Software Factory

CDL
(Component Description Language)
User's Guide

NOTICE FOR USERS:

This User Guide is a general instruction for Open CASCADE study. It may be incomplete and even contain occasional mistakes, particularly in examples, samples, etc. Open CASCADE S.A.S. bears no responsibility for such mistakes. If you find any mistakes or imperfections in this document, or if you have suggestions for improving this document, please, contact us and contribute your share to the development of Open CASCADE Technology: bugmaster@opencascade.com

15 bis rue Ernest RENAN

92136 ISSY LES MOULINEAUX

FRANCE

TABLE OF CONTENTS

1. CDL and Application Architecture

CDL is the component definition language of the Open CASCADE programming platform. Some components, which CDL allows you to create, are specific to Open CASCADE application architecture. These and other components, which you can define using CDL include the following:

- Class (including enumeration, exception)
- Package
- Schema
- Executable
- Client.

A **class** is the fundamental software component in object-oriented development. Because of a very large number of resources used in large-scale applications, the **class** itself is too small to be used as a basic management unit.

So, while the class is the basic data component defined in CDL, this language also provides a way to group classes, **enumerations**, and **exceptions** together – the **package**. A package groups together a number of classes, which have semantic links. For example, a geometry package would contain Point, Line, and Circle classes. A package can also contain enumerations, exceptions, and package methods. In practice, a class name is prefixed with the name of its package e.g. Geom_Circle.

Using the services described in the **packages**, you can construct an **executable**. You can also group together services provided by **packages**.

To save data in a file, you need to define persistent classes. Then, you group these classes in a schema, which provides the necessary read/write tools.
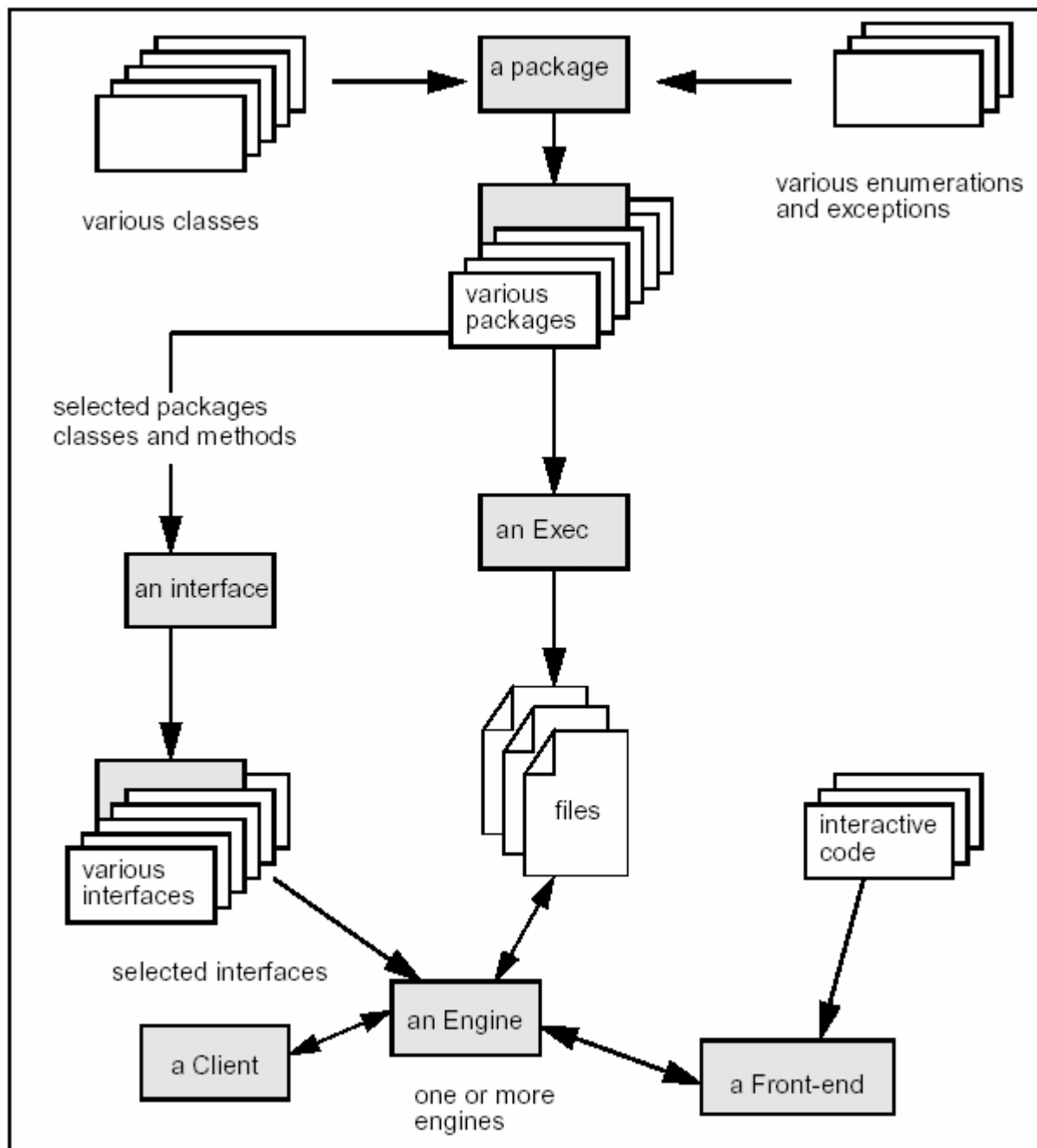
Figure 1. Building an Open CASCADE application

2. Introduction to CDL

2.1 Purposes of the Language

You can use CDL to **define data** in the Open CASCADE environment. CDL allows you to define various kinds of data types supporting the application architecture and development methodology, which you envision. CDL is neither an analysis formalism (e.g. Booch methodology) nor a data manipulation language (e.g. C++).

You use CDL in the **design phase** of a development process to define a set of software components which best model the concepts stated in the application specification.
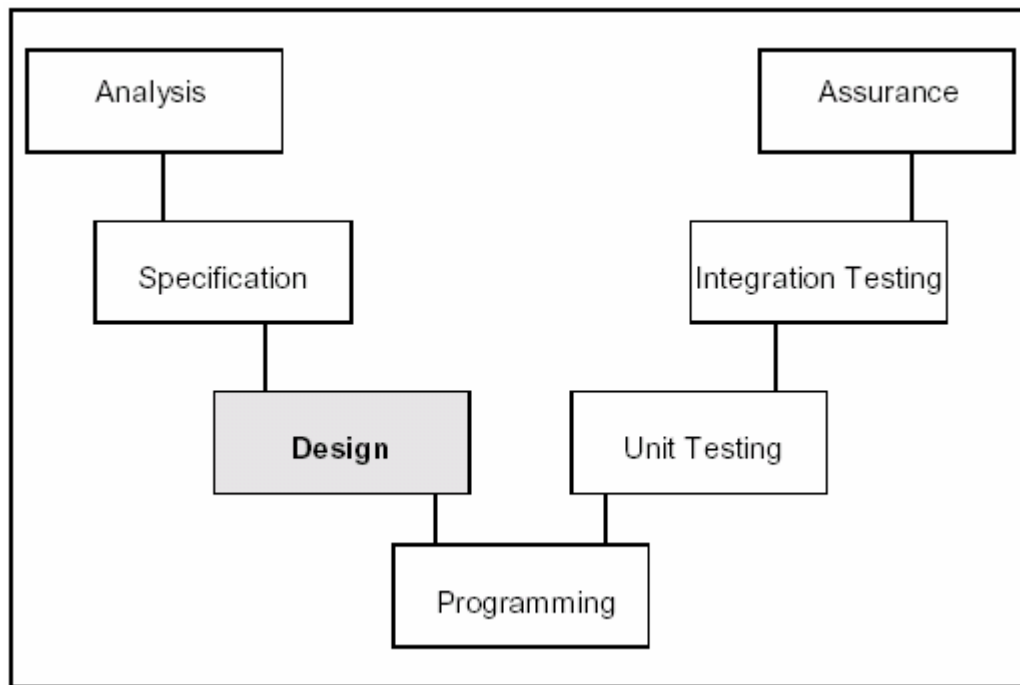
Figure 2. The Development Process

From a structural point of view, CDL is an object-oriented language. It is centered on the notion of the **class** - a data type, which represents an elementary concept. CDL offers various means of organizing classes, mostly under the fundamental form of **packages**. A package contains a set of classes, which share some semantic relationship. This greatly simplifies your task of managing individual classes when confronted with a very large number of them.

Once you have defined the classes and packages using CDL, you can implement their **methods** - i.e., their functionality - in one of the data manipulation languages supported by the Open CASCADE environment (currently C++).

Even though you can describe classes directly in C++ and save them as header files (.hxx), to do so would forfeit all the advantages of using CDL. These are:

- Precise, complete, and easy-to-read description of the software components.

- Creation of a link with the database; object persistence forms part of the predefined environment of the language.

- Multi-language access to the services of an application engine – a specific architectural form created using the CDL tools, which serves as the motor of an application.

2. 2 Overview of CDL

CDL is an object-oriented language. In other words, it structures a system around data types rather than around the actions carried out on them. In this context, an **object** is an **instance** of a data type, and its definition determines how you can use it. Each data type is implemented by one or more classes, which make up the basic elements of the system.

2. 2. 1 Classes

A class is an implementation of a **data type**. It defines its **behavior** and its **representation**.

The behavior of a class is its programming interface - the services offered by its **methods**. The representation of a class is its data structure - the **fields**, which store its data.

Every object is an **instance** of its class. For example, the object *p* of the data type *Point* is an instance of the class Point.

The class Point could be defined as in the example below:

Example

```
class Point from GeomPack
        ---Purpose: represents a point in 3D space.
        is
        Create returns Point;
    fields
        x, y, z : Real;
    end Point;
```

The definition of this class comprises two sections:

- one starting with the keywords **is**

- one starting with the keyword **fields**.

The first section contains a list of methods available to the clients of the class. The second section defines the way in which instances are represented. Once this class has been compiled you could **instantiate** its data type in a C++ test program as in the example below:

Example

```
GeomPack_Point p;
```

2. 2. 2 Categories of Types

You declare the variables of a **data manipulation language** as being of certain data types. These fall into two categories:

- Data types manipulated by handle (or reference)

- Data types manipulated by value
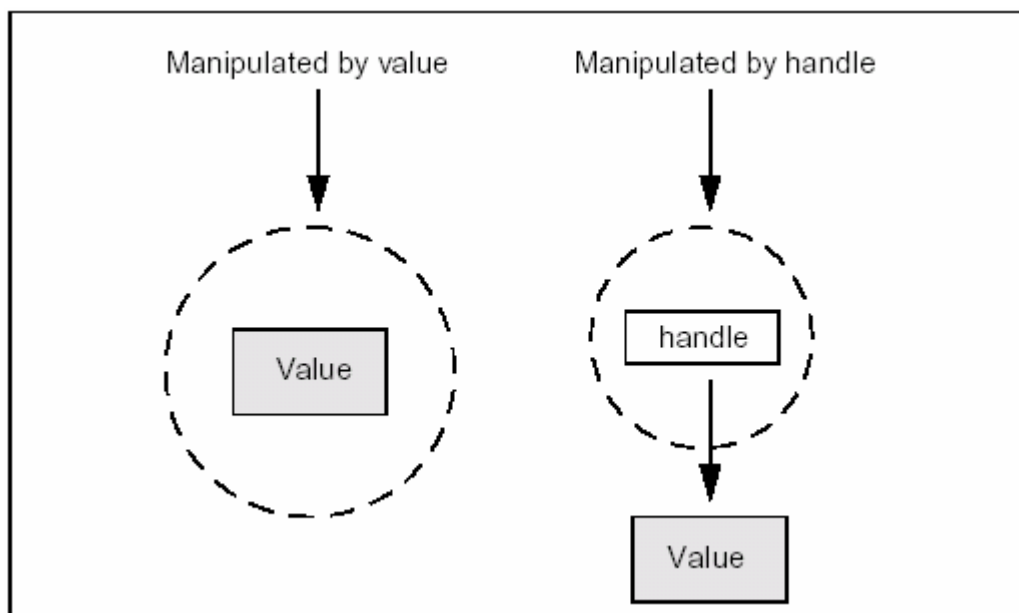
Figure 3. Manipulation of data types

As seen above, you implement data types using classes. However, classes not only

define their data representation and methods available for their instances, but they also define how the instances will be manipulated:

- A data type manipulated by value contains the instance itself.

- A data type manipulated by handle contains a reference to the instance.

The most obvious examples of data types manipulated by value are the predefined **primitive types**: Boolean, Character, Integer, Real ...

A variable of a data type manipulated by handle, which is not attached to an object, is said to be **null**. To reference an object, you need to instantiate the class with one of its constructors. This is done in C++ as in the following syntax:

Example

```
Handle(myClass) m = new myClass;
```

---

2. 2. 3 Persistence

An object is called **persistent** if it can be permanently stored. In other words, you can use the object again at a later date, both in the application, which created it, and in another application.

In order to make an object persistent, you need to declare it in CDL as inheriting from the **Persistent** class, or to have one of its parent classes inheriting from the **Persistent** class.

Note that the classes inheriting from the Persistent class are handled by reference.

Example

```
class Watch inherits Persistent
```

In this example, building the application, you add the Watch class to the corresponding schema of data types.
If, running the application, you instantiate an object of the Watch class, you have the possibility of storing it in a file.
You cannot store objects instantiated from classes, which inherit from the **Storable** class. However, you can store them as fields of an object, which inherits from Persistent.

Note that the objects inheriting from Storable are handled by value.

Example

```
If
        class WatchSpring inherits Storable
     //then this could be stored as a field of a Watch
     //object:
        class Watch inherits Persistent
        is.....
        fields
        name : ConstructorName;
    powersource : WatchSpring;
    end;
```

---

2. 2. 4 Packages

In large-scale long-term development the task of marshalling potentially thousands of classes is likely to quickly prove unmanageable. CDL introduces the notion of **package** of classes containing a set of classes, which have some semantic or syntactic relationship. For example, all classes representing a particular set of electronic components might make up a package called Diode.

As the package name prefixes the class name when implementing such class (in C++ for example), classes belonging to different packages can have the same name. For example, two packages, one dealing with finance and the other dealing with aircraft maneuvers, might both contain a class called Bank, without any possibility of confusion.

Example

```
Finance_Bank
Attitude_Bank
```

---

2. 2. 5 Inheritance

The purpose of inheritance is to reduce development workload. The inheritance mechanisms allow you to declare a new class as already containing the characteristics of an existing class. This new class can then be rapidly specialized for a task at hand. This eliminates the necessity of developing each component "from scratch".

For example, having already developed a class BankAccount, you can quickly specialize new classes - SavingsAccount, LongTermDepositAccount, MoneyMarketAccount, RevolvingCreditAccount, etc..

As a consequence, when two or more classes inherit from a parent (or ancestor) class, all these classes surely inherit the behavior of their parent (or ancestor). For example, if the parent class BankAccount contains the method Print that tells it to print itself out, then all its descendent classes offer the same service.

One way of ensuring the use of inheritance is to declare classes at the top of a

hierarchy as being **deferred**. In such classes, the inherited methods are not implemented. This forces you to create a new class used to redefine the methods. In this way, you guarantee a certain minimum common behavior among descendent classes.

Example

```
deferred class BankAccount inherits Persistent
      is
      .......
      fields
      name : AccountHolderName;
  balance : CreditBalance;
  end;
```

2. 2. 6 Genericity

You will often wish to model a certain type of behavior as a class. For example, you will need a list modeled as a class.

In order to be able to list different objects, the class **List** must be able to accept different data types as parameters. This is where genericity comes in: you first declare a list declared as the generic class **List**, willing to accept any data type (or only a particular set of acceptable data types). Then, when you want to make a list of a certain type of object, you instantiate the class **List** with the appropriate data type.

Example

```
generic class NewList (Item)
   inherits OldList
      is
      .....
      end ;
```

Items may be of any type, an Integer or a Real for example.

When defining the package, add the following line:

Example

```
class NewListOfInteger instantiates
      NewList (Integer);
```

2. 2. 7 Exceptions

The behavior of any object is implemented by methods, which you define in its class declaration. The definition of these methods includes not only their signature (their programming interface) but also their domain of validity.

In CDL, this domain is expressed by **exceptions**. Exceptions are raised under various error conditions. This mechanism is a safeguard of software quality.

2. 2. 8 Completeness

You use CDL to define data types. Such definitions are not considered complete unless they contain the required amount of structured commentary.

The compiler does not enforce this required degree of completeness, so it is the responsibility of the developer to ensure that all CDL codes are properly annotated.

Completeness is regarded as an essential component of long-term viability of a software component.

2. 3 Lexical Conventions

2. 3. 1 Syntax notation

In this manual, CDL declarations are described using a simple variant of the Backus-Naur formalism. Note the following:

- Italicized words, which may also be hyphenated, denote syntactical categories, for example:

    declaration-of-a-non-generic-class

- Keywords appear in bold type:

    class

- Brackets enclose optional elements:

    `identifier [from package-name]`

- Curly braces enclose repeated elements. The element may appear zero or many times:

    `integer ::= digit{digit}`

- Vertical bars separate alternatives:

    passing-method ::= [**in**] | **out** | **in out**

- Two apostrophes enclose a character or a string of characters, which must appear:

    `exponent ::= 'E'['+']integer | 'E-' integer`

    NOTE
    So as to introduce ideas progressively, the examples presented in this manual may be incomplete, and thus not compilable by the CDL compiler.

2. 3. 2 Lexical elements

A CDL source is composed of text from one or more compiled units. The text of each compiled unit is a string of separate lexical elements: **identifiers**, **keywords**, **constants**, and **separators**. The separators (blank spaces, end of line, format characters) are ignored by the CDL compiler, but these are often necessary for separating identifiers, keywords, and constants.

2. 3. 3 Comments

With CDL, you cannot use the expression of all useful information about a development unit. In particular, certain information is more easily expressed in natural language. You can add such information to the CDL description of a data type.

Rubrics and free comments are to be differentiated:

Free comments are preceded by the characters "--" (two hyphens), and they terminate at the end of the line in which they appear.

Example

```
--This is a comment
```

Unlike rubrics, free comments can appear before or after any lexical element. The first written character of the comment itself *must not* be a hyphen. If a hyphen is necessary make sure it is preceded by a blank.

Example

```
-- -List item
```

Rubrics are various types of comments attached to CDL components. A rubric is a comment made up of three hyphens, name of the rubric (without any intermediary space) and then a colon and a space. It is terminated by the beginning of the following rubric, or by the end of the commentary.

Example

```
---Purpose: This is an example of a
--rubric composed of a
--comment which extends to
--four lines.
```

The different categories of rubrics and the form of their content do not depend on the Component Description Language, but on the tool for which it is intended.

The use of commentary is generally governed by the internal programming standards of an enterprise. You are encouraged to use various well-defined rubrics, such as Purpose, Warning, Example, References, Keywords, etc.

These rubrics can be attached to:

- Packages
- Classes
- Methods
- Schemas
- Executables
- Clients

2. 3. 4 Identifiers

An identifier is an arbitrary chain of characters, either letters or digits, but it must begin with a letter.

The underscore "_" is considered to be a letter as long as it doesn't appear at the beginning or the end of an identifier.
Capital and small letters are not equivalent (i.e. AB, Ab, aB, ab are four different identifiers).

2. 3. 5 Keywords

The following is a list of keywords.

alias	any	as	asynchronous
class	client	deferred	end
enumeration	exception	executable	external
fields	friends	from	generic
immutable	imported	inherits	instantiates
is	library	like	me
mutable	myclass	out	package
pointer	primitive	private	protected
raises	redefined	returns	schema
static	to	uses	virtual

In a CDL file, the following characters are used as punctuation:
; : , = ( ) [ ] ' "

2. 3. 6 Constants

There are three categories of constants:

- Numeric
- Literal
- Named

Numeric Constants

There are two types of numeric constants: integer and real.

An **integer** constant consists of a string of digits, which may or may not be preceded by a sign. Integer constants express whole numbers.

Examples

1995   0   -273   +78

A **real** constant may or may not be preceded by a sign, and consists of an integral part followed by a decimal point and a fractional part (either one or both parts may be null, but both parts must always be present). It may also be followed by the letter E to indicate that the following figures represent the exponent (also optionally signed).

Examples

5.0   0.0   -0.8E+3   5.67E-12

Literal Constants

Literal constants include individual characters and strings of characters.

An **individual character** constant is a single printable character enclosed by two apostrophes. (See the definition of the class Character in the Standard Package).

Examples

'B'   'y'   '&'   '*'   '' "

A **string** constant is composed of printable characters enclosed by quotation marks.

Examples

''G''   ''jjjj''   ''This is a character string, isn't it?''

The **quotation mark** can itself appear within a character string as long as it is preceded by a backslash.

Examples

''This film was originally called \''Gone with the Tide\''.''

Named Constants

Named constants are sub-divided into two categories: Booleans and enumerations.

Booleans can be of two types: True or False.

An **enumeration** constant is an identifier, which appears in the description of an enumeration.

3. Software Components

3. 1 Predefined Resources

3. 1. 1 Primitive types

Primitive types are predefined in the language and they are **manipulated by value**.

Four of these primitives are known to the schema of the database because they inherit from the class **Storable**. In other words, they can be used in the implementation of persistent objects, either when contained in entities declared within the methods of the object, or when they form part of the internal representation of the object.

The primitives inheriting from Storable are the following:

Boolean	Is used to represent logical data. It has only two values: *True* and *False*.
Byte	8-bit number.
Character	Designates any ASCII character.
ExtCharacter	Is an extended character.
Integer	Is an integer number.
Real	Denotes a real number (i.e. one with a whole and a fractional part, either of which may be null).
ShortReal	Real with a smaller choice of values and memory size.

There are also non-storable primitives. They are:

CString	Is used for literal constants.
ExtString	Is an extended string.
Address	Represents a byte address of undetermined size.

The services offered by each of these types are described in the Standard Package.

3. 1. 2 Manipulating types by reference (by handle)

Two types are manipulated by handle:

- Types defined using classes inheriting from the **Persistent** class are storable in a file.

- Types defined using classes inheriting from the **Transient** class. These types are not storable as such in a file.
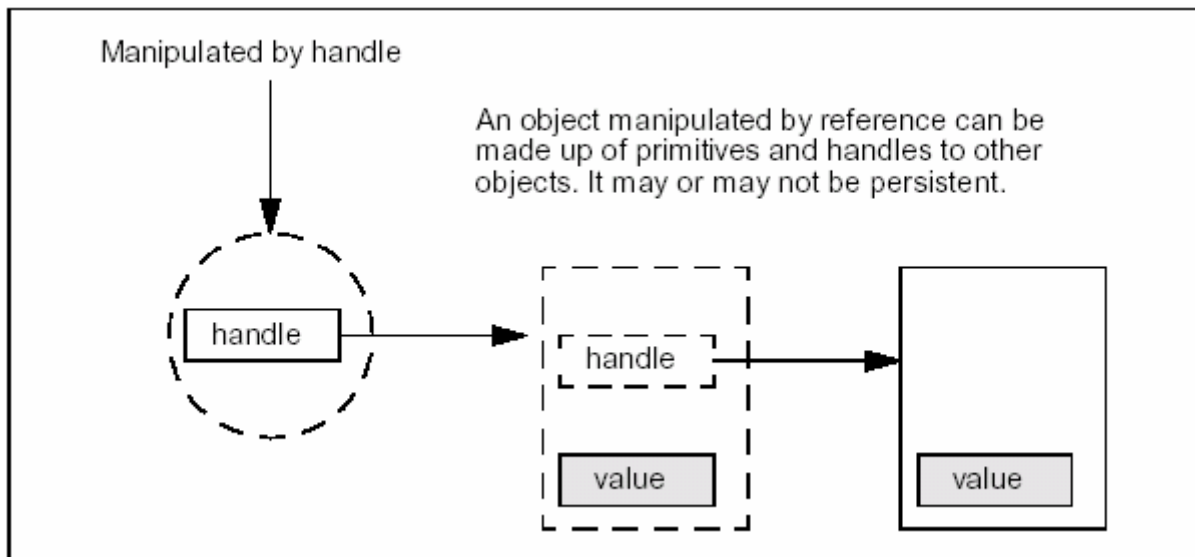
Figure 4. Manipulation of a data type by reference

3. 1. 3 Manipulating types by value

Types, which are manipulated by value, behave in a more direct fashion than those manipulated by handle. As a consequence, they can be expected to perform operations faster, but they cannot be stored independently in a file.

You can store types known to the schema (i.e. either primitives or inheriting from Storable) and manipulated by value inside a persistent object as part of the representation. This is the only way for you to store objects "manipulated by value" in a file.
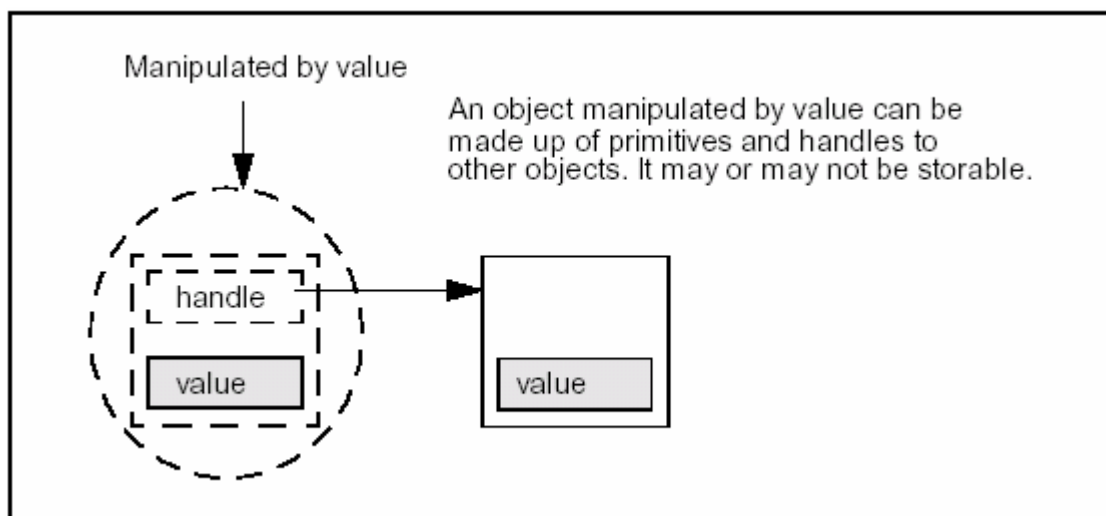
Figure 5. Manipulation of a data type by value

Three types are manipulated by value:

- Primitive types

- Enumerated types

- Types defined by classes not inheriting from Persistent or Transient, whether directly or not

3. 1. 4 Summary of properties

	Manipulated by handle	Manipulated by value
storable	**Persistent**	**Primitive**, **Storable** (storable if nested in a persistent class)
temporary	**Transient**	**Other**

Figure 6. Summary of the relationship for the various data types between how they are handled and their storability.

3. 2 Classes

3. 2. 1 Class declaration

The class is the main system for creating data types under CDL. By analyzing any CDL-based software, you find that classes are the modular units that make up packages. When you describe a new class, you introduce a new data type.

Whatever the category of the described type (manipulated by value, Storable or not, manipulated by handle, Persistent or not) the structure of the class definition remains the same. The syntax below illustrates it:

Example

```
-- declaration-of-a-simple-class ::=
class class-name from package-name
   [uses data-type { ',' data-type } ]
   [raises exception-name { ',' exception-name} ]
   is class-definition
   end [ class-name ] ';'
   data-type ::= enumeration-name | class-name |
exception-name | primitive-type
   package-name ::= identifier
   class-name ::= identifier
   class-definition ::=
   [{member-method}]
   [declaration-of-fields]
   [declaration-of-friends]
```

Class name becomes a new data type, which you can use inside its own definition. Other types appearing in the definition must either be primitive types, previously declared classes, exceptions, or enumerations.

Apart from the types defined in the Standard Package, which are **implicitly visible** everywhere, you need to declare the data types after the keyword **uses**. This concerns both the class behavior and its internal representation.

Exceptions are declared after the word **raises**.

Example

```
class Line from GeomPack
usesPoint, Direction, Transformation
raisesNullDirection, IdenticalPoints
is-- class definition follows here
        -- using Point, Direction and
        -- Transformation objects,and the
        -- NullDirection and Identical-
        -- -Points exceptions.
end Line;
```

The elements, which make up the definition of a class, are divided into four parts:

- the behavior
- the invariants
- the internal representation
- the friend methods and friend classes.
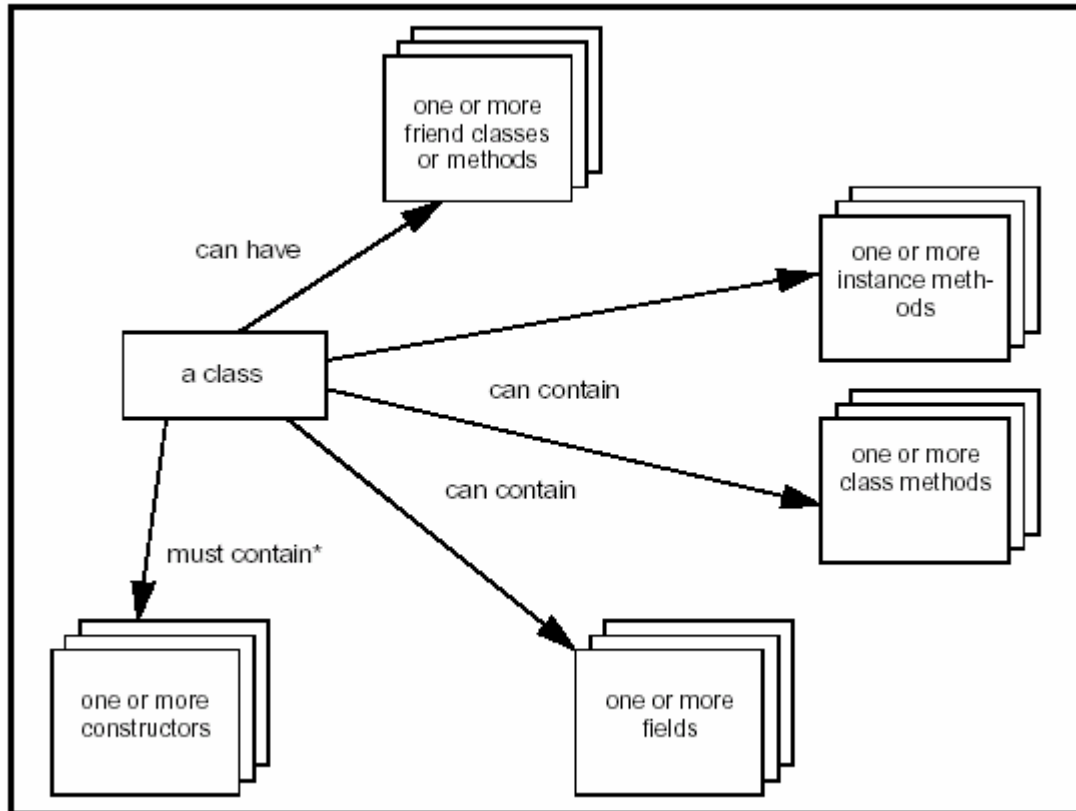
Figure 7. Contents of a class
* a deferred class does not have to contain a constructor

3. 2. 2 Categories of classes

Classes fall into three categories:

- Ordinary classes
- Deferred classes
- Generic classes

Deferred classes

The principal characteristic of a **deferred class** is that you cannot instantiate it. Its purpose is to provide you with a given behavior shared by a hierarchy of classes and dependent on the implementation of the descendents. This allows you to guarantee a certain base of inherited behavior common to all classes based on a particular deferred class. Deferred classes are declared as in the following syntax:

Example

```
-- declaration-of-a-deferred-class ::=
deferred class class-name
[inherits class-name {',' class-name}]
[uses data-type {',' data-type}]
[raises exception-name {',' exception-name}]
is class-definition
end [class-name]';'
```

Generic classes

The principal characteristic of a **generic class** is that it offers you a set of functional behavior allowing you to manipulate other data types. To instantiate a generic class you need to pass a data type in argument. Generic classes are declared as in the following syntax:

Example

```
-- declaration-of-a-generic-class ::=
[deferred] generic class class-name '('generic-type
{',' generic-type}')'
[inheritsclass-name {',' class-name}]
[usesdata-type {',' data-type}]
[raisesexception-name {',' exception-name}]
[{[visibility] declaration-of-a-class}]
is class-definition
end [class-name]';'
generic-type ::= identifier as type-constraint
identifier ::= letter{[underscore]alphanumeric}
type-constraint ::= any | class-name ['('data-type {','
data-type}')']
```

3. 3 Packages

3. 3. 1 Package declaration

Packages are used to group   classes, which have some logical coherence. For example, the Standard Package groups together all the predefined resources of the language. In its simplest form, a package contains the declaration of all data types, which it introduces. You may also use a package to offer public methods and hide its internal classes by declaring them private.

Example

```
-- package-declaration ::=
package package-name
[uses package-name {',' package-name}]
is package-definition
end [package-name]';'
-- package-name ::=
identifier
-- package-definition ::=
[{type-declaration}]
[{package-method}]
-- type-declaration ::=
[private] declaration-of-an-enumeration |
[private] declaration-of-a-class |
declaration-of-an-exception
-- package-method ::=
identifier [simple-formal-part][returned-type
          -declaration]
[error-declaration]
[is private]';'
```

The data types described in a package *may* include one or more of the following data types:

- Enumerations

- Object classes

- Exceptions

- Pointers to other object classes.

Inside a package, two data types *cannot* have the same name.

You declare data types before using them in the definition of other data types.

When two classes are **mutually recursive**, one of the two *must* be first declared in an incomplete fashion.

Grouped behind the keyword **uses** are the names of all the packages containing definitions of classes of which the newly introduced data types are clients.

The methods you declare in a package do not belong to any particular class. **Package methods** *must* carry a name different from the data types contained in the package. Like any other method, they can be overloaded. With the exception of the keyword **me** and the visibility (a package method can *only* be either public or private) package methods are described in the same way as **instance methods**.
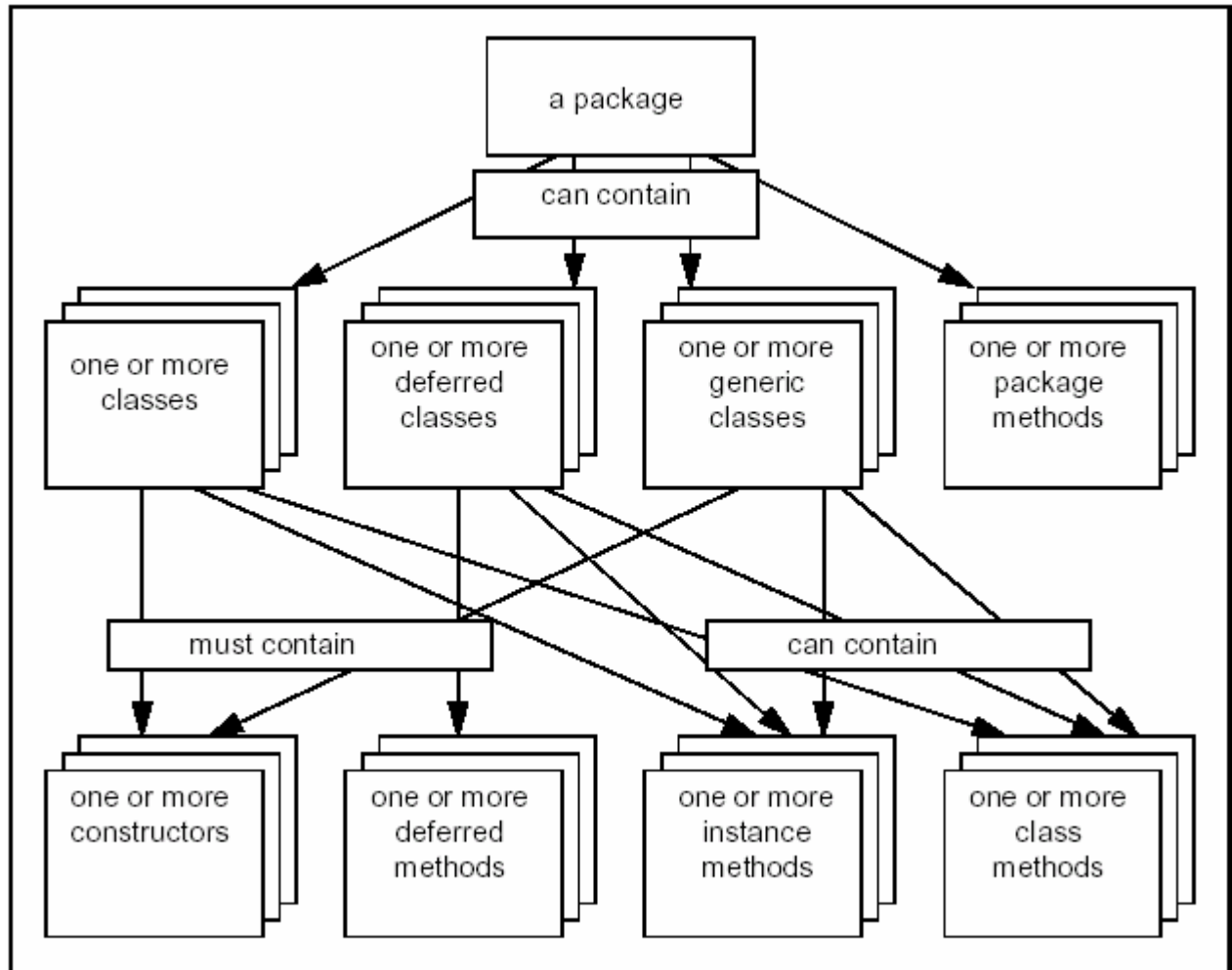
Figure 8. Contents of a package

The example of the package below includes some of the basic data structures:

Example

```
package Collection
    uses
    Standard
    is
exception NoSuchObject inherits Failure;
exception NoMoreObject inherits Failure;
generic class SingleList;
generic class Set;
end Collection;
```

Note that the class Set is declared after the declarations of the NoSuchObject and NoMoreObject exceptions and the SingleList class of which Set is a client. In the same way, the classes Failure, Persistent, and the exception NoSuchObject are defined before they are used. They are defined in the Standard package, which appears after the keyword **uses**.

3. 3. 2 Name space

The **name space** or **scope** of a class extends from the beginning of its declaration up to the end of the package in which it appears.

Sometimes, two classes, which come from separate packages, are both visible to a third package and carry the same name. For example, there might be two different classes both called "Window" in a screen generator package and in an architectural package. As a client of a data type, you can find yourself in the position of having to remove the ambiguity over the origin of this type; you do this by means of the keyword **from**.

Example

```
-- class-name ::=
identifier [from package-name]
-- exception-name ::=
identifier [from package-name]
-- enumeration-name ::=
identifier [from package-name]
```

You can use the keyword **from** everywhere the name of a class, exception, or enumeration appears. As a consequence, as a client of the class "Window" you could write wherever necessary:

Example

```
Window from ScreenGenerator
-- or
Window from ArchitecturalFeatures
```

NOTES
Within the description of a package the keyword **from** *must be used when referencing any data type that is not defined in this package.*

Here is a further example:

Example

```
class Line from Geom
 uses
 Point from Geom2d,
 Point from Geom3d
 is
 -- class definition using
 -- Point from <AppropriatePackage>
 -- wherever Point appears
 end;
```

3. 3. 3 Declaration of classes

You cannot describe a package in one single file. You need to describe it in different units and send them separately to the CDL compiler. Each compilation unit can contain the declaration of a class or of a package. When you describe a class in a unit different than that, which describes its package, you need to specify which package the class belongs to. You do this using the keyword **from**.

If the **from** clause appears in the **uses** clause of the package, it does not need to be repeated elsewhere.

The following example takes the package "Collection" which was presented above, but this time it is divided into three compilation units.

Example

```
-- First compilation unit, the package "Collection" :
package Collection
    uses
    Standard
    is
    exception NoMoreObject inherits Failure from Standard;
exception NoSuchObject inherits Failure from Standard;
generic class SingleList;
generic class Set, Node, Iterator;
end Collection;
-- Second compilation unit, the class "SingleList" :
generic class SingleList from Collection (Item as
Storable)
    inherits
        Persistent from Standard
        raises
        NoSuchObject from Collection
        is
        -- definition of the SingleList class
        end SingleList;
    -- Third compilation unit, the class "Set" :
generic class Set from Collection (Item as Storable)
    inherits
      Persistent from Standard;
    raises
      NoSuchObject from Collection,
      NoMoreObject from Collection
      private class Node instantiates SingleList
      from Collection (Item);
    -- etc....
      end Set;
```

NOTE
It is not explicitly stated that the "Node" class belongs to the "Collection" package. In fact any nested class necessarily belongs to the package of the class, which encompasses it.

Note that a package can hide certain classes (just as it can hide methods) by declaring them **private**. To make a class private, you prefix its description with the keyword **private**. In the example of the "Collection" package, the "SingleList" class

serves only to implement the "Set" class. It is recommended to make it private. You write this as in the following syntax:

Example

```
package Collection
    uses
    Standard
    is
    generic class Set, Node, Iterator;
private generic class SingleList;
exception NoMoreObject inherits Failure from Standard;
end Collection;
```

---

3. 4 Other Data Types

These are:

- Enumerations

- Imports

- Aliases

- Exceptions

- Pointers

3. 4. 1 Enumerations

The **enumerated types** are the second type, which is manipulated by value. Unlike the primitive types they are extensible because they are defined by the user under the form of enumerations. An enumeration is an ordered sequence of named whole constant values called enumeration constants.

Example

```
declaration-of-an-enumeration ::=
numeration enumeration-name
is identifier {',' identifier}
[end [enumeration-name]]';'
enumeration-name ::= identifier
```

The declaration of an enumeration creates an enumerated type. An object of this type can successively take the value of any one of the constants cited in the list.

Example

```
enumeration MagnitudeSign is Negative, Null, Positive;
```

Inside a package, two enumeration constants cannot have the same name, even if they belong to different enumerated types.

Example

enumeration Cars is
    Honda,
    Ford,
    Volkswagen,
    Renault
    end;
enumeration AmericanPresidents is
    Nixon,
    Reagan,
    Ford, -- Error: 'Ford' already defined
    Carter
    end;

---

3. 4. 2 Imports

An **imported type** is one of which which has not been defined in CDL. It is up to the supplier of this data type to ensure compatibility with the CDL language by providing services which allow CDL to recognize the imported data type.

The CDL syntax for declaring an imported type is:

declaration-of-an-imported-type::=
        [**private**] **imported** <typename> ;

Example

```
-- You can define an imported type as follows:
-- In the MyPack.cdl file, you declare the imported
type:
-- package MyPack
   ....
   imported MyImport;
....
   end Mypack;
-- In the MyPack_MyImport.hxx file, you write the
following
-- C++ code:
#ifndef _MyPack_MyImport_HeaderFile
#define _MyPack_MyImport_HeaderFile
#include <Standard_Type.hxx>
typedef unsigned long MyPack_MyImport;
extern const Handle(Standard_Type)& TYPE
(MyPack_MyImport);
-- In the MyPack_MyImport.cxx file, you write the
following
-- C++ code:
#ifndef _MyPack_MyImport_HeaderFile
#include <MyPack_MyImport.hxx>
#endif
const Handle(Standard_Type)& TYPE (MyPack_MyImport)
```

```
{
    static Handle(Standard_Type) _aType =
        new Standard_Type ("MyPack_MyImport",sizeof
            (MyPack_MyImport))
            return _aType;
}
```

Then, add the names of these two files (MyPack_MyImport.hxx, MyPack_MyImport.cxx) to a file called FILES in the src subdirectory of the package. If the file does not exist you must create it.

3. 4. 3 Aliases

An **alias** is an extra name for a type, which is already known. It is declared as in the following syntax:

declaration-of-an-alias::=
```
        [private] alias <type1> is <type2> [from <apackage>] ;
```

Example

```
alias Mass is Real;
---Purpose:
    -- Defined as a quantity of matter.
    -- Gives rise to the inertial and
    -- gravitational properties of a body.
    -- It is measured in kilograms.
```

Having defined *Mass* as a type of *Real*, you can use either *Mass* or *Real* to type an argument when defining a method.

3. 4. 4 Exceptions

In the model recommended by CDL, the principal characteristic of errors is that they are treated in a different place from the place where they appear. In other words, the methods recovering and those raising a given exception are written independently from each other.

Subsequently this poses the problem of communication between the two programs. The principle adopted consists in viewing the exception as both a class and an object. The exception class (by means of one of its instances) is used to take control of an exception, which has been raised.

Consequently, error conditions are defined by means of **classes of exceptions**. Exception classes are arranged hierarchically so as to be able to recover them in groups. They are all descendents of a single root class called "Failure", and it is at the level of this class that the behavior linked to the raising of exceptions is implemented.

```
    declaration-of-an-exception ::=
    exception exception-name inherits exception-name
```

All exceptions share identical behavior, that of the class "Failure". Here are some examples of exception classes:

```
exception NumericError inherits Failure;
exception Overflow inherits NumericError;
exception Underflow inherits NumericError;
```

The use of exceptions as a means to interrupt the normal execution of one program and then take control of another one depends on the programming language used to implement the methods. See the following chapter <u>"Defining the Software Components"</u> on page 32.

3. 5 Schemas

The purpose of a **schema** is to list persistent data types, which will be stored in files by the application. A schema groups together persistent packages. A persistent package is one, which contains at least one persistent class.

```
declaration-of-a-schema ::=
    schema <SchemaName>
    is
      {package <PackageName>; }
      {class <ClassName>; }
    end;
```

Example

```
schema Bicycle
    ---Purpose: Defines the Bicycle schema.
    is
    package FrameComponents;
package WheelComponents;
end;
```

NOTE
Note that it is unnecessary to specify all the dependencies of the packages. It is sufficient to specify the highest level ones. The others on which they depend are automatically supplied.

3. 6 Executables

The purpose of an **executable** is to make an executable program without a front-end. It can be used to test more than one package at a time. An executable is written in a .cdl file as a set of packages.

Example

```
definition-of-an-executable ::=
    executable <ExecutableName>
    is
{
    executable <ExecutablePart>
      [uses [<Identifier> as external]
       [{',' <Identifier> as external}]
       [<UnitName> as library]
       [{',' <UnitName> as library}]
       is
       {<FileName> [as C++|c|fortran|object];}
       end;
      }
    end;
```

Example

```
executable MyExecUnit
    ---Purpose:
    -- Describes the executable MyExecUnit
    is
    executable myexec
    -- the binary file
    uses
    Tcl_Lib as external
    is
    myexec;
-- the C++ file
    end;
-- several binaries can be specified in one .cdl file.
    executable myex2
    is
    myex2;
end;
end;
```

4. Defining the Software Components

4. 1 Behavior

The behavior of an object class is defined by a list of **methods,** which are either **functions** or **procedures**. Functions return an object, whereas procedures only communicate by passing arguments. In both cases, when the transmitted object is an instance manipulated by a handle, its identifier is passed. There are three categories of methods:

Object constructor   Creates an instance of the described class. A class will have one or more object constructors with various arguments or none.

Instance method   Operates on the instance which owns it.

Class method   Does not work on individual instances, only on the class itself.

4. 1. 1 Object Constructors

A constructor is a function, which allows the **creation of instances** of the class it describes.

Example

```
constructor-declaration ::=
Create [ simple-formal-part ] declaration-ofconstructed-
type
[ exception-declarations ]
simple-formal-part ::=
'(' initialization-parameter {';' initialization
parameter}')'
initialization-parameter ::=
identifier {',' identifier} ':' parameter-access datatype
[ '=' initial-value ]
parameter-access ::=
mutable | [ immutable ]
initial_value ::=
numeric-constant | literal-constant | named-constant
declaration-of-constructed-type ::=
returns [ mutable ] class-name
```

The name of the constructors is fixed: "Create". The object returned by a constructor is always of the type of the described class. If that type is manipulated by a handle, you *must* declare it as **mutable**, in which case the content of the instance it references is accessible for further modification.

For example, the constructor of the class "Point"

Example

```
Create (X, Y, Z : Real)
        returns mutable Point;
```

With the exception of the types predefined by the language, all types of initialization parameters *must* appear in the **uses** clause of the class of which the constructor is a member.

When an initialization parameter is of a type which is manipulated by a handle, an access right *must* be associated with it so as to express if the internal representation of the referenced object is modifiable (**mutable**) or not (**immutable**). The default option is **immutable**. For example, the constructor of the persistent class "Line".

Example

```
Create (P : mutable Point; D : mutable Direction)
        returns mutable Line;
```

In the above example "P" and "D" must be mutable because the constructor stores them in the internal representation of the created line, which is mutable itself. An alternative would be to accept immutable initialization parameters and then copy them into the constructor in a mutable form.

The parameters of a native type can have a default value: this is expressed by assigning a constant of the same type to the parameter concerned. Parameters, which have a default value, may not be present when the call to the constructor is made, in which case they take the value specified in the declaration. For this reason, they must all be grouped at the end of the list. For example, the constructor of the persistent class "Vector".

Example

```
Create (D : mutable Direction; M : Real = 1.0)
        returns mutable Vector;
```

A class can have many constructors (in this case, you say they are **overloaded**) provided that they differ in their syntax and that the presence of parameters having default values does not create ambiguities.

The restrictions on their use are expressed by a list of **exceptions** against which each constructor is protected.

Each class must have at least one constructor to be able to create objects of its type.

4. 1. 2 Instance Methods

An instance method is a function or procedure, which applies to any instance of the class, which describes it.

Example

```
declaration-of-an-instance-method ::=
    identifier formal-part-of-instance-method
    [ declaration-of-returned-type ]
    [ exception-declaration ]
    formal-part-of-instance-method ::=
    '(' me [':' passing-mode parameter-access ] {';'
parameter}')'
    parameter ::=
       identifier {',' identifier} ':' passing-mode
parameter-access
    data-type [ '=' initial-value ]
    passing-mode ::=
    [ in ] | out | in out
    parameter-access ::=
    mutable | [immutable]
    declaration-of-returned-type ::=
    returns return-access data-type
    return-access ::=
    mutable |[ immutable ]| any
```

The name **me** denotes the object to which the method is applied: you call this the "principal object" of the method. The passing mode expresses whether the direct content of the principal object or a parameter is either:

- read

- created and returned

- read then updated and returned by the method.

Remember that the direct content of an argument of a type which is manipulated by value contains the internal representation of the object itself. Thus, when the argument is of this type, **out** and **in out** mean that the content of the object will undergo a modification. When the method is a function (as is the case for constructors), all the arguments must be **in** (read). This is the default mode.

In case of an argument of a type manipulated by a handle, the direct content being an object identifier, the passing mode addresses itself to the handle, and no longer to the internal representation of the object, the modification of which is controlled by the access right. An argument of this type declared **mutable** may see its internal representation modified. If declared **immutable**, it is protected. When a parameter is both **in out** and **mutable**, the identifiers passed and returned denote two distinct modifiable objects.

When the returned object is manipulated by a handle it can be declared modifiable or not, or indeterminate (**any**). To return an object with an indeterminate access right means that the method transmits the identifier without changing its state and that the method has no right to alter the access right. This functionality is particularly useful in the case of collections; temporarily storing an object in a structure and unable to modify its state.

With the exception of the types predefined by the language, all types of parameters and returned objects, whether manipulated by a handle or by value, *must* appear in the **uses** clause of the class of which the method is a member.
As is the case for constructors, some parameters can have a default value, provided that they are of primitive or enumerated type. They are passed in the **in** mode, and they are found at the end of the list of arguments.

Overloading of instance methods and use of exceptions and post-conditions is allowed and respects the same rules than constructors.

Note the overloading of "Coord" in the following example of instance methods associated with the persistent class "Point":

Example

```
Coord (me; X, Y, Z : out Real);
---Purpose: returns the coordinates of <me>

Coord (me; i : Integer) returns Real;
---Purpose: returns the abscissa (<i>=1), the
     -- ordinate (<i>=2) or the value (<i>=3) of <me>

SetCoord (me : mutable; X, Y, Z : Real);
---Purpose: modifies the coordinates of <me>

Distance (me; P : Point) returns Real
---Purpose: returns the distance to a point
```

In all these cases, **me** is implicitly an object of type Point. Only "SetCoord" is able to modify the internal representation of a point.

4. 1. 3 Class Methods

A class method is a function or procedure relative to the class it describes, but does not apply to a particular instance of the class.

```
declaration-of-a-class-method ::=
identifier formal-part-of-class-method
[ declaration-of-returned-type ]
[ exception-declaration ]
formal-part-of-class-method ::=
'(' myclass {';' parameter}')'
```

The first parameter **myclass** indicates that the method does not apply to a previously created instance, but to the class itself. The rest of the syntax is identical to that of the instance methods. In particular, access rights (**mutable**, **immutable**, **any**) and the argument passing mode (**in**, **out**, **in out**) must remain unchanged. With the exception of the types predefined by the language, all types of parameters must appear in the **uses** clause of the class of which the method is a member. Overloading of class methods and the use of exceptions and post-conditions is allowed, and it follows the same rules as for constructors and instance methods.

Examples of class methods associated with the class "Real":

Example

```
First (myclass) returns Real;
---Purpose: returns lower limit of reals

Last (myclass) returns Real;
---Purpose: returns upper limit of reals
```

---

4. 1. 4 Package Methods

Package methods are methods which are members of a package. They are frequently used for library or application initialization, or for offering an application programming interface to the sources to the package. They are sometimes methods used for development purposes but which are not made available to final end-users of the package.

```
package-method ::=
identifier [simple-formal-part][returned-type-declaration]
[exception-declaration]
[is private]';'
```

---

4. 1. 5 Sensitivity to Overloading

When there is more than one method of a class, several methods share the same name but have different syntax, you say the method is overloaded.

In order that the methods can be considered distinct, they must differ either in the number of parameters, or one of their parameters must be of a different type. In particular, you *cannot* overload a method if you merely modify it as follows:

- The type of the returned object when the method behaves as a function
- The name or the mode of passing a parameter

  (**in**, **out**, or **in out**)
- The mutability of passed objects

  (**mutable**, **immutable**, **any**)
- Default value of a parameter.

4. 2 Internal Representation

Each object contains its own state in a private space in the memory. This state consists of a set of **fields**, which include or reference other objects.

Example

```
declaration-of-the-internal-representation-of-a-class
::=
fields field {field}
field ::=
identifier {',' identifier} ':' data-type ['['integer
{','integer}']']';'
```

A copy of all the defined fields exists locally in each instance of the class. This group of fields will be initialized by the class constructors when the object is instantiated.

Fields *must not* have the same name as any method of the class in which they appear. When the field type is followed by a list of integer constants between square brackets, the data will take the form of a multi-dimensional array containing objects of this type.

The following example shows two equivalent ways of describing three fields of the "Real" type:

Example

```
fields
x, y, z: Real;
coord: Real[3];
```

Depending on their type, Object fields have one of the two forms. When the field is of the "manipulated by handle" type, it corresponds to an identifier. In this case, the contents of the object can be shared by other objects or by a handle in a program. When the field is of a "manipulated by value" type, it contains the value of the object. In this case you say the object is **embedded**.

4. 3 Exceptions

Exceptions describe exceptional situations, which can arise during the execution of a method. With the raising of an exception, the normal course of program execution is interrupted. The actions carried out in response to this situation are called treatment of exception.

```
exception-treatment ::= raises exception-name {','
exception-name}
```

Each exception name corresponds to a class of exceptions previously defined as being susceptible to being raised by the method under which it appears. Exception classes must all appear in the **raises** clause of the class of which the method is a member. The class of exceptions is analogous to the class of objects described in this manual.

Take for example the method which returns the x, y, or z coordinate of a point.

Example

```
Coord (me; i : Integer) returns Real
---Purpose:
-- Returns the abscissa (<i>=1)
-- the ordinate (<i>=2)
-- or the value (<i>=3)
-- of <me>.
raises OutOfRange;
-- if <i> is not equal to 1, 2, or 3.
```

Instance methods are likely to raise certain exceptions called **systematic exceptions** which do not have to appear. They are:

NullObject   Raised when the principal object does not exist.

ImmutableObject   Raised when a method tries to modify an immutable principal object.

TypeMismatch   Raised if an argument typed by association is of an unsuitable type.

These exceptions are described in the Standard Package (System Toolkits).

4. 4 Inheritance

4. 4. 1 Overview

The notion of inheritance comes from a development strategy according to which you begin by modeling data in the most general fashion. Then you specialize it more and more so as to correspond to more and more precise cases.

For example, to develop a basic geometry, you can first of all consider the group of geometric objects, and then differentiate the points, vectors, and curves. You can specialize the latter into conic sections, and then decompose them into circles, ellipses, and hyperbolae. Then, the class of conics is considered as a sub-class of curves, and a super-class of circles.

A sub-class has at least the behavior of its super-classes. Thus, a circle could be viewed as a conic, a curve, or even as a geometric object. In each case, the applicable methods belong to the level where you view the class. In this case, you say that the sub-class inherits the behavior from its super-classes.

Example

```
declaration-of-a-sub-class ::=
class class-name
inherits class-name
[uses data-type {',' data-type}]
[raises exception-name {',' exception-name}]
```

```
is class-definition
end [class-name]';'
```

A class cannot inherit one of its descendent classes; nor can it inherit a native type. All the classes of a system can be described in a non-cyclic diagram called the **inheritance graph**.

The definition of a sub-class is identical to that of a simple class. Note that a super-class *must not* appear in the **uses** clause of the sub-class, even if it appears in the definition of the sub-class. The behavior of a sub-class includes as a minimum all instance methods and protected methods of its super-classes.

NOTE
Note that constructors and class methods are never inherited.

4. 4. 2 Redefining methods

Certain inherited methods can be redefined.

Example

```
declaration-of-a-redefined-method ::=
identifier formal-part-of-instance-method [returnedtype-
declaration]
[declaration-of-exceptions]
  is redefined [visibility]';'
```

A redefined method must conform to the syntax described in the super-class where it appears. The exceptions contained in the super-class can be renewed, and others may be added as long as they inherit from an ancestor class.

The **redefined** attribute can be applied neither to a constructor, nor to a class method, since neither of them can be inherited. If the redefined method is private or protected, the visibility must be exactly repeated in the redefinition. For further details on visibility, refer to "Visibility" on page 48.

Example

```
SquareDistance (me; P : Point) returns Real
is redefined private;
```

With regards to the internal representation, all fields defined in the super-classes are, by default, inherited, but they can also be redefined.

4. 4. 3 Non-redefinable methods

Instance methods, which are declared virtual are redefinable in descendent classes, and you can force this redefinition by making a method **deferred**. For more details, see the next section.

Example

```
declaration-of-a-non-redefinable-method ::=
identifier formal-part-of-instance-method [returnedtype-
declaration]
[declaration-of-exceptions]
 is virtual [visibility]';'
```

All methods are static by default. To enable redefinition in all the child classes, add "**is virtual;**" when declaring the method.

You must also be able to forbid redefinition. A redefinable method can become non-redefinable if you declare: **is redefined static**.

4. 4. 4 Deferred Classes and Methods

The presence of certain classes in the inheritance graph can be justified purely by their ability to force certain behavior on other classes, in other words, to make other classes provide various services.

The CDL language allows you to describe a class, which introduces methods without implementing them, so as to force its descendent classes to define them. These are called **deferred classes**; the non-implemented methods are also termed **deferred methods**.

Example

```
declaration-of-a-deferred-class ::=
deferred class class-name
 [inherits class-name
[uses data-type {',' data-type}]
[raises exception-name {',' exception-name}]
is class-definition
end [class-name]';'
declaration-of-a-deferred-method ::=
identifier formal-part-of-instance-method [returnedtype-
declaration]
[declaration-of-exceptions]
is deferred [visibility]';'
```

Only instance methods can be deferred.

It is sufficient for a class to contain one deferred method for it to be a deferred class. It can contain any number of deferred methods (or none).

A deferred class may still have an internal representation but one or more **non-protected** constructors would be necessary to initialize them. The constructors must

be visible in the sub-classes.

The constructors of a deferred class are called **Initialize** (not **Create**). They are **protected** by default, and do not return any object. You cannot create an object of a deferred class type.
For example, consider the class "Point", and its declaration as deferred.

Example

```
deferred class Point inherits Geometry is
Initialize;
---Purpose: Initializes the point.
Coord (me; X, Y, Z : out Real)
---Purpose: Returns the coordinates
is deferred;
SetCoord (me : mutable; X, Y, Z : Real)
---Purpose: Modifies the coordinates
is deferred;
Distance (me; P : Point) returns Real;
---Purpose: Returns the distance from the point P
end Point;
```

Notice that the function "Distance" is not deferred. Although this class contains no representation, this method is programmable by calling "Coord".

In a sub-class of a deferred class, all deferred methods, which have been inherited, *must* be implemented, then redeclared (the attribute **redefined** is useless for this purpose), unless the sub-class is itself deferred.

A non-deferred method can be redefined as a deferred one, in which case it will be declared as follows: **is redefined deferred**.

The notion of deferred class is very useful. The advantage of introducing it, as was previously shown in the deferred class "Point", is that the corresponding resources will be available even before being implemented. Later, you can add different representations to Point (for example, spherical or Cartesian coordinates) without having to modify client programs.

Thanks to the possibility of redefining methods, this approach does not have any negative impact on performance: a method implemented at the level of a deferred class can be reprogrammed in one of its sub-classes while taking into account the data representation.

4. 4. 5 Declaration by Association

At the heart of a class hierarchy, object identifiers are compatible in the ascendant sense. Since the Conic class is descended from the Curve class, an identifier of type Curve can reference an object of type Conic (remember that the behavior of Curve is applicable to Conic). In other words, you can assign a reference to a Conic to an identifier of type Curve, but *not* vice versa.

For example, once the classes have been compiled you could write a C++ test program in which you instantiate a Conic but reference it with a handle to a Curve:

Example

```
Handle(Curve) c = new Conic
```

This same rule applies to parameters of methods; that is to say, you can call a method with identifiers corresponding to a sub-type of that specified in its declaration. To illustrate this, you go back to the "Distance" method of the "Point" class:

Example

```
Distance (me; P : point) returns Real;
```

Conforming to the rule of type compatibility, you could make a call to the method "Distance" with reference to an object from a class descended from "Point". Consequently, if "SphericPoint" is a sub-class of "Point" and therefore inherits this method, it will be possible to calculate the distance between two "SphericPoint", or between a "SphericPoint" and a "Point", without having to redefine the method.

On the other hand, sometimes you may want to force two parameters to be exactly of the same type, and thus not apply the rule of type compatibility. To do this, you need to associate the type of the concerned parameters in the method declaration.

Example

```
association-typing ::=
like associated-parameter
associated-parameter ::=
me | identifier
```

NOTE
Note that identifier is the name of a parameter, which appears
first in the formal part of the declaration of the method.

You can use this technique, which consists in declaring by association, to declare a method that will exchange the content of two objects, or a method, which copies another object:

Example

```
Swap (me : mutable; With : mutable like me);
DeepCopy (me) returns mutable like me;
```

Make sure *not* to write the Swap method as in the syntax below:

Example

```
Swap (me : mutable; With : mutable Point);
```

In this case **me** may be a CartesianPoint or a SphericalPoint, while "With" can only be a Point.

4. 4. 6 Redefinition of Fields

The creation of a hierarchy of classes should be viewed as a means to specialize their behavior, (e.g. a circle is more specialized than a conic section). The more you specialize the object classes, the more it is justified to call into question the inherited fields in order to obtain greater optimization. So, in the description of the internal representation of a sub-class, it is possible not to inherit all of the fields of the super-classes. You then say the fields have been redefined.

Example

```
redefinition-of-the-representation-of-a-class ::=
redefined redefinition-of-a-field {',' redefinition-of-a-
field}','
redefinition-of-a-field ::=
[field-name] from [class] class-name
```

Redefinition of fields can *only* be done in classes manipulated by a handle.

This declaration appears at the beginning of the definition of the internal representation of the sub-class, which breaks the field inheritance. The non-inherited fields are all those which come from the class specified behind the rubric **from**.

4. 5 Genericity

4. 5. 1 Overview

Inheritance is a powerful mechanism for extending a system, but it does not always allow you to avoid code duplication, particularly in the case where two classes differ only in the type of objects they manipulate (you certainly encounter this phenomenon in all basic structures). In such cases, it is convenient to send arbitrary parameters representing types to a class. Such a class is called a **generic class**. Its parameters are the generic types of the class.

Generic classes are implemented in two steps. You first declare the generic class to establish the model, and then instantiate this class by giving information about the generic types.

4. 5. 2 Declaration of a Generic Class

The syntax is as follows:

Example

```
declaration-of-a-generic-class ::=
[deferred] generic class class-name '('generic-type
{','generic-type}')'
[inherits class-name
[uses data-type {',' data-type}]
[raises exception-name {',' exception-name}]
is class-definition
end [class-name]';'
generic-type ::=
identifier as type-constraint
type-constraint ::=
any | class-name ['('data-type {','data-type}')']
```

The names of generic types become new types, which are usable in the definition of a class, both in its behavior (methods) and its representation (fields). The generic type is only visible inside the generic class introducing it. As a result, it is possible to have another generic class using the same generic type within the same package. When you specify the type constraint under the form of a class name, you impose a minimum set of behavior on the manipulated object.

This shows that the generic type has as a minimum the services defined in the class. This can be any kind of a previously defined class, including another generic class, in which case you state exactly with what types they are instantiated.

When the generic type is constrained by the attribute **any**, the generic class is intended to be used for any type at all, and thus corresponds to classes whether manipulated by a handle or by value.

No class can inherit from a generic class.

A generic class can be a deferred class. A generic class can also accept a deferred class as its argument. In both these cases any class instantiated from it will also be deferred. The resulting class can then be inherited by another class.

Below is a partial example of a generic class: a persistent singly linked list.

Example

```
generic class SingleList (Item as Storable)
    inherits Persistent
      raises NoSuchObject
      is
      Create returns mutable SingleList;
    ---Purpose: Creates an empty list
      IsEmpty (me) returns Boolean;
    ---Purpose: Returns true if the list <me> is empty
            SwapTail (me : mutable; S : in out mutable
SingleList)
      ---Purpose: Exchanges the tail of list <me> with <S>
            -- Exception NoSuchObject raised when <me> is
empty
      raises NoSuchObject;
    Value (me) returns Item
    ---Purpose: Returns first element of the list <me>
      -- Exception NoSuchObject raised when <me> is
empty
      raises NoSuchObject;
    Tail (me) returns mutable SingleList
      ---Purpose: Returns the tail of the list <me>
            -- Exception NoSuchObject raised when <me> is
empty
      raises NoSuchObject;
    fields
      Data : Item;
    Next : SingleList;
    end SingleList;
```

Even though no object of the type "SingleList" IS created, the class contains a constructor. This class constitutes a model, which will be recopied at instantiation time to create a new class which will generate objects. The constructor will then be required.

Example

```
generic class Sequence(Item as any, Node as
SingleList(Item))
inherits Object
. . .
end Sequence
```

In the above example, there are two generic types: Item & Node. The first imposes no restriction. The second must at least have available the services of the class SingleList instantiated with the type with which Sequence will itself be instantiated.

In the **incomplete declaration of a generic class**, the keyword **generic** must appear.

Example

```
generic class SingleList;
generic class Sequence;
```

4. 5. 3 Instantiation of a Generic Class

The syntax is as follows:

Example

```
instantiation-of-a-generic-class ::=
    [deferred] class class-name
    instantiates class-name '('data-type {','
                                    data-type}')';'
```

Instantiation is said to be **static**. In other words, it must take place before any use can be made of the type of the instantiated class. Each data type is associated term by term with those declared at the definition of the generic class. These latter ones, when they are not of the type **any**, restrict instantiation to those classes, which have a behavior at least equal to that of the class specified in the type constraint, including constructors. Note that this is not guaranteed by inheritance itself.

For example, let's instantiate the class "Sequence" for the type "Point":

Example

```
class SingleListOfPoint instantiates SingleList(Point);
class Sequence instantiates
        Sequence(Point,SingleListOfPoint);
```

The instantiation of a generic deferred class is a deferred class (the **deferred** attribute must be present during instantiation). An instantiated class cannot be declared in an incomplete fashion.

4. 5. 4 Nested Generic Classes

It often happens that many classes are linked by a common generic type. This is the case when a base structure provides an iterator, for example, in the class "Graph". A graph is made up of arcs, which join together the nodes, which reference objects of any type. This type is generic both for the graph and for the node. In this context, it is necessary to make sure that the group of linked generic classes is indeed instantiated for the same type of object. So as to group the instantiation, CDL allows the declaration of certain classes to be nested.

Example

```
declaration-of-a-generic-class ::=
    [deferred] generic class class-name '('generic-
        type{','generic-type}')'
    [inherits class-name {',' class-name}]
    [uses data-type {',' data-type}]
    [raises exception-name {',' exception-name}]
    [{[visibility] class-declaration}]
    is class-definition
end [class-name]';'
    class-declaration ::=
    incomplete-declaration-of-a-class |
    declaration-of-a-non-generic-class |
```

```
instantiation-of-a-generic-class
```

Nested classes, even though they are described as non-generic classes, are generic by construction, being inside the class of which they are a part. As a consequence, the generic types introduced by the **encompassing class** can be used in the definition of the nested class. This is true even if the generic type is only used in a nested class. The generic types still *must* appear as an argument of the encompassing class. All other types used by a nested class *must* appear in its **uses** or **raises** clauses, just as if it were an independent class.

Nested classes are, by default, **public**. In other words, they can be used by the clients of the encompassing class. On the other hand, when one of the nested classes is declared **private** or **protected**, this class *must not* appear in any of the public methods of the other classes. It cannot be used in a protected field because then it could be used in a sub-class, which implies it would not be private.

The following example shows how to write the Set class with its iterator.

Example

```
generic class Set (Item as Storable)
     inherits Persistent
     private class Node instantiates SingleList (Item);
     class Iterator
        uses Set, Node
        raises NoSuchObject, NoMoreObject
        is
        Create (S : Set) returns mutable Iterator;
     ---Purpose: Creates an iterator on the group <S>
        More (me) returns Boolean;
     ---Purpose: Returns true if there are still elements
           -- to explore
        Next (me) raises NoMoreObject;
     ---Purpose: Passes to the following element
        Value (me) returns any Item raises NoSuchObject;
     ---Purpose: Returns the current element
        fields
        Current : Node;
     end Iterator;
     is
        Create returns mutable Set;
     ---Purpose: Creates an empty group
        IsEmpty (me) returns Boolean;
     ---Purpose: Returns true if the group is empty
        Add (me : mutable; T : Item);
     ---Purpose: Adds an item to the group <me>
        Remove (me : mutable; T : item) raises
           NoSuchObject;
     ---Purpose: Removes an item from the group <me>
        etc.
        fields
        Head : Node;
     end Set;
```

Note that in their fields, both "Set" and "Iterator" are clients of another class, "Node". This last can be effectively declared **private** for it only appears in fields which are themselves private.

The instantiation of a generic class containing nested classes remains unchanged. The same declaration is used to instantiate the encompassing class and the nested classes. These latter will have their name suffixed by the name supplied at instantiation, separated by "Of". For example, you instantiate the class "Set" described above for the type "Point" as follows:

Example

```
class SetOfPoint instantiates Set(Point);
```

In doing so, you implicitly describe the classes "NodeOfSetOfPoint" and "IteratorOfSetOfPoint", which are respectively the result of the concatenation of "Node" and "Iterator" with "Of" then "SetOfPoint".

Note that in the incomplete declaration of an encompassing class, all the names of the nested classes *must* appear behind that of the encompassing class.

```
incomplete-declaration-of-a-generic-class ::=
[deferred] generic class-name {',' class-name};
```

For example, an incomplete declaration of the above class "Set" would be as in the example below:

Example

```
generic class Set, Node, Iterator;
```

Only the encompassing class can be deferred. In the above example only the class "Set" can be deferred.

4. 6 Visibility

4. 6. 1 Overview

A field, method, class, or package method is only available for use if it is **visible**.
Each of these components has a default visibility, which can be explicitly modified during class or package declaration. The three possible states of visibility are:

- Public
- Private
- Protected

4. 6. 2 Visibility of Fields

A field is **private**. It can never be public - this would destroy the whole concept of data encapsulation. The attribute **private** is redundant when it is applied to a field. This means that a field is only visible to methods within its own class.

A field can be declared **protected** which means that it becomes visible in subclasses of its own class. Its contents can be modified by methods in subclasses.

```
field ::=
identifier {',' identifier} ':' data-type
['['integer{','integer}']']
[is protected]';'
```

Example

```
fields
    Phi, Delta, Gamma : AngularMomenta [3]
    is protected ;
```

4. 6. 3 Visibility of Methods

Methods act on fields. Only methods belonging to a class can act on the fields of the class; this stems from the principle of object encapsulation. Methods can be characterized in three ways: by default, methods are **public**. Methods can be declared **private** or **protected** to restrict their usage.

Public methods	Are the default and are generally the most common. They describe the behavior of a class or a package, and they are callable by any part of a program.
Private methods	Exist only for the internal structuring of their class or their package. Private class methods can only be called by methods belonging to the same class. Private package methods can only be called by all methods belonging to the same package and its classes.
Protected methods	Are private methods, which are also callable from the interior of descendent classes.

If you want to restrict the usage of a method, you associate with it a visibility as follows :

```
-- declaration-of-the-visibility ::=
is visibility
visibility ::= private | protected
```

The declaration of the visibility of a method appears at the end of its definition, before the final semi-colon. The attribute **private** indicates that the method will only be visible to the behavior of the class of which the method is a member; **protected** will propagate the visibility among the sub-classes of this class.

For example, add to the class "Line" an internal method allowing the calculation of the perpendicular distance to the power of two, from the line to a point.

Example

```
SquareDistance (me; P : Point) returns Real
is private;
```

4. 6. 4 Visibility of Classes, Exceptions, & Enumerations

The visibility of a class is the facility to be able to use this class in the definition of another class. The visibility of a class extends from the beginning of its declaration up to the end of the package in which it appears. You have seen that the keyword **uses** allows extension of this visibility to other packages.

As was explained in the section on "Name Space", any ambiguity, which arises from having two classes with the same name coming from different packages, is dealt with by the use of the keyword **from**.

A class declared **private** is only available within its own package.

4. 6. 5 Friend Classes & Methods

In certain cases, methods need to have direct access to the private or protected parts of classes of which they are clients. Such a method is called a **friend** of the class, which is accessed. For example, you declare a method to be a friend when a service can only be obtained via the use of another non-descendent class, or perhaps when this will help to improve performance.

Classes can also be declared friends of other classes. In this case all the methods of the friend class will have access to the fields and methods of the host class. The right is **not reciprocal**.

Friend classes or methods are declared inside the class, which reveals its private and protected data or methods to them. This helps in managing the continuing evolution of a class, helping to recognize and to avoid the creation of side effects.

Example

```
declaration-of-friends ::=
friends friend {','friend}
    friend ::=
    identifier from [class] class-name [formal-part] |
Defining the Software Components 67
identifier from [package] package-name [formal-part] |
    class] class-name
    formal-part ::=
    simple-formal-part |
    formal-part-of-instance-method |
    formal-part-of-class-method
```

The formal part *must* be presented if the method contains one; thus this can be overloaded without necessarily propagating the friend relationship among its homonyms. The keyword **class** allows you to avoid certain ambiguities. For example, it removes any confusion between "method M from class C" and "method M from package P".

As an example, take a method, which calculates the perpendicular distance between a line and a point. Suppose this method needs to access the fields of the point. In the class "Point" you would write:

Example

```
friends Distance from Line (me; P : Point)
```

A method can be a friend to many classes. The class to which the method belongs does *not* need to appear in the **uses** clause of other classes of which it is a friend. When the methods of a class are all friends of another class, you can establish the friendship at the level of the class.

	Public	Private	Protected
Field	Does not exist	*Default* Visible to methods in its own class and in friend classes	Visible to methods in its own class, sub-classes and friend classes
Method	*Default* Callable anywhere	Callable by methods in its own class and in friend classes	Callable by methods in its own class, sub-classes and in friend classes
Class	*Default* Visible everywhere with the use of the **from** rubric	Visible to classes in its own package	Does not exist
Package Method	*Default* Callable everywhere with the use of the **from** rubric	Visible to classes in its own package	Does not exist
Nested Class	*Default* Visible to the clients of the encompassing class	Visible to the encompassing class and other classes nested in the encompassing class	Does not exist

Figure 9. Visibility of various components

Appendix A. Syntax Summary

This summary of the CDL syntax will aid in the comprehension of the language, but does *not* constitute an exact definition thereof. In particular, the grammar described here accepts a super-set of CDL constructors semantically validated.

(1) capital ::=

'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' |
'I' | 'J' | 'K' | 'L' | 'M' | 'N' |
'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' |
'W' | 'X' | 'Y' | 'Z'

(2) non-capital ::=

'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' |
'i' | 'j' | 'k' | 'l' | 'm' | 'n' |
'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' |
'w' | 'x' | 'y' | 'z'

(3) digit ::=

'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' |
'8' | '9'

(4) underscore ::=

'_'

(5) special character ::=

' ' | '!' | '"' | '#' | '$' | '%' | '&' | ''' |
'(' | ')' | '*' | '+' | ',' | '-' |
'.' | '/' | ':' | ';' | '<' | '=' | '>' | '?' |
'@' | '[' | '\' | ']' | '^' | '`' |
'{' | '|' | '}' | '~'

(6) printable character::=

            capitals | non-capitals | digits | underscore |
            special characters

(7) letter ::=

            capital | non-capital

(8) alphanumeric ::=

            letter | digit

(9) identifier ::=

            letter{[underscore]alphanumeric}

(10) integer ::=

            digit{digit}

(11) exponent ::=

            'E'['+']integer | 'E-'integer

(12) numeric-constant ::=

            ['+']integer '.' integer[exponent] | '-'integer
            '.' integer[exponent]

(13) literal-constant ::=
         '''printable character''' | '~'{printable character}'~'

(14) package-name ::=
         identifier

(15) enumeration-name ::=
         identifier [**from** package-name]

(16) class-name ::=
         identifier [**from** package-name]

(17) exception-name ::=
         identifier [**from** package-name]

(18) constructor-name ::=
         'Create' | 'Initialize'

(19) primitive-type ::=
         'Boolean' | 'Character' | 'Integer' | 'Real'

(20) data-type ::=
         enumeration-name | class-name | exception-name
         | primitive-type

(21) passed-type ::=
         data-type | **like me** | **like** identifier

(22) passing-mode ::=
         [**in**] | **out** | **in out**

(23) parameter-access ::=
         mutable | [**immutable**]

(23A) return-access ::=
         mutable | [**immutable**]| **any**

(24) value ::=
         numeric-constant | literal-constant |
         identifier

(25) parameter ::=
         identifier {',' identifier} ':' passing-mode
         access-right passed-type ['=' value]

(26) simple-formal-part ::=
         '('parameter {';' parameter}')'

(27) formal-part-of-instance-method ::=
         '(**me** [':' passing-mode access-right] {';'
         parameter}')'

(28) formal-part-of-class-method ::=
         '(**myclass** {';' parameter}')'

(29) visibility ::=
         private | **protected**

(30) redefinition ::=

        static | **deferred**

(31) definition-level ::=

        `redefinition` | **redefined** [`redefinition`]

(32) declaration-of-constructed-type ::=

        returns [**mutable**] `class-name`

(33) declaration-of-returned-type ::=

        returns `return-access passed-type`

(34) declaration-of-errors ::=

        raises `exception-name {',' exception-name}`

(35) declaration-of-visibility ::=

        is `visibility`

(36) declaration-of-attributes-of-instance-method ::=

        is `visibility` | **is** `definition-of-level`
        `[visibility]`

(37) constructor ::=

        `constructor-name [simple-formal-part]`
        `[declaration-of-constructed-type]`
        `[declaration-of-errors]`
        `[declaration-of-visibility]';'`

(38) instance-method ::=

        `identifier formal-part-of-instance-method`
        `[declaration of returned type]`
        `[declaration-of-errors]`
        `[declaration-of-attributes-of-`
        `instancemethod]';'`

(39) class-method ::=

        `identifier formal-part-of-the-class-method`
        `[declaration of returned type]`
        `[declaration-of-errors]`
        `[declaration-of-visibility]';'`

(40) package-method ::=

        `identifier [simple-formal-part]`
        `[declaration-of-returned-type]`
        `[declaration-of-errors]`
        [is private]`';'`

(41) member-method ::=

        `constructor` | `instance-method` | `class-method`

(42) formal-part ::=

        `simple-formal-part` |
        `formal-part-of-instance-method`|
        `formal-part-of-class-method`

(43) friend ::=

        `identifier` **from** [**class**] `class-name [formal-part]`
        |
        `identifier` **from** [**package**] `package-name [formal-part]` |

[**class**] class-name

(44) field ::=

identifier {',' identifier} ':' data-type
['['integer {',' integer}']']
[**is protected**]';'

45) redefinition-of-field ::=

[field-name] **from** [**class**] class-name

(46) declaration-of-fields ::=

fields [**redefined** redefinition-of-field {','
redefinition-of-field}';']
field {field}

(47) declaration-of-an-alias::=

[**private**] **alias** <class-name1> **is** <class-name2> [**from**
<package-name>]

(48) declaration-of-friends ::=

friends friend {',' friend}

(49) class-definition ::=

[{member-method}]
[declaration-of-fields]
[declaration-of-friends]

(50) declaration-of-an-exception ::=

exception exception-name
inherits exception-name

(51) declaration-of-an-enumeration ::=

enumeration enumeration-name
is identifier {',' identifier}
[**end** [enumeration-name]]';'

(52) incomplete-declaration-of-a-non-generic-class ::=

[**deferred**] **class** class-name';'

(53) incomplete-declaration-of-a-generic-class ::=

[**deferred**] **generic class** class-name {',' class-
name}';'

(54) declaration-of-a-non-generic-class ::=

[**deferred**] **class** class-name
[**inherits** class-name
[**uses** data-type {',' data-type}]
[**raises** exception-name {',' exception-name}]
is definition-of-a-class
end [class-name]';'

(55) type-constraint ::=

any | class-name ['('data-type {',' data-
type}')']

(56) generic-type ::=

identifier **as** type-constraint

(57) declaration-of-a-generic-class ::=
      [**deferred**] **generic class** class-name '('generic-type
      {',' generic-type}')'
      [**inherits** class-name
      [**uses** data-type {',' data-type}]
      [**raises** exception-name {',' exception-name}]
      [{[visibility] declaration-of-a-class}]
       is class-definition
      end [class-name]';'

(58) instantiation-of-a-generic-class::=
      [**deferred**] **class** class-name
      instantiates class-name '('data-type
      {',' data-type}');'

(59) declaration-of-a-class::=
      incomplete-declaration-of-a-non-generic-class
      |
      incomplete-declaration-of-a-generic-class |
      declaration-of-a-non-generic-class |
      declaration-of-a-generic-class |
      instantiation-of-a-generic-class

(60) type-declaration ::=
      [**private**] declaration-of-an-enumeration |
      [**private**] class-declaration |
      declaration-of-an-exception

(61) package-definition ::=
      [{type-declaration}]
      [{package-method}]

(62) package-declaration ::=
      package package-name
      [**uses** package-name {',' package-name}]
       is package-definition
      end [package-name]';'

(63) executable-declaration ::=
   executable executable-name
      is
      {
   executable executable-part
      [**uses** [identifier **as external**]
        [{',' identifier **as external**}]
        [unit-name **as library**]
        [{',' unit-name **as library**}]
    is
        {file-name [**as** C++|c|fortran|object]; }
      end ';'
     }
    end ';'

(64) schema-declaration ::=
   schema schema-name
   is
      [{**package** package-name ';' }]
      [{**class** class-name ';' }]
      end ';'

Appendix B.

5. 1 Comparison of CDL & C++ Syntax for Data Types manipulated by Handle and by Value

CDL	Handle	Value
Permanent	Persistent	Storable
Temporary	Transient	Any
Reading	Immutable	In
Writing	Mutable	Out
Read/Write	Mutable	In out
Return	Not specified: any	Without copy: --C++ return const&

C++	Handle	Value
C++ Declaration	Handle(PGeom_Point) p1;	gp_Pnt p2;
C++ Constructor	p1 = new PGeom_Point (p2);	p2(0.,0.,0.);
C++ Method	x = p1 -> XCoord();	x = p2.XCoord();