト指向言語に拡張。class というツールを利用し、プリプロセスにより C 言語に変換。Andrew のオブジェクト指向プログラミングは独特。ソフト開発を自在に行うには、かなりの勉強が必要。
Choice [30]	Roy Campbell らによって開発されたオブジェクト指向 OS のフレームワーク。ハードウェア、アプリケーションなどのあらゆるインタフェースを抽象化し様々な環境に対応できる OS。実装は C++によるが、ガベージコレクション、実行時サブクラス化、クラスオブジェクト導入など先進的機構を持つ。

や、OpenDoc の開発ツールキットである OpenDoc Development Framework(ODF) [84] がある。

1.5.4 フレームワークを用いた開発の問題点

フレームワークは、第 1.5.2節で示したように、クラス派生によりプログラムを拡張する手法をとっている。クラス派生は、プログラミング言語によって直接支援され、クラスの実装方法を変更すれば、コンパイル時に変更内容を静的に反映させられる。しかし、クラス派生では、継承関係はコンパイル時に確定されるので、実行時に実装方法を変更することはできない。加えて、クラス派生によりクラスの実装を継承する場合、あるクラス派生による継承が部分的に新しい問題領域に適さない場合に、より抽象度の高いスーパークラスからクラスを派生したり、スーパークラス自体を変更する必要が生じる。

この問題の解決策として、クラスのインタフェースをのみを継承し、実装を継承しないという方法 (サブタイピング) がある。プログラマは、インタフェースが同じである限り、様々なオブジェクトを動的に結合／連係させ、より複雑な機能を得ることができる。このようなプログラム手法をオブジェクトコンポジションと呼ぶ。オブジェクトコンポジションで用いるオブジェクトは、インタフェースの継承のみを行い、実装は個別に行っている

ためクラスのカプセル化が保たれている。

クラス派生による拡張を行うフレームワークでは、拡張のためにクラス階層を熟知している必要があり、フレームワークを習得するのに時間がかかる。加えて、多くのフレームワークは独自のプログラミング規約を持っており、初心者はフレームワークに慣れるまでにかなりの労力を要する。例えば、ET++ [5] の場合は、前述したデザインパターンによってフレームワークの特性が記述されており、理解のための障壁は低いと考えられる。しかし実際のプログラミングでは、デザインパターンでは分解しきれない独自の手法が必要であり、かなりの熟練したプログラマでなければ利用は難しい。具体的な要因として、以下のような点が挙げられる。

- オブジェクトコンポジションが取り込まれていない。例えば、プルダウンメニューの構成変更にクラスを派生しなければならず、メニューアイテムのオブジェクトを追加／登録するだけですむような構成になっていない。
- オブジェクトの生成や消滅が行われる箇所がまちまちである。
- オブジェクトが生成される順番をプログラマが意識しなければならない。
- 任意のオブジェクト間の通信手段を欠いている。
- 新規クラスの開発には、動的にクラス情報を入手するために膨大なマクロを記述しなければならない。

これらの傾向は、多くのフレームワークで共通して見られる。フレームワークは、クラスライブラリと比較して、大規模ソフトウェアの開発において、高い再利用の効果を発揮すると思われる。しかし、フレームワークの利用を普及させるには、オブジェクトコンポジションを採り入れたり、構造に一貫性を持たせる等の工夫により、フレームワークを用いるための障壁を低くする必要がある。

1.6 コンポーネントを用いた開発

1.6.1 コンポーネントを用いた開発とは

ソフトウェア部品の利用形態として、ブラックボックス化されたインスタンス (コンポーネント) を対象にすることで開発効率を高めようとするアプローチがある。アプリケーションフレームワークが、アプリケーション全体を 1 つの骨組みとして開発するのに対して、コンポーネントの利用では、ある程度の大きさの部品 (たとえばスプレッドシートのシートなど) を複数用意して、それらを組み立てることによってアプリケーションを構築する (オブジェクトコンポジション)。それぞれの部品はある程度汎用性を考慮しているため、そのような部品だけで構成できるアプリケーションは短期間に効率よく開発可能である。また、部品の組み立てはプログラミングを必要としないため、ユーザ自身によるソフトウェア開発が可能だとされている。

1.6.2 コンポーネントの例

コンポーネントを用いた開発では、以下のような点でこれまでのオブジェクト指向のアプローチと異なる。

- クラスの利用からインスタンスの利用へ
- 実装の利用からインタフェース (サービス) の利用へ
- アプリケーション中心からタスク (ドキュメント) 中心へ
- アーキテクチャ独立支援

オブジェクト指向は難解であり、エンドユーザにはその理解が負担となっていた。そのため、これまで見てきたクラスライブラリやフレームワークなど、オブジェクト指向をベースとしたプログラム開発がなかなか普及しなかった。

一方コンポーネントを用いたプログラムでは、エンドユーザでもすぐに使えるブラックボックス的な利用に割り切り、拡張性を犠牲している。しかし、コンポーネントを用いることにより、エンドユーザでもあるアプリケーションドメインの技術者が直接コンポーネントを用いてソフトウェアの構築ができるようになった。コンポーネントベースのプログラミングは、エンドユーザの一部をプログラマとして取り込むことにより、プログラム人口の裾野を広げ、最終的にソフトウェアの生産性を高めるアプローチであるとも言える。

現在では、ActiveX/COM [67]、OpenDoc/CORBA [71] [34]、Java Beans [22] など、様々なソフトウェアプラットフォームや、計算機アーキテクチャで動作可能なコンポーネントの仕様が公開され、多くのベンダーがコンポーネントの市場に参入している (表 1.3)。

1.6.3 コンポーネントを用いた開発の問題点

エンドユーザへの部品の提供を完結したコンポーネントで行うという形態は、今後定着する可能性が高いと思われる [52]。しかし、コンポーネントの適用は、定型化されたアプリケーションドメインに限られる。それ以外の非定型な分野では、新たにコンポーネントを開発する必要があるが、非定型な処理を汎用的に扱えるコンポーネントの開発は難しい。

コンポーネントはインタフェースだけが明示されており、他のクラスの実装が継承されていないため、カプセル化が強固に保たれている。しかし、実装を継承しないので差分プログラミングが成り立たない。そのため、コンポーネントの拡張性は貧弱なものであり、非定型分野のコンポーネント開発に大きな問題になると考えられる。

また、新たな専用コンポーネントを開発する場合のプログラミングは、前述のクラスライブラリやデザインパターンのレベルに留まっており、非プログラマであるエンドユーザには行えない。コンポーネントの開発は、コンポーネント開発のためのライブラリ (VC++ の AppWizard で用いる MFC や ATL(ActiveX Template Library) など) を用いても、コンポーネントの内部構成やアーキテクチャを熟知したプログラマでなければ困難である。

一般にコンポーネントのプログラミングは、ActiveX の開発環境である VC++の AppWizard [16] や、JavaBeans の開発環境である Visual Cafe,JBuilder [22] などのグラフィカ

表 1.3: 主なコンポーネント製品

製品名	開発元	備考
Delphi[86]	Borland	Delphi コンポーネント。Object Pascal を用いてホワイトボックス的な拡張も可能。グラフィカルな統合環境でソフトウェアの開発が可能。
IntelligentPad[88]	日立, 富士通	北海道大学 田中譲研究室で開発されたパッドを使ったソフトウェア開発環境。パッド同士をスロット結合することによりプログラミング可能。パッド自体の開発は C++で行う。
APPGALLARY	日立	IntelligentPad 同様、部品の結線によりプログラミングが可能。スクリプト言語も備える。
VBX[78]	Microsoft	Microsoft が開発したブラックボックス的コンポーネントの草分け。同社の開発環境 Visual Basic 上で利用可能なコンポーネントとして広く流通。その後 OCX として、Visual Basic 以外でも利用可能な部品に発展
ActiveX[67]	Microsoft	Microsoft が提唱する COM(Componet Object Model) に従ったコンポーネント (コントロール)。Visual Basic,Visual C++などの各種開発環境で利用できるだけでなく、WWW を介してブラウザ上での利用も可能。
PowerObject	Oracle	Basic ライクなスクリプト言語で記述可能。様々なデータベースにアクセスするためのコンポーネントが充実。見栄えのよいクエリプログラムが容易に開発可能。
JBuilder[61]	Borland	Java で記述したコンポーネントと開発ツールとのインタフェースを Java で標準化したコンポーネント JavaBeans の一開発環境。JavaBeans のガイドラインに従って作成されたコンポーネントは、JavaBeans 対応の各種開発ツール上で共通に使用することができる。

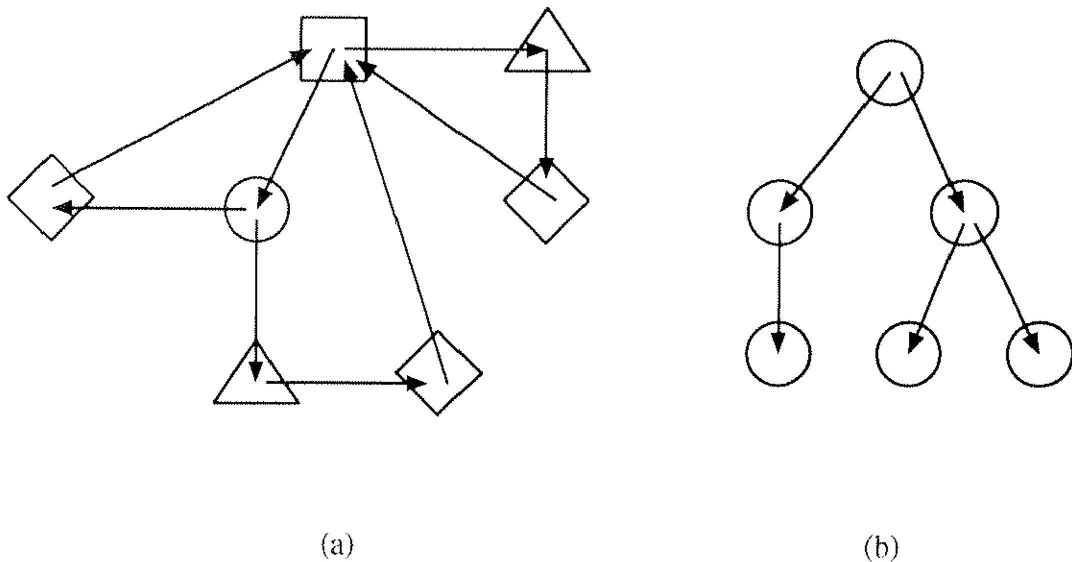

図 1.4: オブジェクトグラフ

ルな開発環境を用いる。しかし、これらの開発環境の助けを借りたとしても、新規コンポーネントの作成はかなり煩雑なプログラミングを必要とする。

1.7 考察

オブジェクト指向を中心とした現在のプログラミング手法は、クラスから、コンポーネントを対象としたインスタンスの利用へと移行しようとしている。

しかし、従来のフレームワークを用いた開発では、すべてのコンポーネントを単一の方法でアクセスできないため (すべてのコンポーネントが同じに見えない)、フレームワークの利用にはコンポーネント毎の仕様を習得しなければならない。また、コンポーネント間の接続関係についても、デザインパターンでは吸収できない各フレームワーク毎のプログラミング規約があり、統一性を欠いている。

これらの問題に対して、筆者が提案するフレームワークである Nuts では、以下のような特徴を持ったコンポーネントアーキテクチャを提案している。

- コンポーネントのインタフェースの統一 (全体構造の木構造化)
- 継承の利用による柔軟な拡張性 (差分プログラミング)

Nuts コンポーネントアーキテクチャの利点をオブジェクトグラフで説明する。現在のコンポーネントベースプログラミングは、図 1.4(a) に示すように、部品の種類毎に結合する相手が決まっている。プログラマはそれぞれのコンポーネントの種類を気にしつつ、それらを適切に接続する技術を修得しなければならない。

これに対して、結合のためのインタフェースを統一したプログラミングでは、図 1.4(b) のように、すべてのコンポーネントを区別なく扱えるため、コンポーネント間の結合を機械的に行うことができる (相手の種類などを気にする必要がない)。また、構造に規則性を持たせることにより、構造の理解が容易になるとともに、その変更も容易になる。

例えば、オブジェクトグラフを図 1.4(b) のように木構造にすれば、木構造が持つフラクタルな性質を利用したプログラムが可能である。一つの木構造は複数の部分木からなっており、部分木もその中に含まれる部分木から成り立っている。このように、木構造は全体と部分が区別されないフラクタルな構造をもっている。コンポーネントベースのプログラミングに、フラクタルな構造を当てはめた場合、単一の部品と、複数の部品が組み合わさってできた複合部品、さらにそれらが組み合わさった高機能部品を区別することなく扱える。これは、プログラミングを行う際に、部品やその塊を扱う上で有効な性質である。

さらに、継承を用いることで、クラス派生に基づいた差分プログラミングにより、コンポーネントを柔軟に拡張ができることが望ましい。しかし、現在のコンポーネントベースプログラミングは、インスタンスの利用を対象にした完全なブラックボックス的なものであり、クラス派生の利点が失われている。

すなわち、現在のコンポーネントベースプログラミングは、オブジェクト指向プログラミングの「オブジェクト」および「カプセル化」の特徴は持っているが、「継承」「多相性」の特徴を完全には持たない。これは、コンポーネントの普及を促進しようとする狙いがあってのことであり、エンドユーザに近いレベルのプログラマには受け入れやすいが、大規模かつ柔軟なソフトウェア開発を担うプログラマの要求を充分には満たしていない。これらのプログラマの要求を満たすためには、差分プログラミングによる拡張性を維持したコンポーネント、およびそれらのコンポーネントを統一的に扱える枠組が必要である。

以下の章では、ホワイトボックス的/ブラックボックス的プログラミングの観点から現在のオブジェクト指向プログラミングの問題点を整理する。その上で、両者の利点を兼ね備えた、オブジェクト指向に基づく新しいコンポーネントベースプログラミングである Nuts フレームワークを提案する。