14 オブジェクト指向プログラミング

14.1 オブジェクト指向設計

前章では、プログラムを構造化するための言語サポートについて議論したが、 「プログラムはどのようにモジュールに分解されるべきなのか」という問いに答えようとはしなかった。通常、このテーマはソフトウェア工学のコースで学習されるはずだが、オブジェクト指向プログラミング（OOP）と呼ばれる分解方法の1つは非常に重要であり、現代のプログラミング言語はこの方法を直接サポートしている。次の2章では、OOPをサポートする言語のトピックを取り上げる。

プログラムを設計する場合、機能または操作の観点から要件を検討するのが自然なアプローチです。例えば、航空会社の予約係の仕事をサポートするソフトウェアの要件は次のようになる：

1. Accept the customer's destination and departure date from the clerk.
2. Display the available flights to the clerk.
3. Accept the reservation request for a specific flight from the clerk.
4. Confirm the reservation and print the ticket.

各関数に対応するモジュールと、他の関数を呼び出す「メイン」モジュールがある # 図14.1に示すような設計に自然に変換される。

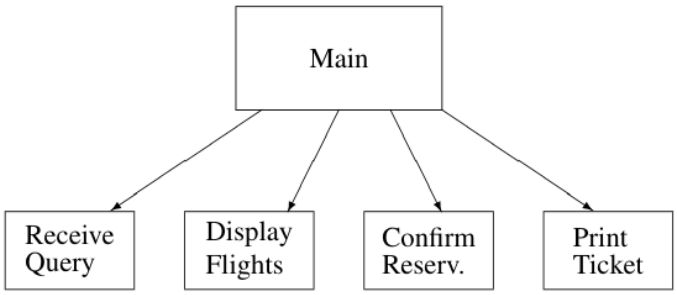

図14.1： 機能分解

残念なことに、この設計はロバストではない。例えば、航空会社が老朽化した表示端末を

交換することで労働条件を改善したとする。同様に、他の航空会社との新しいコードシェアリングの取り決めも、大規模な修正を必要とする。

ロバストでない設計は、納品されるソフトウェアシステムを信頼性のない不安定なものにしてしまう。そうでなければ、すべてのアプリケーションはポケット電卓のプログラムのように効率的に配線されてしまうだろう。

そうでなければ、すべてのアプリケーションはポケット電卓のプログラムのように効率的に配線されてしまうだろう。 # ソフトウェアは、設計基準の焦点を変えることによって、より堅牢で信頼性の高いものにすることができる。正しい問いは、「そのソフトウェアは何に対して動作しているのか？機能重視から、外部デバイス、内部データ構造、そしてオブジェクトと総称される実世界モデルに重点を置くように変えるのだ。モジュールは「オブジェクト」ごとに構築され、オブジェクトの実装に必要なすべてのデータと操作を含むべきである。この例では、図14.2に示すように、いくつかのオブジェクトを特定することができる。

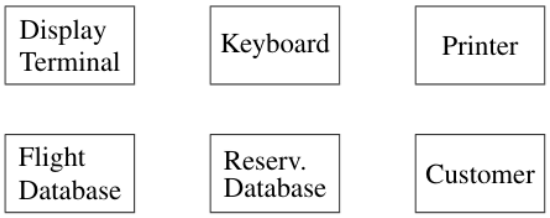

図14.2： オブジェクト指向設計

表示端末やプリンタのような外部デバイスはオブジェクトとして識別され、 フライト情報や予約のデータベースもオブジェクトとして識別される。さらに、「顧客」オブジェクトを設計した。このオブジェクトの目的は、フライトが確 定してチケットが発行されるまで、店員がデータを入力する架空のフォームをモデル化することである。この設計は修正に強い：

- 別の端末を使用するための修正は、Terminalオブジェクトに限定できる。このオブジェクトのプログラミングは、顧客データを実際のディスプ レイやキーボードのコマンドにマッピングするので、新しいハードウェアにマッピングするだけで、 Customerオブジェクトを変更する必要はない。
- コード共有は確かにデータベースの全面的な再編成を必要とするかもしれないが、プログラムの他の部分に関する限り、2文字の航空会社コードは別のものと同じである。

オブジェクト指向設計は、現実世界のオブジェクトのモデリングに限定されるものではない。これは、これまで強調してきたプログラミング言語のコンセプトのひとつである「抽象化」と直結しています。データ構造を実装するモジュールは、抽象的なデータ型のインスタンスであるオブジェクトと、そのデータを操作するための操作として設計し、プログラムすることができる。抽象化は、データ型の表現をオブジェクトの中に隠すことで達成される。

実際のところ、オブジェクト指向設計と 「普通の 」プログラミングの主な違いは、 普通のプログラミングでは、あらかじめ定義された抽象概念に制限されるのに対して、 オブジェクト指向設計では、独自の抽象概念を定義できるということです。

例えば、浮動小数点数（第9章）は、コンピュータにおける難しいデータ操作の有用な抽象化に過ぎない。すべてのプログラミング言語が、私たちが必要とするすべてのオブジェクト（複素数、有理数、ベクトル、行列などなど）に対してあらかじめ定義された抽象化を含んでいれば便利なのだが、便利な抽象化に終わりはない。最終的にプログラミング言語は立ち止まり、その作業をプログラマーに委ねなければならない。

プログラマーはどのようにして新しい抽象化を生み出すことができるだろうか？1つの方法は、コーディング規約やドキュメントを使うことである（「配列の最初の要素は実数部、2番目の要素は虚数部である」）。あるいは、プログラマーが新しい抽象化を明示的に定義できるように、言語がAdaのプライベート型のような構造を提供することもできる。OOPは普通の言語でも可能（であるべき）だが、プログラミングの他の考え方と同様、その概念を直接サポートする言語で行うのが最も効果的である。OOPをサポートする言語で使用される基本的な構成要素は、前章で説明した抽象データ型ですが、オブジェクト指向設計はより一般的で、外部デバイスや現実世界のモデルなどを抽象化することにまで及ぶことを理解することが重要です。

オブジェクト指向設計は非常に難しい。というのも、何かをオブジェクトにする価値があるかどうかを判断するには、多くの経験と判断が必要だからです。オブジェクト指向設計を初めて使う人は、熱中しすぎて何でもオブジェクトにしてしまいがちです。その結果、プログラムが非常に緻密で退屈なものになり、この手法の利点がすべて失われてしまいます。情報隠蔽の概念に基づいた、最も良い経験則を使うべきだ：

> Every object should hide one major design decision.

自問自答することは役に立つ：「と自問することが役立ちます。

予約システムのために選択された特定の表示端末とプリンタは、明らかにアップグレードの対象である。同様に、データベースの構成に関する決定も、システムが成長するにつれてパフォーマン スを向上させるためだけであれば、変更される可能性が高い。一方、顧客データフォームは変更される可能性が低いので、別のオブジェクトにする必要はないとも言える。Customerオブジェクトを作成するという私たちの設計上の決定に同意しないとしても、オブジェクト指向設計が設計上の問題を議論したり、ある設計と別の設計の利点を論じたりするのに適したフレームワークであることには同意するはずです。

以下のセクションでは、2つの言語を使ってOOPの言語サポートについて説明します： C ++とAda 95である。まずC ++について見てみよう。C ++は、モジュールのサポートさえないC言語の上に、OOPのための単一の統合された構造を追加することによって設計された。次に、すでに部分的にOOPをサポートする多くの機能を備えていたAda 83に、いくつかの小さな構成要素を追加することによって、Ada 95で完全なオブジェクト指向プログラミングがどのように定義されたかを見ていく。

14.2 C++におけるオブジェクト指向プログラミング

A programming language is said to support OOP if it includes constructs for:

- カプセル化とデータの抽象化
- Inheritance

- Dynamic polymorphism

前章のカプセル化とデータ抽象化の議論を思い出してみよう。

Adaパッケージのようなモジュールは、計算資源をカプセル化し、インターフェイスの仕様のみを公開する。データの抽象化は、他のユニットからアクセスできないプライベート部分にデータ表現を指定することで実現できる。C ++におけるカプセル化と抽象化の単位は、データ型とサブプログラムの宣言を含むクラスである。インスタンスと呼ばれる実際のオブジェクトは、このクラスから生成される。C ++におけるクラスの例を以下に示す：

```
class 飛行機データ { pub
lic：
    char *get_id(char *s) const{return id;}
    void set_id(char *s)      {strcpy(id, s);}
    int get_speed( ) const    {return speed;}
    void set_speed(int i)     {speed = i;}
    int get_altitude( ) const {return altitude;}
    void set_altitude(int i)  {altitude = i;}
private:
    char id[80];
    int speed;
    int altitude;
};
```

この例では、前章の例を拡張し、各航空機のデータ用に別のクラスを作成している。このクラスは別のクラス、例えば多くの飛行機を格納するためのデータ構造を定義するクラスなどで利用できるようになった：

```
class Airplanes {
public:
void New Airplane(Airplane Data, int &); void Get
 Airplane(int, Airplane Data &) const; private：
    Airplane_Data database[100];
    int current_airplanes;
    int find_empty_entry( );
};
```

各クラスは、一連のデータ宣言をカプセル化するように設計されている。プライベート部分のデータ宣言は、再コンパイルは必要ですが、そのクラスを使用するプログラム（そのクラスのクライアントと呼ばれます）に変更を加えることなく変更することができます。クラスは、クラス内部の更新データ値を取得する一連のインターフェース関数を持つ。

なぜ飛行機データを普通のパブリックレコードとして宣言するのではなく、別のクラスにする必要があるのか疑問に思うかもしれません。内部表現が変更される可能性があると考えるのであれば、データはクラスの中に隠すべきです。

例えば、ある顧客は高度を英語の単位（フィート）で測りたがり、別の顧客はメートル単位で測りたがるとします。飛行機データ用に別のクラスを定義することで、どちらの顧客にも同じソフト ウェアを使うことができ、アクセス関数の実装だけを変更することができます。

この柔軟性の代償として、データ値へのアクセスには必ずサブプログラム # が必要になります。 call:

```
Aircraft_Data a;              // Instance of the class
int alt;

alt = a.get_altitude( );      // Get value hidden in instance
alt = (alt * 2) + 1000;
a.set_altitude(alt);          // Return value to instance
```

データ値へのアクセスには、パブリックレコードであれば単純な代入文ではなく # サブプログラムが必要になります：

```
a.alt = (a.alt * 2) + 1000;
```

アクセス関数が意図した処理を不明瞭にしてしまうため、 # プログラミングは非常に面倒で、結果として得られるコードは読みにくく、 理解しにくいものになりかねません。そのため、クラスは抽象データ型の実装の詳細を隠すことで得られる明確な利点がある場合にのみ定義されるべきである。

しかし、カプセル化のために実行時のオーバーヘッドが大きくなる必要はない。この場合、この関数はインライン関数[1]であり、サブプログラムのコールとリターンのメカニズム（第7章参照）は使用されません。この場合、関数はインライン関数です。インライン化はスペースと時間を交換するものなので、非常に小さなサブプログラム（2、3命令以下）に限定すべきである。サブプログラムをインライン化する前に考慮すべきもう1つの要素は、コンパイル条件が追加されることである。インライン化されたサブプログラムを修正した場合、すべてのクライアントを再コンパイルしなければならない。

14.3 Inheritance

セクション4.6で、Adaではある型を別の型から派生させることで、派生した型が親型に定義された値のコピーと操作のコピーを持つことができることを示した。親型があれば

Ada

```
package Airplane_Package is
   type Airplane_Data is
      record
         ID: String(1..80);
         Speed: Integer range 0..1000;
         Altitude: Integer range 0..100;
      end record;
```

[1] In C++ a function can be inline even if the body is written separately and not in the class declaration.

```
procedure New Airplane(Data: in Airplane Data; I: out Integer); pro
cedure Get Airplane(I: in Integer; Data: out Airplane Data); end Ai
rplane Package ;
```

派生型は別のパッケージで宣言することができる：

Ada

```
type New_Airplane_Data is
   new Airplane_Package.Airplane_Data;
```

派生型を操作する新しいサブプログラムを宣言し、親型のサブプログラムを 新しいサブプログラムで置き換えることができます：

Ada

```
procedure Display_Airplane(Data: in New_Airplane_Data);
         -- Additional subprogram
procedure Get_Airplane(Data: in New_Airplane_Data; I: out Integer);
         -- Replaced subprogram
         -- Subprogram New_Airplane copied from Airplane_Data
```

派生型は型のファミリーを形成し、ファミリー内の任意の型の値を、ファミリー内の別の型の値に変換することができる：

Ada

```
A1: Airplane_Data;
A2: New_Airplane_Data := New_Airplane_Data(A1);
A3: Airplane_Data := Airplane_Data(A2);
```

さらに、プライベート型から派生させることもできます。もちろん、派生した 型のサブプログラムはすべて、親型のpublicサブプログラムで定義されている 必要があります。

Adaの派生型の問題点は、拡張できるのは操作だけで、型を構成するデータコンポーネントは拡張できないということだ。例えば、超音速機のマッハ数[2]を既存のデータに加えて保存するように、航空管制システムを変更しなければならないとする。1つの可能性は、単に既存のレコードを修正して追加フィールドを含めることである。この解決策は、プログラムの初期開発中に修正するのであれば、受け入れられる。しかし、システムがすでにテストされ、インストールされているのであれば、既存のソースコードをすべて再コンパイルし、チェックする必要のない解決策を見つける方がよいだろう。

この解決策は継承を使うことである。継承とは、既存の型を拡張する方法であり、操作の追加や変更だけでなく、型にデータを追加することもできる。C++では、これを別のクラスから派生させることで行う：

C++

```
class SST Data: public Airplane Data { priva
te :
   float mach;
public:
```

[2] 音速に換算した速度。

```
    そのため、既存のコードに影響を与える必要はありません。 # flo
    at get mach( ) const {return mach;}; void set mach(float m)
    {mach = m;};
};
```

派生クラスSST Dataは、既存のクラスAirplane Dataから派生したものである。これは、基底クラス[3]で定義されているすべてのデータ要素とサブプログラムが、派生クラスでも利用可能であることを意味する。さらに、派生クラスSST Dataの値にはそれぞれ追加のデータコンポーネントmachがあり、派生型の値に適用できる新しいサブプログラムが2つあります。

The derived class is an ordinary class in the sense that instances can be declared and subprograms invoked:

C++

```
SST_Data s;

s.set_speed(1400);          // Inherited subprogram
s.set_mach(2.4);            // New subprogram
```

set machと呼ばれるサブプログラムはSST Dataクラス内で宣言されたものであり、 set speedと呼ばれるサブプログラムはベースから継承されたものである。派生クラスは、基底クラスを修正したり再コンパイルしたりすることなくコンパイルおよびリンクできることに注意してください。

14.4 C++における動的ポリモーフィズム

When one class is derived from another class, you can *override* inherited subprograms in the derived class by redefining the subprogram:

```
クラス SST Data: public Airplane Data publ {
ic :
    int get_speed( ) const;   // Override
    void set_speed(int);      // Override
};
```

呼び出しが与えられた場合

```
obj.set_speed(100);
```

Airplane Dataから継承されたサブプログラムとSST Dataの新しいサブプログラムのどちらを呼び出すかは、コンパイル時にオブジェクトobjのクラスに基づいて決定されます。これはスタティック・バインディングまたはアーリーバインディングと呼ばれ、プログラムが実行される前に決定され、各呼び出しは常に同じサブプログラムになります。

[3]Parent type in Ada is roughly the same as base class in C++.

しかし、継承の要点は、類似のプロパティを持つクラス群を作成することであり、これらのクラスのいずれかに属する値を変数に代入することが可能であることを期待することは合理的である。このような変数に対してサブプログラムが呼ばれた場合、何が起こるだろうか。どのサブプログラムを呼び出すかは、実行時に決定されなければならない。なぜなら、変数に格納されている値は、実行時までわからないからである。実行時にサブプログラムを選択する機能を表す用語として、動的多相性、動的バインディング、後期バインディング、実行時ディスパッチなどがある。C ++では、動的バインディングが可能なサブプログラムを示すために仮想関数が使用される。

performed：

```
クラスAirplane Data { pr
ivate：
    ...
public:
    virtual int get_speed( ) const;
    virtual void set_speed(int);
    ...
};
```

親クラスの仮想サブプログラムと同じ名前とパラメータシグネチャを持つ # 派生クラスのサブプログラムも仮想とみなされます。virtual指定子を繰り返す必要はありませんが、分かりやすくするために繰り返すのは良い習慣です：

```
class SST_Data : public Airplane_Data {
private:
    float mach;
public:
    float get_mach( ) const;        // New subprogram
    void set_mach(float m);         // New subprogram
    virtual int get_speed( ) const;// Override virtual subprogram
    virtual void set_speed(int);    // Override virtual subprogram
    ...
};
```

ここで、基底クラスへの参照パラメータを受け取る手続きupdateを考えてみましょう：

```
void update(Airplane_Data & d, int spd, int alt)
{
    d.set_speed(spd);          // What type does d point to ??
    d.set_altitude(alt);       // What type does d point to ??
}

Airplane_Data a;
SST_Data s;
```

```
void proc( )
{
   update(a, 500, 5000);   // Call with Airplane_Data
   update(s, 800, 6000);   // Call with SST_Data
}
```

派生クラスの考え方は、派生値は基本値（おそらくは追加フィールドを持つ）であるというものである。updateの中で、コンパイラはdがAirplane Dataの値なのかSST Dataの値なのかを知る術がありません。そのため、2つの型に対して異なる定義がされているset speedをコンパイルする方法がない。したがって、コンパイラーは、dが指す値のクラスに応じて、実行時に正しいサブプログラムに呼び出しをディスパッチするコードを作成しなければならない。procの最初の呼び出しは、Airplane Data型の値をdに渡しているので、set speedの呼び出しはAirplane Dataクラスで定義されたサブプログラムにディスパッチされ、2番目の呼び出しはSST Dataで定義されたサブプログラムにディスパッチされる。

動的ポリモーフィズムの利点を強調しておこう。プログラムの大部分を、仮想サブプログラムの呼び出しを使って完全に汎用的に書くことができる。派生クラス群の中の特定のクラスに処理を特化するのは、実行時に仮想サブプログラム上でディスパッチすることによってのみ行われる。さらに、派生クラスをファミリーに追加する必要が生じた場合でも、既存のコードを修正したり再コンパイルしたりする必要はありません。例えば、別のクラスを派生させるとしよう：

```
class Space_Plane_Data : public SST_Data {
   virtual void set_speed(int);   // Override virtual subprogram
private:
   int reentry_speed;
};

Space_Plane_Data sp;
update(sp, 2000, 30000);
```

(i)新しいサブプログラムがset speedをオーバーライドし、(ii)updateの正式なパラメータdの値に追加フィールドreentry speedが含まれていても、updateの定義を含むファイルを再コンパイルする必要はない。

動的ポリモーフィズムはいつ使われるのか？

仮想サブプログラムと通常の非仮想サブプログラムを持つ基底クラスを宣言し、 追加フィールドを持つクラスを派生させ、両方のサブプログラムに新しい宣言を与えましょう：

```
class Base_Class {
```

```
private:
    int Base_Field;
public:
    なぜなら、派生オブジェクトはベースオブジェ
    クト（＋余分な情報）であり、余分な情報は代
    入の際に無視できるからです（図14.3）；
};
クラス Derived Class : public Base Class { pr
ivate:
    int Derived_Field;
public:
    virtual void virtual_proc( );
    void ordinary_proc( );
};
```

次に、クラスのインスタンスを変数として宣言してみましょう。派生クラスの値を基底クラスの変数に代入することができる。

```
Base_Class    Base_Object;
Derived_Class Derived_Object;
if (...) Base_Object = Derived_Object;
```

なぜなら、派生オブジェクトはベースオブジェクト(プラスアルファの情報)であり、 # 代入ではそのプラスアルファの情報は無視できるからです(図14.3)。

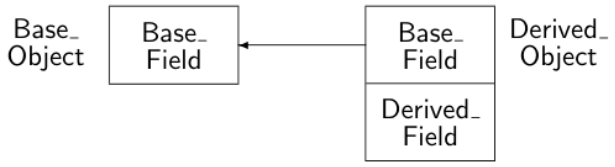

図14.3： 派生オブジェクトの直接代入

さらに、サブプログラムの呼び出しは（仮想かどうかにかかわらず）一義的であり、コンパイラは静的バインディングを使用できます：

```
Base Object.virtual proc( ); Ba
se Object.ordinary proc( ); Der
ived Object.virtual proc( ); De
rived Object.ordinary proc( );
```

しかし、間接的な割り当てが行われ、派生クラスへのポインタがベースクラスへのポインタに代入されたとします：

```
Base_Class*    Base_Ptr = new Base_Class;
Derived_Class* Derived_Ptr = new Derived_Class;
if (...) Base_Ptr = Derived_Ptr;
```

[4]See section 15.4 on assignment of objects from base to derived classes.

この場合、ベース・ポインタは完全な派生オブジェクトを指し、切り捨ては行われないため、セマンティクスは異なります（図14.4）。この場合、ベースポインタは完全な派生オブジェクトを指し示し、切り捨ては行わ # ないため、セマンティクスは異なります（図14.4）。

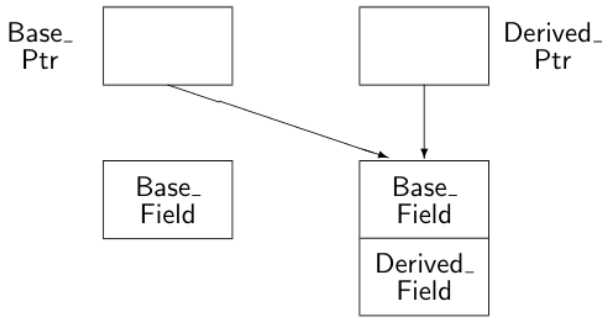

図14.4： 図14.4: 派生オブジェクトの間接的な割り当て

全てのポインタは、指定された型に関係なく、同じように表現される。

注意すべき重要な点は、ポインタの代入の後、コンパイラーは指定されたオブジェクト の型に関する情報を一切持たないということです。したがって、呼び出しをバインドする方法がありません：

```
Base_Ptr− >virtual_proc( );
```

図14.4: 派生オブジェクトの間接的な代入 # 呼び出しを正しいサブプログラムにバインドする手段がないため、動的ディスパッチ を行わなければならない。同じような状況は、上記のように参照パラメータを使用した場合にも発生します。プログラマーは通常、変数と指定オブジェクトを区別しないので、この状況は混乱を招く可能性があります。以下の文の後に

```
int i1 = 1;
int i2 = 2;
int *p1 = &i1;          // p1 points to i1
int *p2 = &i2;          // p2 points to i2
p1 = p2;                // p1 also points to i2
i1 = i2;                // i1 has the same value as i2
```

型が完全に同じである限り、これは確かに正しいのですが、派生クラスから基底クラスへの代入では、切り捨てのために正しくありません。継承を使用する場合、指定されたオブジェクトがポインタ宣言で設計された型と同じ型であるとは限らないことを覚えておく必要があります。

C++における動的ポリモーフィズムのセマンティクスには落とし穴がある。しかし、オーバーライドされた普通のサブプログラムがクラス内に存在することもある：

```
Base_Ptr = Derived_Ptr;
Base_Ptr− >virtual_proc( );// Dispatches on designated type
Base_Ptr− >ordinary_proc( );// Statically bound to base type !!
```

仮想サブプログラムの呼び出しは、指定されたオブジェクトの型（この場合は Derived Class ）に従って実行時にディスパッチされ、通常のサブプログラムの呼び出しは、ポインタの型（この場合は Base Class）に従ってコンパイル時にバインドされます。この違いは危険です。なぜなら、非仮想サブプログラムを仮想サブプログラムに変更したり、逆に変更したりすると、基底クラスから派生したクラス・ファミリー全体にバグが発生する可能性があるからです。

> C++における動的ディスパッチは、ポインタまたは参照を通じて行われる仮想サブプログラムの呼び出しに対して行われる。

Implementation

ここで、暗黙のインデックスは指定されたオブジェクトの最初のフィールドであると仮定します。 # 先ほど、派生クラスでサブプログラムが見つからない場合、サブプログラムの定義が見つかるまで祖先クラスを検索することを指摘しました。静的バインディングの場合、検索はコンパイル時に行うことができます。コンパイラは派生クラスの基底クラスを検索し、その基底クラスを検索するというように、適切なサブプログラム・バインディングが見つかるまで検索します。そして、そのサブプログラムに通常の手続き呼び出しをコンパイルすることができる。

仮想サブプログラムが使用されている場合、実行時まで実際に呼び出すサブプログラムが分からないため、状況はより複雑になります。仮想サブプログラムが、参照やポインタではなく、特定の型のオブジェクトで呼び出される場合でも、スタティック・バインディングを使用できることに注意してください。そうでない場合、どのサブプログラムを呼び出すかは、(1)サブプログラムの名前、(2)オブジェクトのクラスに基づいて決定される。しかし、(1)はコンパイル時に分かっているので、必要なのはクラスに対するcase文をシミュレートすることだけである。

図14.5） # 通常の実装は少し異なり、仮想サブプログラムを持つ各クラスに対して、ディスパッチテーブ ルが管理される。クラスの各値は、ディスパッチテーブルにインデックスを 「ドラッグ 」しなければならない。

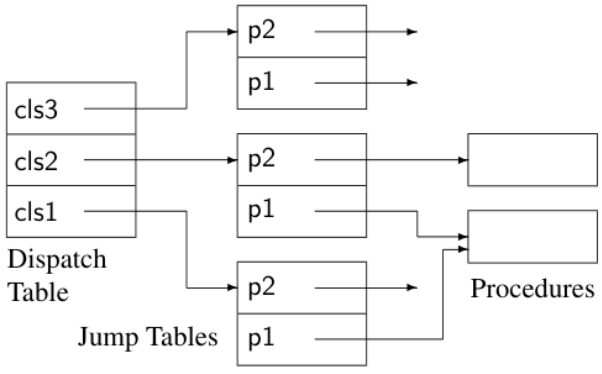

図14.5: 動的ポリモーフィズムの実装

ディスパッチ・テーブルは、そのクラスが定義されている派生ファミリーの # ディスパッチ・テーブルです。ディスパッチ・テーブルのエントリーはジャンプ・テーブルへのポインターで、各ジャンプ・テーブルには各仮想サブプログラムのエントリーがあります。これは、クラスが仮想サブプログラムをオーバーライドしていない場合に起こります。cls2はp2をオーバーライドしているがp1はオーバーライドしていない。

ディスパッチ・サブプログラム呼び出しptr- >p1()に遭遇すると、暗黙のインデックスが指定されたオブジェクトの最初のフィールドであると仮定して、以下のようなコードが実行される：

```
load    R0,ptr          オブジェクトのアドレスを取得
load    R1,(R0)         指定オブジェクトのインデックス
load    R2,&dispatch    を取得 ディスパッチテーブルの
add     R2,R1           アドレスを取得 ジャンプテーブ
load    R3,(R2)         ルのアドレスを取得 手続きのア
load    R4,p1(R3)       ドレスを取得 手続きを呼び出す、
call    (R4)            アドレスはR4にある
```

さらに最適化しなくても、実行時のオーバーヘッドは比較的小さく、さらに重要なことは固定的であるため、ほとんどのアプリケーションでは動的ポリモーフィズムの使用を控える必要はありません。それでもオーバーヘッドは存在するので、ランタイムディスパッチ は慎重に分析した上で使用すべきです。動的ポリモーフィズムが「良いアイデア」だからといって過剰に使用したり、「非効率的」だからといって使用を控えたりするような極端なことは避けてください。

動的多相性は、基本クラスから派生するクラスの固定セット（固定サイズのディスパッチテーブルを使用可能）と、オーバーライド可能な仮想関数の固定セット（各ジャンプテーブルのサイズも固定）に限定されるため、固定オーバーヘッドが得られることに注意してください。動的ポリモーフィズムが無制限の実行時探索なしで実装できることを示したのは、C++の大きな功績である。

14.5 Ada 95におけるオブジェクト指向プログラミング

Ada 83は、パッケージ構造によるカプセル化を完全にサポートし、派生型による継承を部分的にサポートしています。派生型は完全な継承をサポートしていない。なぜなら、新しい型を派生させた場合、新しい操作を追加できるだけで、新しいデータコンポーネントを追加することはできないからである。また、ポリモーフィズムはバリアント・レコードの静的ポリモーフィズムのみである。Ada 95では、型を導出するときにレコードを拡張できるようにすることで、完全継承がサポートされている。親レコード型が継承の対象であることを示すには、タグ付きレコード型として 宣言しなければならない：

```
package Airplane_Package is
   type Airplane_Data is tagged
      record
         ID: String(1..80);
         Speed: Integer range 0..1000;
         高度： 整数範囲 0；
      end record;
End Airplane Package；
```

タグはPascalのタグやAdaのバリアントレコードの判別子と似ており、互いに派生する異なる型を区別するために使用されます。これらの構文とは異なり

タグ付きレコードのタグは暗黙的なものであり、プログラマが明示的にアクセスする必要はない。抽象データ型を作るには、型をprivateとして宣言し、privateの部分に完全な型宣言を記述します：

```
パッケージ Airplane Package is
   procedure Set ID(A: in out Airplane Data; S: in String);
   function Get ID(A: Airplane Data) return String; procedur
   e Set Speed(A: in out Airplane Data; I: in Integer); func
   tion Get Speed(A: Airplane Data) return Integer; procedur
   e Set Altitude(A: in out Airplane Data; I: in Integer); f
   unction Get Altitude(A: Airplane Data) return Integer ;
private
   飛行機データ型はタグ付きレコード
   である。
         ID: String(1..80);
         Speed: Integer range 0..1000;
         Altitude: Integer range 0..100;
      end record;
end Airplane_Package;
```

タグ付き型の宣言を含むパッケージ仕様の中で定義されたサブプログラムは(その 型に対する定義済みの操作とともに)プリミティブ操作と呼ばれ、派生時に継承 されるサブプログラムです。継承はタグ付き型を拡張することで行われる：

```
with Airplane_Package; use Airplane_Package;
package SST_Package is
   type SST_Data is new Airplane_Data with
      record
         Mach: Float;
      end record;
   procedure Set_Speed(A: in out SST_Data; I: in Integer);
   function Get_Speed(A: SST_Data) return Integer;
end SST_Package;
```

この派生型の値は、親型であるAirplane Dataの値のコピーに、追加レコード フィールドMachを加えたものである。この型に対して定義される操作は、プリミティブ操作のコピーである。もちろん、派生型に対して他の無関係なサブプログラムを宣言することもできる。

Adaにはプリミティブ操作を呼び出すための特別な構文はない：

```
A: Airplane_Data;
```

[5] If necessary, the tag can be read by using an attribute on an object of a tagged type.

```
Set_Speed(A, 100);
```

オブジェクトAは構文的には普通のパラメータであり、その型からコンパイラはどのサブプログラムを呼び出すかを推測することができます。このパラメータは、どのサブプログラムが選択されるかを制御するため、制御パラメータと呼ばれる。制御パラメータは最初のパラメータである必要はなく、複数あってもよい（ただし、すべて同じ型であることが条件）。C++では、クラスで宣言されたサブプログラムを呼び出すために特別な構文が使用される：

C++

```
Airplane_Data a;
a.set_speed(100);
```

オブジェクトaはメッセージセットspeedの区別された受信者である。区別された受信者は暗黙のパラメータであり、この場合、オブジェクトaに対してスピードが設定されることを示す。

動的ポリモーフィズム

Ada95の動的ポリモーフィズムについて説明する前に、Adaと他のオブジェクト指向言語との用語の違いに対処しなければならない。C ++では、データ型を表すのにクラスという用語を使い、そのデータ型のインスタンスを作成するのに使う。Ada95では、他の言語ではクラスやインスタンスとして知られているタグ付き型やオブジェクトであっても、型やオブジェクトという用語を使い続けている。クラスという単語は、共通の祖先から派生したすべての型の集合を示すために使用されます。クラスという言葉の新しい用法とC ++での用法を混同しないように注意してください。

タグ付き型Tには、T'Classと表記されるクラス全体型と呼ばれる型が関連付けられている。型Tから派生した型はすべてT'Classでカバーされる。クラスワイド型は制約がなく、制約のない配列のオブジェクトを宣言するのと同じように、制約のない型のオブジェクトを宣言することはできません：

```
type Vector is array(Integer range <>) of Float;
V1: Vector;                          -- Illegal, no constraint
```

型Airplane Dataはタグ付きレコードA1: Air
plane Data'Classである；

```
.. end record;
 - Illegal, no constraint
```

しかし、初期値が与えられていれば、クラス全体の型を持つオブジェクトを宣言することができる：

```
V2: Vector := (1..20=>0.0);          -- OK, constrained

X2: Airplane_Data;                   -- OK, specific type
X3: SST_Data;                        -- OK, specific type
A2: Airplane_Data'Class := X2;       -- OK, constrained
A3: Airplane_Data'Class := X3;       -- OK, constrained
```

配列と同様に、クラスワイド・オブジェクトは一度制約を受けると、その制約 を変更することはできません。クラスワイド型は、制約のない配列と同様に、クラスワイド型のパラメータを取 るサブプログラムのローカル変数の宣言で使用することができます：

```
procedure P(S: String; C: in Airplane_Data'Class) is
   Local_String: String := S;
   Local_Airplane: Airplane_Data'Class := C;
begin
   ...
end P;
```

動的ポリモーフィズムは、実際のパラメータがクラス全体の型であるのに対して、フォーマル・パラメータがクラスに属する特定の型である場合に発生する：

```
with Airplane_Package; use Airplane_Package;
with SST_Package; use SST_Package;
procedure Main is
   procedure Proc(C: in out Airplane_Data'Class; I: in Integer) is
   begin
      Set_Speed(C, I);      -- What type is C ??
   end Proc;

   A: Airplane_Data;
   S: SST_Data;
begin     -- Main
   Proc(A, 500);          -- Call with Airplane_Data
   Proc(S, 1000);         -- Call with SST_Data
end Main;
```

しかし、Set Speedには2つのバージョンがあり、その正式なパラメータ は親型か派生型のどちらかになります。実行時にCの型は呼び出しによって変わるため、呼び出しを曖昧にしないために動的なディスパッチが必要になります。

図14.6は、ディスパッチにおけるフォーマル・パラメーターと実パラメー ターの役割を理解するのに役立つでしょう。図14.6を見れば、ディスパッチにおけるフォーマル・パラメータとアクチュアル・パラメータの役割が理解できるだろう。つまり、サブプログラムが呼び出されたときに初めて、実パラメータの型がAirplane DataなのかSST Dataなのかがわかる。しかし、図の下に示したプロシージャー宣言は、それぞれ特定の型のフォーマル・パラメーターを持っています。矢印で示すように、呼び出しは実際のパラメータの型に従ってディスパッチされなければなりません。

ディスパッチは必要な場合にのみ行われることに注意してください。コンパイ ラが呼び出しを静的に解決できる場合は、そうします。もしコンパイラが静的に解決できるのであれば、そうします。 # 以下の呼び出しは、クラス全体の型ではなく、特定の型を持つ実際のパラメータとの 呼び出しであるため、ディスパッチする必要はありません：

```
Set_Speed(A, 500);
Set_Speed(S, 1000);
```

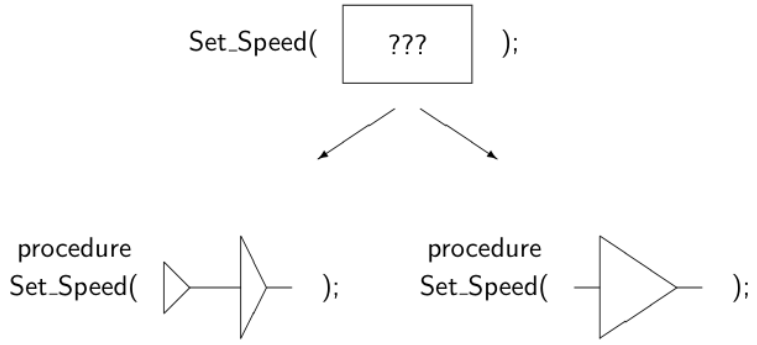

図14.6： Ada 95における動的ディスパッチ

同様に、フォーマルパラメータがクラス全体の型である場合、ディスパッチ は必要ありません。Procへの呼び出しは、1つの明確な手続きへの呼び出しです。正式なパラメータはクラス全体の型であり、クラスがカバーする任意の型の実際のパラメータにマッチします。図14.7を参照すると、仮にSet Speedの宣言が次のようになっていたとする：

```
procedure Set_Speed(A: in out Airplane'Class: I: in Integer);
```

このポインタは、クラス全体の型がカバーする任意のオブジェクトを指定することができます。全ての呼び出しが同じサブプログラムになるので、ディスパッチは必要ありません。

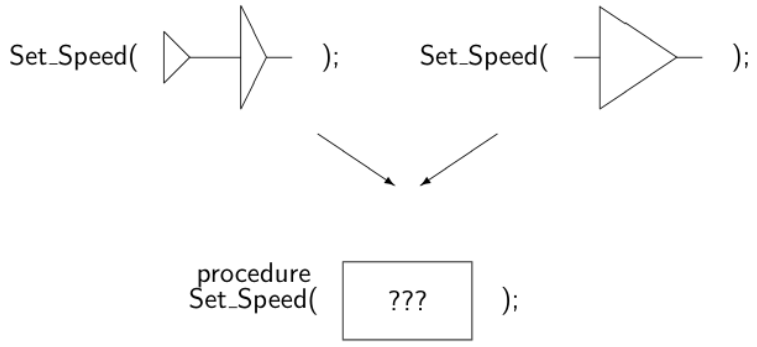

図14.7: クラス全体の正式パラメータ

アクセス型の指定型はクラス全体の型である。この型のポインタは、クラス全体の型がカバーする任意のオブジェクトを指 定することができ、ディスパッチは単にポインタをデリファレンスすることで行える。

```
型 Class Ptr is access Airplane Data'Class; P
tr： クラス Ptr := new 飛行機データ；

if (...) then Ptr := new SST_Data; end if;
```

[6]Ada 95 also defines an anonymous access type that can dispatch without dereferencing.

```
Set_Speed(Ptr.all);          -- Ptr が指す型は?
```

> Ada95における動的多相性は、実際のパラメータがクラス全体の型であり、フォーマルパラメータが特定の型である場合に発生する。

ディスパッチの実行時実装はAda 95とC ++で似ていますが、ディスパッチされる条件は全く異なります：

- C ++では、ディスパッチが行われるためにはサブプログラムが仮想的に宣言 されている必要があります。仮想サブプログラムへの間接呼び出しは全てディスパッチされる [7] 。
- Ada 95では、継承されたサブプログラムはオーバーライドすることができ、 暗黙のうちにディスパッチされるようになる。ディスパッチは、特定の呼び出しによって必要な場合にのみ行われる。

Adaアプローチの主な利点は、動的ポリモーフィズムを使用するかどうかを事前に指定する必要がないことである。これは、仮想サブプログラムと非仮想サブプログラムの呼び出しの間に意味上の違いが存在しないことを意味する。飛行機データがタグ付きで定義されているが、導出は行われていないとする。この場合、すべての呼び出しが静的に解決されるシステム全体を構築することができる。その後、派生型が宣言されれば、既存のコードを修正したり再コンパイルしたりする必要なく、ディスパッチを使用することができる。

14.6 Exercises

1. トップダウン・プログラミングと呼ばれるソフトウェア開発手法では、高レベルの抽象的な操作でプログラムを記述し、プログラミング言語文に到達するまでその操作を徐々に洗練していくことを提唱している。この方法をオブジェクト指向プログラミングと比較してみよう。

2. Aircraft Dataを抽象データ型にするか、クラスのフィールドを公開するか。

3. 既存のコードを再コンパイルすることなく、タグ付き型を持つC ++クラスまたはAda 95パッケージを継承できることを確認する。

4. タグ付きItem型を宣言し、Itemのキュー項を指定し、Boolean、Integer、 Characterのそれぞれに in ついてItemから派生する。

5. C++で異種キューを書きなさい。

6. C ++では、参照パラメータに対してディスパッチが発生しますが、 通常のパラメータに対しては発生しないことを確認してください。

7. Ada 95では、タグ付き型は非公開で拡張できる：

[7] コンパイラは動的なディスパッチを最適化することができます。例えば、実際のパラメータが参照やポインタではなく変数である場合、 Ada 95のように呼び出しを静的に解決することができます。

飛行機パッケージを使用; 飛行機パッケージを使用; S
STパッケージを使用 # 8.

```
    type SST_Data is new Airplane_Data with private;
    procedure Set_Speed(A: in out SST_Data; I: in Integer);
    function Get_Speed(A: SST_Data) return Integer;
private
    ...
end SST_Package;
```

What are the advantages and disadvantages of private extension?

8. Ada 95またはC ++コンパイラが生成する機械命令について、動的多相性を調べ てください。