15 オブジェクト指向プログラミングの詳細

この章では、オブジェクト指向言語に存在する追加的な構成要素について調査する。これらは単に言語に追加された巧妙な機能ではなく、オブジェクト指向プログラミング技法を使いこなせるようになるためには習得しなければならない技術的な構成要素である。詳細は、各言語の教科書を参照されたい。

The chapter is divided into six sections:

1. クラスの構造化

   - 抽象クラスは、1つ以上の継承クラスが実装可能な抽象インターフェイスを 作成するために使用されます。
   - ジェネリクス(Ada)やテンプレート(C++)を継承と組み合わせることで、 クラスを他のクラスとパラメータ化することができます。
   - 多重継承： 多重継承: クラスは2つ以上の親クラスから派生し、それぞれのクラスから データと操作を継承することができる。

2. プライベートコンポーネントへのアクセス： あるいは、派生クラスやクライアントにエクスポートすることは可能でしょうか?

3. クラスデータ クラスデータ: このセクションでは、クラスにおけるデータコンポーネントの作成と 使用について説明します。

4. エッフェル： Eiffel:Eiffel言語は、プログラムを構造化する唯一の方法としてOOPをサポートするように設計された。Eiffelの構成要素を、既存の言語にOOPのサポートが追加されたAda 95やC ++の構成要素と比較することは有益である。

5. (a)クラスを使用することと継承することのトレードオフとは?(b)継承はどのようなことに使用できるのか?(c) オーバーロードとオーバーライドの関係は?

6. We conclude with a summary of methods for dynamic polymorphism.

15.1 クラスの構造化

Abstract classes

あるクラスが基底クラスから派生する場合、基底クラスは必要なデータと操作のほとんどを含んでおり、派生クラスは追加のデータを追加したり、いくつかの操作を追加したり変更したりするだけであるという前提がある。

多くの設計では、基底クラスは派生クラス群全体に共通する操作を定義するフレームワークと考えた方がよいでしょう。例えば、I/Oやグラフィックスのクラス・ファミリーは、getやdisplayのような共通操作を定義することができます。Ada95とC++は、このような抽象クラスをサポートしています。

抽象クラスはSetデータ構造[1]を定義し、派生クラスは2つの異なる方法でSetを実装します。抽象クラスはSetデータ構造[1]を定義し、派生クラスは2つの異なる方法でSetを実装します。Ada 95では、abstractという単語は抽象型とその型に関連する抽象サブプログラムを表します：

Ada

```
package Set_Package is
    type Set is abstract tagged null record;
    function Union(S1, S2: Set) return Set is abstract;
    function Intersection(S1, S2: Set) return Set is abstract;
end Set_Package;
```

抽象型のオブジェクトを宣言することはできませんし、抽象サブプログラムを呼び出すこともできません。抽象型のオブジェクトを宣言することはできませんし、抽象サブプログラムを呼び出すこともできません。型は具象型を派生させるための枠組みとしてのみ機能し、サブプログラムは具象サブプログラムでオーバーライドする必要があります。

まず、ブール値の配列を使って集合を実装する型を派生させる：

Ada

```
with Set_Package;
package Bit_Set_Package is
    type Set is new Set_Package.Set with private;
    function Union(S1, S2: Set) return Set;
    function Intersection(S1, S2: Set) return Set;
private
    type Bit_Array is array(1..100) of Boolean;
    type Set is new Set_Package.Set with
        record
            Data: Bit_Array;
        end record;
end Bit_Set_Package;
```

もちろん、演算を実装するためのパッケージ本体が必要である。

派生型は具体的なデータ構成要素と操作を持つ具体的な型であり、他の型と同様に使用することができる：

Ada

```
with Bit_Set_Package; use Bit_Set_Package;
procedure Main is
    S1, S2, S3: Set;
begin
```

[1] これは集合のための有用なクラスのほんの一部分である。

```
    S1 := Union(S2, S3);
end Main;
```

もちろん、操作を実装するためにパッケージ本体が必要である。 # ここで、プログラムの別の部分で、配列の代わりにリンクリストを使うような、 セットに関する別の実装が必要になったとする。抽象型から追加の具象型を導出し、それを以前の実装の代わりに、あるいは追加して使用することができます：

Ada

```
with Set_Package;
package Linked_Set_Package is
    type Set is new Set_Package.Set with private;
    function Union(S1, S2: Set) return Set;
    function Intersection(S1, S2: Set) return Set;
private
    type Node;
    type Pointer is access Node;
    type Set is new Set_Package.Set with
        record
            Head: Pointer;
        end record;
end Linked_Set_Package;
```

実際、コンテキスト節を置き換えるだけで、既存のユニットで使用されている実装を変更することができます：

Ada

```
with Linked_Set_Package; use Linked_Set_Package;
procedure Main is
    S1, S2, S3: Set;
begin
    S1 := Union(S2, S3);
end Main;
```

C ++では、抽象クラスは純粋な仮想関数を宣言することで作成され、その関数は「初期値」0 で示されます[2] C ++における集合の抽象クラスは以下のようになります：

C++

```
class Set {
public:
    virtual void Union(Set&, Set&) = 0;
    virtual void Intersection(Set&, Set&) = 0;
};
```

抽象クラスのインスタンスを定義することは不可能である：

[2]The syntax was chosen to be similar to the syntax for a null pointer, using “0” rather than a new keyword.

C++

```
class Bit Set : public Set { p
ublic:
.
virtual void Union(Set&, Set&); virtual
void Intersection(Set&, Set&); private:
.
    int data[100];
};

class Linked Set : public Set { pu
blic:
.
    仮想 void Union(Set&, Set&);
    virtual void Intersection(Set&, Set&);
private:
    int data;
    Set *next;
};
```

具体的な派生クラスは、他のクラスと同様に使用することができます：

C++

```
void proc( )
{
    Bit_Set b1, b2, b3;
    Linked_Set l1, l2, l3;

    b1.Union(b2, b3);
    l1.Union(l2, l3);
}
```

抽象クラスのインスタンスを定義することはできません。Ada 95では、2つの集合を受け取り、3番目の集合を返す普通の関数が定義されている。C ++では、セットの1つが区別されたレシーバーとなる。の意図する解釈は

```
    b1.Union(b2, b3);
```

Union操作の区別されたレシーバーであるインスタンスb1は、2つのパラメーターb2とb3に対する操作の結果を受け取り、それを使って内部データの現在値を置き換える。

集合クラスでは、UnionやIntersectionのような名前を使う代わりに、「+」や 「*」のような 定義済みの演算子をオーバーロードすることを好むかもしれません。これはC++でもAda 95でも可能です。

抽象クラスのすべての実装は、クラス全体の型Set'Class．抽象クラス全体の型の値は、正しい具体的な特定の型、つまり正しい実装にディスパッチされる。このように、抽象型と抽象操作によって、プログラマは実装に依存しないソフトウェアを書くことができる。

Generics

セクション10.3では、Adaのジェネリックサブプログラムについて説明しました。 # ジェネリックサブプログラムは、プログラマがサブプログラムのテンプレートを作成し、 様々な型に対してサブプログラムをインスタンス化することを可能にします。例えば、リストを保持するパッケージは、リスト要素の型についてジェネリックになる。さらに、リストをソートできるように、リスト要素を比較する関数をジェネリックにすることもできます：

Ada

```
generic
   type Item is private;
   with function "<"(X, Y: in Item) return Boolean;
package List_Package is
   type List is private;
   procedure Put(I: in Item; L: in out List);
   procedure Get(I: out Item; L: in out List);
private
type List is array(1..100) of Item; e
nd List パッケージ ;
```

このパッケージは任意の要素型に対してインスタンス化することができます:[3]。

Ada

```
package Integer_List is new List_Package(Integer, Integer."<");
```

インスタンス化によって新しい型が生成され、その型のオブジェクトを宣言して使うことができます：

Ada

```
Int_List_1, Int_List_2: Integer_List.List;

Integer List.Put(42, Int List 1);
Integer List.Put(59, Int List 2) ;
```

Adaには、実際のパラメータを離散型や浮動小数点型のような特定のクラスの型に 制限するためにコントラクトモデルで使用される、ジェネリックな形式パラメータ表 記の豊富なセットがあります。Ada95では、これが一般化され、プログラマが定義した型のクラスを指定するためのジェネリック形式パラメータが使えるようになりました：

Ada

```
with Set_Package;
generic
   type Set_Class is new Set_Package.Set;
package Set_IO is
   ...
end Set_IO;
```

[3] A generic subprogram formal parameter declared as follows:

```
with function "<"(X, Y: in Item) return Boolean is <>;
```

は、インスタンス化の際に適切なシグネチャを持つサブプログラムが表示されている場合、実際のパラメータが与えられていなければ、デフォルトでそれを使用できることを指定します。これにより、Integer.「<」などと書く必要がなくなります。

この指定は、Bit SetやLinked Setのようなタグ付き型Setから派生した型であれば # 汎用パッケージをインスタンス化できることを意味します。ユニオンなどのSetの操作はすべてジェネリックパッケージ内で使用できます。なぜなら、コントラクトモデルによって、インスタンス化がSetから派生した型で行われることが分かっているからです。

Templates

C ++では、クラステンプレートを定義することができます：

C++
```
template ¡class Item¿
   class List {
      void put(const Item &);
   };
```

クラステンプレートが定義されると、templateパラメータを与えることで # このクラスのオブジェクトを定義することができます：

C++
```
List¡int¿ Int_List1;
          // Int_List1 is a List instantiated with int
```

Adaのように、C++では、プログラマが特定のクラスのインスタンス化で使用する特定のサブプログラムを提供したり(特殊化と呼ばれる処理)、そのクラスに存在するデフォルトのサブプログラムを使用したりすることができます。

AdaのジェネリックとC++のテンプレートには重要な違いがあります。Adaでは、型を定義するジェネリック・パッケージのインスタンス化によって、特定の型を含む特定のパッケージが得られる。オブジェクトを得るには、もうひと手間かかる。C ++では、インスタンス化によって直接オブジェクトが得られ、特定のクラスは定義されません。別のオブジェクトを定義するには、テンプレートを再度インスタンス化すればよい：

C++
```
リスト<int> Int List2;          // Another object
```

コンパイラとリンカは、同じ型によるすべてのインスタンス化を追跡し、 クラステンプレートの操作のコードがオブジェクトごとに複製されないように する責任がある。

言語間のさらなる違いは、C++ではコントラクト・モデルを使用しないため、 インスタンス化がテンプレート自体のコンパイル・エラーを引き起こす可能性が あるということです（セクション10.3参照）。

Multiple inheritance

派生クラスの議論は、常に1つのベースからの派生という観点から行われてきた。オブジェクト指向の設計では、1つのクラスが2つ以上の既存のクラスの特徴を持つことがあります。これを多重継承と呼びます。図15.1は、AirplaneがWinged VehicleとMotorized Vehicleから多重継承できることを示しています。つのクラスが与えられたとする：

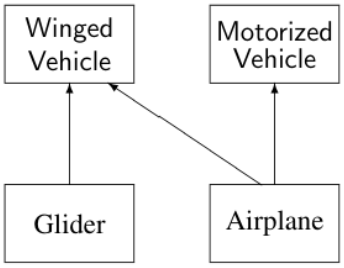

Figure 15.1: Multiple inheritance

C++

```
class Winged_Vehicle {
public:
    void display(int);
protected:
    int Wing_Length;
    int Weight;
};
```

```
クラス Motorized Vehicle { p
ublic：
    void display(int);
protected:
    int Power;
    int Weight;
};
```

クラスは多重継承によって派生させることができる：

C++

```
class Airplane :
    public Winged_Vehicle, public Motorized_Vehicle {
public:
    void display_all( );
};
```

C ++では、クラステンプレートは次のように定義できます # List Int List2; # 複数継承を使うために解決しなければならない問題は、Weightやdisplayのような、 複数の基底クラスから継承されるデータや操作をどうするかということです。C ++では、多重定義されたコンポーネントによって引き起こされる曖昧さは、スコープ解決演算子を使って明示的に解決しなければならない：

C++

```
void Airplane::display all( )
{
    Winged Vehicle::display(Wing Length)；
```

```
    Winged Vehicle::display(Winged Vehicle::Weight); Motor
    ized Vehicle::display(Power); Motorized Vehicle::displ
    ay(Motorized Vehicle::Weight);
};
```

継承の要点は、変更されていないデータやベースの操作に直接アクセスできるようにすることなので、これは残念なことです。多重継承の実装は、セクション14.4で説明した単一継承の実装よりもはるかに困難です。詳しくは注釈付きリファレンスマニュアルのセクション10.1cから10.10cを参照してください。

OOPにおける多重継承の重要性については、多くの議論があります。Eiffelのように多重継承を推奨するプログラミング言語もあれば、Ada 95やSmalltalkのように多重継承の機能を持たない言語もある。これらの言語では、多重継承が解決できる問題は、言語の他の機能を使ってエレガントに解決できると主張している。たとえば、Ada 95のタグ付き型のジェネリック・パラメータは、既存の抽象化を組み合わせて新しい抽象化を作成するために使用できることを前述した。明らかに、多重継承の可用性は、オブジェクト指向の原則に従ったシステムの設計とプログラミングに深い影響を与える。したがって、言語に依存しないオブジェクト指向設計を語ることは困難である。

15.2 Javaにおけるカプセル化

第13.1節で、C言語にはカプセル化構文がないことを説明し、第13.5節で、C ++のスコープ解決演算子と名前空間構文が、グローバル名の可視性に対するC言語の粗いアプローチを改善していることを述べた。互換性を保つために、C ++はカプセル化構文を持たない。Adaには、モジュールへのカプセル化をサポートするパッケージ構文があり（13.3節参照）、パッケージの仕様と実装(本体)を別々にコンパイルすることができる。

Javaにはパッケージと呼ばれるカプセル化構成があるが、紛らわしい ことに、この構成はAdaのパッケージよりもC++の名前空間に近い！パッケージはクラスの集合体である：

Java

```
飛行機パッケージ；

public class Airplane_Data
{
    int speed;                              // Accessible in package
    private int mach speed;                 // Accessible in class

    public void set speed(int s) {...}      // Globally accessible
    ; public int get speed( ) {...};
```

```
}

public class Airplane Database
{
   public void new_airplane(Airplane_Data a, int i)
   {
   if (a.speed>1000) {                    // OK !
      a.speed = a.mach_speed;             // Error !
   }

   private Airplane_Data[] database = new Airplane_Data[1000];
}
```

パッケージは複数のファイルに分割できますが、1つのファイルには1つ のパッケージのクラスしか含めることができません。

publicとprivateという指定子は、C++の指定子と似ています: publicは、そのメンバがクラス外のどこからでもアクセスできることを意味し、privateは、クラスの他のメンバへのアクセスを制限することを意味します。もし指定子が与えられなければ、そのメンバはパッケージの中で見ることができる。この例では、クラスAirplane Dataのメンバint speedに指定子がないので、クラスAirplane Database内のステートメントでアクセスできます。メンバmach speedはprivateと宣言されているので、それが宣言されているAirplane Dataクラス内でのみアクセス可能です。

同様に、クラスにはアクセシビリティ指定子があります。この例では、両方のクラスがpublicと宣言されていますが、これは他のパッケージがこれらのクラスの任意の（public）メンバにアクセスできることを意味します。クラスがprivateと宣言されている場合は、そのパッケージ内でのみアクセス可能です。例えば、プライベートクラスAirplane Fileを宣言し、パッケージ内でデータベースを格納するために使用したいとします。

パッケージはJavaのソフトウェア開発において重要です。 # パッケージを使うことで、外部インターフェイスを明示的に制御しつつ、 関連するクラスをグループ化できるからです。階層的なライブラリ構造は、ソフトウェア開発ツールの構築を単純化します。

他の言語との比較

Javaのパッケージは、C++の名前空間と同様の方法で、グローバルな命名とアクセシビリティを制御する役割を果たします。この例で宣言された内容を考えると、Javaのプログラムには # 以下のようなものが含まれることになる：

Java

```
Airplane Package.Airplane Data a; a.
set speed(100);
```

クラス名とメソッド名はpublicだからだ。パッケージのソースコード全体を調べなければ、どのクラスがインポートされているかはわかりません。パッケージの名前空間を開き、直接見ることができるようにするインポート文があります。この構文は、C++やAdaでの使用と似ている。

パッケージを明示的に宣言する必要はありません。もし宣言しなければ、ファイル内のすべてのクラスは共通の匿名 to パッケージに属しているとみなされます。

JavaとAdaの主な違いは、Adaではパッケージの仕様とパッケージ本体が分離されている ことです。これは単にコンパイルのサイズを小さくするための便利な機能ではなく、大規模なソフトウェア・システムを管理、開発、保守する上で重要な要素である。パッケージ仕様は凍結することができ、その実装と他のユニットの開発を並行して進めることができる。Javaでは、パッケージの 「インターフェイス 」は、単にすべてのパブリック宣言を集めたものである。Javaで大規模なシステムを開発するには、パッケージの仕様を抽出し、 仕様と実装の一貫性を保証するためのソフトウェア・ツールが必要になる [5] 。

パッケージ構造は、JavaにC++に対する1つの大きな利点を与えている。パッケージ自体が階層的な命名規則を使用しているため、コンパイラとインタプリタが自動的にクラスを見つけることができる。例えば、標準ライブラリにはjava.lang.String.toUpperCaseという関数がある。toUpperCaseはjava.lang.Stringパッケージの関数です。toUpperCaseは、java.lang.Stringパッケージの関数です。Javaライブラリは、階層ディレクトリとして実装することができます（する必要はありませんが）。階層名は、言語の外でのみ意味を持つことに注意してください。サブパッケージは、その親に対して特別なアクセス権を持ちません。これは、Adaの子パッケージが、プライベート宣言のエクスポートを禁止するルールに従って、親のプライベート宣言にアクセスできるのとは対照的です。

プライベートコンポーネントへのアクセス

Friends in C++

C ++のクラス宣言の中に、フレンド・サブプログラムやフレンド・クラスの宣言を含めることができます：

```
class Airplane_Data {
private:
    int speed;

    フレンドクラスCL；

};
```

サブプログラムprocとクラスCLのサブプログラムは、Airplane Dataのプライ ベートコンポーネントにアクセスできる：

```
void proc(const 飛行機データ & a, int & i)
{
    i = a.speed;                // OK, we're friends
}
```

サブプログラムprocは、このように参照パラメータやポインタを使用して、クラスの内部コンポーネントを渡すことができます。こうして 「友達 」は、抽象化の秘密をすべて公開することになる。

[5] In Eiffel, this is called the *short* form of a class (Section 15.5).

プライベートな要素へのアクセスを許可する動機は、ケイパビリティと呼ばれる特定の権限を明示的に付与するように設計されたオペレーティング・システムから得ています。OOPの目的のひとつは、閉じた再利用可能なコンポーネントを作ることだからだ。設計の観点からは、フレンドは問題がある。というのも、誰がそのコンポーネントを使用しているかという知識をコンポーネントに求めることになるからだ。フレンド構造のもう1つの深刻な問題は、抽象化を再考する代わりに、プログラムの問題を 「パッチ 」するためにフレンドが使われすぎる可能性があることだ。フレンドを使いすぎると、注意深く設計された抽象化が明らかに壊れてしまう。

フレンドの有効な使い方は、抽象化が2つの異なる要素で構成されている場合である。この場合、互いにフレンドである2つのクラスを宣言することができる。例えば、Keyboardクラスが文字をエコーするためにDisplayクラスへの直接アクセスを必要とする場合、逆にDisplayクラスがタッチスクリーンインターフェースから取得した文字をKeyboardクラスの内部バッファに入れる必要があるとする：

```
class Display {
private:
    void echo(char c);
    friend class Keyboard;   // Let Keyboard call echo
};

class Keyboard {
private:
    void put_key(char c);
    friend class Display;      // Let Display call put_key
};
```

つまり、サブプログラムを不必要にpublicにするか、2つのクラスが1つの大きなクラスに統合される。

friendのもう1つの使用法は、C ++クラスのサブプログラムが、obj1.proc(obj2)という呼び出しでobj1のような区別されたレシーバを持つという事実に関連する構文上の問題を解決することである。このため、パラメータが対称であるはずのサブプログラムに非対称性が生じる。標準的な例は、算術演算子のオーバーロードである。複素数の 「+」をオーバーロードし、浮動小数点パラメーターを暗黙のうちに複素数値に変換できるようにしたいとしよう：

```
complex operator+(float); comp
lex operator+(complex) ;
```

ここで、xまたはyは浮動小数点数であり、もう一方は複素数である。x+yはx.operator+(y)と等価であり、複素数型の区別されたレシーバーでディスパッチされるからである。

しかし、xがfloat型である場合の2番目のx+yの宣言は、float型でディスパッチしようとするが、演算子はcomplexクラスで宣言されている。

解決策は、これらの演算子をクラスの演算としてではなく、クラスのフレンドとして宣言することである：

```
friend complex operator+(complex, complex); f
riend complex operator+(complex, float); frie
nd complex operator+(float, complex);
```

演算子 「+=」をメンバ関数として定義し(ARMのp.249を参照)、「+」をクラス外の普通の関数として定義することができる：

```
complex operator+(float left, complex right)
{
    complex result = complex(left);
    result += right;          // result is distinguished receiver
    return result;
}
```

Access specifiers in C++

あるクラスが別のクラスから派生した場合、派生クラスが基底クラスのコンポーネントにアクセスできるかどうかを確認する必要があります。以下の例では、データベースはプライベートであると宣言されているので、派生クラスからはアクセスできない：

```
クラス Airplanes {
 private：
    Airplane_Data database[100];
};
class Jets : public Airplanes {
    void process_jet(int);
};
void Jets::process jet(int i)
{
    Airplane_Data d = database[i];   // Error, not accessible !
};
```

Jets クラスのインスタンスが宣言された場合、そのインスタンスにはデータベー # ス用のメモリが格納されますが、そのコンポーネントは派生クラスのサブプ # ログラムからはアクセスできません。

C ++ には3つのアクセス指定子がある：

[6] I would like to thank Kevlin A.P. Henney for showing me how to do this.

- パブリックなコンポーネントは、クラスのどのユーザーからもアクセスできる。
- 保護されたコンポーネントは、そのクラス内およびそのクラスから派生した # クラス内でアクセス可能である。
- A private component is accessible only within the class.

この例では、データベースがprivateではなくprotectedである場合、派生クラス Jetsからアクセスすることができます：

```
class Airplanes  {
protected:
    Airplane_Data database[100];
};
class Jets : public Airplanes {
    void process_jet(int);
};
void Jets::process_jet(int i)
{
    Airplane_Data d = database[i];   // OK, in derived class
};
```

この場合、publicとprotectedの両方のコンポーネントがprivateになります。継承されたコンポーネントを操作するには、publicまたはprotectedのサブプログラムを使用するのが、派生クラスにとっても良い方法でしょう。そうすれば、内部表現が変更されても、修正する必要があるのは少数のサブプログラムだけだ。

C ++は、派生時にクラスコンポーネントのアクセシビリティを変更できるようにします。通常、（すべての例で行っているように）public派生を使用します。しかし、private派生も可能で、その場合はpublicとprotectedの両方のコンポーネントがprivateになります：

```
class Airplanes {
protected:
    Airplane_Data database[100];
};
class Jets : private Airplanes {        // private derivation
    void process_jet(int);
};
void Jets::process_jet(int i)
{
    Airplane_Data d = database[i];   // Error, not accessible
};
```

Ada における子パッケージ

Adaでは、パッケージ本体だけがプライベート宣言にアクセスできる。このため、C++ではprotected宣言の共有が可能ですが、パッケージ間で private宣言を直接共有することはできません。

Ada 95では、子パッケージと呼ばれる追加的な構造化機能によって、プライベート宣 言を共有することができます。子パッケージは、既存のパッケージを変更したり再コンパイルしたりすることなく拡張したい場合に非常に便利ですが、ここではこの目的での子パッケージの使用に限定して説明します。

Given the private type Airplane_Data defined in a package:

パッケージ飛行機 パッケージは

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

この型は子パッケージで拡張することができます：

```
package Airplane_Package.SST_Package is
   type SST_Data is tagged private;
   procedure Set_Speed(A: in out SST_Data; I: in Integer);
private
   type SST_Data is new Airplane_Data with
      record
         Mach: Float;
      end record;
end Airplane_Package.SST_Package;
```

パッケージP1とその子P1.P2があった場合、P2は親P1のスコープに属します。子パッケージのprivate部分と本体内では、親のprivate宣言が見える：

```
package body Airplane_Package.SST_Package is
   procedure Set_Speed(A: in out SST_Data; I: in Integer) is
   begin
      A.Speed := I;        -- OK, private field in parent
   end Set_Speed;
end Airplane_Package.SST_Package;
```

もちろん、子パッケージの public 部分が親の private 部分にアクセスすることはできません。

[7] A child package can be declared private in which case its visible part *can* access the private part, but then the child package cannot be used outside of the parent and its descendants.

15.4 Class data

コンストラクタとデストラクタ

同様に、デストラクタはオブジェクトが破棄されるときに呼び出されます。実際、どの言語で定義されたオブジェクト（変数）でも、メモリの確保と解放のためだけであれば、変数の生成時と破棄時に処理を行う必要がある。オブジェクト指向言語では、プログラマーはそのような処理を指定することができる。

C++では、どのクラスに対してもコンストラクタとデストラクタを定義することができます。コンストラクタの構文は、クラス名を持つサブプログラムであり、デストラクタの構文は、名前の前に記号「~」を付けたものである：

C++

```
クラス Airplanes {
 private：
    Airplane_Data database[100];
    int current_airplanes;
public:
    Airplanes(int i = 0) : current airplanes(i) {
    }; ~Airplanes( ) ;
};
```

Airplanesデータベースの作成時に、飛行機のカウントはパラメータiの値を受け取る：

```
Airplanes a1(15);          // current_airplanes = 15
Airplanes a2;              // current_airplanes = 0
```

データベースが破棄されると、デストラクタ（図示せず）のコードが実行される。

It is possible to define several constructors which are overloaded on the parameter signatures:

C++

```
クラス Airplanes {
 public：
    Airplanes(int i = 0) : current airplanes(i) { };
     Airplanes(int i, int j) : current airplanes(i+j
    ) { }; ~Airplanes( ) ;
};

Airplanes a3(5,6);         // current_airplanes = 11
```

C ++にもコピーコンストラクタがあり、オブジェクトが既存のオブジェクトの値で初期化されるとき、より一般的には、あるオブジェクトが別のオブジェクトに代入されるときに、プログラマが定義した処理を行うことができる。C ++におけるコンストラクタとデストラクタの完全な定義は非常に複雑である。詳細は『注釈付きリファレンス・マニュアル』の第12章を参照のこと。

Ada 95では、明示的なコンストラクタとデストラクタは通常宣言されません。変数の単純な初期化には、レコード・フィールドのデフォルト値を使用すれば十分です：

Ada

```
タイプAirplanesはタグ付きレ
コードである。
   現在の飛行機 Integer := 0; end record
   ;
```

or discriminants (Section 10.5):

```
型Airplanes(Initial: Integer)はタグ付きレコ
ードである。
   現在のAirplanes： Integer := Initial; end
   record；
```

明示的なプログラマ定義処理は、Controlled型と呼ばれる抽象型から型を導出することで可能です。この型は、初期化、最終化、調整（代入）のための抽象サブプログラムを提供し、必要な特定の処理でオーバーライドすることができます。詳細については、Ada 95言語リファレンスマニュアルの7.6節で説明されている Ada.Finalizationパッケージを参照してください。

Class-wide objects

クラスのインスタンスごとにメモリが割り当てられる：

C++

```
class C {
   char s[100];
};

C c1, c2;                    c1, c2それぞれに対して # ／／ 100文字
```

時として、クラスのすべてのインスタンスに共通する変数を持つことは有用である。例えば、各インスタンスに通し番号を割り当てる場合、最後に割り当てられた番号を記録する変数lastを保持します。Adaでは、パッケージ本体に通常の変数宣言を含めることでこれを行う：

Ada

```
package body P is
   Last: Integer := 0;
end P;
```

while in C++, a special syntax is needed:

C++

```
class C {
    static int last;          // Declaration
    char s[100];
};

int C::last = 0;              ファイル外からアクセス可能な定義
```

この場合のstatic指定子は、クラス全体のオブジェクトが1つ割り当てられることを意味します。この場合、指定子staticは、クラス全体のオブジェクトが1つ割り当てられることを意味します。クラス定義の外側で明示的にstaticコンポーネントを定義する必要があります。静的クラスコンポーネントは、ファイルスコープでの静的宣言とは異なり、外部リンケージを持ち、他のファイルからアクセスできることに注意してください。

アップコンバージョンとダウンコンバージョン

セクション14.4では、派生クラスの値をC++で暗黙的に基底クラスの値に変換する方法について説明しました。これはアップコンバージョンと呼ばれ、子孫から祖先のいずれかに上方変換されるからである。また、派生型は（余分なフィールドを持つため）「広い」のに対して、基底型は「狭い」ため、派生ファミリーのすべての型に共通するフィールドのみを持つことから、狭義（narrowing）とも呼ばれます。アップコンバージョンが発生するのは、派生型の値がベース型の変数に直接代入されるときだけであり、ある変数から別の変数にポインタが代入されるときには発生しないことを思い出してください。

基底型の値から派生型の値へのダウンコンバージョンは許されない。しかし、基底型へのポインタの場合を考えてみよう：

C++

```
Base_Class*       Base_Ptr = new Base_Class;
Derived_Class*    Derived_Ptr = new Derived_Class;

if (...) Base_Ptr = Derived_Ptr;
Derived_Ptr = Base_Ptr;    // What type does Base_Ptr point to ?
```

この場合、割り当てを拒否する理由はありません。この場合、代入を拒否する理由はありません。一方、指定されたオブジェクトが実際には基底型である場合、ダウンコンバージョンを試みていることになり、代入は拒否されるべきです。このケースに対処するため、C++は指定オブジェクトの型を条件とする動的キャストを定義している：

C++

```
Derived_Ptr = dynamic_cast<Derived_Class*>Base_Ptr;
```

指定されたオブジェクトが実際に派生型である場合、変換は成功します。そうでない場合、ヌルポインタ0が代入され、プログラマはそれをテストすることができます。

すでにAda83では、互いに派生した2つの型の間で明示的な変換が許されていた。派生型は全く同じ構成要素を持っているので、これは何の問題も生じなかった。

異なる表現を持つことは可能ですが（セクション5.9を参照）、どちらの表現も構成要素の数と型が同じであるため、型変換は完全にうまく定義されています。派生型から基底型へのアップコンバージョンの場合、タグ付き型への派生型変換の拡張は即座に行われる。不要なフィールドは切り捨てられる：

Ada

```
S: SST_Data;
A: Airplane_Data := Airplane_Data(S);
```

一方では、拡張時に追加されたフィールドに値を供給するために拡張集約が使用される：

Ada

```
S := (A with Mach => 1.7);
```

フィールドSpeedなどは値Aの対応するフィールドから取られ、余分なフィールドMachは明示的に与えられる。

When attempting to down-convert a class-wide type to a specific type, a run-time check is made and an exception will be raised if the class-wide object is not of the derived type:

Ada

```
procedure P(C: Airplane_Data'Class) is
   S: SST_Data;
begin
   S := SST_Data(C);      -- What type is C ??
exception
   when Constraint_Error => ...
end P;
```

15.5 Eiffelプログラミング言語

エッフェルプログラミング言語の中心的な特徴は以下のとおり：

- エッフェルは、既存の言語にOOPのサポートを移植するのではなく、オブジェクト指向言語として一から構築された。
- Eiffelでは、プログラムを構成する唯一の方法は、互いにクライアントであるか、 あるいは互いに継承するクラスのシステムである。
- 継承が主な構造化構成要素であるため、（継承によって関連づけられる）クラスの標準ライブラリが言語の中心となる。
- 言語 ”の一部ではありませんが、洗練されたプログラミング環境がEiffelチームによって開発されました。この環境には、クラスの表示と修正、インクリメンタルコンパイル、テストとデバッグのための、言語に依存したサポートが含まれている。

Eiffelは、Ada 95やC ++のような動的ポリモーフィズムに加え、静的な型チェックを主張している点で、Smalltalk(同様の特徴を持つ)とは一線を画している。Eiffelは、セクション11.5で説明したように、アサーションを言語に統合することで、信頼性の高いプログラミングをサポートする試みをさらに進めている。

Eiffelでは唯一のプログラム単位はクラスであり、CやC++のようなファイルはなく、Adaのようなパッケージもない[8] Eiffelの用語は他の言語とは異なる。サブプログラム（手続きや関数）はルーチンと呼ばれ、オブジェクト（変数や定数）は属性と呼ばれ、クラスを構成するルーチンと属性はクラスの機能と呼ばれる。基本的に、関数と定数は区別されません。Adaの列挙リテラルのように、定数は単にパラメーターを持たない関数とみなされます。EiffelはC ++のように静的に型付けされており、代入文やパラメーターの受け渡しはコンパイル時にチェックできる適合型を持っていなければなりません。しかし、この言語にはAdaのようなサブタイプや数値型などの豊富な型チェック機能はありません。

クラスが宣言されると、機能のリストが与えられる：

```
class Airplanes
feature                          -- “public”
   New Airplane(Airplane Data): Integer is
      do
         ...
      end; -- New_Airplane
   Get Airplane(Integer): 飛行機データは
      do
         ...
      end; -- Get_Airplane
feature { }                      -- “private”
   database: ARRAY[Airplane_Data];
   current_airplanes: Integer;
   find_empty_entry: Integer is
      do
         ...
      end; -- find_empty_entry
end; -- class Airplanes
```

C ++と同様に、機能セットはグループ化することができ、各グループでアクセシビリティを異 なるように指定することができます。指定子が空集合「{ }」である特徴グループは、private指定子のように、他のクラスにはエクスポートされません。しかし、C++のpublic指定子やAdaのパッケージ指定のpublic部分とは異なり、読み取りアクセスのみがエクスポートされます。さらに、feature-specifierにクラスのリストを明示的に記述することもできます。これらのクラスは、C++のfriendsのように、グループ内の機能にアクセスすることが許可されます。

[8] 言語をこの単一の構文に制限することの欠点は、クラスの集合からプログラムを作成する方法を指定するための追加のツールを環境に含まなければならないことです。

Eiffelでは、定義済みの型とプログラマーが定義した型との間に実際の区別はありません。データベースはEiffelライブラリで定義済みのARRAYクラスのオブジェクトです。もちろん、「array 」は非常に一般的な概念であり、配列の構成要素型を示すにはどうすればよいのでしょうか。その答えは、プログラマーが任意のデータ型をパラメータ化するのと同じ方法、つまりジェネリックスを使用することです。定義済みのARRAYクラスには、構成要素の型を指定するためのジェネリック・パラメーターが1つあります：

```
class ARRAY[G]
```

ARRAY型のオブジェクトが宣言された場合、[9]実際のジェネリック・パラメータを 与えなければならない。事前に定義された複合型を宣言するための特別な構文を持つAdaやC++とは異なり、Eiffelではすべてが単一の構文ルールと意味ルールを使用してジェネリッククラスで構成されます。

ジェネリックはEiffelで広く使用されています。ライブラリには多くのジェネリッククラスの定義が含まれており、特定の要件に合わせて特殊化することができるからです。ジェネリックは、Adaと同様にジェネリッククラスとインスタンス化の間の契約を達成するために制約を付けることもできます（セクション10.3を参照）。パターン・マッチの代わりに、実際のジェネリック・パラメータが派生しなければならないクラスの名前を与えることで制約が表現されます。例えば、以下のジェネリック・クラスはREALから派生した型によってのみインスタンス化できます：

```
class Trigonometry[R -> REAL]
```

Eiffelクラスでは、機能の仕様と実行可能なサブプログラムとしての実装が分離されていないことにお気づきかもしれません。パッケージを個別にコンパイルされた仕様と本体に分けるAdaとは異なり、すべてが同じクラス宣言に含まれていなければならない。このように、Eiffel言語は、そのシンプルさの代償として、プログラミング環境に多くの作業を要求する。特に、この言語では、実質的にインターフェイスであるショートフォームが定義され、環境は要求に応じてショートフォームを表示する責任を負う。

Inheritance

全てのクラスは型を定義し、システム内の全てのクラスは1つの階層に配置される。階層の最上位にはANYというクラスがある。代入と等号はANY内で定義されるが、クラス内でオーバーライドすることもできる。継承の構文は、C ++と似ています：継承されたクラスは、クラス名の後にリストされます。Airplane Dataというクラスがあるとする：

```
class Airplane_Data
feature
    Set_Speed(I: Integer) is ...
    Get_Speed: Integer is ...
feature { }
    ID: STRING;
```

[9]Note that the declaration does not create the array; this must be done in a separate step which will also specify the size of the array.

```
    Speed: Integer;
    Altitude: Integer;
end; -- class Airplane_Data
```

以下のように継承できる：

```
class SST_Data inherit
    Airplane_Data
        redefine
            Set_Speed, Get_Speed
        end
feature
    Set_Speed(I: Integer) is ...
    Get_Speed: Integer is ...
feature { }
    Mach: Real;
end; -- class SST_Data
```

基底クラスのすべての機能は、エクスポート属性を変更せずに継承されます。しかし、派生クラスのプログラマーは、継承された機能の一部または全部を自由に再定義することができます。再定義する機能は、inherit-clauseに続くredefine-clauseで明示的に列挙する必要があります。再定義の他に、単純に名前を変更することもできます。継承された機能は、それが基底クラスでプライベートであったとしても、クラスから再エクスポートすることができることに注意してください（以前に隠された実装に侵入することを許さないC ++やAda 95とは異なります）。

Eiffel環境では、クラスのフラットバージョンを表示することができます。このフラットバージョンは、現在有効なすべての機能を表示します。これにより、クラスのインターフェイスが明確に表示され、プログラマーは階層を 「掘り下げる 」必要がなくなり、何が再宣言され、何が再宣言されなかったのかを正確に確認することができます。

Eiffel, like C++ but unlike Ada 95, uses the distinguished receiver approach so there is no need for an explicit parameter for the object whose subprogram is being called:

```
A: Airplane_Data;
A.Set_Speed(250);
```

Allocation

Eiffelには明示的なポインタはない。すべてのオブジェクトは暗黙的に動的に割り当てられ、ポインタによってアクセスされる。ただし、プログラマーはオブジェクトを拡張宣言することができ、その場合、オブジェクトはポインタなしで割り当てられ、アクセスされる：

```
database: expanded ARRAY[Airplane_Data];
```

(C++やAda 95では、以前に隠されていた実装に侵入することはできない。) # さらに、クラスは拡張されたものとして宣言することができ、そのクラスのすべての オブジェクトは直接割り当てられる。言うまでもなく、定義済みのIntegerやCharacterなどは拡張型である。代入演算子や等号演算子が与えられた場合、 # Eiffel環境はそのオブジェクトを表示することができる：

```
X := Y;
```

ただし、コンパイラは可能な限り静的バインディングを最適化する。 # X、Y、どちらでもない、または両方が展開されるかによって、4つの可能性がある。AdaとC++では、ポインタの代入が意図されているときと、指定されたオブジェクトの代入が意図されているときを区別するのはプログラマーの責任です。Eiffelでは、代入はプログラマーにとって透過的であり、それぞれの可能性の意味は言語で注意深く定義されている。

間接的な代入の利点は、基底クラスの型を持つ通常のオブジェクトが、その基底クラスから派生した任意の型の値を持つことができるということです：

```
A: 飛行機のデータ; S
: SSTのデータ ;
```

```
A := S;
```

もし割り当てが静的であれば、オブジェクトAにはSの余分なフィールドMachを格納する「余地」がない。これをAda 95やC ++と比較すると、クラス全体の型と、特定の型を保持する代入のためのポインタという追加の概念が必要になります。さらに、Eiffelでは代入文のコピーをシャローコピーとディープコピーに区別しています。シャローコピーはポインター（拡張オブジェクトの場合はデータ）をコピーするだけですが、ディープコピーはデータ構造全体をコピーします。継承された代入の定義をオーバーライドすることで、どのクラスでもどちらの意味も選択できます。

動的ポリモーフィズムはすぐに続く：

```
A.Set_Speed(250);
```

コンパイラは、現在Aに保持されている値の特定の型が、Aの基本型Airplane Dataなのか、Airplane Dataから派生した型なのかを知る方法がない。サブプログラムSet Speedは、少なくとも1つの派生クラスで再定義されているため、ランタイム・ディスパッチを行う必要があります。特別な構文やセマンティクスは必要ないことに注意してください。すべての呼び出しは潜在的に動的ですが、コンパイラーは可能な限り最適化して静的バインディングを使用します。

Abstract classes

Eiffelの抽象クラスは、C++やAda 95の抽象クラスに似ている。クラスまたはクラス内の機能は、遅延として宣言することができる。延期されたクラスは、延期された機能をすべて有効にすることによって、つまり実装を提供することによって、具象化されなければならない。C ++やAda 95とは異なり、型が遅延されたオブジェクトを宣言することができることに注意してください：

```
deferred class Set ...          -- Abstract class

class Bit_Set inherit Set ...-- Concrete class
```

```
S: Set;                 -- Abstract object !
B: Bit_Set;             -- Concrete object

!!B;                    -- Create an instance of B
S := B;                 -- OK, S gets a concrete object
S.Union(...);           -- which can now be used
```

多重継承

Eiffelは多重継承をサポートしています：

```
クラス Winged Vehicle 機
能
    Weight: Integer;
    display is ... end;
end;

class Motorized_Vehicle
feature
    Weight: Integer;
    display is ... end;
end;

class Airplane inherit
    Winged_Vehicle, Motorized_Vehicle
    ...
end;
```

複数の継承が許可されている場合、1つの名前が2つ以上の祖先から継承され ている場合に曖昧さを解決する方法を指定しなければならない。Eiffelのルールは基本的に非常にシンプルである（継承階層のすべての可能性を考慮しなければならないため、正式な定義は難しいが）：

ある機能が祖先クラスから複数のパスで継承されている場合、その機能は共有されます。

rename句とredefine句は、必要に応じて名前を変更するために使用できる。この例では、Airplaneクラスは Weightフィールドを1つだけ継承している。明らかに、このクラスは2つのWeightフィールド、1つは機体用、もう1つはモーター用を持つように意図されています。これは、継承された2つのオブジェクトの名前を変更することで実現できます：

```
class Airplane inherit
    Winged_Vehicle
        rename Weight as Airframe_Weight;
```

```
    Motorized_Vehicle
        Weight を Engine Weight にリネームする；
    ...
end;
```

ここで、サブプログラムdisplayをオーバーライドしたいとする。どちらのサブプログラムを再定義しているのか曖昧になってしまうので、redefineは使えない。解決策は、継承されたサブプログラムの定義を解除し、新しいサブプログラムを書くことです：

```
class Airplane inherit
    Winged_Vehicle
        undefine display end;
    Motorized_Vehicle
        undefine display end;
feature
    display is ... end;
end;
```

Eiffelのリファレンスマニュアルでは、多重継承における曖昧さを解決するための rename、rendefine、undefineの使い方について詳しく説明している。

15.6. 設計上の考慮点

継承と合成

継承は、オブジェクト指向設計で使用できる構造化の1つの手法に過ぎない。もっと単純な方法は、ある抽象化を別の抽象化の中に含めるコンポジションです。あるレコードを別のレコードの中に含めることができることはご存じでしょうから、コンポジションについてはすでにご存知でしょう：

```
with Airplane_Package;
package SST_Package is
    type SST_Data is private;
private
    type SST_Data is
        record
            A: Airplane_Data;
            Mach: Float;
        end record;
end SST_Package;
```

and in C++, a class can have an instance of another class as an element:

```
class SST_Data {
private:
```

```
    Airplane_Data a;
    float mach;
};
```

モジュールのカプセル化をサポートすれば、自動的に抽象化を合成することができる。モジュールのカプセル化のサポートがあれば、自動的に抽象化を合成することができます。ジェネリックスは、型チェックされた言語ではどのような場合でも必要ですが、抽象化を合成するためにも使うことができます。しかし、継承には高度な言語サポート（Adaではタグ付きレコード、C ++では仮想関数）が必要であり、動的ディスパッチには実行時のオーバーヘッドがかかる。

動的なディスパッチが必要な場合は、もちろん、合成よりも継承を選択する 必要があります。しかし、動的なディスパッチングを行わない場合、この2つの選択は、純粋にどちらの方法が 「より良い 」設計を生み出すかを決定する問題である。C ++では、動的ディスパッチングを行う場合、基底クラスが作成されるときに、1つ以上のサブプログラムが仮想であることを宣言することによって決定する必要があります。これらのサブプログラムのみがディスパッチされます。Ada 95では、動的ディスパッチ は、タグ付き型の制御パラメータで宣言されたすべてのサブプログラムに対して 発生する可能性があります：

```
type T is tagged ...;
procedure Proc(Parm: T);
```

バインディングが静的か動的かの実際の判断は、呼び出しごとに個別に行われる。単純なレコードで十分な場合は、継承を使用しないでください。

この2つの方法の基本的な違いは、コンポジションが既存の閉じた抽象化を単純に利用 するのに対し、継承は抽象化の実装に関する知識を持つことである。閉じた抽象化のユーザーは、実装の変更から保護される。継承を使用する場合、基底クラスは、その変更が派生クラスに与える影響を考慮せずに変更することはできません。

一方、継承では、派生クラスによる効率的な直接アクセスが可能になります。さらに、派生クラスでは実装を変更することができますが、コンポジションでは既存の実装を使用することに制限されます。簡潔に言うと、コンポジションではモジュールを 「買う 」のも 「売る 」のも簡単であるのに対し、継承ではモジュールの供給者の 「パートナー 」になることができる。

There is no danger in a well-designed use of either method; problems can occur when inheritance is used indiscriminately as this can create too many dependencies among the components of a software system. We leave a detailed discussion of the relative merits of the two concepts to specialized texts on OOP. For the pro-inheritance viewpoint, see Meyer's book (*Object-oriented Software Construction*, Prentice-Hall International, 1988), especially Chapters 14 and 19. Compare this with the pro-composition viewpoint expressed in: J.P. Rosen, "What orientation should Ada objects take?", *Communications of the ACM*, 35(11), 1992, pp. 71–76.

Uses of inheritance

継承の使い方をいくつかのカテゴリに分けるのは便利です。

[10]This classification is due to Ian Muang and Richard Mitchell.

SSTは飛行機として振る舞う。これはコード共有のための継承の単純な使い方です。Airplaneに適した操作はSSTにも適しており、必要に応じてオーバーライドすることができます。

Linked SetとBit SetはSetを多相的に置換可能である。共通の祖先から派生することで、異なる実装の集合を同じ操作で処理することができます。さらに、祖先の型に基づいて、その型ファミリ全体の要素を含む異種データ構造を作成することもできる。

共通のプロパティは複数のクラスで継承される。このテクニックはSmalltalkやEiffelのような大規模なライブラリで使用され、共通のプロパティは祖先クラス（アスペクトクラスと呼ばれることもある）に因数分解される。例えば、Comparableクラスは 「i」のような関係演算を宣言するのに使われ、IntegerやFloatのようなそのような演算を持つクラスはComparableを継承する。

あるクラスから論理関数を継承し、別のクラスからその実装を継承することで、クラスを作成することができる。典型的な例はバウンデッド・スタックで、その機能はStackから、実装はArrayから（多重に）継承する。より一般的には、多重継承によって構築されたクラスは、いくつかのアスペクトクラスからその機能を継承し、さらに1つのクラスからその実装を継承する。

これらのカテゴリーは相互に排他的でも網羅的でもなく、ソフトウェア設計においてこの強力な構造を使用するためのガイドとして意図されています。

オーバーロードとポリモーフィズム

オーバーロードはポリモーフィズム(「多形式」)の一形態ですが、この2つの概念は全く異なる目的で使用されます。オーバーローディングは、全く異なる型を操作するサブプログラムに同じ名前を 与えるための便宜的なものとして使われます。例えば

C++

```
void proc put(int);
void proc put(float);
```

共通名は便宜上使用されているだけであり、intとfloatの間には何の関係もないため、#はオーバーロードです。一方

C++

```
仮想 void set speed(int);
```

は1つのサブプログラムであるが、たまたま飛行機の種類によって実装が異なる。

オーバーロードと動的ポリモーフィズムを混在させることは技術的に困難である。派生クラス内で、基底クラスに登場するサブプログラムのオーバーロードを試みないでください：

C++

```
クラス SST Data : public Airplane Data { publi
c :
    void set_speed(float);    // float rather than int
};
```

The rules of C++ specify that this subprogram neither overloads nor overrides the subprogram in the base class; instead it *hides* the definition in the base class just like an inner scope!

Ada 95 allows overloading and overriding to coexist:

Ada

```
with Airplane_Package; use Airplane_Package;
package SST_Package is
    type SST_Data is new Airplane_Data with ...
    procedure Set_Speed(A: in out SST_Data; I: in Integer);
            -- Overrides primitive subprogram from Airplane_Package
    procedure Set_Speed(A: in out SST_Data; I: in Float);
            -- Overloads, not a primitive subprogram
end SST_Package;
```

親型にFloatパラメータを持つSet Speedプリミティブサブプログラムは存在しないため、2番目の宣言は単に同じ名前をオーバーロードした無関係なサブプログラムです。これは合法であっても、型のユーザが混乱する可能性が高いため、避けるべきです。SSTパッケージだけを見ても(コメントなしでも!)、どのサブプログラムがオーバーライドで、どのサブプログラムが単なるオーバーロードなのかわかりません：

Ada

```
procedure Proc(A: Airplane_Data'Class) is
begin
    Set_Speed(A, 500);      -- OK, dispatches
    Set_Speed(A, 500.0);    -- Error, cannot dispatch !
end Proc;
```

15.7 動的ポリモーフィズムのメソッド

We conclude this chapter by summarizing dynamic polymorphism in languages for object-oriented programming.

スモールトークでは、サブプログラムを呼び出すたびに、そのサブプログラムが 見つかるまで継承階層を探索する動的ディスパッチが必要になります。

Eiffel サブプログラムの呼び出しはすべて動的にディスパッチされます（静的バインディングに最適化されている場合を除く)。Smalltalkとは異なり、オーバーライドの可能性はコンパイル時にわかるので、ディスパッチにはジャンプテーブルに基づく固定オーバーヘッドがある。

C ++ 明示的に仮想宣言され、ポインタや参照を介して間接的に呼び出されるサブプログラムは、動的にディスパッチされる。ランタイムディスパッチには固定オーバーヘッドがあります。

Ada 95 動的ディスパッチは、タグ付き型のプリミティブサブプログラムで、 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暗黙的に使われます。ランタイムディスパッチには固定オーバーヘッドがあります。

言語設計は、動的ポリモーフィズムに必要な明示的なプログラミングとオーバーヘッドに違いがあり、これらはプログラミングスタイルとプログラム効率に影響を与えます。この原則を明確に理解することで、オブジェクト指向言語を比較し、どの言語を選んでも優れたオブジェクト指向プログラムを設計・作成する能力を向上させることができます。

15.8 Exercises

1. Implement the Set packages in Ada 95 and the classes in C++.

2. Ada 95の抽象型やC++の抽象クラスはデータコンポーネントを持つことができますか? もしそうなら、それらは何に使われるのでしょうか?

Ada

```
type Item is abstract tagged
    record
        I: Integer;
    end record;
```

3. Write a program for a heterogeneous queue based on an abstract class.

4. 集合パッケージ／クラスを、単なる整数要素型ではなく、一般的な要素型で実装しなさい。

5. Eiffelの多重継承を詳しく学び、C++の多重継承と比較してください。

6. Eiffelにおける多重継承の標準的な例として、リストと配列の両方を継承して実装された固定サイズのリストがあります。多重継承を持たないAda 95では、このようなADTをどのように記述するのでしょうか?

7. C ++で保護されたデータを定義することの危険性は? これはAda 95の子パッケージにも当てはまりますか?

8. 子パッケージを多用するAda 95の標準ライブラリの構造を勉強しなさい。C++の標準的なI/Oクラスの構造と比較しなさい。

9. Study the package Finalization in Ada 95 which can be used to write constructors and destructors. Compare it with the C++ constructs.

10. What is the relationship between assignment statements and constructors/destructors?

11. Give examples of the use of class-wide objects.

12. 12.C++のフレンド指定子とJavaのパッケージ構成子の関係は？

13. C ++では、派生クラスのメンバを可視化するためにprotected指定子を使用する。パッケージ構文は、Javaのprotectedの概念にどのような影響を与えるか。

14. Compare interface in Java with multiple inheritance in C++.

15. 15.C++の名前空間とJavaのパッケージの違いを、特にファイルや入れ子に 関する規則に関して分析しなさい。

16. Compare clone and equals in Java with these operations in Eiffel.