Data Abstraction Mechanisms in Object-Oriented Programming

Haitao Li

School of Computer Science
McGill University, Montreal

July 1993

A Thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements of the degree of Master of Science

To my wife, Xiaohong

Acknowledgements

I would like to express sincere gratitude and appreciation to my thesis supervisor Dr. Hafedh Mili, for leading me to the field of object-oriented programming and software reuse, offering me extensive guidance throughout my graduate studies, and supporting me financially to complete my degree.

I am also grateful to Prof. Tim Merrett, my co-supervisor, for giving me every help during my studying at McGill.

Special thanks to my parents and my wife for moral support and encouragement to fulfill my studies in Canada.

Abstract

Existing OO modeling methodologies prescribe notations, processes, and guidelines that, if followed, ensure that analysis-leve' OO model reflect application semantics. As we move into design, implementation-level considerations may distort analysis-level models, and the transition is seamless no more. In this thesis, we describe data definition facilities in *SoftClass* an experimental CASE tool for software reuse– that aim at maintaining the integrity of application data models throughout the development lifecycle, while maximizing opportunities for code reuse. In SoftClass, analysis-level data models describe application-semantics and are organized in an inheritance hierarchy based on shared application-semantics. At the design-level, we maintain two kinds of data models: 1) generic data structures, used as implementation templates for analysis-level models, and organized along "implementation inheritance" hierarchies, and 2) realizations of analysis-level application models, which consist of mapping an analysis-level model to a generic data structure. Design-level representations of application objects may be seen as belonging to two independent hierarchies, and we show how each hierarchy offers some opportunities for reuse. We show how data abstraction supports a high-level program design language that is both easy-to-use and that supports some design validation. We conclude by outlining directions for further research.

Résumé

Les méthodologies de modélisation orientée-objet stipulent que les modèles objet du niveau analyse reflètent la sémantique du domaine d'applications. Au moment de passer à la *conception*, des considérations d'implantation telles que la réutilisation de code et la performance, peuvent entrer en jeu et modifier la structure d'héritage. Ceci a pour effet de nuire à la clarté conceptuelle des modèles, et de créer une fissure entre l'analyse et la conception; reproche que l'on fait souvent aux méthodologies de développement traditionelles. Dans ce mémoire, nous proposons des mécanismes de définition de données aux niveaux analyse et conception, qui permettent de séparer, conceptuellement, et dans les faits, l' *héritage sémantique*, implicite dans le domaine d'applications, de l'*héritage d'implantation*, permettant de réutiliser les structures d'implantation telles la représentation des données et les algorithmes. Ce travail s'est effectué dans le contexte d'un outil CASE expérimental appelé *SoftClass*. Dans SoftClass, les modèles objet du niveau analyse représentent la sémantique du domaine d'applications. La représentation des objets du domaine d'applications au niveau conception est faite en projetant les modèles correspondants du niveau analyse sur des *structures de données génériques*. Ces structures sont elles mêmes organisées selon une hiérarchie d'implantation. Nous montrerons que cette représentation des données offre plus de flexibilité au niveau de la conception, tout en assurant le maximum de réutilisation possible. Cette représentation sert aussi de base à un langage de conception de programmes ("pseudo-code") qui est à la fois facile d'usage et qui supporte certaines validations. Nous conclurons le mémoire en soulignant quelques directions prometteuses de recherche.

Contents

List of Figures

Chapter 1

Introduction

Software reuse has been recognized as an effective solution to the software crisis, as far as improving software productivity and quality [Gog84, Cox90, Mey87]. However, software reuse is still far from common. Some of the reasons are nontechnical but managerial/organizational problem, while the main obstacles are still technical [Mey87].

Several tenets of object-orientation make OO software more reusable: 1) *information hiding*, which shields clients (objects, modules) from implementation changes in servers, 2) *genericity* and *overloading*, which parameterize functionality by abstracting out some type-dependencies, and 3) *inheritance*, which provides a conceptual framework for reusing software components.

Most object-oriented development methodologies make the distinction between object-oriented analysis and design. The purpose of OO analysis is to model the *application domain* in terms of objects and patterns of activities/interactions between objects. Existing OO development methodologies (e.g. [Coa91, Rum91, Wir90]) prescribe that design builds on the basic class structure identified at the analysis level – one aspect of the much vaunted seamless transition – by adding high level (control) and low level (utility) application-independent classes. Methodologies recognize that in some cases, some reconstructing of the basic application classes may be warranted to accommodate some implementation-level concerns such as performance and code reuse. A number of researchers have observed that existing class hierarchies at the design or the code level do not always "make sense" (see [Coo92, Cox90]). In [Coo92], Cook thoroughly studied Smalltalk-80's collection classes and found a number of discrepancies between the protocols that classes implement and their place in the code class hierarchy. In [Cox90], Cox studied a commercial class library and dwelt on the extent to which the place of some classes in the hierarchy did not make sense. Both authors explained the discrepancies by the fact that a reuser/user of a class library is a *client* who approaches the class library more from a requirements (analysis-level) point of view. In other words, the analysis-level inheritance hierarchies that carry only application-semantics may be distorted or reconstructed when pure "computation-semantics" (the data structure representations of analysis-level objects) are taken into account at the design and the implementation level.

In the traditional object-oriented programming, the *programming language level* inheritance requires that subclasses reuse superclasses' *data representations*, meaning that the common data members of subclasses should have the same data representations as that of superclasses. Therefore, it precludes partial reuse of superclasses using different data representations and implementations. When we explore high level functions that usually depend only on the application, or that depend slightly on the data structures, we find that such functions (operations) can be and should be reused despite their different underlying data representations [Mil93]. For example, for **Adult** objects, we may want **Adult** to have the **Get_Num_Of_Sons (Adult)** operation that may look like:

```
Get_Num_Of_Sons (self: in Person, Num: out INTEGER)
  {
    local each_child : Person;
    local counter : INTEGER;

    counter = 0;
    For_Each (each_child, Get_Children (self))
      {
        if  (Is_Male (each_child))  then
          counter = counter + 1;
      };
     Num = counter;
  }
```

No matter how **Adult** is represented, this operation can be reused by different implementations of **Adult**, if some discipline is followed in accessing **Adult**'s data components.

The language level inheritance hierarchy, as an *implementation hierarchy*, is organized according to implementation-related information. The application-dependent information is mingled with the specifics of the data structures chosen to represent the application objects. We feel that the two orthogonal dimensions – the application-semantics of analysis-level concerns and the pure data structures representations of design-level concerns should be kept separately during the design and implementation, with each keeping their own inheritance hierarchies.

The motivation of this thesis is to build a representation of objects that distinguishes between application inheritance hierarchies and implementation inheritance hierarchies so as to reuse objects with the same application-semantics even if they have different underlying data structures.

Our approach may be summarized as follows:

1. Extend parameterized class concept by introducing **name parameters** which are formal parameters for field names. With both **name parameters** and **type parameters** together, traditional parameterized classes become fully parameterized classes. We call them *Generic Data Structures* or GDSs. GDSs are design level classes and support only computation-related operations such as accessors and iterators. GDSs are organized in an inheritance hierarchy based on common field types and operations, i.e. based on shared *implementation* characteristics.

2. Distinguish data objects at the analysis level (*Application Objects* or AOs) from their design level (*Application Data Structures* or ADSs). The latter are implementation of the former using GDSs. Instead of having one single design-level class hierarchy which mixes application-semantics and implementation-specifics, we maintain a design-level hierarchy of the GDSs based on shared implementation characteristics and an analysis-level hierarchy of the AOs based on shared application-specific external behaviour. The AO hierarchy carries only application-semantics; GDS hierarchy focuses only on computation-semantics. In consequence, the ADSs can be seen as belonging to the two hierarchies. They inherit application-semantics from AOs' hierarchy and inherit data representations and other computation-semantics from GDSs' hierarchy.

3. Combine the data abstraction mechanisms such as overloading and information hiding along with genericity and inheritance which are already mentioned, to support class design at the design level.

The benefits from our approach are:

- Design level application-dependent operations can be reused even if the underlying implementations are different.

  1) It may be desired to have multiple implementations for the same analysis-level class for different performance requirements. Application-dependent operations of one implementation can be reused by another implementation using a different data representation.

  2) When a subclass at analysis level is implemented using different data representations from its superclass(es) at the design level, it has no inheritance relationship with its superclass(es), but it *can* reuse design level application-dependent operations from its superclasses.

- Computation-related operations can be reused by application-specific classes.

Basic operations such as accessors/selectors, constructors/destructors, iterators, can be obtained from generic data structures, where the basic functions are in a parametric fashion. The application-specific classes can reuse these functions by instantiating formal **name parameters** and **type parameters**.

This work was inspired and conducted in the *SoftClass* project which is carried out at the University of Quebec at Montreal. SoftClass is an experimental OO CASE tool for software reuse. Responsible for designing a representation of data objects and code representation at the detailed design level, the author implemented the data object definition/classification/transformation tools and the PDL (Program Design Language – for detailed design) compiler for SoftClass, using the approach which is summarized above and will be presented in details in this thesis.

This thesis can be divided into two parts: the first part (Chapter 2 to Chapter 4) presents the literature review that is related to our approach; the second part (Chapter 5 to Chapter 7) is an elaboration of our approach.

Chapter 2 provides a historical review of programming and shows the major developments that led to object-oriented programming, and discusses in detail the concepts and the promising features of object-oriented programming.

In order to explain how the object-oriented models are applied to software development cycle, in particular, how the seamless transformation from analysis level to design level may be distorted and damaged because of unavoidable design considerations, we dedicate Chapter 3 to discussing object-oriented analysis (OOA) in general, where only application-semantics are considered, and object-oriented design (OOD) in general, where purely implementation issues have to be taken into account, and examining the different foci and the different developing knowledge involved at these two stages, and finally raising the issue that the separation of application-semantics from purely implementation representation during the design and the implementation level is demanded.

Since we use genericity, inheritance and overloading to support our approach, in Chapter 4, we discuss various polymorphisms in programming languages, show how the roles they play in software reuse, and explain our extension of parametric polymorphism   genericity to classes.

As this work is conducted in *SoftClass*, the framework and related context of SoftClass are discussed in Chapter 5.

In Chapter 6 and 7, we elaborate our approach in detail and show how to embody the reuse method into software development cycle.

We will conclude with discussion of further works related to our approach in Chapter 8.

Chapter 2

Object-Oriented Programming

In this chapter, we first review programming history to see how abstraction played its role and show major developments that led to object-oriented programming. We then discuss object-oriented programming in detail from both concept and programming language point of view.

2.1 History Review of Programming

Programming has evolved from real machine-orientation to virtual machine-oriented, to real world-orientation (object-orientation). And it is a history of abstraction [Wir90].

2.1.1 Assembly Languages

In the early days of computing, programmers wrote programs in binary digits. There was no distinction between instructions and data. *Assembly languages* were abstractions that relieved people from remembering the exact bit sequences of which specific machine instructions are composed. People used symbols instead, to remember the code pattern that is an abstraction of the binary instructions. However, at this stage, programs were totally machine-dependent, i.e programs written in one particular assembly language could not be ported to other platforms.

2.1.2 High-Level Languages

Control Structure Abstraction and User-Defined Data Structures

To some extent, high-level languages freed people from machine-specifics. Structured programming languages abstracted the control structures - sequence, branch and loop that are the basics of human logic. The facilities to construct user-defined data structures in high-level languages such as Pascal and C also make it easier for programmers to design more complicated data structures than the ones provided in the assembly languages. Each statement of a high-level language corresponds to one or several machine instructions, which are created by a compiler or an interpreter. People can write the program without worrying about the underlying machines.

Procedure Abstraction

Procedure abstraction illustrates another application of abstraction in programming. Programmers can group sequence of statements together into one unit – procedure or function , and invoke the procedure or function by one statement. If we say that control structures are high-level abstractions that are built into the languages, the procedure abstraction offers a user-defined multi-level abstraction. Programmers could deal with the computer at such abstract level they wished to have performed with these user-defined procedures, and they can further construct higher level procedures until reaching the highest level such that the whole program can be invoked by one procedure call (e.g.,“main()” in C).

2.1.3 Abstract Data Types and Modules

Abstract data types (ADTs) shifted the focus from function-based approach towards a *data-based* approach [Cox87]. An ADT is composed of a data structure and associated functions or procedures supporting the data structure. Because ADTs' functions or procedures offer stable external interfaces to users, ADTs allow programmers to write code without worrying about the specific form in which the data is represented, i.e., programmers can code at the abstract level of *what* can be done with the *data* versus *how* it is to be done. The details of the representation of the data are hidden. The users of the data only know the external behaviour of the data without necessarily knowing how the data is implemented.

Ada [DOD83] implements ADTs by *packages*. The following example is an implementation of geometric 2-D point ADT using Ada's package with separated package specification and package implementation [Car85]:

```
package point1 is
   function makepoint(x:Real, y:Real) return Point;
   function x_coord(P:Point) return Real,
   function y_coord(P:Point) return Real;
end point1;

package body point1 is
   function makepoint(x:Real, y:Real) return Point;
        -- implementation of makepoint

   function x_coord(P:Point) return Real;
        -- implementation of x_coord

   function y_coord(P:Point) return Real;
        -- implementation of y_coord
end point1;
```

However, structured programming languages (abstraction of machine) and ADTs (abstraction of data) have one thing in common: they are both *computing-oriented*. The *Object-oriented* approach focuses on the real world.

2.2 What is Object-Oriented

Object-orientation looks at a system as a a collection of objects that *encapsulate* data and functions together (like ADT). The use of objects allows additional abstraction mechanisms, in particular, *inheritance*.

Object-oriented programming = ADTs + Object type + Inheritance [Dan88].

The concepts of *object* and *inheritance* come from classification and knowledge representation theories [Str88, Kor90]. These concepts are extremely important when doing analysis and design, and are preserved until coding level.

Structured programming is carried out by first finding all the things that need to be done (functions), and then recursively decomposing each function into smaller functions until the language statements level is reached. Therefore, Structured programming concerns itself with the implementation of the programming: the sequences that compose each function and the data structures manipulated by them. Its first question is *how* [Wir90].

Object-oriented programming initially starts with the intent of the program: the *what*. Its goal is to find the objects and their connections. It apportions responsibilities of storing data and manipulating data to objects [Wir90]. Each object knows how to store its own information and how to perform its own operations. A system then is a collection of objects that know how to play their roles within the system.

Although object-oriented programming only reached the limelight in the nineteen-eighties, its origins can be traced back to the sixties from Simula [Dah66], which is as old as the principles of structured programming.

Object-oriented is defined differently by different people, nevertheless, the following general concepts, which have been agreed on as basic to object-orientation, are discussed here: *objects, classes, inheritance*, and *polymorphism and dynamic binding.* Unlike conventional software development techniques, object-oriented modeling is a unifying paradigm that is consistent from analysis, design and implementation phases of the software development life cycle [Kor90]. Objects, classes and inheritance are identified and constructed at the beginning, and preserved and enriched during design and implementation. Polymorphism and dynamic binding are language-level concepts. We will discuss these terms both from a conceptual point of view (high level – analysis and design) if any, and from language's point of view (low level)

2.2.1 Objects and Messages

Object-orientation models the real world in terms of *objects*. It views everything as an object, whether concrete or abstract.

From a conceptual point of view, each *object* has its own *features* [Mey87] that are composed by two parts: 1) *attributes* or *properties* of the object, which describe the characteristics, and 2) *actions* or *behaviour* of the object, which specify how it plays its role and how it responds to the external environment.

From a language point of view, each object is an entity that has the capability to store information, to manipulate the stored information, and/or to carry out some activities. Therefore, an *object* is an integrated unit of data and associated *operations* – functions or procedures manipulating the data or performing some transformations. In fact, high level attributes or properties are represented by data, and operations are used to elaborate the behaviour of the object[1].

Objects communicate by *sending each other messages*. The object to which a message is sent is called *receiver*. A *message* is a request asking the receiver to perform some activity. A message may request activity and information the receiver will need in order to complete the activity. An object must *understand* the messages it receives. By *understand*, we mean that the object must have an action specified which matches the action requested by the message. From a run-time perspective, sending a message to an object is actually invoking a function or procedure call associated with that object. This will be discussed in Section 2.2.4 in detail.

Only an object can access its own data directly. When an object needs to access other objects' data, it must communicate with those objects by sending them messages. The object hides its data from other objects and allows that data to be accessed only via its own methods. This is called *information hiding*. This is im-

[1]Operations are called *methods* in object-oriented terminology

portant because it protects the object's data from arbitrary and unintended use and among other things, it protects the object's data from corruption.

Different kinds of objects store different kinds of information and have different capabilities to manipulate the data or perform different activities. Objects coordinate with each other by *uniformly* sending messages to accomplish all kinds of complicated functionalities.

Objects and messages promote modular design. The implementation of an object does not depend on the internal details of other objects; it only depends on how they respond to messages. This is called *encapsulation*. *Encapsulation* is the result (or act) of hiding the implementation details of an object from its user[2].

Encapsulation separates an object's interface from its implementation. Other objects only know *what* the object can do without knowing *how* it does it. This allows objects implementations to be modified without requiring the applications that use them to be modified.

David Taylor made an analogy of objects to cells [Tay88]. Cells are organized packages that combine related information, which is contained in the DNA and protein molecules within the nucleus of the cell, and methods, which are carried out by structures outside the nucleus. The cell is surrounded by a membrane that both protects and hides the internals of the cell from outside intrusion. The membrane hides the complex internal structures and presents a relatively simple interface to the rest of the cells and organism. Cells can not read each other's protein molecules or directly change each other's structures; they can only read and change their own. Instead, they send chemical requests to one another.

2.2.2 Classes and Instances

An object is any thing, real or abstract, about which we store data and those operations that manipulate the data. Objects that respond to the same messages in the

[2]Encapsulation is realized by class defintion, which will be discussed in the next section.

same way are grouped together. A group of objects that are related in this way is called a *class*, an object in a group is called an *instance* of a class.

All the instances of a class share the same kind of attributes but do not share the same values of attributes, and the same behaviour[3]. Class is a static concept: it is a definition of all the objects belonging to the class. For example, a **Rectangle** class states that each individual rectangle has such properties as *origin, extent, color* ... and a behaviour like *compute_area, draw* .... However, each individual rectangle may have a different size, position or color. From a programming language's point of view, a class is an *Abstract Data Type (ADT)*.

The **Rectangle** class example may be implemented as follows in C++:

```
class Rectangle : public Shape {

private:
  Point extent;

public:
virtual void draw();
virtual float compute_area();
...


void Rectangle::draw()
  {
    //implementation of function draw()
    ...
  }

float Rectangle::compute_area()
  {
    //implementation of function compute_area()

    return ((extent.getX() - origin.getX()) *
            (extent.getY() - origin.getY()))
  }
```

[3]This is class-based object-orientation There are other approaches such as *delegation*, which allow instance-level inheritance [Lie86, Ste87]

```
...

}
```

From the above example, we can notice that the attributes of objects are also objects *extent* is of class **point**, and therefore, complex objects are composed of other (simple) objects, these objects, in turn, may be composed of objects. The operation of an object may require that other objects cooperate and share responsibilities **compute_area** uses **Point** objects and associated operations **getX, getY** to accomplish the computation.

Encapsulation gives us the advantages of data protection, it also brings about other advantages such as reducing complexity and minimizing dependency. Let us consider how encapsulation is realized in class definition. But first, we introduce two concepts class as *client* and class as *supplier*. As we saw in the **Rectangle** example, a complex object may be composed of other objects. This is defined at class level. A class that relies on or uses another is said to be a **client**; the other class is the **supplier**.

Protecting data

This is implemented by restricting the right to access the data portion of the class. Only the instances of the class may directly access their data. Other objects must access the data through the operations offered by the supplier class. In C++, one can restrict the access right using *private, restrict* or *public.*

Reducing complexity

The implementation details of the class are private to the class. A supplier class only exposes the interface to client classes. Some of the data or operations of a class may be only for internal uses and therefore, should not be exposed to outside objects. For example, a **VCR** class has a very complex structure and a myriad of operations, however, a user is only interested in some particular features it offers, such as *the state of the indicator* and some operations such as *Play, Record*, etc.. This greatly reduces the complexity of classes from client classes' point of view.

Minimizing dependency

As we mentioned earlier, complex objects are composed of other objects. The complex class uses other supplier classes to define its data portion. As the implementation details are hidden from clients, supplier classes can be free to modify their internal implementation detail without affecting the client classes, provided that they keep the same external interfaces.

In some object-oriented languages, such as Smalltalk, a class itself is also treated as an object. A class that defines class-like objects is called *metaclass*. In Smalltalk, a metaclass has only one instance that is the class itself. Class-level objects also have capabilities to store information about the class as a whole, and to support class-level operations. For example, the **Average_Height** attribute for **Person** class reflects an attribute of all the instances of **Person** class, while the **Get_Population** operation is a class-level operation.

2.2.3 Inheritance, Subclasses and Superclasses

Objects and classes can can be traced back to ADTs, but inheritance is a unique contribution of the object-oriented paradigm. It is inheritance combined with the object and class concepts, that characterize object-oriented programming [Dan88].

Inheritance is the ability of one class to define its data structure and behaviour of its instances as specialization/extension of the definition of another class or classes.

A *subclass* describes a group of objects that inherit information and behaviour from an existing class. The class from which a subclass inherits is called *superclass*. A superclass is also called a *base class* in C++, while a subclass is called *derived class*. Let us elaborate on the inheritance relation by an example.

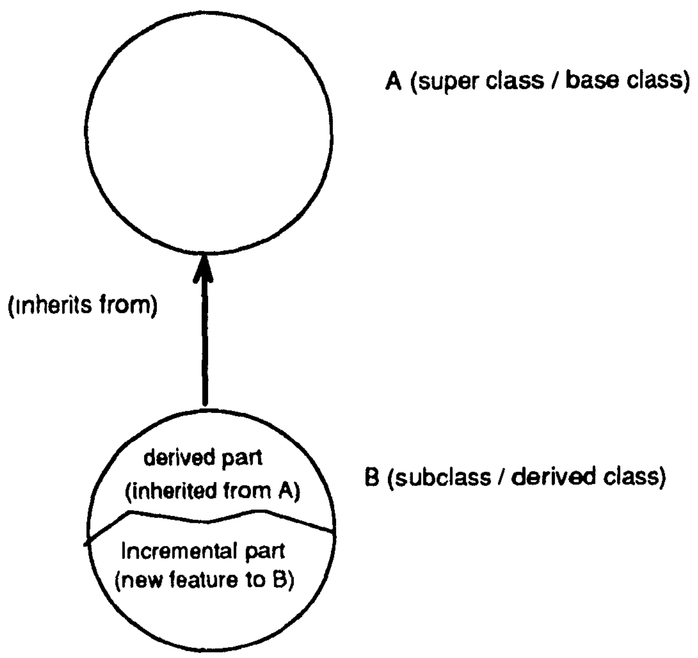


Figure 2.1: Inheritance

Supposing class **B** inherits from class **A** (see Figure 2.1 [Kor90]). Class **A** is referred to as the base class and class **B** is referred to as the derived class. Class **B** is composed of two parts: the derived part, which is *inherited* from base class **A**, and the incremental part, which are newly-added features, specific to **B**.

From this, we can see that inheritance actually reflects the generalization-specialization relation among classes. A derived class is specialized from its base classes. And a base class is a general form of its derived classes.

A derived class usually adds its own attributes and behaviour to define its own unique features in addition to those (attributes and behaviour) inherited from its base class. It may also modify those features inherited.

Inheritance and Reuse

Inheritance produces two major promising advantages: inheritance for extension and inheritance for reuse.

- **Inheritance for extension (Reuse without Modification)**

  Inheritance allows us to design a new class of objects as a refinement of another For example, when defining classes of geometric figures to be displayed on a window system, a **Shape** class is defined first, then **Rectangle** and **Circle** classes are added as subclasses of the **Shape** class. Later, someone might need **Triangle**. *Without affecting other classes*, the class **Triangle** can be added directly as a subclass of **Shape**, and **Triangle** inherits all the data structures and operations from **Shape**, only new features specific to **Triangle** are added. This is the way inheritance helps extend classes.

- **Inheritance for reuse (Reuse with Modification)**

  Actually, extension includes reuse. Here *reuse* has specific meaning: reuse by modifying (overriding, cancelling) features from existing classes. They have the inheritance relation only for the purpose of reuse. For example, **Array** and **Stack** class: a **Stack** can be implemented using an **Array**, therefore, we can make **Stack** a subclass of **Array**, but **Array** and **Stack** are not in a generalization/specialization relation. **Stack** inherits only *part* of the features of **Array**. e.g., **Stack** only allows access to the first element, while **Array** allows access to every element. Therefore, **Stack** needs to void part of the operations offered by **Array**.

Abstract Classes

In cases of extension and reuse, *abstract classes* play an important role. An *abstract class* is a class which springs from a group of classes which share one or more than one features (data components or operations). It actually abstracts out the common features among classes, but itself does not make much sense as an ordinary class, i.e., the instances of such class are meaningless and therefore should be never be used in any program. Therefore, it exists just for the purpose of organization of inheritance relations between other classes. Abstract classes more often appear at the late stages of design and implementation. they are rarely found at the analysis stage, since analysis focuses on understanding the objects that are "visible" in the problem domain.

In the above example, when a new class **Triangle** is needed, it can be added directly as a subclass of **Shape**. We may realize that **Triangle** and **Rectangle** share some common features, since they are both *polygons*. A better approach would be to create an abstract class **Polygon** to abstract the common features/attributes of the existing class **Rectangle** and the one to be created - **Triangle**, so that the features in **Polygon** can be used by **Triangle** without duplication. Furthermore, we could have other shapes other than polygons, such as circles, to be subclasses of **Shape**, but they do not have common features/attributes with polygons besides those from **Shape**.

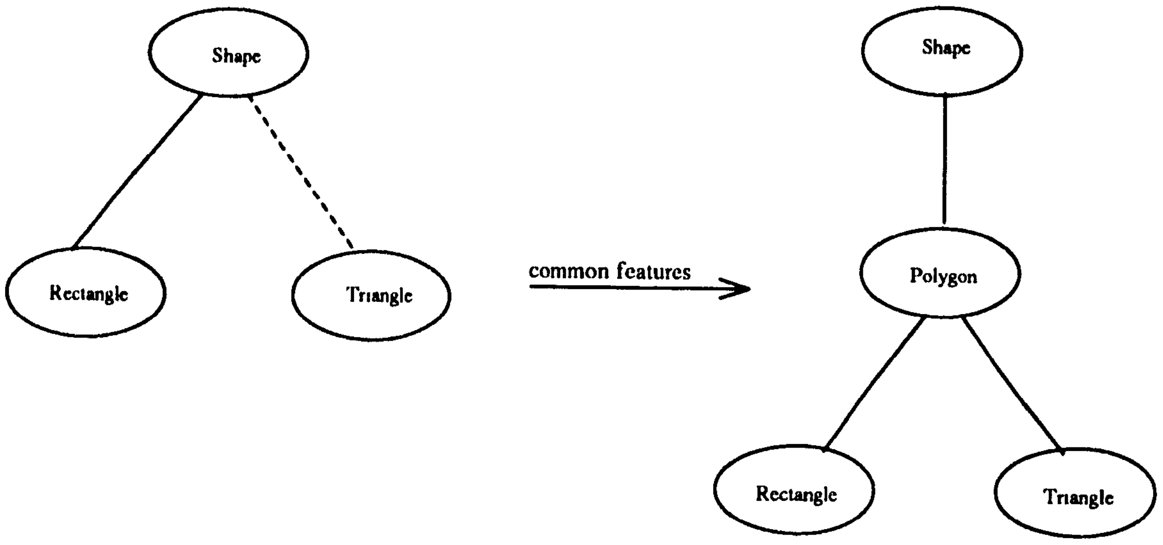


Figure 2.2: Abstract Class

In case of reuse with modification (overriding, cancellation), there are opportunities for misusing inheritance. For instance, most OOP languages allowed derived classes to *rename*, *reimplement*, *duplicate* or *void* the features inherited from its base class, to change its interface, and to support a behaviour which is completely unrelated to that of its base class. What constitutes the proper use of inheritance is a widely debated topic [Cox90, Coo92].

Multiple Inheritance

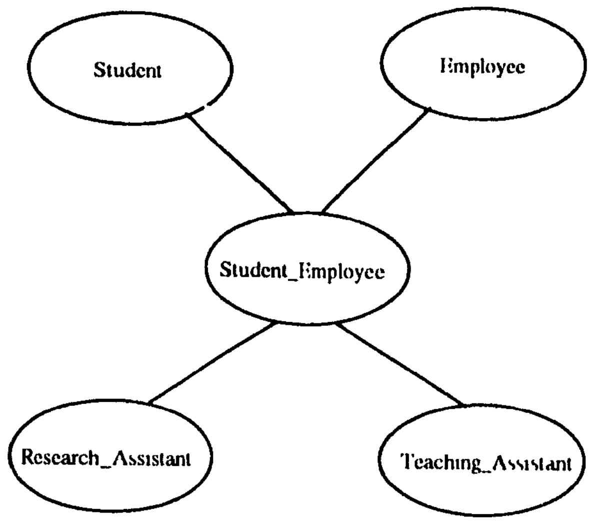


Figure 2.3: Multiple Inheritance

When a class has more than one immediate parent, it may inherit features from all of them. This is called *multiple inheritance*. For example, (Figure 2.3), a system is being developed for a School Administration System, where a **Student** and **Employee** class have been defined, and a new class whose instances are both Students and Employees is required. The designer may create a **Student-Employee** class as a subclass of both **Student** and **Employee**.

Multiple inheritance may bring conflicts such as name-clashes. This can be solved by different approaches renaming (Eiffel [Mey92]) or redefining features (C++ [Str86, Lip92]) at the subclass.

Ducournau etc. introduced monotonic conflict resolution mechanisms for inheritance [Duc92].

2.2.4 Polymorphism and Dynamic Binding

There are many kinds of polymorphism. We will dedicate Chapter 4 to discuss fully the various kinds of polymorphism with emphasis on a reuse perspective.

Generally, *polymorphism* means the ability of an object to take more than one form. In OO, polymorphism and dynamic binding are used at the language level. Together they support the inheritance mechanism in object-oriented programming languages [Car85].

In an object-oriented programming language, a polymorphic reference is one which refers to instances of more than one class over time. In strongly typed object-oriented language, a polymorphic reference has both a *static type* and a *dynamic type.* The static type is that specified in the declaration in the program - compile-time type. The static type determines all the valid types acceptable to the reference – the class itself and all of its subclasses. The dynamic type of a polymorphic reference may change during program execution but all within the valid types defined by the static type.

For example, assume that a variable *aPolygon* is declared as an instance of **Polygon**, and assume that **Polygon** has two subclasses **Triangle** and **Rectangle**. The declaration of *aPolygon* is confined to refer to instances of **Polygon**, instances of **Triangle**, and instances of **Rectangle**. The *binding* to actual type of the variable is only determined at run-time. This is called *dynamic binding.* The static type of *aPolygon* is always **Polygon**, while the dynamic type may be **Polygon, Triangle** or **Rectangle** depending on run-time.

Dynamic binding also refers to *method selection*, which refers to the binding of a message to the method code to be executed in response to that message.

Suppose that the procedure **display** is defined as an operation of class **Polygon,** but redefined in **Triangle**. The message sent to *aPolygon*, *aPolygon* **display**, will be invoked depending on the run-type dynamic type of *aPolygon*. If its run-time type is **Polygon**, the call will be bound to the method defined by **Polygon**. If the dynamic type of *aPolygon* is **Triangle**, then the call would be bound to the method as redefined in class **Triangle**.

The dynamic binding and polymorphic reference together realize the code reuse by subclasses.

Suppose that another method called **setColor** is defined as an operation of class **Polygon**, but not defined in **Triangle**. The message sent to *aPolygon* that refers to an instance of **Triangle** – *aPolygon* **display** will actually invoke the operation defined by **Polygon**, and therefore **Triangle** reuses the code form **Polygon**.

2.3 Object-Oriented Programming Languages

2.3.1 When Can a Programming Language Be Called an OO Language?

An object-oriented programming language must support the following notions: 1) classes, 2) inheritance, and 3) polymorphism and dynamic binding [Kor90, Dan88].

Stroustrup argued that there is an important distinction between a language *supporting* certain features of programming style and a language *enabling* such features. "A language does not support a technique if it takes exceptional effort or skill to write such programs." [Str88]

However, there is also a distinction between a language *disciplining* a program style and a language *supporting* such style. For example, Smalltalk forces[4] users to write every single code by defining class (subclass relation, instance variables and associated methods) and to invoke any action by sending messages to certain objects. On the other hand, a claimed C++ programmer can write programs without using any object-oriented notions.

It is also true that object-oriented programming does not necessarily require an object-oriented programming language. For example, message passing can be ob-

[4]Of course, there also exists good and bad style in Smalltalk programming

tained by *message schedulers* [Bla89], in which a function Send(x,p) is invoked, where x is the object to which the message is sent, and p is a parameter record that contains the actual message and its parameters. It is inevitable that efficiency is lost, since the parameter record has to be assembled before each function invocation.

Dynamic binding can be simulated by linking each object x with the functions possible upon it. Every object is implemented as a record with one operation pointing to a table of all possible function variations. And polymorphism is achieved by coercion (type conversion). The problem of unreliability is obvious and it can only be reduced by extra disciplined programming guidelines.

2.3.2 Dynamic Binding vs Typing

Programming languages can be classified according to the extent of type declaration and type checking provided at compile time, such as strongly typed, weakly typed and typeless. A type is viewed as a set of values. The type system is introduced to impose constraints over the possible values [Dan88]. When we say a variable belongs to a certain type, we actually restrict the possible values it can carry and distinguish it from other sets of values (types).

Type constraints are specified through the declaration of variables and arguments of functions. The compiler may check whether the uses of the variables or functions satisfy (are consistent with) the declarations. Most of the traditional languages are strongly typed. At the other end of the spectrum, we also have untyped programming languages. There are two meanings to *untyped*: 1) it means that there is *no* type or only *one* kind of type such as Lisp; 2) it means that there are different types, but the user doesn't need to specify types of variables, as type information can be inferred at run-time or compile-time, ML [Mil84] belongs to this category.

The same distinctions apply to object-oriented programming languages. In object-oriented programming languages, classes are types. Since objects not only have data but also have associated operations, type checking involves both data type compatibility and the determining proper operations applied to an object.

Since a reference of a certain class (static type) can refer to either instances of the declared class, or instances of its subclasses, type-checking would take place within the class hierarchy.

2.3.3 A Comparison of Smalltalk, Eiffel and C++

A comparison of several programming languages with object-oriented features is given in [Bla89]. In this section, we compare Smalltalk, Eiffel and C++ as representatives of three trends: 1) C++ is a representative of object-oriented extensions of conventional programming languages, 2) Eiffel is a new object-oriented language with multiple inheritance, and 3) Smalltalk is the most consistent implementation of object-oriented principles.

The following criteria were established for the language comparison by Blaschek etc [Bla89]:

1. Inheritance mechanism: C++ and Eiffel support multiple inheritance, Smalltalk supports single inheritance but can emulate multiple inheritance.

2. Run-time reliability: Eiffel and Smalltalk are more reliable than C++.

3. Uniformity of data structure: Smalltalk is a pure object-oriented language, all data types are classes. C++ retains all the elements from conventional languages. In Eiffel, basic data types are not classes.

4. Documentation value: The readability of Eiffel programs is higher than the Smalltalk and C++.

5. Memory management: C++ does not provide garbage collection but the other two both have garbage collectors.

6. Efficiency: C++ has very high run time efficiency. Smalltalk has the lowest efficiency. And Eiffel stands between.

7. Language complexity: Smalltalk's syntax is very simple. Everything is done by sending messages. C++ is most complex among the three.

The choice of a particular language depends on the different application environment. C++ is the first choice OO language if programmers know C and if run-time efficiency is the highly desired. If high safety and multiple inheritance top the priority list, then Eiffel might be the best choice. Smalltalk can always be recommended if efficiency is not the leading criterion.

Chapter 3

OOA and OOD, and the Comparison of Inheritance Hierarchies of Analysis and Design

Compared to traditional structured analysis and design (SA/SD) methodology, Object-oriented modeling is a unifying model, spanning all phases of software development [Kor90]. However, the transformation from analysis to design is not trivial [Cha92, Cha92a]. In this chapter, we first give a brief introduction of the advantages of object-oriented model over the traditional life-cycle models. Then we examine the OO analysis and OO design methodologies in general, illustrate the differences between the two stages, and finally point out that in order to keep the clarity of the inheritance relation of application-semantics during the design, a separated application-semantics hierarchy should be maintained along with the pure implementation hierarchy

3.1 Introduction

The fundamental weakness of the conventional life-cycle (waterfall) model is its imbalance between analysis and synthesis [Agr86]. The waterfall model was formulated on the basis of: 1) the fact that machine-time cost was very high (limited hardware techniques available and limited opportunities of access computers), as a consequence, lots of things were needed to be done before moving to hard coding on the machines, 2) lack of tools to support earlier development stages. The problem of the differences between data flows in structured analysis and hierarchies of tasks in structured design (no unifying model) makes it difficult to integrate the phases. The problem of no emphasis on reuse is reflected by the fact that each system is built from scratch and is difficult to maintain and to extend later on [Kor90].

The object-oriented paradigm addresses each of these issues. Object-orientation brings consistency through the software development life cycle: 1) the designer's model is similar to the analyst's model, and 2) the development process is iterative [Kor90].

Object oriented analysis and design model the world in terms of objects that have properties and behaviour, and events that trigger operations that change the state of the objects.

As in other methodologies, analysis helps us understanding the problem domain, and design fills in enough detail to generate code. Design is a realization of the analysis, but implementation-language independent.

Martin and Odell give two types of object-oriented models for analysis and design: a model of the object types and their structures (object schema) and a model of what happens to the objects (event schema) [Mar91].

The first model, which is referred to as *Object Structure Analysis (OSA) and Object Structure Design (OSD)*, concerns object types, classes, relationships among objects, and inheritance.

The other model, called *Object Behavior Analysis (OBA) and Object Behavior Design (OBD)*, concerns the behaviour of objects and what happens to them over time.

For different problem domains, we use different models. In some applications, OSA and OSD are more important than OBA and OBD. In others, it is the opposite.

In extremely complicated applications, both approaches are needed, since these two models reflect the two aspects of the real world, and only by understanding both the structure and behaviour can we really have deep insight about the problem

The model we present here is a hybrid object-oriented model for analysis and design which is based on [Coa91, Coa91a, Boo91].

3.2 Object-Oriented Analysis (OOA)

Analysis is the activity that yields a description of what the problem domain is composed of, and of what the target system is supposed to do, detailing functional, performance and resources requirements. This concept could be the basis for a contract between the client and developer and aims to produce clear input to the designer.

Object-oriented analysis (OOA) describes a target system in terms of 1) objects (entities) and their attributes and/or components, 2) descriptions of behaviour of objects and 3) inheritance hierarchy of objects.

The steps of OOA can be divided into [Coa91]:

1. Identify objects in the target system
2. Identify structures and relations among objects
3. Define attributes of each object
4. Specify behaviour of each object
5. Classify inheritance interrelation among objects

3.2.1 Identify Objects

Objects need to reflect both the problem domain and the system's responsibilities [Coa91].

For example, a problem domain might include:

```
Person
  Name
  Address
  Weight
  Height
  Age
  Salary
  ...
```

Yet given system's responsibility for a particular application may only include:

```
Person
  Name
  Address
  Age
```

With OOA, an analyst studies the overall problem domain, filters those aspects that are not within the system's responsibilities, abstracts those aspects of interest and models them accordingly.

Identifying objects can be carried out using various approaches, depending on the scale of the target system. A small system only yields a "flat" set of objects. When the target system is very large, directly identifying objects becomes tricky. Large systems have various objects involved, and very complex interactions among objects. The well-established divide-and-conquer strategy has to be applied, which here means top-down approach.

The target system can be divided into several relatively independent sub-systems. Each sub-system is independent in the sense of its functionality. And all sub-systems are coordinated within the whole problem domain in the way of communication and cooperation to fulfill the functionality as a whole [Cha92a]. If the problem domain is so complicated that sub-systems need to be further divided into smaller sub-systems, multiple layers of sub-systems may be used to understand and analyze the problem domain.

There are several ways to investigate the problem domain and identify the objects: 1) observe first-hand, 2) actively listen to problem domain experts, 3) check previous OOA (reuse analysis level knowledge), and 4) obtain information from client's requirement documentation [Coa91].

Candidate objects can be found in the following categories:

- Structures: relationships among objects - generalization-specialization relation and whole-part relation,
- Other systems with which the systems under consideration will interact,
- Devices,
- Thing and events remembered,
- Roles played,
- Operational procedures,
- Sites,
- Organizational units etc.

A candidate object must satisfy the following criteria:

- Is it in problem domain?
- Does it satisfy the domain-based requirement?
- Does the system need it to store data (information)?
- Does it need to offer some behaviour required by the system?
- Can it be represented by another object or can it be obtained through specialization from an existing general object?
- Does it need to be divided into a group of objects that are generalization/specialization related?

3.2.2 Identify Structures and Relations Among Objects

There are three kinds of relations among objects:

- Generalization/specialization relation
- Whole/part relation
- Association - instance connections

Generalization/Specialization Relation

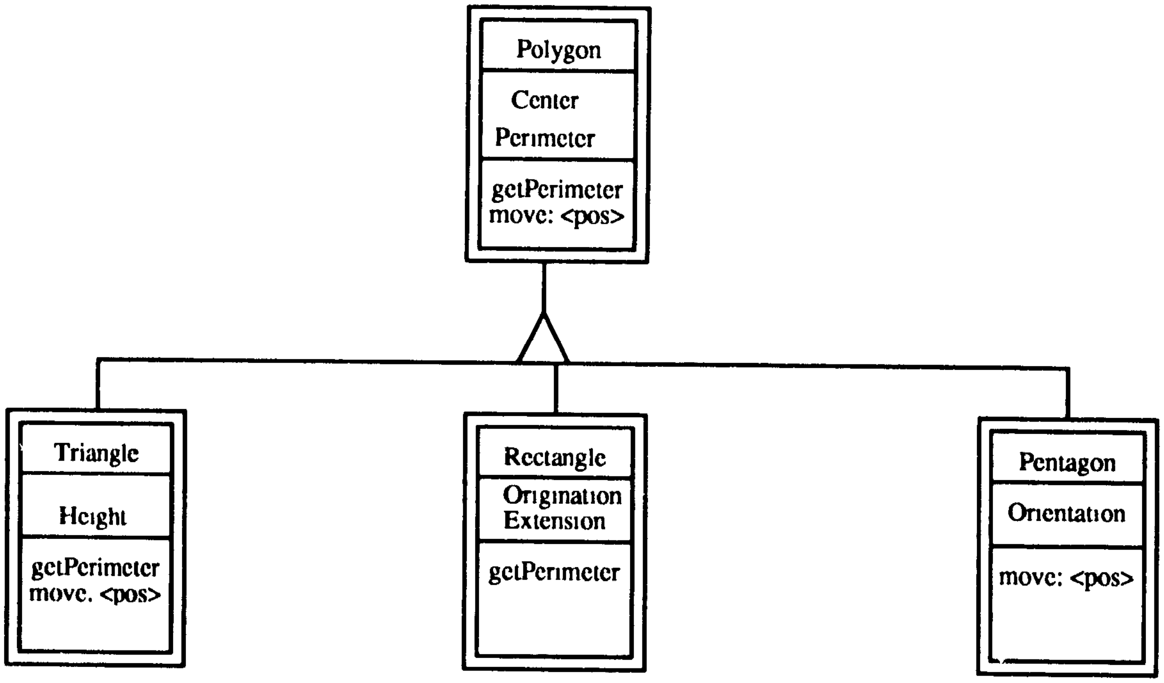


Figure 3.1: Generalization/Specialization Relation

Generalization/specialization can be viewed as an "is a" or "is a kind of" relation. Figure 3.1 shows an example of a generalization/specialization relation where Polygon is in generalization form and it has three specializations: Triangle, Rectangle and Pentagon. A Triangle is a kind of Polygon. Within generalization/specialization, inheritance applies such that specialized objects will inherits the attributes and behaviour from the general objects.

To identify generalization/specialization relations among objects, first consider each class as a generalization, and examine its potential specializations using the following criteria:

- Is it in problem domain and within system's responsibility?
- Will there be inheritance?

Similarly, consider each class as a specialization, examine its potential generalizations using the same rules mentioned above.

A specialized class may be obtained from more than one generalized class, and the hierarchy will contain multiple inheritance.

Generalization/specialization structures highlight additional specialization, explicitly capture commonalities, encourage reuse and therefore reduce redundancy of attribute and behaviour specification.

Whole/Part Relation

Whole/part relation can be thought of as a "has a" relation. Figure 3.2 shows an example where a Text is consisted of Paragraphs. Whole/part relations help people understand complex structures by decomposing them into parts, and help understand functionalities as a whole by dividing responsiblity and their coordinations among the components [Coa91].

Whole/part relations take several formats in practice:

- Assembly-parts
- Container-contents
- Collection-members

The purpose of finding whole/part relations is to help decompose complex systems, "whole" or "parts" need to be identified only they are in the problem domain and share in the target system's responsibilities.

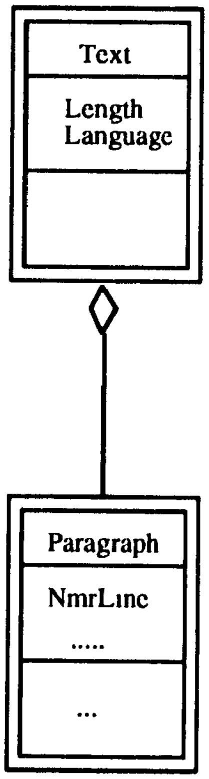


Figure 3.2: Whole/Part Relation

Object Connection – "Side-By-Side" Interrelation

Objects associate together to perform certain kinds of functionalities. In non-trivial systems, objects need to have connections with other objects that are neither generalization/specialization nor whole/part related, but "side-by-side" related. Objects coordinate and communicate with other objects, delegate other objects or combine with other objects together to perform certain functionalities [Coa91]. Figure 3.3 shows an example where Person objects and Company objects are in an Employed by/Employ relation.

3.2.3 Define Attributes

There is no fixed requirement to work from defining attributes to specifying behaviour. Actually both perspectives are valuable. People usually work back and forth between

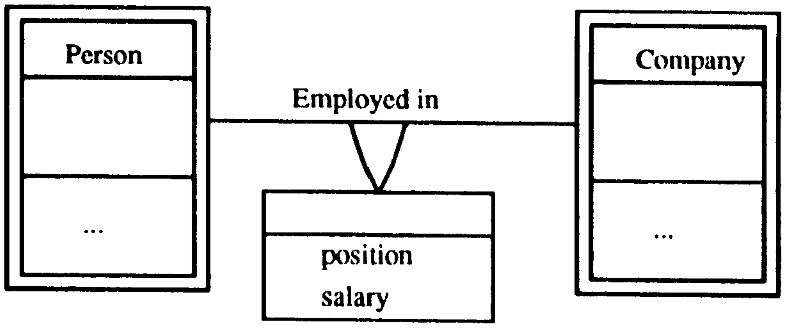


Figure 3.3: Association - "side-by-side"

these two activities.

"Attributes are properties, characteristics and qualities that are ascribed to an object" [Coa91]. Attributes are used to describe states (values) kept within an object. These attributes must reflect both the problem domain and the system's responsibilities.

To define the attributes of an object is to specify the object in more detail in the aspect of the states and information that are needed for the particular application [Coa91].

At the analysis level, representation of the attributes should be of no concern. The main job should focus on understanding the system and specifying the responsibilities.

When defining attributes, generalization/specialization relation should be considered at the same time in order to position attributes properly to obtain inheritance and clean structures among objects.

3.2.4 Specify Behavior

Objects are encapsulation of data and operation. The attributes reflect the states of an object or information kept within the object, while the operations are manipulation of these attributes and other related objects. These operations perform a set of functionalities and offer certain services responsible for exhibiting.

Attributes combined with behaviour (operations or services) together specify a unique object. Operations can be classified into algorithm-simple and algorithm-complex categories.

Algorithm-simple operations are such operations as *creating, connecting, accessing, releasing.* Such operations are usually implementation-dependent at design and implementation level, since they are closely related to underlying data representations.

The other kind of operations fall into *calculation, monitoring, transformation* categories. Usually these kinds of operations are application-specific, but data-representation-independent at the design or implementation level.

To specify an operation, two steps may be taken:

- From the architectural point of view, message connection or processing dependency has to be identified. i.e., all the related objects and their needed behaviour required by the operation in order to fulfill the responsibility.

- For detailed logic sequence, each substep and the control structure are specified.

Similar to the process of defining attributes, when specifying behaviour, generalization/specialization relations should be considered at the same time in order to position behaviour properly to obtain inheritance and clean structures among objects.

3.2.5 Classify Objects within Inheritance Hierarchies

Generalization/specialization relations between objects yields a hierarchy (or a set of hierarchies) called *inheritance hierarchy.* As discussed in the previous sections, the classification of the inheritance hierarchy may be done immediately, but it is done within a small scope only local to the objects of interest. From the whole system or subsystem point of view, reexamining the inheritance hierarchy is desired in order to get a cleaner hierarchy for easier understanding and potential reuse. This will affect the following levels for the development cycle.

3.3 Object-Oriented Design (OOD)

Design is the activity that yields an artifact description of how a target system will work. The design satisfies the requirement, but remains *implementation language independent.* Design can be viewed as an implementation of analysis. It provides a *computational* description of a program that meets analysis requirements. It differs from implementation. "Implementation activities are *environmental*, providing an expression of the design suitable for the target environments; i.e., using particular programming languages, particular tools and systems, and particular configurations" [Cha92a].

Object-oriented design (OOD) can be viewed a mapping of analysis objects with generic data structures along with detailed description of the behaviour identified at the analysis level on the basis of the chosen data representation.

Although OOD uses the same model as OOA, the notions of OOD are different from those of OOA because design activities need to be performed with reference to *computational models.* OOA involves in identifying application-semantics attributes for each object, while OOD will choose a proper data structure to represent each attribute, and then each object. Furthermore, attributes and parts of analysis notions are both mapped to components at the design-level. The design level inheritance hierarchy is built based on both their attributes (application-specifics) and their underlying data representations (data structure-specifics).

OOD can be divided into the following steps:

1. Class design
2. Program design
3. Classify objects within an implementation hierarchy

3.3.1 Class Design

Class Design produces a definition of the representational and algorithmic properties (attributes) of classes according to the declarative constraints specified within OOA. The representations of classes are in a abstract fashion, i.e., high-level abstract data structures independent of any programming languages.

At the analysis phase, the structures of objects are classified into generalization/specialization relation, whole/part relation and "side-by-side" relation. At the design level, the notions to express the structures of classes are restricted to *compositional* notion.

Attributes and components identified at the analysis are replaced by components at the design level. "Side-by-side" connections among several objects can be represented in one of the two ways: 1) Each object contains fields referring to other objects; or 2) Association objects (tuples), which are design-level objects used to represent and coordinate analysis-level association between objects.

For example, instances of the **Company** class and instances of the **Person** class may have *employer/employee* association. At the design stage, we can let **Company** have pointer references to **Persons** as their employees. It is another choice to create an association object class **Employer_Employee** whose instances consist of tuples that refer to the instances of both **Company** and **Person** class.

At the design level, nearly all objects are both *clients*, i.e., using other objects either by reference or by sending messages, and *servers*, i.e, used by other objects either by being referred to or by being sent messages. Therefore, we have to consider class design from either point of view. Champeaux etc. give some guidelines about composition of components at the design-level [Cha92].

1. Classes as servers

   The goal of designing classes as servers is to design each class to be amenable for use as a component of other classes. *Design for reuse* is a key consideration.

This involves:

- Design of (abstract) classes rather than one-shot objects abstract out as many attributes and operations as possible from subclasses so that it is easy to extend later and to gain more reuse.
- Design of classes interfaces (accessor, operations) rather than of attributes and transitions, because subclasses can override operations but can not override attributes (components).
- Design of services and protocols (access control, locking, etc) so that objects may be used predictably and reliably by others [Cha92].
- Minimization of representational and informational demands upon clients (low coupling).

2. Classes as clients

   When composing a class using other defined classes, *design with reuse* is one of the main criteria. The concerns for designing classes as clients are following.

   - Block-box reuse. Minimization of representational and informational demands upon severs, thus allowing a broad range of concrete object types to be employed as components. This includes the use of capability-based (abstract) rather than implementation-based (concrete) specification of internally accessed objects. For example, we use "Next_Element(aCollection)" to get the next element in a collection instead of using "aCollection[i+1]" for an ARRAY or "aCollection→next" for a LinkedList.
   - Using implementation-independent delegation rather than concrete subclassing as the compositional technique of choice. i.e., if we want to reuse the functionality of a class A, instead of creating a subclass of class A, we build another class with its data containing the instance of class A.

- Minimization of protocol demands upon servers. This will reduce the dependence on the server. When some changes occur in server, the client will be affected at a minimal price.
- Design of coordination schemes such as trigger, transactions, to maintain static and dynamic invariant both among components and between components and self.

3.3.2 Program Design

Program Design produces the detailed algorithm for each operation on objects in an abstract fashion (abstract control structures of programming languages). It requires a more concrete reference model for describing the program logics – the computational structure.

Operations on classes can be classified into two categories: 1) base functions or uncomputational functions, such as constructor/destructor, accessor/selector, modifier and so on, and 2) computational or complex functions, which are usually application-specific and use other base functions to perform a given functionality.

The base operations then include field operations that are used for access and modification. Martin and Odell examine two different categories of operations for single-valued fields and multivalued fields [Mar91]. For single-valued field, only two fundamental operations are needed: **Assign** and **Get**. Multivalued fields require at least five fundamental operations: **Add, Remove, Detect, Get_using** and **Apply**. **Add, Remove** and **Detect** are all self-explanatory, **Get_using** operation allows the programmer to select a specific method that retrieves objects that satisfy the given boolean function. **Apply** operation extends beyond the retrieve-only limitation of the **Get_using** operation, it allows users to select a specific method which *updates* contained object.

Notice that such base operations are data structure dependent but application-independent. These operations can, therefore, be mapped directly to the operations offered by the chosen data structure.

As for computation-loaded operations, the description of the logical sequences and of the control structures is the most significant. The description can be either formal and precise, or informal and coarse. Usually a *Programming Description Language (PDL)* is used to formally describe a program.

When designing computed functions, the following issues need to adhered [Mar91]:

- Function expression are reusable
- Function expression can be eager or lazy
- Controlling redundancy

3.3.3 Classify Objects within an Implementation Hierarchy

The inheritance hierarchy at the design level is, first and foremost, an implementation hierarchy. Second, the hierarchy at the design level should keeps the generalizational hierarchy.

Some aspects that are related only to design are considered:

1. *Abstract classes* are introduced.

2. As fields are used to represent *attributes, components* and *references*, of objects at the analysis phase, subclasses that share the fields from superclasses will inherit all the characteristics including but not limited to attributes.

3. For efficiency reasons, classes that have no members and operations, but that may help understand the system during analysis, may not be mapped to classes at the design level, and may be replaced using flags in the subclasses. For example, given an application, we may identify two classes at analysis level: class **Vehicle** and class **Car**, with class **Car** as a subclass of class **Vehicle**.

However, in our application, we only deal with instances of class **Car**. In such case, we may eliminate class **Vehicle** during design and only keep class **Car**.

4. Access rights are considered during design such as making a class as private or public.

The design level inheritance hierarchy should follow these guidelines [Wir90]:

- Model a "kind-of" hierarchy
- Locate common attributes and operations as high as possible in the hierarchy
- Abstract classes should not inherit from concrete classes
- Eliminate classes that do not add functionalities

3.4 Inheritance Hierarchy at Analysis Level versus Design Level

Although OOA and OOD use similar models so that the transformation from analysis to design stage is easier than in traditional structured methodologies, the transformation is not trivial. The main reason is that the two levels focus on different aspects. Analysis is most concerned with understanding application-semantics, and least concerned with the computational representations. Design, on the other hand, is concerned with choosing a representation for the objects in the problem domain, and considers more in the computer-perspective, although in an abstract form, since it is independent of specific languages and systems.

3.4.1 The Difference between the Two Hierarchies

The different foci of analysis and design yield different hierarchies, which can be compared along the following perspectives:

1. The classification rule

   The analysis-level hierarchy is a generalization/specialization hierarchy. Specialized classes (object types) inherit properties and behaviour from general class(es). The classification of generalization/specialization relation is based on the common properties and behaviour.

   When moved to design level, analysis level classes are mapped to certain data representations. Subclass relations are based on both analysis-level properties and data representations.

2. Classes deleted and added at the design stage

   Some object types at the analysis level are helpful to decompose and then to understand the system, but they themselves add no members or operations, Such classes may be deleted from design and therefore will not exist in the hierarchy.

   Design may also produce new classes that can not be found at the analysis stage. *Abstract classes* and *classes of association objects* are such kinds of classes

3.4.2 What is Missing from Implementation Hierarchies

At the design level, subclasses inherit data structures and the application-semantics from superclasses. If the shared attributes of superclasses and subclasses (at analysis) are represented using different data structures, then subclasses (at design) will not be able to inherit from superclasses[1]. In other words, the application-semantics may yield to the data structure representations, with loss of some inheritance relations.

3.4.3 Keeping Two Separate Hierarchies

What has been missing from design hierarchies can compensated by keeping the analysis-level hierarchy until design and making the implementation hierarchy only

[1]Although in untyped OO language, such as Smalltalk, there is no type declaration specified for the fields, the type restriction is implicitly required

data representation-dependent but application-semantics-independent so that the application-specifics will be preserved no matter how the design implements the analysis, and that the inheritance relations of each hierarchy can keep their clarity.

An analysis-level class which is differently implemented in existing software from the one we are currently developing, can not share the data representation. However, the application-specifics are the same regardless the different implementation, and therefore, the application-dependent operations should be reused.

An analysis-level subclass which is differently implemented from its superclass(es), can not share the data representation. However, the subclass-relation that is identified at the analysis level, implies that the application-semantics should be inherited by the subclass regardless the data representation, and therefore, the application-dependent operations should also be inherited.

We will elaborate our approach in Chapter 5 and 6. Since a general genericity will be used to describe application-independent data structure representations in our approach, we first look at the various polymorphisms, especially genericity, in the next chapter, and see how they play roles in software reuse. Finally we introduce our extension of genericity.

Chapter 4

Polymorphism and Reuse

4.1 Polymorphism

Polymorphism in general, means the ability to have more than one form. In programming languages, polymorphism means the ability to have more than one *type*. Polymorphism in programming languages is reflected in the following aspects [Car85]:

- *Polymorphic constants or values* are values that may have more than one type. e.g., " 1 " may be treated as an integer or a real value in different context.
- *Polymorphic variables* are variables that may have more than one type. e.g., in C++, a variable declared as of class **A**, may be used to refer to instances of class **A** and instances of all the subclasses of class **A**.
- *Polymorphic functions* are functions whose actual parameters can have more than one type. e.g., " + " operation can be used for integer addition or real addition.
- *Polymorphic types* may be defined as types whose operations are applicable to actual parameters of more than one type. e.g., a function whose parameters are declared as certain types can be applied to all the subtypes of corresponding declared types.

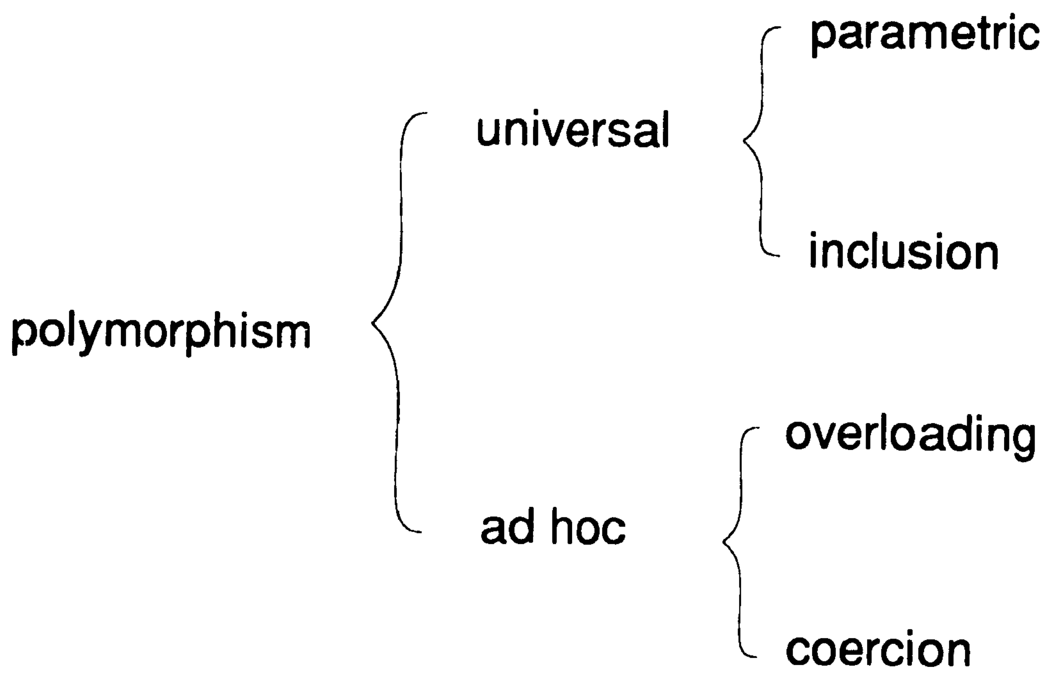


Figure 4.1: Varieties of polymorphism

Cardelli and Wegner classified the different kinds of polymorphism [Car85] (see Figure 4.1). This classification extends Strachey's work [Str67] by introducing a new form of polymorphism called *inclusion polymorphism*. Strachey distinguished informally, between two kinds of polymorphism: *Parametric polymorphism* and *Ad-hoc polymorphism*. In the remaining in this section, we discuss the various kinds of polymorphism. In Section 4.1.1, we discuss overloading and coercion. Next, we examine parametric polymorphism. And finally, we investigate inclusion polymorphism.

4.1.1 Overloading and Coercion

In the above classification, ad-hoc polymorphism is further divided into two categories: *overloading* and *coercion*.

Overloading

Overloading means that the same name is used to denote different functions, while the compiler uses context to determine which actual function is denoted by a particular instance of the name.

Overloading *itself* is not true polymorphism in the sense that instead of an operation applicable to many types of arguments, we allow the name of the operation to be used for many types. The code actually executed is quite different from one to an other. However, overloading is not just a syntactic construct allowing the same name for different semantic objects: the desire to overload functions or procedures reflects the similarities of their structures and behaviour semantics. For example, the operation name "add" can be overloaded to denote to adding an element to an array, to a link-list, to a queue or to a set. The use of the same name for these four operations embodies the similarities of the container structures and the behaviour of the four data types.

Coercion

Coercion is actually a semantic operation to convert the argument to the type desired in a situation that would otherwise result in a type error. A coercion can be explicitly specified by users, or implicitly deduced by the compiler. In both cases, the type conversion operation is automatically inserted before invoking the expected function or procedure.

Coercion is not true polymorphism either. An operation may appear to accept arguments of many types, but a type-conversion operation has to be applied first before the operation can use them. Therefore, the operation actually has only one type.

However, coercion allows users to omit semantically needed type conversions. When properly used, such a form of abbreviation brought by coercion may reduce source program size and improve program readability.

For example, considering the algorithmic addition operator "+" in an OO programming language, we may have the following situations when only considering INTEGER and REAL [Car85]:

```
(1) 3 + 4
(2) 3.0 + 4
(3) 3 + 4.0
(4) 3.0 + 4.0
```

Without coercion, INTEGER class and REAL class will have to implement two overloaded operations "+"[1] each. After introducing coercion, it is possible to have INTEGER class implement INTEGER addition (which applies to the only case where both argument are INTEGERs), and REAL class implement REAL addition (which applies to the situation where one of the arguments is REAL).

4.1.2 Parametric Polymorphism

Parametric polymorphism is obtained when a function works uniformly on a number of types; these types usually have some common structures. In parametric polymorphism, a polymorphic function may have implicit or explicit type parameters that stand for the types of the arguments for each instantiation of the function.

Polymorphic functions are also called *generic functions.* Ada's generic packages and C++ templates are cases of generic functions. It is not the generic function that is executed directly, but the application of the generic function by instantiating appropriate formal parameters of the generic function. Therefore, from a programming language's point of view, generic functions are second-order notions, i.e., they are not run-time notions.

Unlike overloading, parametric polymorphism is a true polymorphism in the sense

[1]For a commutative operation, the receiver of the message can be either one of the arguments determined by the compiler or interpreter

that the same code of a parametric function will be executed for arguments of any allowed types after it has been instantiated.

4.1.3 Inclusion Polymorphism

Inclusion polymorphism is used to model subtypes and inheritance. The idea of a type being a subtype of another type has already existed in traditional programming languages such as Pascal, where a subrange type is a subtype of an ordered type such as INTEGER. In object-oriented programming languages, inclusion subtypes are applied to more complex data structures (classes). For example, a type representing **Rectangle**, can be made a subtype of a more general type **Polygon**. Every object of a subtype can be used in a supertype context, in the sense that every Rectangle is a Polygon. Therefore, a variable declared as a **Polygon** can be used to refer to instances of **Polygon** and also to refer to instances of any subtype of **Polygon** such as **Rectangle**.

In inclusion polymorphism, the functions (operations) defined for the supertype can be applied to this type and also to all the subtypes of this type. In the above example, any operation defined by class **Polygon** can be uniformly applied to **Rectangle**, unless it has been redefined by the class **Rectangle**.

4.2 Reuse by Function, Package and Class

Software reuse has been categorized in different ways: 1) product reuse, 2) human resource reuse, and 3) software components reuse, etc [Mey87]. With software components reuse, the unit of reuse has evolved from routines (procedures or functions) or routine libraries, to modules or packages (Modula-2 [Wir83] and Ada [DOD83]), to classes in object-oriented programming.

4.2.1 Routines

A "reusable routine" is a single function which can be directly linked into a program. Programmers do not need to code it repeatedly line by line. Functions such as data conversions, arithmetical functions, and statistical routines, have been utilized since the early days of programming.

The limitations of reusable routines are that:

- The range of reuse is limited to a few domains, such as mathematics, statistics, basic data structure operations, I/O routines.
- It remains a very low level of reuse. There are no good criteria to organize the related operations together. In most cases, the reusable routines are organized into libraries according to functionalities.

4.2.2 Packages

Ada and Modula-2 offer facilities to build modules or packages, which are units of a group of associated functions for particular data structures. The organization of functions has shifted from individual function to a group of related functions.

One of the limitations of reusable packages is that, short of introducing the inheritance mechanism, redundancy becomes inevitable.

Embley and Woodfield developed an environment that can record the Ada packages along with the relations – general/special, generic/instantiated, etc, – among these packages [Emb87].

4.2.3 Classes

Classes and class inheritance overcome the shortcomings of the previous approaches. The advantages of using classes are reflected in the following aspects:

- A class is a unit of data structure and associated operations. It provides a good way to organize related functions together.

- The encapsulation mechanism supplied by classes protects the object's data from arbitrary and unintended uses, which among other things, protects the object's data from corruption (see Section 2.2.1)

- The inheritance mechanism reduces the redundancy between classes by using subclass relations.

Chapter 2 and 3 have shown the advantages of reuse offered by the object-oriented model – mainly the concept of class and inheritance (inclusion polymorphism). In the next section, we see how genericity plays the role in reuse.

4.3 Genericity and Reuse

The fundamental idea of using parameterization is to maximize program reuse by storing programs in an as general form as possible. Programmers can later construct a new program module by instantiating appropriate parameters associated with the generic module. Such capacity is available in Ada in the form of generic packages, and in C++ in the form of templates.

Usually, the formal parameters of polymorphic functions are restricted to *type parameters*, or some *value parameters*. The actual application of a polymorphic function is obtained by substituting the formal type and/or value parameters.

The generic functions concept has been extended to classes *generic classes* – when genericity and object-orientation are combined together. In this case, the generic parameters are applied not only to one individual function but also to the group of functions (operations) that are associated with the generic class. For example, the generic class **Array(element-type: type-parameter; size: INTEGER)** has one formal type parameter, **element-type**, and one value parameter, **size**, which may appear in the operations associated with class **Array** such as **Add(), Delete(), Get()**, etc. Examples of instantiation of application of the generic class **Array** include **Array(REAL, 20)** or **Array(Person,100)** where **Person** is a user-defined class.

One of the assumptions of genericity is operation overloading. With overloading, the client programmer may write the same code when using different implementations of the same data abstraction. With genericity, the implementer may write a single module for all instances of the same implementation of a data structure, applied to a number of types of data structures. i.e., the operations related to the generic parameters within a generic operation have to be overloaded for the data types that the generic operation's formal parameters can accept.

Another assumption of genericity is that all the operations related to the generic parameters within a generic operation should be valid, i.e., those operations should be defined by the types that instantiate the formal parameters. However, neither Ada's generic packages nor C++'s templates features reliable use of parameterized programming because neither has the mechanism to check whether the substitutions are valid or not. OBJ and its descendant OBJ2, developed by Goguen, [Gog84, Fut87] use the concepts of *theories, views* and *module expressions* to meet such needs. The instantiation of a parameterized module to an actual parameter, using a particular view will result in creating a new module. In OBJ and OBJ2, the only top level entities are "modules", which are either "objects" or "theories", and "views", which relate theories to modules. Objects contain executable code, while theories contain nonexecutable assertions Theories and views make it possible to construct varieties of module expressions. Specifically:

- *Theories* are used to define the properties required of an actual parameter for it to be meaningfully substituted for the formal parameters of a given parameterized module.

- *Views* are used to express that a given module satisfies a given theory in a particular way.

- *Module expressions* are used to modify modules, by adding, deleting or renaming functionality.

An important feature of Goguen's parameterized programming permits parameterized modules to be modified, either before or after instantiation, so that they may be applied to various applications. The possible modifications of a module include: 1) enriching it by adding new functionalities; 2) renaming some of its functions, and 3) restricting it by eliminating some of its functions. OBJ's module expression mechanism accomplishes these modifications, and guarantees the satisfaction of selected program properties given in the form of theories.

4.4 Our Extension of Genericity – All Application-Specific Classes are Parameterized

Although genericity has been used in building a library of reusable components for some application domain, not only for the basic data structure components, it is only meaningful in an organization that has a long-term management commitment to an official reuse plan. In reality, genericity is more suitable for basic data structure-like, application-domain independent, low-level components. This use of genericity frees us from repeatedly coding purely computing-related, accessor/selector-like operations. When genericity is extended to classes in object-oriented programming, we can benefit more from their combination.

4.4.1 Extending Generic Parameters to Name Parameters

Traditional generic parameters are restricted to *type parameters*, or *value parameters* We can extend the generic parameters to *name parameters* usually the field names of record types. For example, we may define a generic class **Record1** by

Record1 = Record(field1 : STRING; field2 : INTEGER; field3 : Array(element-type))
where **element-type** is type parameter, **field1, field2, field3** are name parameters.

An example of application of the generic class **Record1** is **Owner = Record(Name : STRING; ID : INTEGER; Books : Array(Book))**. Such a class could be obtained by instantiating all the formal parameters defined by **Record1** including traditional type parameters – **element-type** substituted by **Book** and the newly introduced name parameters – **field1, field2** and **field3** substituted by **Name, ID and Books** respectively. Another class **Course = Record(Title : STRING; Credit : INTEGER; Students : Array(Student))** can be obtained in the same fashion.

The introduction of name parameters makes it possible to instantiate a number of application-specific classes by a single generic class. We will see the significance of this concept in Chapter 6.

4.4.2 Introducing Inheritance to Generalized Classes

Parameterized polymorphism can be viewed as a *horizontal reuse*. A generic class **LinkedList(element-type)** is the common structure of all the applications of **LinkedList** such as **LinkedList(INTEGER)**, **LinkedList(STRING)**, etc. After introducing the name parameters, any application-specific class can find its underlying generic class common with some other application-specific classes. For the example mentioned above, generic class **Record1** is the same underlying generic class for class **Owner** and class **Course**

Inclusion polymorphism is a mechanism for *vertical reuse*. Any operation defined at a superclass can be uniformly applied to its subclasses unless it is redefined by its subclasses. Generic classes also have inheritance relations among them. "Sub-generic-classes" inherit computation-semantics (data components and related operations) from "super-generic-classes". We will see the details in Chapter 6.

Before moving to the detailed discussion of our approach in the next part of this thesis, in the next chapter, we give an overview of *SoftClass* project in which this research is carried out and our approach is realized.

Chapter 5

SoftClass Project

SoftClass is an object-oriented CASE tool developed in the context of a software reuse project financed jointly by the *Centre de Technologie Tandem de Montreal* (CTTM, an R&D division of Tandem Computers Inc.), and government research granting agencies. CTTM specializes in developing systems for managing Tandem's distributed applications. The orientation of the project changed during the first months. Initially, we intended to develop an object-oriented development methodology based on an object-oriented model tailored to specifying what might be called "nearly decomposable" systems of collaborating agents [Mil90]. However, due to a number of factors, including the lack of support for object-oriented languages on Tandem computers at the time, and the expected lengthy feed-back loop for such a project, we chose a two-pronged approach: 1) theoretical work on OO software development, and in parallel, 2) exploring software adaptation procedures (i.e. specialization, composition, and transformation, see section 5.1), as independently of the underlying software development technology as possible, while ensuring that the *abstractions* embodied in the model described in [Mil90] are supported; hence the strong object-oriented flavor in both the underlying principles and implementation of SoftClass. In section 5.1, we provide a brief discussion of issues in software reuse. Section 5.2 provides an overview into SoftClass. The representation of software components in SoftClass is described in section 5.3. Section 5.4 describes SoftClass's representation of development knowledge in general, and development traces for specific products, with a view towards automating maintenance. In section 5.5, we briefly describe the specific contribution of this work to SoftClass.

5.1 Software Reuse

Software reuse is seen by many as a key factor to improving software productivity and quality [Fre87]. The software engineering literature abounds with horror stories about low programmer productivity, poor software quality, and poor management of large software projects [Tha81]. Several factors hinder software reuse including, the infancy of software development as an engineering discipline, market pressures, management shortsightedness with regard to consenting to the initial investments needed to implement software reuse methodologies, personnel training and education, and the lack of methodologies and tools to support software reuse in particular and software development in general [Fis87]. Research on reusing existing software has traditionally focussed on reusing code, and tended to focus on library-oriented issues and techniques [Pri87]. While substantial code reuse rates were sometimes achieved [Gru88], productivity gains remained marginal because of the diminishing share of coding in the overall development lifecycle [Hor84].

Interest in reusing software since the earlier stage of development spurred a host of new research problems and a myriad of solutions [Fre87]. In [Mil90], Mili provided a general framework for the study of software reuse in an attempt to identify the important research issues and categorize existing research in software reuse. Viewing software development as a knowledge intensive problem-solving activity, software development involves using and reusing a body of development knowledge which increases with experience. AI researchers and cognitive scientists have long divided problem-solving knowledge into procedural knowledge (*skills*) and declarative knowledge (*facts* and *semantic knowledge*). In the context of software development, *skills* consist of development procedures, such as the process of deriving structure charts from functional diagrams in structured design. Semantic knowledge consists of things such as application-domain knowledge, knowledge of programming concepts (e.g. algorithms, data structures and their properties), while "facts" consists of instances of previously solved "problems". Accordingly, we identified a spectrum of reuse methodologies based on the range of knowledge reused, ranging from procedure-oriented reuse to data-oriented reuse (i.e the reuse of facts). Procedure-oriented reuse is similar to Horowitz's *reusable processor* concept [Hor84] and includes things such as automatic programming where development knowledge is embodied in automated procedures that transform specifications in one form or another into executable programs [Els82]. Data-oriented reuse focuses mainly on computer-based memorization and recall of past realizations of software products and is best exemplified by code reuse efforts which focus on library-based techniques. In the between, we have a whole range of knowledge-based approaches which apply more or less heuristic transformations to more or less generic development templates as in [Bas87]. In this paper, we deal with the data-oriented end of the spectrum.

Roughly speaking, data-oriented reuse involves two major issues: 1) retrievability of existing software components, and 2) the adaptability of those components. Retrievability of software components involves content-based indexing for which we need a good indexing vocabulary. The quality of indexing vocabularies involves a set of *objective* criteria as well as *subjective* ones. Objective criteria include: 1) the coverage of the vocabulary, i.e., the extent to which all the concepts relevant to the domain of discourse are represented, 2) the precision of the vocabulary, to the extent that it discriminates between distinguishable concepts, and 3) consistency, i.e., the extent to which each concept is represented by a single index term. Subjective criteria relate to the ease of use of the indexing vocabulary and have to do with things such as the choice of wording for indexing concepts, and the ease with which a user can locate the exact wording for a concept he is looking for. Designing indexing vocabularies for software components is a major concern in the relatively recent research trend generically described as *domain analysis* [Pri87].

Retrievability also requires sophisticated retrieval algorithms. Within the realm of controlled vocabulary indexing of software components, classification has been shown to yield superior results to boolean retrieval [Pri87]. Prieto-Diaz used additional heuristics (*biases*) to break ties, by taking into account reuse related attributes such as the length (and thus complexity) of retrieved components and the experience of the "reuser" [Pri87]. Keyword-based classification has been considered even in object-oriented programming environments to complement browsing of class hierarchies [Hel91]. SoftClass uses a keyword-based classification of software components, as well as a number of graph-based concept matching algorithms [Mil93b, Mil93c].

Depending on the kind and level of genericity of usable components, the adaptability of reusable components may take one of several forms:

1. ***specialization***: a generic or parameterized software component is somehow specialized to take into account the specifics of the problem at hand. With template-based approaches (e.g. [Bas87]) program templates are *instantiated* for specific parametric values In object-oriented programming, subclassing is used to drive new classes from existing ones. Inheritance has historically been associated with specialization, although only restricted forms of inheritance imply specialization in the real sense (e.g. code wrappers in Flavors [Moo86]).

2. ***composition***, whereby reusable components are reassembled to satisfy new requirements, which are not satisfied, closely or remotely, with any individual component. Work on module interconnection languages addresses reuse by composition [Pri87], and relies on the standardization and transparency of module interfaces. For these reasons, reuse by composition is one of the favorites for object-oriented reuse theoreticians, as exemplified by the works of Kaiser

[Kai87] and Goguen [Gog86].

3. *transformations*, which deal mostly with modifying existing, non-generic, software components to adapt them to new requirements. There has been relatively few efforts in this direction, with some notable exceptions [Ara86].

SoftClass is built, in part, to explore heuristic software transformations using analogical operators.

Data-oriented reuse, be it with OO classes, development templates, or project-specific, non-generic, software components, depends on the existence of a library of reusable components. Retrievability requires an attributed representation of software components. Adaptability requires an explicit representation of those functional and structural aspects of software components that need to be manipulated by adaptation procedures.

5.2 Overview

The SoftClass project was aimed at enhancing software reuse for distributed management software the industrial sponsor's main line of products. We were to prioritize one of two alternative research goals: 1) research and develop methodologies for reusing existing software developed using structured methods, and 2) research and develop methodologies and tools for developing reusable software– with a keen interest in object-orientation. Both were retained, but with an emphasis on former during the first half of the project (just ended), and an emphasis on the latter in the second half, to accommodate both near-term returns (the former) and long-term ones (the latter). Providing for a paradigm shift half-way through the project posed a number of problems, both conceptual and practical, which slowed our progress on the first front, but that, we hope, will pay off handsomely in the long run if our sponsors don't run out of money and/or patience in the meantime!

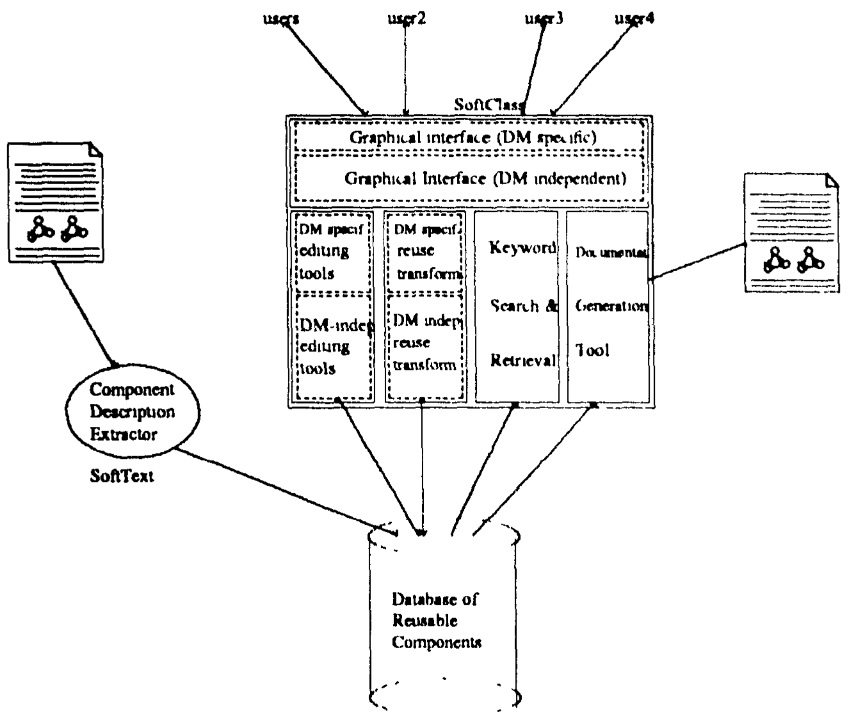

users
user2
user3
user4
SoftClass
Graphical interface (DM specific)
Graphical Interface (DM independent)
DM specif editing tools
DM-indep editing tools
DM specif. reuse transform
DM indep reuse transform
Keyword Search & Retrieval
Documentat Generation Tool
Component Description Extractor
SoftText
Database of Reusable Components

Figure 5.1: SoftClass's overall architecture. SoftClass was developed in Smalltalk80. Its interface to the database (INGRES) is written in C. SoftText and SoftIndex are developed in Lisp and C. "DM" means "Development Methodology". Because of SoftClass's object-oriented architecture, DM-specific tools are subclasses of the corresponding DM-independent tools in which we abstracted the common functionalities. The "functional" separation between the various components is in some cases "mental" as different functionalities may be implemented by the same class.

Early in the project, we focused on the development of a technology-independent representation of software components that can handle both structured method abstractions, and object-oriented abstractions, at any level of development. In SoftClass, the term "software component" refers to incarnations of software products at different stages of development, including requirements, analysis, and design. Software products consist of either procedures or data, and the model does not distinguish between the two. Work on reuse techniques focussed on: 1) repackaging existing (structured) software to fit the chosen model, and 2) computer support for design with reusable components. The sources considered for existing software consisted of traditional software documentation (e.g. requirements documents), as well as outputs of CASE tools. With software documentation, repackaging relies on: 1) a tool called

SoftText that uses a theoretical model of technical writing to extract a skeleton software architecture, and 2) a simple automatic indexer– called *SoftIndex*– that matches parts of the document to specific vocabularies to support later retrieval. SoftText complements the extracted information with CASE tools export files, when such files are available. Figure 5.1 shows an overview of the SoftClass tool set.

Support for design with reusable components is embodied in a CASE tool, also called *SoftClass*. In addition to issues of component classification and retrieval, we attempted to address the following problems:

1. Given that no component was found that closely matches the requirements, which combination of components (and in what combination) might satisfy them, if any?

2. Given the (partial) description of a desired software component (presented as a query to the software components library) and the closely matching description of a retrieved component, which transformations should be applied to the retrieved component so that it satisfies the desired requirements?

3. Given that a transformation has been applied to a retrieved component at a given development stage (e.g. analysis), which transformation(s) should be applied to its subsequent level descriptions (e.g. design)?

Focus on these problems was motivated by empirical data suggesting that developers are quick to fall back on developing from scratch when a reusable component requires non-trivial modifications.

The first problem can be seen as a retrieval problem, and provides the basis for bottom-up development. An exact solution to this problem is impractical, if not theoretically impossible, although heuristic methods can be developed that provide potential solutions. Except in trivial cases, the second problem is notoriously difficult: choosing the right transformation often requires a very precise specification language, and knowledge about programming and about the application domain. However, once a transformation/modification has been enacted by the user at a given development stage, a "trace" of the subsequent development decisions can help automate the propagation of these transformations across development stages (third problem). SoftClass uses mappings to record development transformations. For those development transformations/activities that can be automated, the mappings are given in a functional format, and when triggered, automatically produce the target descriptions. For those development transformations/activities that require developer intervention, the mapping is recorded after the fact. In either case, they enable us to predict the effect of *local* changes at one level of development on successive levels.

We have made forays into object-oriented software development The author has developed representations for generic abstract data types at the analysis and design level, with support for inheritance and automatic classification for newly defined types. Further, the author has also developed a compiler for an algorithm design language (PDL) that allows some measure of detailed design validation versus architectural design validation, which is already supported

For the remaining part of the project, work will focus on: 1) domain analysis, and 2) the development of an object-oriented development *methodology* and tools to support it.

5.3 Representing Software Components

In SoftClass, the software components are described by: 1) *global attributes*, and 2) an *internal structure*. Global attributes includes things such as "Function", "Performance Requirements", "Date of Creation", "Author", as well as the external interfaces (input/output variables) of the component. They cover the information embodied in the administrative and black box description part of questionnaires, but provide much richer semantics as shown below. Attribute "values" consist of: 1)

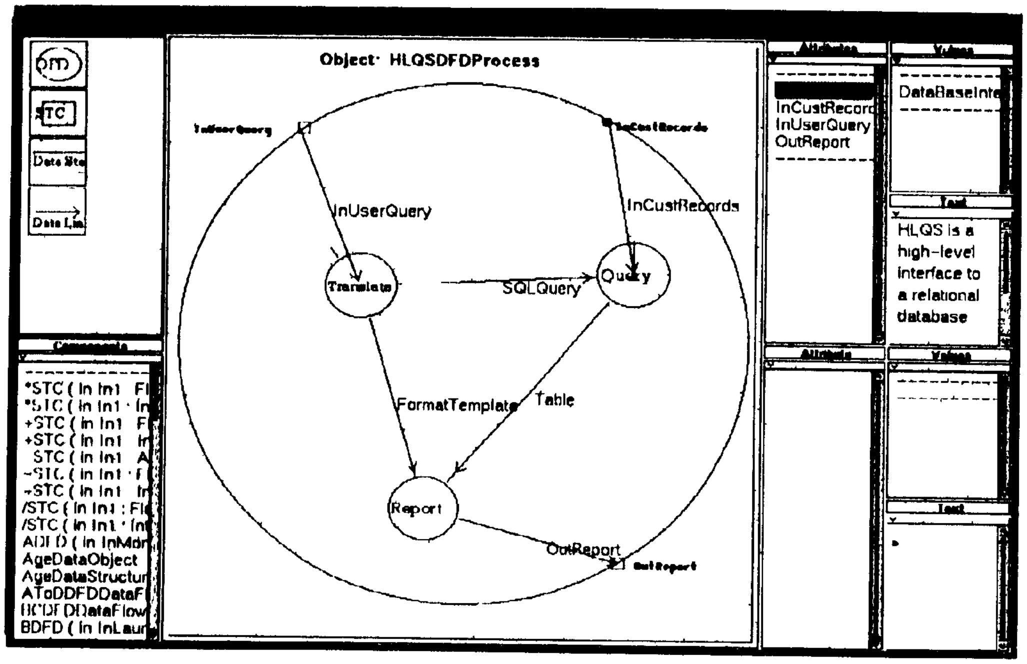


Figure 5.2: Graphical representation of the internal structure of a process-like component using SoftClass's graphical component editor.

terms from a controlled vocabulary, and 2) a textual description. The terms belong to predefined (or computable) semantic hierarchies (i.e. taxonomies), and support various retrieval algorithms [Mil93a]. The internal structure is similar to the clear-box description part of questionnaires, with the following differences: 1) data flows are considered as distinct software components, and not simply artifacts for connecting software components, and 2) the relationships between a component's subcomponents are represented by explicit data structures manipulated by the CASE tool (SoftClass). Referring to the HLQS example in Figure 5.2, HLQS has 6 software components, 3 links and 3 nodes, we call them Tr (Translate), Qy (Query), Rt (Report), SqQy (SqlQuery), FtTe (Format-Template), and Tbl (Table). The internal structure of HLQS includes the relations/bindings between the various components, and between the components and HLQS itself, as shown below:

R_1 **User-Query(HLQS) = I1(Tr)**
R_2 O1(Tr) → Origin(SqQy)
R_3 Dest(SqQy) → I1 (Qy)
R_4 O2(Tr) → Origin(FtTe)
R_5 Dest(FtTe) → I2(Rt)
R_6 **Database(Qy) = Database(HLQS)**
R_7 O1(Qy) → Origin(Tbl)
R_8 Dest(Tbl) → I1(Rt)
R_9 **O1(Rt) = User-Report(HLQS)**

where I1, I2 (O1, O2) represent input (output) variables of process-like software. Origin and Dest (for destination) represent the end points of data flow-like objects. The equalities (shown in bold) *relate HLQS 's global attributes to its components global attributes*, and are called *In/Out interface relations*.The remaining are called *value propagation relations* and are internal to HLQS. This rather cumbersome notation has advantages. For instance, both kinds of structural relations entail some compatibility constraints between the various input/output variables and the data flows. In SoftClass, specifying a value-propagation relation (e.g. by binding a data flow to a process) triggers such constraints, which notify the developer if a violation (i e. data incompatibility) is detected. Other constraints can be triggered at will, providing an "on-the-fly" development validation. Further, developers need not manipulate such relations since they are created automatically from drawings using the graphical interface (see below).

In SoftClass, all software component descriptions are instances of description templates, or *categories.* Roughly speaking, for a given development methodology, we have one category for each type of component (e.g. process/procedure versus data), and for each level of development. Categories are themselves arranged in a (Smalltalk) class hierarchy to take advantage of similarities between categories. Each category is characterized by: 1) a set of relevant attributes and their semantics, 2) a set of permissible component categories, and 3) a set of permissible generic internal relations. For example, in SA/SD, the *type* of data objects (an attribute) applies only to design level descriptions. Also, a data object can be decomposed into more elementary data objects (permissible component types), while processes can be decomposed into more processes and data flows, and the structural relations will be of different nature, with an emphasis on aggregation for data objects, and functional composition for process-like components. SoftClass provides a fairly simple to use tool for defining/editing categories, called *SoftCategoryBrowser* Such a tool would typically be used by administrators. [1]

SoftClass categories supports far richer semantics for attributes than those offered in questionnaires. For a given attribute and a given category, we can specify:

1. the indexing vocabulary for the attribute. When that vocabulary is a taxonomy, it suffices to specify the root,

2. the number of keyword values allowed (cardinality constraints) for a given instance (software component) of the category, and in case several are allowed, compatibility constraints between the value, if any,

3. whether the attribute is mandatory or optional. When defining instances of the category, developers are automatically prompted for mandatory attributes, but must explicitly bring up optional attributes,

4. whether an attribute is shared between a component and its subcomponents, in which case its values are automatically filled for the subcomponents, and

5. whether an attribute remains invariant across development stages, in which case its value is automatically filled for subsequent level descriptions. For example, the "purpose" of a software module remains the same, independently of whether we describe the module at the analysis or the design level. It suffices to specify it for the analysis level.

[1] For those familiar with Smalltalk80, the tool is very similar (actually inherited from) Smalltalk's class browser

The last three options considerably alleviate the process of specifying software components. For example, a study of design-level objects based on IEEE documentation standards and on the software development methodology used at the industrial sponsor, identified some thirty four (34) attribute for procedures. Twenty five (25) of those can be inferred from related descriptions. Of the remaining nine, two can be inferred from other knowledge in the system, and three are related to quality control This leaves us with the familiar "Purpose" application-dependent function), "ComputationalProcedure" (algorithmic function), Inputs and Outputs for an overwhelming majority of components. And yet, a total of 34 attributes can generally be used for search purposes.

In the current implementation of SoftClass, software components can be specified either through a tool similar to the one used for specifying categories[2] or through the graphical interface (see Figure 5.2). The interface has been designed in such that a way that it can easily support other notations/development methodologies such as object-oriented methodology. Users can bring up components on display by either: 1) selecting and dragging a button (one per category) from the palette (upper left), and supplying the name, or 2) selecting and dragging a name from the 'Components' list (left bottom). The right side shows two "browsers" for the attributes of the components selected in the tool, the top part for the currently selected component in the display, and the lower part for the current selection in the 'Components' list. The lower part shows the attributes for HLQS, the object being edited. The 'Values' list shows the list of keyword values for the currently selected attribute ('Function'). The list shows the value 'DataBaseInterrogation'. The 'Text' part is an editor to enter a textual description of the attribute Users can add/remove keyword values. When adding values, the user is presented with a hierarchical pop-up menu of the keyword hierarchy form which he can select a value.

[2]Such a tool is too cumbersome for components, and not visual enough Categories are defined once in a blue moon, and the tool is adequate for that purpose

5.4 Representing Pairs of Products in SoftClass

For the purposes of supporting the reuse paradigms, we need to represent mappings between descriptions of software products at successive levels of development. Consider the following scenario: assume that the developer starts the structured analysis of HLQS. Assume that the users required that queries be entered in a restricted natural language format. The developer first specifies the attributes of HLQS, such as its function, intended users, and so forth. Assume that a QBE (Query by Example [Zlo75]) interface has already been developed in-house. Before proceeding to decompose HLQS, the developer searches the database of product descriptions for one that matches HLQS (product retrieval is discussed below). The search returns the analysis level description of QBE, whose attributes match most of those of HLQS (e.g. all but the format of the user's query) While the developer cannot reuse the description of QBE as-is, he reuses the structure of QBE and adapts it to HLQS. In particular, he reuses the modules Query and Report as-is, but replaces ScreenInput of QBE by a new component Translate, the two differing in the format of users' input (see Figure 5.3).

As the developer proceeds to designing HLQS, he knows that he can reuse the design of Query and Report developed for QBE, but doesn't quite know how to arrange them, along with the design of Translate, in a structure chart. By analogy to the difference between the DFD decomposition of HLQS and QBE namely, replacing ScreenInput by Translate, he infers that HLQS and QBE have a similar structure, and has only to worry about designing Translate (see Figure 5.4). For this inference to take place, the developer needs a mapping between the structure of QBE at the DFD level and that at the (preliminary) design level. Such a mapping must include both mappings between DFD level and design level representation of software components, but also a mapping between their inter-relationships. At the representation level, both mappings between individual software components and mappings between

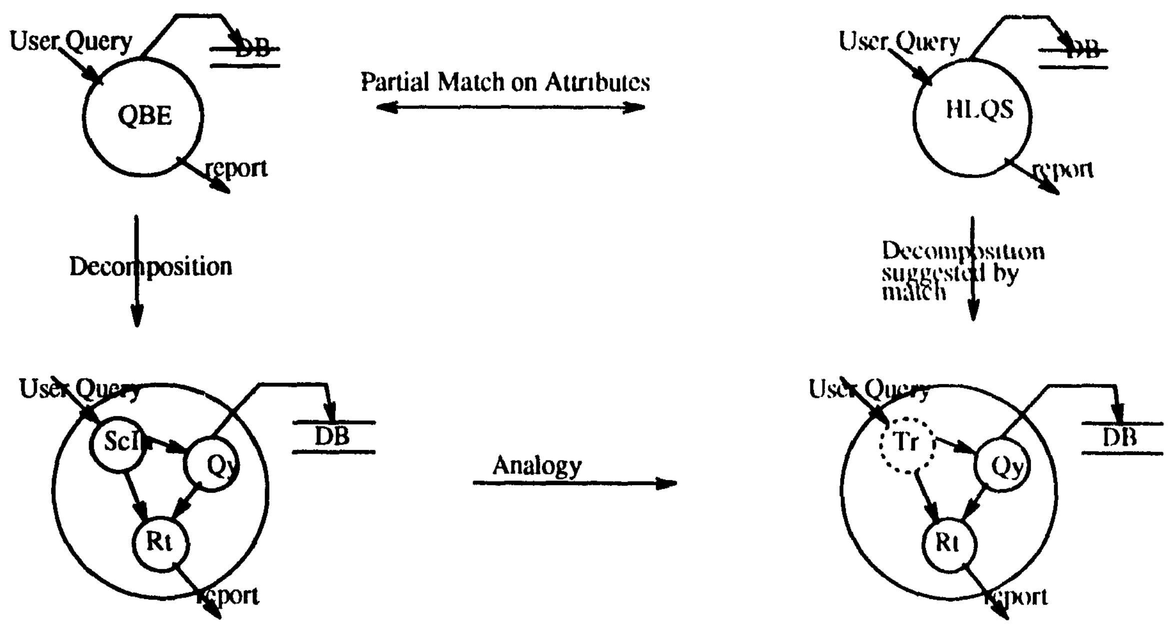


Figure 5.3: A partial (but substantial) match between the attributes of QBE and HLQS suggests adapting the structure of QBE to that of HLQS

components' inter-relationships can be represented by relation objects (i.e. subclasses of RelationObject above). The semantics of these relations depend on the development methodology at hand. For a given methodology, we define a kernel of generic mappings which may be combined in various ways, in the same way that the structural relation MultipleBind was defined as the combination of simple binary bindings between two objects' attributes. We have developed a kernel of such mappings for SA/SD. We shall illustrate it below.

As a first approximation, "bubble"s[3] of a DFD end up as modules ("boxes"s[4]) at the preliminary design level. A mapping between DFD bubbles and design modules establishes consistency requirements between values of their attributes. For instance, while the sets of attributes applicable to these objects are different, there is a significant overlap. Attributes such as "Function" (or purpose) and "IntendedUsers" are applicable to both analysis level and design level objects, and their values remain

[3]a notation for process/procedure components at the analysis level

[4]a notation for process/procedure components at the preliminary design level

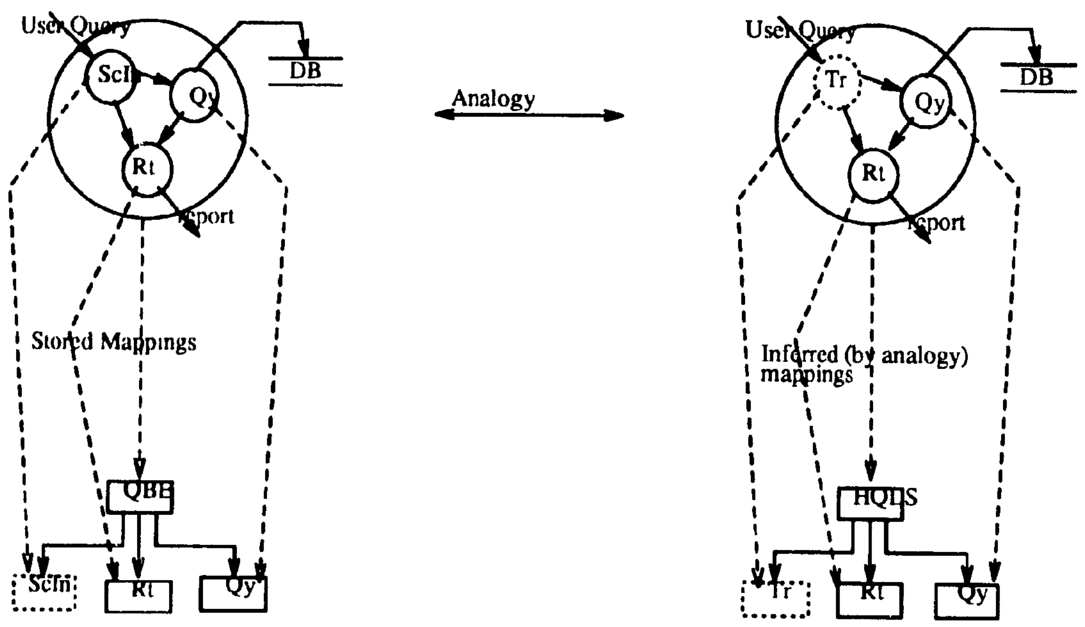


Figure 5.4: analogical reuse requires mappings between successive level representations of software components, and mappings between their interrelationships. Only mappings between individual components are represented graphically (dashed arrows).

unaffected by the design process. However, attributes such as "Performance" evolve. While they represent a requirement (upper limit) at the analysis level, they become measurable at the design level once the algorithm and data structures manipulated are designed. Notice that some analysis level attributes do not apply at the design level, and conversely, some design level attributes have no equivalent at the analysis. Such attributes are not handled by the mapping. Let M_Process$_{DFD \rightarrow STC}$ be the (binary) mapping between process objects at the analysis and design levels. We have:

$$((X,Y) \in M\_Process_{DFD \rightarrow STC}) \rightarrow (Function(X) = Function(Y)) \wedge$$

$$(IntendedUsers(X) = IntendedUsers(Y)) \wedge$$

$$(Perfromance(Y)\ IsConsistentWith\ Performance(X)) \wedge \ldots$$

Note that the above equation expresses a *necessary* condition for design object X to map to analysis object X. Nothing short of automating development would allow us to ascertain the sufficiency of these conditions[Mil91]. Such a mapping can be used in two ways: 1) the developer creates a design level representation Y for an analysis object X, then creates an instance of M_Process$_{DFD \rightarrow STC}$ with the pair (X,Y); this will check the consistency of the attributes of X and Y, or 2) the developer asks M_Process$_{DFD \rightarrow STC}$ to create an instance whose pair's first element is X; this will create object Y, and assign values to the attributes that are subject to an equality.

We have developed a kernel of mappings between DFD structures and structure charts, and transformations on structure charts based on the SA/SD development methodology [Pet87]. The reasons for this choice are twofold: 1) CTTM, the sponsor of this research, will keep using SA/SD for some time to come, and 2) SA/SD has been faulted by OO programming proponents for the "seamful" transition between analysis and design; if our model can handle SA/SD, it can handle more seamless development methodologies such as OOA/OOD [Boo91]. Figure 5.5 shows a simple mapping between DFD structures and STC (Structure Chart) structures.

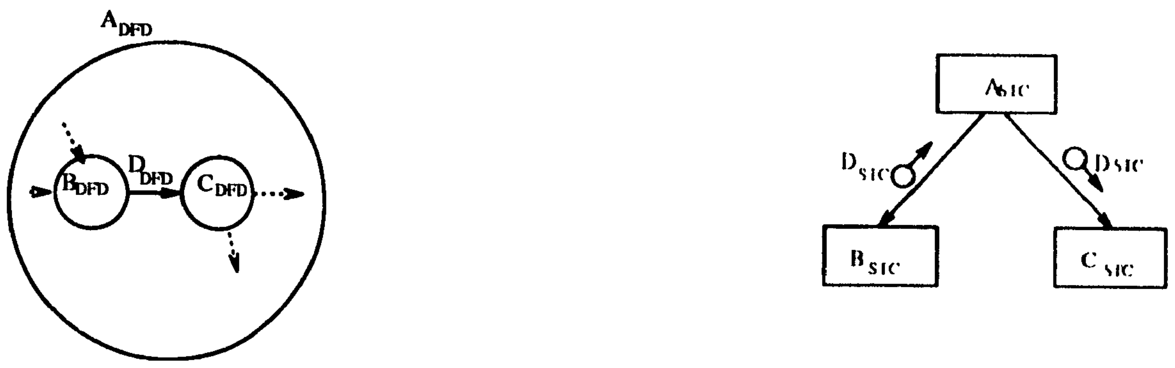


Figure 5.5: An elementary mapping between DFD structures and STC structures

Let A_{DFD} be a DFD "bubble" with two sub-components B_{DFD} and C_{DFD}, exchanging a data flow D_{DFD}. Informally, the mapping to the STC level can be described as follows: M_Process$_{DFD \rightarrow STC}$ maps A_{DFD}, B_{DFD}, and C_{DFD}

to A_{STC}, B_{STC}, and C_{STC}, respectively, and D_{DFD} is mapped to D_{STC} (by M_Dataflow$_{DFD \rightarrow STC}$), such that

- A_{STC} *calls* B_{STC} and C_{STC},
- D_{STC}, is a local variable to A_{STC}, and
- $D_{STC}$is an output parameter of B_{STC} and an input parameter to C_{STC}.

Needless to say, this mapping, used alone, trivializes structured design, and in practice, the final architectural design of a software component may bear little resemblance to the component hierarchy at the analysis level. For example, transformation-center and transaction-center design entail a post-analysis grouping of DFD bubbles (under what Peters calls "Boss" [Pet87]) based on an analysis of data flows. Further, the architecture may undergo a number of refinements as "boxes" are split and/or merged together to enhance things such as coupling, fan-in, fan-out, scope of control [You79] and others [Mil91]. However, we believe that structured design can be decomposed into a sequence of simple mappings and transformations. It is the choice of the *sequence* of these mappings and of the objects to which they are applied that makes structured design difficult and non-automatable. Of course, architectural design is only part of the design, but it is about how much can be inferred from data flow charts alone.

5.5 Contributions of this work to SoftClass

Our contributions to SoftClass are summarized as follows:

- Implementing a representation of data objects (analysis level and design level). The representation framework is consistent with that for process/procedure components.

- Supporting reuse from the analysis level. An automatic classification tool for the analysis level objects and another automatic classification tool for the design level generic data structures are built into object design facilities.

- Formalizing the transformation from analysis to design for the case of data objects.

- Implementing an algorithm (program) design language compiler for detailed level code validation.

In the next two chapters, we will present this work in detail. We discuss representations of data objects in Chapter 6 and the program design language compiler in Chapter 7, including implementation issues.

Chapter 6

Data Design in SoftClass

As discussed in the last chapter, SoftClass is an OO CASE tool aiming at software reuse. The data design facilities in SoftClass are responsible for designing data objects at the analysis and at the design level. Another important feature of the data design facilities in SoftClass is to enhance reuse of design-level data objects that have the same analysis-level semantics, even if their data representations are different. In traditional OO software development methods, such reuse is limited by the implementation inheritance hierarchy, because reused software components have to have the same *data representations* (data structures). In our approach, this restriction is relieved by reserving the analysis-level data objects semantics until the design level, and introducing the generic mechanism to all design-level data objects.

6.1 Overview

In SoftClass, we distinguish between three kinds of data objects:

1. *Application Objects* (AO), which are analysis- (specification-) level representations of the objects of the application. Application objects (AOs) are characterized by a set of application-meaningful (only analysis-concerned and only related to application domain) attributes and components, and a set of application-meaningful operations,

2. *Generic Data Structures* (GDS), which are design-level descriptions of parameterized data structures. Generic data stuctures are parameterized by both types, *à la* generic packages in Ada e.g , and component names. They also support a number of operations, mainly ones that access the components of the structure by name or by property,

3. *Application Data Structure* (ADS), which are design-level descriptions of application objects. Application data structures (ADSs) are obtained by "fitting" an analysis-level application object (AO) "over" a generic data structure (GDS). This fitting resolves/instantiates the parameters of the GDS, and enables developers to sketch an algorithmic "realization" for the application-meaningful operations.

Application objects only carry the application semantics and the behaviour descriptions. GDSs are building blocks or templates for application data structures which combine application-semantics from application objects with representation and operations from GDSs to fulfill the behaviour specification of application objects into operations.

Application objects are organized in an inheritance hierarchy based on common application-level semantic contents (attributes and/or components) and application-level operations. Generic data structures are organized in an inheritance hierarchy based on common data types and organization in the programming sense and access operations, i.e., based on shared *implementation* characteristics. Application data structures, which correspond to the joining of a data object and a generic data structure, may be seen as belonging to two parallel hierarchies from which they inherit different things: they inherit application semantics from the AO hierarchy, and design/implementation semantics from the GDS hierarchy.

For any pair of application data structures, four situations instead of the usual two – may occur:

1. they share application semantics and implementation semantics,

2. they share only application semantics,

3. they share only implementation semantics, and

4. they share neither.

While the first three cases present – to different degrees – opportunities for code reuse, only in the first two cases would the system automatically put the pair in a class-subclass relationship. Further, only the first case is 100% safe. Some manual double-checking is required in the second case to ensure that the inherited operations would function correctly in the subclass, the operations that can be reused are application-specific and data structure-independent operations or high-level operations. In the third case, reuse takes place by re-"instantiating" the inherited generic operation with the appropriate name and type parameters. In this case, low-level, access-like or purely data structure-specific operations are reused. Intuitively, the GDS hierarchy enhances code reuse by abstracting out application-semantics from ADS classes; while the AO hierarchy preserves the application-semantics into the design and implementation stage, and promote code reuse by inheriting application-semantics without having same underlying (generic) data structure.

Notice that the operations attached to data components are represented by process-like components, as shown in Chapter 5. In particular, operations may be decomposed into more elementary operations. Further, the entire apparatus set up in SoftClass for SA/SD-like processes such as architectural representation (structure charts), analysis → design mappings, search tools, etc - may be applied to the operations attached to data objects. Accordingly, operations have a "global scope" in SoftClass, and in order to enable different data objects to support different versions of the same operation (overloading), we had to implement "overloading" within the tool itself, i.e., allow two different processes to have the same "user-oriented name", i.e., the name of the component as presented by SoftClass to the user, versus the internal name of the component. For a number of theoretical and practical reasons, software components are represented by Smalltalk classes. Up until we tackled the representation, the internal name of a component (class name) consisted of the "user-oriented" name of the component, concatenated with the name of its category (see Section 5.2), to distinguish between its representations at different stages of development. For example, the component HLQS "HLQS" is the user-oriented name shown in Figure 5.1 is represented by the class HLQSDFDProcess. Its representation at the architectural design level would be a class called HLQSSTCProcess. This nomenclature was no longer appropriate when we allowed operations supported by different data objects to have the same "name". We had to add a variable to the representation of software components called "stringName", which is the "user-oriented" name of the component. Hence, internally, two components may have different (Smalltalk class) names but have the same "user-oriented" name. SoftClass distinguishes between them based on their internal name; the user can distinguish between them based on their signature.

In the remainder of this chapter, we discuss the three kinds of data objects in the order given above. For each kind of data object, we discuss the principles underlying its representation and the SoftClass functionalities and tools used to manipulate it.

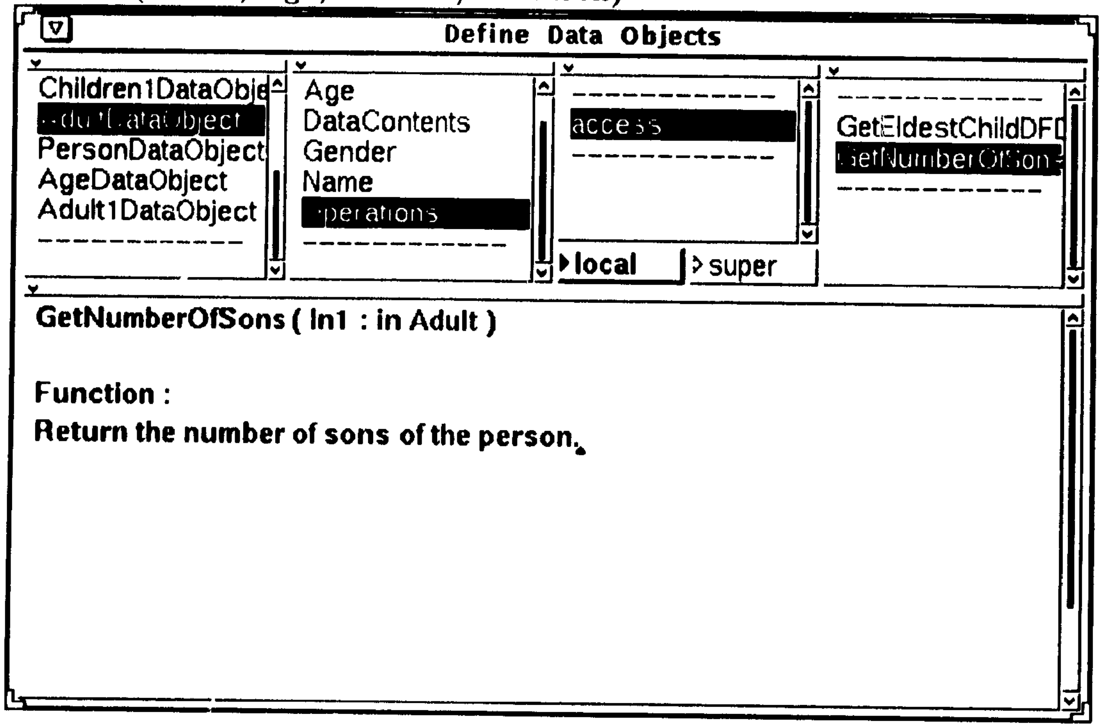
6.2 Application Objects

6.2.1 Application Objects

As mentioned above, AOs are analysis level representations of the data objects within an application. At this level, one is concerned with implementation-independent, semantic properties of data objects. As mentioned in Section 5.2, all software components in SoftClass are described by their attributes and their internal structure. In this case, the attributes of a data object correspond to its application-meaningful attributes, in the sense of OO analysis methods such as [Coa91].

For example, assume we want to design **Adult** AO in a particular application, we may get:

Adult (Name, Age, Gender, Children)



GetNumberOfSons (In1 : in Adult)

Function :
Return the number of sons of the person.

Figure 6.1: Tool for defining application objects. The left view contains list of application objects known to SoftClass. The list in the second view from the left contains the attributes for the currently selected application object (Age, Gender, Name, and Children), as we well as "DataContents" and "Operations". By selecting "Operations", the third list from the left shows the list of protocols. In this case, there is only one ("access"), and that protocol has two operations. The lower text window shows the "definition" of the selected operation. The "local" versus "super" buttons allow the user to switch between locally defined operations and inherited ones.

An **Adult** object consists of four attributes – *Name, Age, Gender,* and a collection of *Children*. We do not care about how to represent these four attributes at this moment. Instead we do care about the application-semantics, in particular, the behaviour of **Adult** objects needed in the particular application. We may want **Adult** to have **Get_Num_Of_Sons (Adult)** operation and **Get_Eldest_Child (Adult)** operation, etc. However, we do not need to know the details about how these operations will be implemented Instead, what we need now are only the specifications of these operations.

Figure 6.1 shows the tool (browser) for specifying application objects. In the above example, we are examining the definition of AdultDataObject, which is supposed to have four attributes, Age, Gender, Name, and Children "DataContents" is an attribute shared by all application objects used for retrieval purposes so that developers may search for an application object without knowing its name or the names of its attributes/components.

Application Objects are characterized by 1) attributes, 2) components and 3) relation between attributes and components, and 4) behaviour specifications. For the time being, the internal structure consists simply of an enumeration of the components. Further, for most information system applications, no object components are relevant to the application, as in a person's limbs in a personnel management application.

6.2.2 Application Objects Hierarchy

All the application objects are organized in a inheritance hierarchy in which sub-Application-Object will inherit the semantics (attributes, etc) and behaviour specifications from super-Application-Object. e.g., **Person, Juvenile** and **Adult** AOs may be defined as **Person (Name, Age, Gender) , Juvenile (Name, Age, Gender) , Adult (Name, Age, Gender, Children). Juvenile** and **Adult** will be sub-Application-Objects of **Person** in the hierarchy. Therefore **Juvenile** and **Adult** should inherit the semantics and the behaviour from **Person** (see Figure 6.2).

New application objects are placed in the inheritance hierarchy which is NOT a tree – in one of two mutually non-exclusive ways: 1) automatically, as new attributes are added/removed, and 2) manually, by the user/developer. Automatic classification uses the inclusion rule between set of attributes. For example, when we first created

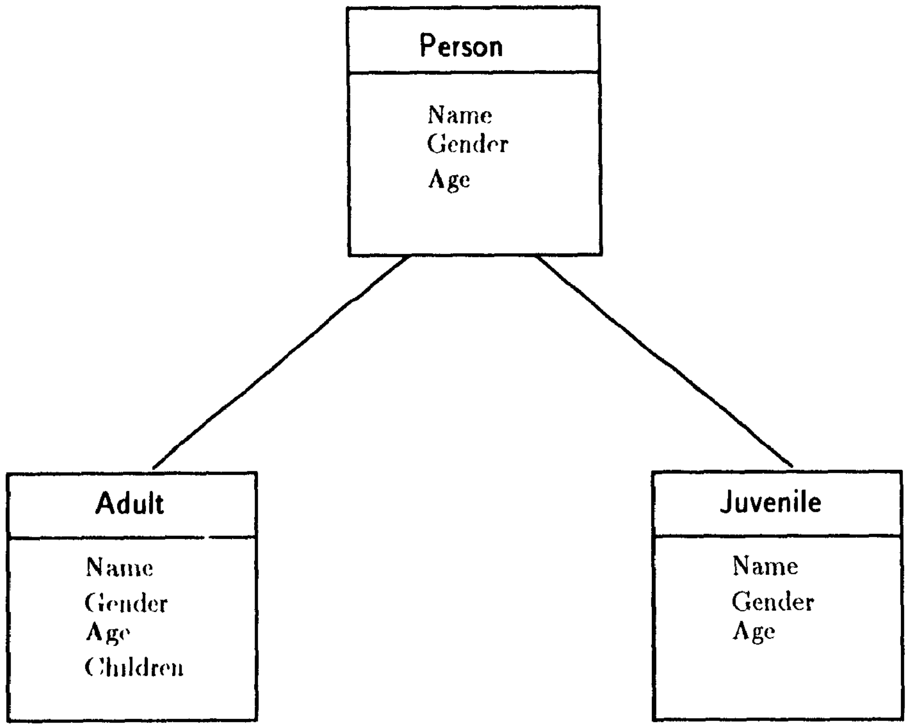


Figure 6.2: Application Object hierarchy

Adult, as new attributes are added to **Adult**, **AdultDataObject** is pushed down the application object hierarchy, becoming a child of those of its "previous siblings" that already had the newly added attribute. For example, if the system had known about **Person** beforehand, with attributes "Name", "Age", and "Gender", it would have automatically placed **Adult** as a "subclass" of **Person**, and rightly so in this case. Despite its naivete, we believe that such a classification algorithm (see Figure 6.3 about the AO automatic classification algorithm) is an invaluable analysis aid:

1. It may reveal inconsistent use of nomenclature by blindly – based on attribute names pushing new application objects down the wrong paths (case of homonyms),

2. It may help developers complete the specifications of their application objects where, at each step - i.e., each time a new attribute is added – the new siblings of the application object suggest alternative ways of completing the specification

```
Classify_AO(new_AO, root_AO)
  0.  new_AO_attribute_set <- all attributes of new_AO
  1.  root_AO_attribute_set <- all attributes of root_AO
  2.  IF new_AO_attribute_set ⊇ current_AO_attribute_set THEN
        FOR each sub_AO of root_AO DO
          Classify_AO(new_AO, sub_AO) ;
          make new_AO a sub_AO of root_AO;
          EXIT;
      ELSE return
```

Figure 6.3: Application Object Automatic Classification Algorithm

of the object – in effect turning the classification algorithm into a browsing tool – hence avoiding the proliferation of attribute names that mean essentially the same thing (synonyms);

3. It might be all that is required, i.e., the user may very well be entirely satisfied with result of automatic classification.

Manual classification may be used when the user/developer knows beforehand the application object from which he wishes to inherit by name, for e.g., as the result of a search on "Data Contents", or simply by displaying the application object hierarchy.

Finally, notice that while automatic classification uses only attributes for the time being, e.g., an application-object inherits both attributes and operations. For the time being, SoftClass does not care about conflicting operations in the case of multiple inheritance. Notice however that users can redefine inherited operations locally[1]. In the case of multiple inheritance, a user may chose to redefine one of the conflicting operations locally, in effect masking the other one.

[1]By pushing the "super" button in Figure 6.1, and selecting one of the operations (right most list), one of the options available is to accept the operation locally The new operation may then be edited in a separate tool

6.3 Generic Data Structures

6.3.1 Generic Data Structures

In SoftClass, generic data structures correspond to application-independent, programming constructs/artifacts. In terms of level of abstraction, they are half-way between abstract classes an interface/signature, without implementation – and fully-implemented abstract data types, in terms of including the algorithmic design for some of the operations. Typically, GDSs might include a **SortedList** such that the description of its operation "InsertInOrder(InOut L: SortedList; In V: ValueType)" includes the algorithm/pseudo-code for inserting a value in an ordered list, without referring specifically to the way nodes and successor links are implemented (e.g., with dynamic pointers, versus as an index in an array of records). The nuances will be come clearer as we see some examples. We distinguish between three kinds of GDSs: 1) *atomic GDSs*, such as integer, float, character, string, which represent indivisible literals, 2) *records*, and 3) *collections*, which consist of things such as arrays, sets, and all sorts of recursive data structures, such as linked lists, trees, graphs, etc. All GDSs are described by the set of operations they support, grouped in "protocols"[2], such as "constructors","accessors", "iterators" "search", etc.

All but atomic GDSs are parameterized. We use two kinds of parameters: 1) *Type parameters* for all kinds of GDSs, à la generic packages in Ada, and 2) *Name parameters*, used mostly for record GDSs as discussed in Section 4.5.1.

For example, we might define a GDS **LinkedList[ElementType]**, where **ElementType** is the type of the data contents of the "nodes" – however they are implemented. For an actual application data structure, we might substitute **ElementType** by **EmployeeRecord**, for example, and obtain a list of employees. Operations of such GDS are those that are purely data structure-dependent. Such kinds of operations can be reused by any application-specific classes using **LinkedList**. Operations

[2]Using Smalltalk's jargon, where a protocol is a list of closely related methods

like **Add_element()** , **Delete_element()** and so on belong to this category.

Name parameters are used to parameterize operation names. Consider the record GDS R1 shown below:

```
R1 [F1,F2,F3] =
  {
     F1: STRING;
     F2: INTEGER;
     F3: STRING
  }
```

In this case, the field names F1,F2,F3 are *name parameters*. R1 might support the operations "GetF1(In R1; Out STRING)", "SetF1(InOut R1, In STRING)" Such a GDS may be used to implement the application object (AO) **Person(Name, Age, Gender)** by mapping **Name** to **F1**, **Age** to **F2** and **Gender** to **F3** In this case, "GetAge(...)" is also mapped to "GetF1(...)", i.e., the algorithm for "GetAge(. .)" is obtained from that of "GetF1(...)" by making the proper name substitutions. We could use the same record **R1** to "implement" the application object **Course (Title, Credits, Textbook)**, with totally different application semantics. In this case **F1** maps to **Title**, and "GetF1(...)" maps to "GetTitle(...)".

Generic data structures may have both kinds of parameters, as shown with the example record GDS R2:

```
R2 (name-parameter[F1,F2,F3,F4], type-parameter[element-type]) =
         {
            F1: LinkedList [element-type]
            F2: STRING;
            F3: INTEGER;
            F4: STRING
         }
```

Using this structure, we can build the generic operation "GetTotalNumberOfF1" defined as follows:

```
GetTotalNumberOfF1(In self: R2; Out num : INTEGER)
  {
    num := GetNumberOfElements(GetF1(self));
  }
```

where we assumed that linked lists support the operation "GetNumberOfElements", and R2 supports the operation "GetF1". R2 can be used to implement **Adult** using the following mappings: 1) **F1** → **Children**, 2) **ElementType** → **Person** meaning that the attribute **Children** is a list of **Persons** – , 3) **F2** → **Name**, 4) **F3** → **Age**, and 5) **F4** → **Gender**. Using this mapping, "GetTotalNumberOfF1", becomes "GetTotalNumberOfChildren", and does exactly that.

6.3.2 Generic Data Structures Hierarchy

In SoftClass, generic data structures (GDS) are organized along an inheritance hierarchy. A GDS R1 is a descendant of a GDS R2 *if R1 has all the component types of R2*, and possibly more. Unlike application objects (AOs) where inheritance was based on attribute set inclusion, in this case we use the *types*, rather than the *names* of the data structure components (see Figure 6.4). For example, according to our definition, R2 above inherits from R1, although field names clash. We choose to ignore field names because those names are bogus names that will ultimately be replaced by application meaningful names. In the <R2,R1> example, it is clear that field F2 of R1 (of type INTEGER) maps to field F3 of R2. However, F1 and F3 of R1, may map to either F2 and F4 of R2, respectively, or F4 and F2, and it is important to indicate which, once and for all, when we create the inheritance link between R2 and R1[3].

In general, it is useful to think of inheritance relations as being defined by a

[3]Among other things, it is important to keep track of which methods are applicable to which field.

combination of a superclass, and a list of mappings (pairs). This is similar to renaming in Eiffel [Mey92].

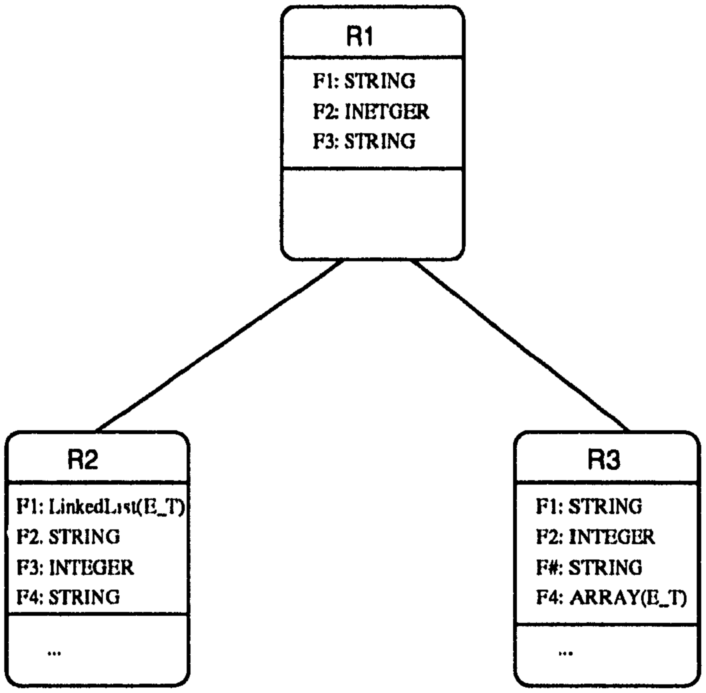
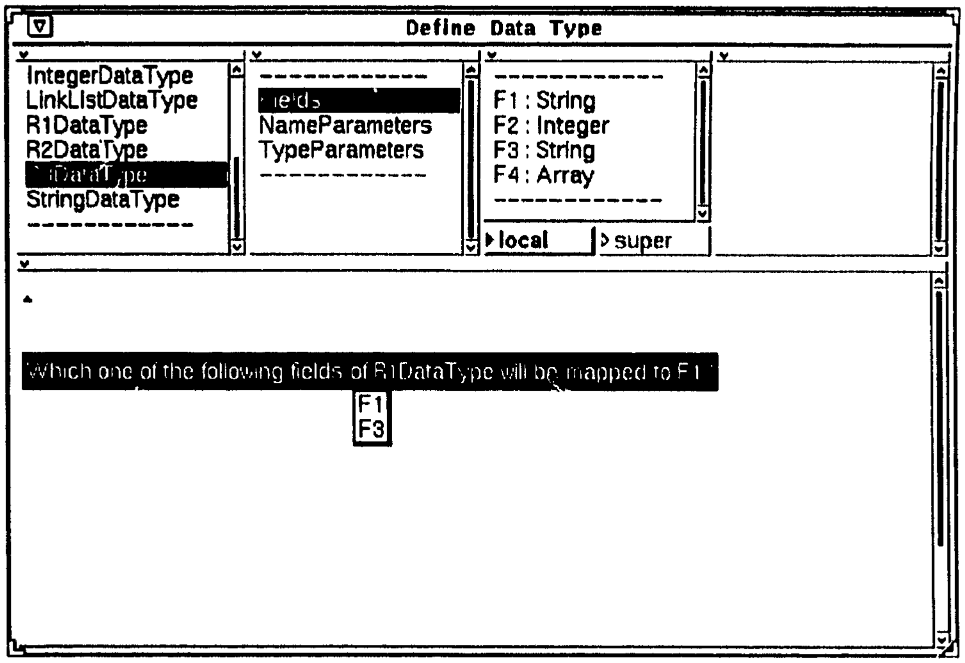


Figure 6.4: Generic Data Structure Hierarchy

In the above example, one interpretation of the relation between R1 and R2 is R1 ((F2 F1) (F3 F2) (F4 F3)). Another would be R1 ((F2 F3) (F3 F2) (F4 F1)). A GDS may inherit from several GDSs, in which case, it would maintain on such mapping for each immediate superclass. Assume that we define two record GDSs, R3 and R4, as follows:

```
R3 (name-parameter [F1,F2,F3,F4],
    type-parameter[ElementType2]) =
     {
        F1: STRING;
```



Which one of the following fields of R1DataType will be mapped to F1

Figure 6.5: The GDS tool for automatic classification

```
        F2: INTEGER;
        F3: STRING;
        F4: ARRAY[ElementType2]
     }

R4 (name-parameter[F1,F2,F3,F4,F5],
    type-parameter[ElementType,ElementType2]) =
     {
        F1: LinkedList[ElementType];
        F2: STRING;
        F3: INTEGER;
        F4: STRING;
        F5: ARRAY[ElementType2]
     }
```

R4 is a subclass of both R2 and R3. We have:

R4 **superclasses**: R2 ((F1 F1) (F2 F2) (F3 F3) (F4 F4)),

R3 ((F2 F1) (F3 F2) (F4 F3) (F5 F4))

In addition to "GetTotalNumOfF1 (...)" defined for R2 above, R4 can also inherit the operation "GetTotalNumOfF4 (In self: R3; Out num: INTEGER)" for R3.

```
Classify_GDS(new_GDS, root_GDS)
  0.  new_GDS_type_set <- all field types of new_GDS
  1.  root_GDS_type_set <- all field types of root_GDS
  2.  IF new_GDS_type_set ⊇ current_GDS_type_set THEN
        FOR each sub_GDS of root_GDS DO
          Classify_GDS(new_GDS, sub_GDS) ;
          Mapping_Field_Name(new_GDS, root_GDS);
          make new_GDS a sub_GDS of root_GDS;
          EXIT;
      ELSE return
```

Figure 6.6: Generic Data Structure Automatic Classification Algorithm

```
Mapping_Field_Name(new_GDS, root_GDS)
  FOR each new_GDS_field DO
    IF only one new_GDS_field is the same type as root_GDS_field THEN
      new_GDS records this field name mapping automatically;
    ELSE
      prompt to the user the choices and let the user decide mapping;
```

Figure 6.7: Generic Data Structure Field Name Mapping Algorithm

Such an operation is identical to "GetNumTotalOfF1", except that F1 is replaced by F4. Because of the mapping, R4 "sees" these methods as "GetTotalNumOfF1" and "GetTotalNumOfF5", respectively.

SoftClass automatically classifies generic data structures based on their component types, when there is no ambiguity. In case of ambiguity – e.g., case of R1 and R3 above – the tool for defining generic data structures prompts developers for the ambiguous mappings. Figure 6.5 shows that when the user asks for automatic classification of R3 GDS, the tool first finds that R3 should be sub GDS of R1. However, the tool also finds that F1 of R3 can be mapped to either F1 or F3 of R1. In such cases, the tool prompts to the user to make the choice. Figure 6.6 and 6.7 show the algorithm for automatic classification of GDSs. Developers may also classify generic data structures *manually*, i.e., by specifically naming their immediate superclasses.

Notice that when a GDS is classified, the inherited operations that are relative to renamed fields are not actually renamed and redefined locally. However, by recording the mapping, SoftClass can dynamically "reconstruct" such local definitions, should the need arise (see Chapter 7).

6.4 Application Data Structure

6.4.1 Application Data Structure

Application Data Structures (ADS) are design-level representation objects. They are obtained by mapping an application object over a generic data structure. As another way of looking at it, they are obtained by binding the parameters of a GDS to the specifics of an application object. Figure 6.8 illustrates how the application data structure **AdultRecord** may be obtained by mapping the application object **Adult** over GDS **R3**.

Assume that we want to implement **Adult (Name, Age, Gender, Children)** application object, the given example in the previous section. R2 and R3 are GDS candidates. Suppose we have **Person** application object that is already fully implemented by mapping to another GDS (e.g., R1) and we finally get **Person** ADS. If we chose R2 as the GDS to represent **Adult** application object, we obtain it by instantiating all the generic parameters:

R2(name-parameter (Name, Age, Gender, Children) , type-parameter (Person)).

This mapping is similar to that between generic data structures that are in an inheritance relation: it renames data field names, and makes the proper substitutions in the code of the operations. Figure 6.9 shows that when the user is mapping the **Adult** AO to the **R3** GDS, the tool prompts to the users to confirm the mapping of each individual attribute to proper field. After the mapping, the field name pairs are recorded and all GDS operations can be easily obtained by field name substitution

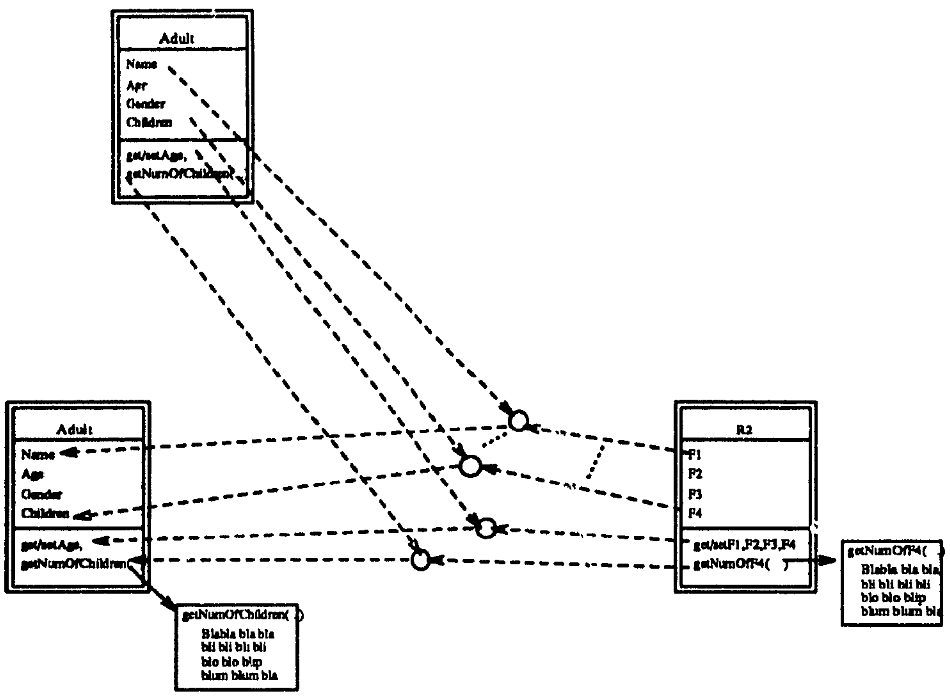


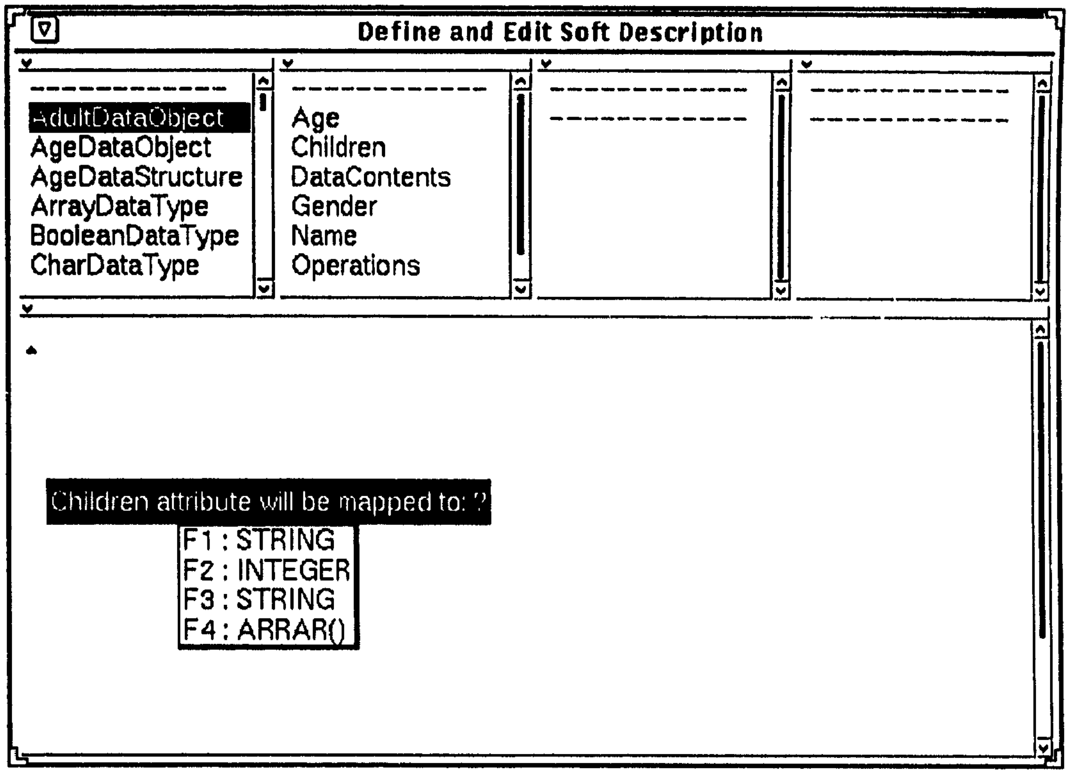
Figure 6.8: Creating a design-level representation (ADS) for an application object (AO).

in the code. For example, by mapping **Children** of the application object **Adult** to field **F4** of **R3** GDS, we realize that the operation "GetTotalNumberOfChildren" identified at the analysis level has been implemented by R3 GDS under the name of "GetTotalNumberOfF4" that might be written as:

```
GetTotalNumberOfF4(In self: R3; Out num: INTEGER)
 {
  num := GetNumberOfElements(GetF4(self));
 }
```

Thus, a design-level representation of "GetTotalNumberOfChildren" is created by making the proper substitution in the (pseudo-) code of the "GetTotalNumberOfF4". In particular, we get:

```
GetTotalNumberOf*Children*(In self: *Adult*; Out num: INTEGER)
 {
  num := GetNumberOfElements(Get*Children*(self));
```



Children attribute will be mapped to: ?

F1 : STRING
F2 : INTEGER
F3 : STRING
F4 : ARRAR()

Figure 6.9: The GDS tool for automatic classification

}

We could have mapped **Adult** to **R3** GDS, and obtained another ADS called it **AdultRecord3** –, which is distinct from **AdultRecord2**, and the two may co-exist in the system.

Notice that we automatically create the design-level representation (algorithmic design) for the operation only if the application object has an equivalent in the generic data structure. In particular, we do not systematically create an operation in the application data structure for every generic operation of the generic data structure. For example, if R2 GDS implemented the generic operation "GetMiddleF4(...)", we do not necessarily create the operation "GetMiddleChildren(...)", unless it has been identified at the analysis level. This kind of *encapsulation* is highly desired in most of applications. A GDS offers all the operations needed to access, construct/release, iterate its components and to manipulate its structure.

The operations of an application object, identified at analysis stage, can be roughly divided into two categories as described in [Mar91]: 1) base operations 2) high-level, application-specific, or computational operations. Base operations, using our separate GDS hierarchy, can be obtained by proper *name* or *type* substitution. On the other hand, for every application-specific operation, we will not necessarily have an equivalent generic operation. For example, we may be interested in the number of sons a given **Adult** person has. The fulfilled operation of **Get_Num_Of_Sons (Adult)** may look like:

```
Get_Num_Of_Sons (in self Adult, out Num: INTEGER)
 {
 local each_child : Person;
 local counter : INTEGER;
 counter = 0;
 For_Each (each_child, Get_Children (self))
  {
  if (Is_Male (each_child)) then
   counter = counter + 1;
  };
 Num = counter;
 }
```

but such an operation has no equivalent in the GDS, and consequently, its pseudo-code will have to be entered explicitly by the developer. However, as we will see in Chapter 7, the program design language (pseudo-code) compiler draws a number of inferences that make this task easier.

6.4.2 Application Data Structure Hierarchy – a Bi-Inheritance-Hierarchy

Our way of creating application data structures results in a significant reuse of generic code. however, we would also want to reuse code corresponding to operations with very specific application-semantics, illustrated by the - not so specific "GetNumberOfSons" operation. Under what conditions could we reuse that code (or pseudo-code)? For any given pair of application data structures ADS1 and ADS2, one of four situations may occur:

1. The corresponding (AO)s are identical or have an inheritance relationship, and the corresponding (GDS)s are identical or have an inheritance relationship,
2. Only the corresponding (AO)s are identical or in an inheritance relationship,
3. Only the corresponding (GDS)s are identical or in an inheritance relationship, and
4. None of the above.

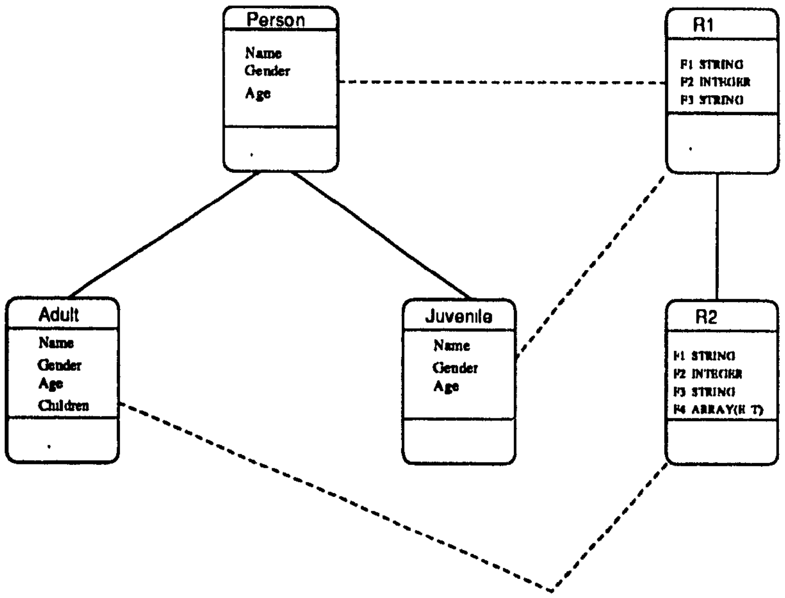


Figure 6.10: Application Data Structure hierarchy

The first and last case are straight-forward, although they stand at the opposite end of the reuse spectrum. For the last case, there is very little chance of code reuse, and we should not try to force reuse.

With the first case, we can safely reuse code between ADS1 and ADS2, and the reused operations means the same thing in both classes. In the third case, we could reuse all of the operations of ADS1 in ADS2, if we wanted to, but most likely, they would not make sense. We could imagine using R2 GDS to implement a company department, say **Department(NameOfDepartment, DeptNumber, NameDeptHead, Employees)**. By replacing "GetChildren" by "GetEmployees" - a substitution that could be automated using the various mappings, we could reuse "GetNumberOfDaughters" to get the number of female employees of the department, but that would not make sense, because a user looking for an operation which returns female employees will not intuitively consider a "get number of daughters". This is the kind of implementation inheritance that a number of researchers and partitioners advised against. [Coo92, Cox90].

In the second case above, there is a great deal of reuse that can be explored and should be taken advantage of. In this case, ADS2 could reuse all of the *application-semantics* of ADS1 - which it already does - but not necessarily the implementation or representation. Again, there are two kinds of possibilities in this case:

- the corresponding AOs are identical.

  This may be the case that an AO may have two or more versions of implementation at the design or implementation. We can imagine the same software product will ultimately be run in different environments or ported to different platforms, and therefore, may have different performance requirements. Or, more generally, an existing software product may have some AOs that have the same or similar application-semantics as a new product in which different data structures must be used to implement them. For example, for the same **Adult** application object, we may want to use **ARRAY** instead of **LinkedList** to represent **Children** attribute, or we could use R3 to implement **Adult**.

- the corresponding AOs are in an inheritance relationship.

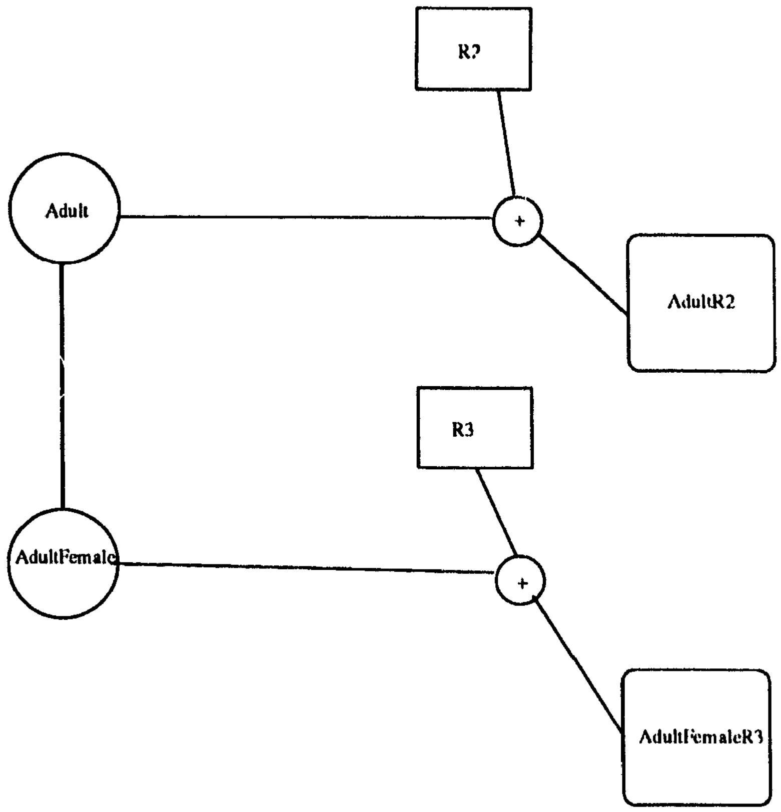


Figure 6.11: Adult mapping to R2 and AdultFemale mapping to R3

It may not make sense for one software product to have AOs that are in an inheritance relationship implemented using different representations. Our goal is to reuse code between different implementations of the same software products, or to reuse existing software components to construct new software products. In such cases, there will be more opportunities for reuse by eliminating the strict, not always desired requirement of having the same underlying data structures. Suppose that in one existing software product, we have the AO **Adult** and its implementation **AdultR2**, obtained by mapping **Adult** to the GDS **R2**. Assume that a new application requires that we define an **AdultFemale** AO which is a subclass of **Adult** from a conceptual point of view, but that for efficiency consideration, we chose to implement **AdultFemale** using R3 instead of R2, to yield **AdultFemaleR3** (see Figure 6.11). We may need some operations such as "GetNumberOfSons" that have been fully implemented by **AdultR2**.

According to traditional, implementation-based inheritance, there is no chance for *systematic* reuse. This is because that, at design or implementation stage, application-semantics has to give up the priority to the underlying data representation. It is true that we can not reuse those *low-level, data structure-dependent* operations with different data structure implementation. "Add_Element(...)" would mean different thing for an **ARRAY** or **LinkedList**. However, these low-level access operations will be redefined as a result of mapping **AdultFemale** to **R3**. Nevertheless, for those application-specific and data structure-independent (with disciplines) operations such as "GetNumberOfSons" can be and should be reused.

Let us take a look at the pseudo-code of "GetNumberOfSons" for **AdultFemaleR3**. It will be:

```
Get_Num_Of_Sons (in self AdultFemale, out Num: INTEGER)
 {
 local each_child : Person;
 local counter : INTEGER;
 counter = 0;
 For_Each (each_child, Get_Children (self))
   {
   if (Is_Male (each_child)) then
     counter = counter + 1;
   };
 Num = counter;
 }
```

notice that the *For_Each()* operation, an iterator operation of ARRAY is different from an implementational point of view (ARRAY vs Linked_list), and if not following some discipline, it may be named different from each other (i.e. For_Each_Of_Array() and For_Each_Of_Linked_list()).

Compared to the the same operation of **AdultR2**, the operation of **AdultFemaleR3** also uses four classes - **AdultFemale, Person, INTEGER** and **ARRAY**.

Obviously here, **Person** ADS is the same GDS (**PersonR1**) and therefore, operation "Is_Male()" is also the same one. The access operation "Get_Children()" for **Adult-Female**, which is actually obtained by proper substitution from "GetF1()" from R3 GDS, has the same function by selecting the proper component. The only difference is the iteration operation "For_Each" for ARRAY while **AdultR2** uses "For_Each" for Linked_List. However, if we keep the same syntax for ARRAY iterator operation by overloading the operation name, i.e., **For_Each (out element-type, in ARRAY)**, by replacing **Adult** with **AdultFemale**, we can reuse the same code of this application-dependent operation for another **Adult** Application Object implementation **AdultFemaleR3**.

In general, if ADS1 and ADS2 are inheritance-related in the AO hierarchy but not in GDS hierarchy, the application-specific operations can be reused provided that the following checking and design principles are satisfied:

1. First, we need to find out all the client classes and their operations used in the *original* operation. We will check each of these operations one by one.

2. These operations might be 1) GDS operations (in our example, they are iteration "For_Each()" for ARRAY or Linked_List and "+" for INTEGER); or 2) the operations are obtained from different GDS ("GetChildren()" for R2 or R3); or 3) other ADS operations ("Is_Male()" for **Person** in the example).

3. For the GDS operations, if the client classes are the same in both original and desired operation (INTEGER GDS and operation "+" in our example), it is completely safe. We can directly use it. If the the client classes are different (Linked_List vs ARRAY), some discipline is required in design of generic data structure, where we should aim for abstraction and overloading. Iterator operation for both ARRAY and Linked_List are required using the same name and syntax.

4. For the operations obtained from different GDS, there is no problem, because they are parameterized in the same way, such as "GetChildren()".

5. Since the application-semantics is inherited, the client GDS will have the same semantics also. Therefore, we only need to check if they are using the same GDS. If they are, which means they are the same ADS also, there is no problem. If not, we also need to discipline the operation such that it will have the same *external* behaviour. Here encapsulation plays a very important role. In this example, since **Person** ADS is identical, the operations used from it are safe to be used also, such as "Is_Male()".

6.4.3 Operation Classification

From the above discussion, we distinguish three kinds of operations for any ADS:

1. operations that are obtained from GDS. These kinds of operations always have no application-semantics, but highly data structure-dependent. We may call such operations *application-independent operations* (AIO).

2. operations that have application-semantics and also data structure-dependent. For example, the "Is_Male()" operation may be implemented as "return (Gender == 1)" or "return (Gender == 'Male')" or "return (Gender == 'M'), depending on whether **Gender** is represented as an INTEGER, a STRING or

a CHARACTER. We call this sort of operations *low-level application-specific operations* (LLASO).

3. operations that heavily application-semantics dependent, but data representation (GDS) independent. "Get_Num_Of_Sons()" is such kind of operations. We call this kind of operations *high-level application-specific operations* (HLASO).

Using our approach, the first kind of operations can be reused from GDS. And we just argued in the previous sections that the third kind of operations can also be reused. Only the second kind of operation need to be rewritten. Fortunately, they are usually trivial operations.

Application-independent operations can be easily distinguished. The distinction may be blurred between *low-level* or *high-level application-specific operations.* Intuitively, there are some rules that help distinguish them:

1. If all the client classes but the ADS2 itself are identical in original and the desired operation, and the components of ADS2 used are not changed, it is a HLASO.

2. If the components of ADS1 are used and they are different from the corresponding ADS2, and also literal values are involved, it is a LLASO, and therefore should be rewritten.

3. If there exist client classes or components of the receiver class whose GDS are different, such as in "Get_Num_Of_Sons()", the component Children is different, besides the design discipline (overloading operations and encapsulating the literal-like details) to guarantee the external behaviour keep unchanged, recompiling the code may be needed after proper substitution.

6.5 Implementing Representations of AOs, GDSs and ADSs in SoftClass

Recall in Section 5.3, process/procedure software components are represented using global attributes, and internal relations among attributes and components. The attributes values have both keyword and text format.

AOs, GDSs and ADSs are represented as software components in a consistent fashion with process/procedure components.

The application-domain attributes of each AO is represented as "global attribute"s the AO class, the sub AOs, if any, are represented as "components" as the AO (aggregation).

Since GDSs can be roughly classified into two categories: singleton and complex data structures which are composed of fields. Therefore, the components of a complex GDS are represented as "field components", each of which has an "DataType" attribute to store the field type information.

An ADS in SoftClass is a combination of the AO (its analysis level format) and the GDS (its underlying implementation): 1) all the application-domain attributes are inherited by the ADS and also represented as "global attribute"s; 2) since each application-domain attribute is mapped to a field in the GDS, the field type is stored as the keyword value of the attribute; 3) all the sub AOs at the analysis level of the corresponding AO are transformed into sun ADSs of the ADS.

The representation of operations (process/procedure components) of data objects is implemented as follows: All the operations of a data object are organized into different protocols (just like in Smalltalk). The protocol list is conceptually represented as an attribute named "Operations" for AOs and GDSs. As for ADSs, we use "Operations" attribute to represent the protocol list inherited from their corresponding AOs, another "Generic Operation" attribute to represent the protocol list from its underlying GDS.

In SoftClass, each operation is actually a process/procedure software component. From a process/procedure component's point of view, it is an operation belonging to one or more data objects. We know it from Chapter 5 that the input/output parameters of a process/procedure component are represented as attributes of the component. In details, the keyword name indicates the parameter name; the input/output property is defined according to proper keyword category - keywords for parameters are classified into "input", "output" and "input/output" categories; the parameter types are represented as the values of the keywords. The text for the parameter attributes is used for description of the parameter. Process/procedure components are associated with data objects according to, in most cases, the *input* parameters of each process/procedure component.

The representation of the detailed design level operations are represented differently from those of the analysis and architectural design level, since more detailed code needs to be entered by the developer. In the next chapter, we will discuss the PDL (Program Design Language) for algorithm design, and its compiler which realizes our reuse approach and validates pseudo-code.

Chapter 7

PDL – A Program Design Language for Detailed Design

7.1 What is PDL

A number of CASE tools support some sort of pseudo-code/algorithm design language. However, not much is done with the pseudo-code, with the exception, perhaps, of the signatures of procedures. As part of the SoftClass project's efforts towards automating some of the change propagation across development stages, it is desired to develop an intuitive and easy to use, yet "formal" and somewhat verifiable design language. *Programming Design Language* or PDL is such a language.

Design and programming are the same activity, raising the same issue – structure, modularity, correctness, balancing between function and data aspects etc. A PDL is for design what a programming language is for implementation. The only difference is in the abstraction level of the virtual machines being programmed. SoftClass wants users/developers to be able to write something close to

```
For each employee:
   compute weekly salary,
   print check,
   send evaluation report to supervisor
```

without having to worry about whether the personnel records are kept in a file, an array, a linked list, and without worrying about, or having to learn the syntax of yet another design/programming language. Further, as a experimental CASE tool, it is not desired to spend too much time designing such a language and make sure it is expressive, complete, exhaustive, etc. The syntax of PDL can be found in the last page of this chapter (Figure 7.6).

In order to remain close to the user, short of performing fancy natural language processing, we had to adopt an extensible "object-oriented design language". The language is object-oriented in the sense that it supports information hiding, inheritance (two inheritance hierarchies) and overloading. It is extensible in the sense that, with the exception of some very basic control structures (branches and loops), the "vocabulary" of the language is as wide as what the language compiler finds in SoftClass in terms of GDS, ADS and operations. Figure 7.1 shows the pseudo-code for the operation "GetNumOfSons" defined on **AdultRecord**. For each detailed design-level representation of a process-like component, the system creates an empty procedure with the signature and the terminator "EndModule". A developer enter "most" of what is between the first and last line. We say "most" because the local variable declarations are semi-automatically added by the compiler. This will be illustrated in the next section.

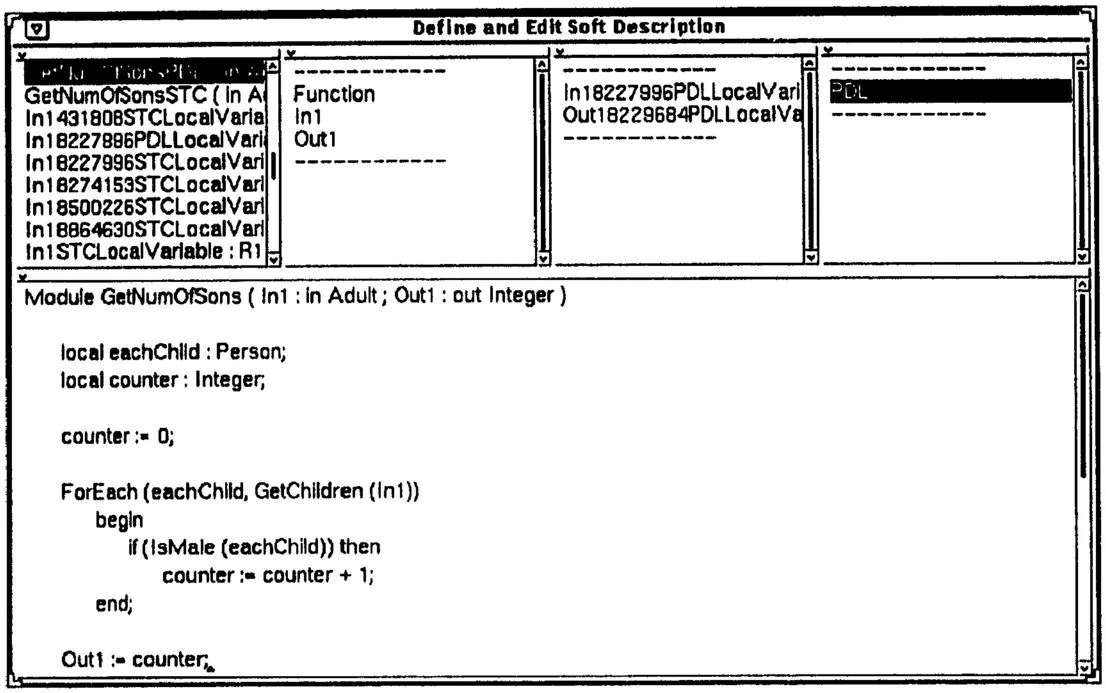
7.2 Algorithm Validation

PDL is used to describe algorithm within the an operation. PDL compiler is responsible for the algorithm validation. Algorithm validation includes twofold: syntactic validation and semantic validation (see Figure 7.2).

Syntactic and lexical validation involves the following:

1. Validating control structures: the PDL compiler looks for the proper delimiters of various control structure.

2. Validating object names and operator names (alphanumeric with proper hyphenation characters).



```
[illegible]
GetNumOfSonsSTC ( In A
In1431808STCLocalVaria
In18227886PDLLocalVari
In18227996STCLocalVari
In18274153STCLocalVari
In18500226STCLocalVari
In18864630STCLocalVari
In1STCLocalVariable : R1
```

```
-------------
Function
In1
Out1
-------------
```

```
-------------
In18227996PDLLocalVari
Out18229684PDLLocalVa
-------------
```

```
-------------
PDL
-------------
```

```
Module GetNumOfSons ( In1 : in Adult ; Out1 : out Integer )

    local eachChild : Person;
    local counter : Integer;

    counter := 0;

    ForEach (eachChild, GetChildren (In1))
        begin
            if (IsMale (eachChild)) then
                counter := counter + 1;
        end;

    Out1 := counter;
```

Figure 7.1: PDL Tool

Semantic validation involve the following:

1. All data objects are locally or globally declared;

2. All data manipulations (operations) are valid.

7.2.1 Data Object Validation

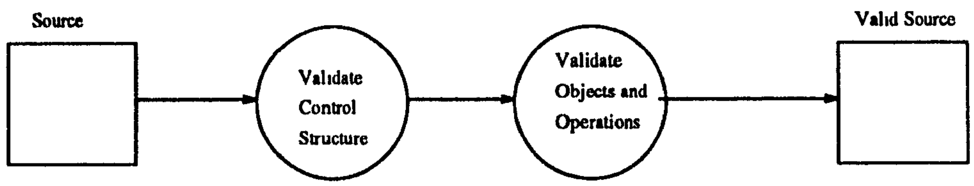
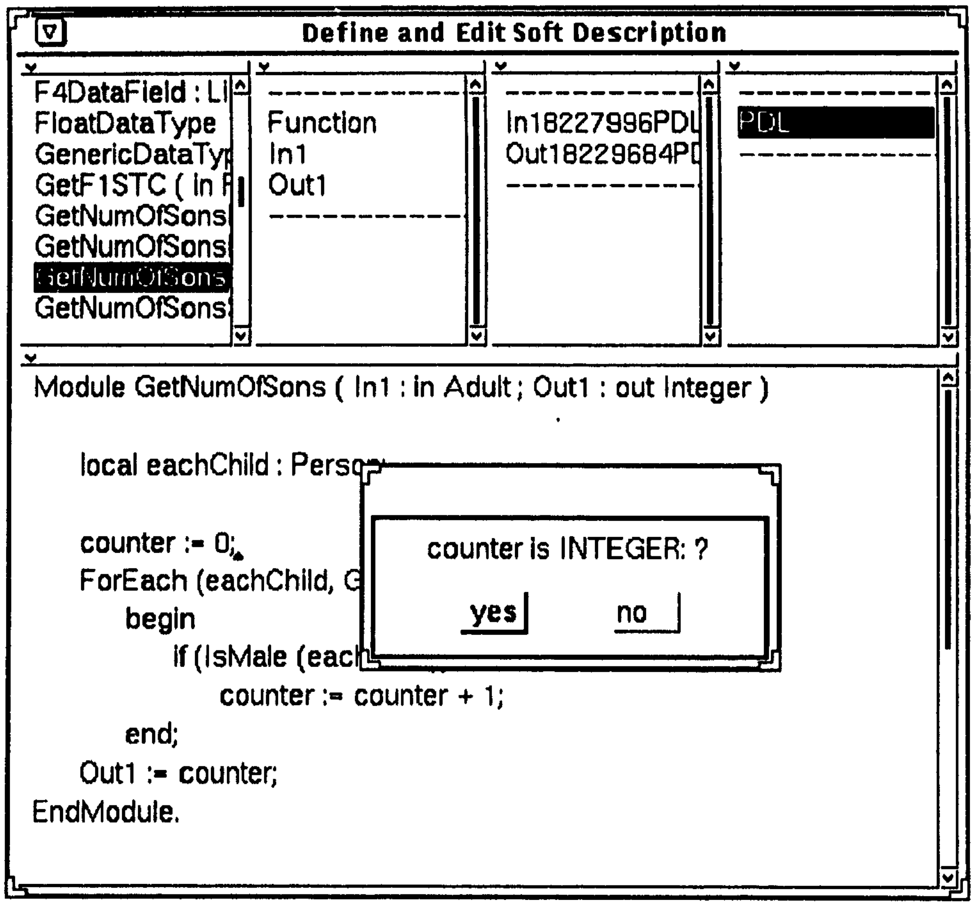


Figure 7.2: The Process of Algorithm Validation

```
0.  IF a constant value is assigned to the undeclared variable THEN
      the variable's type is defaulted as the constant's type and prompt user to confirm
    ELSE
1.  IF the variable appears as an parameter of an operation THEN
    (a)  IF non of the parameters of the operation is known THEN
           prompt the user to ask for type of any parameter
    (b)  ELSE (i.e., some parameter's type is known)
      the undeclared variable is inferred from the operation
      which is found according to the known parameter.
```

Figure 7.3: Variable Type Inference Algorithm

As mentioned above, the PDL compiler can semi-automatically add variable declarations when it has enough knowledge about the variables to infer their type and their scope. When the compiler sees a symbol it does not know about - basically, not present as a component in SoftClass, the compiler tries to make inferences about the nature of the symbol depending on the context of its use. The syntax is sufficiently unambiguous to distinguish easily a procedure/operation from a data object. If the compiler identifies a data object, it tries to guess its scope and type. For our example, compiler infers that 1) "counter" and "eachChild" are both data objects (from syntax), 2) that they are both local (could not find them elsewhere in the system) 3) that "counter" is "an instance" of INTEGER class, and 4) that "eachChild" is an instance of Person. Type inferences are fairly simple (and simplistic) when the object are used as arguments to recognized functions: the compiler takes by default – to be confirmed by the user – the most general ADS/GDS that supports the operation. Figure 7.4 shows that when the user does not declare "counter" variable, the PDL compiler infers that "counter" might be an "INTEGER" when it hits the first assignment statement.



```
Module GetNumOfSons ( In1 : in Adult; Out1 : out Integer )

    local eachChild : Perso

    counter := 0;
    ForEach (eachChild, G
        begin
            if (IsMale (eac
                counter := counter + 1;
        end;
    Out1 := counter;
EndModule.
```

Figure 7.4: The PDL compiler interacts with the user to confirm inferred type.

7.2.2 Operation Validation

In addition to ensuring that all data objects are properly declared, the compiler validates all the operations on data objects. This is done in two steps: 1) make sure that the data object has an operation with the same name as the operation being checked (name checking), and 2) check the conformity of the types of the call parameters with the formal ones (type-checking and parameters validation).

The type-conformity rule can be described as follows:

Suppose we have one desired type called **D**, and one type being checked called **C**. We say types are valid when:

1. **D** and **C** are of the same (name equivalent) Data structure. If not, then
2. **C** is descendant of **D** in Application Data Structure. If not, then
3. **D** and **C** are from the same GDS. If not, then
4. **C** is descendant of **D** only in GDS hierarchy.

In SoftClass, binary operators such as "+", "-", are treated as ordinary functions and procedures with two input parameters and one output parameter. So operator validation is processed in a unified way.

Input/output parameter-checking works in the following way:

1. Check that the number of call parameters matches the number of formal parameters. If not, parameter-checking fails.
2. Check the input/output property, which is based on the following rules: If the formal parameter is either output or input&output parameter, the actual parameter can only be a variable, or either an output or input&output parameter of the module being compiled. If the checking fails, parameter-checking fails.
3. Check the actual-formal parameter type, which is based on the rules explained above. If the checking failes, parameter-checking fails.

The overall validation of an operation goes as follows:

If the object in question belongs to a GDS, then the compiler checks whether that GDS implements the operation. If it does not find the operation in that GDS, it climbs the inheritance hierarchy, *possibly changing the name of the operation it is looking for to take into account the renaming of data fields through inheritance* (see section 6.3 for details).

If the object in question is an application data structure, the compiler uses the algorithm shown in Figure 7.5.

In other words, the compiler looks for the operation within the application data structure hierarchy. If it does not find it, it finds the name its generic equivalent would have, and tries to find that generic equivalent, in the same way that the compiler

```
0.  currentADS <- class of argument to OPERATION
1.  IF the currentADS implements OPERATION THEN
      IF parameters are valid according to the rules above THEN
        EXIT with valid flag
      ELSE EXIT with unvalid flag
    ELSE
      currentADS <- currentADS superclass
    IF currentADS = NIL THEN
      GOTO 2.
    ELSE GOTO 1.
2.  current GDS <- GDS implementing class of argument to OPERATION
3.  map OPERATION back to OPERATION_GDS, using mapping of ADS to
      currentGDS
4.  IF the currentGDS implements OPERATION_GDS THEN
      OPERATION valid
      EXIT
    ELSE
      currentGDS <- current GDS superclass
    IF currentGDS = NIL THEN
      OPERATION not valid
      EXIT
    ELSE
      OPERATION_GDS <- equivalent of old OPERATION_GDS in new
        currentGDS (possibly with renaming)
      GOTO 4.
```

Figure 7.5: Operation Validation Algorithm

validate operations on instances of GDS. In case the operation to be validated has several arguments, we try the above algorithm for every positional argument of the operation. Once the operation is found, the compiler checks the other arguments for conformity with the definition of the operation using the checking rule described above.

Let us take a look at how the PDL compiler validates operation **Get_Num_Of_Sons()** of **Adult** Application Data Structure.

Get_Num_Of_Sons() calls four operations altogether.

1. **Get_Children (in self : Adult)**

The PDL compiler first checks if there is an operation called **Get_Children()**

attached to **Adult** Application Data Structure. The compiler finds that there is no such operation. Then PDL compiler climbs Application Data Structure hierarchy to see if there is such an operation at **Person** Application Data Structure which is the a super Application Data Structure of **Adult**. If it is not found there either. the PDL compiler continues to search along the inheritance path up to the ıoot data structure.

If the PDL compiler fails to find such an operation in Application Data Structure hierarchy, then the compiler goes to the GDS hierarchy. First, the compiler finds that record **R2** is the GDS that represents the **Adult** Application Object. It then renames **Get_Children()** to **Get_F4()** from the name-parameter dictionary. And finally it gets the operation at GDS R2.

2. **For_Each (in/out each_child : Person, in link-list (Person))**

   The PDL compiler validates this operation in the same way as the above operation.

3. **Is_Male (in/out each_child : Person)**

   This operation is an application-dependent operation attached to **Person** Application Data Structure. The PDL compiler will first perform input/output parameter-checking, and find that the operation valid.

4. **+ (in/out counter : integer, in 1 : integer)**

   Since all the actual parameters are GDSs, the PDL compiler goes to the GDS hierarchy to perform operation validation. Actually, it finds that the operation is one attached to **integer** GDS.

Such an elaborate verification scheme relieves developer/analysts from having to explicitly define the most mundane data manipulation operations. We mentioned in last chapter that when we create application data structures, only the operations that were defined at the analysis level and that have a GDS equivalent are created automatically; the others have be added manually. Given that at the analysis level, we wish not to overburden analysts with having to specify mundane operations – such as access operations, this mechanism enables the compiler to verify that such operations "make sense". For example, the compiler might infer that "GetName" is a sensible operation for an **PersonRecord**, even if it is not explicitly defined, either for **PersonRecord**, or for its ancestors.

7.3 Implementing PDL in SoftClass

In SoftClass, the internal relation of a process/procedure component at the analysis and preliminary design level is represented in terms of data-flow relation among interfaces of the component and its subcomponents. At the detailed design level, such an internal relation is embedded in the text of PDL pseudo-code along with other code entered by the developer for fully implementing the operation. It is the PDL compiler's responsiblity to validate the code the developer entered and to extract *in reverse* the previous level representation if needed.

The PDL compiler in SoftClass is implemented with the help of a parser-generator which is supplied by the Smalltalk library. With the parser-compiler, the PDL parser is obtained by writing special methods for specifying the PDL syntax (see Figure 7.6). For example, the code to parse a "while" statement, which is an option of "statement" grammar unit, is:

```
while =
    #while expression [compare: withType: #Boolean at: mark]
    #do  statement
```

where "while" is the grammar unit name for "while" statement; the identifiers following "#" sign will be parsed as "reserved word"s; "expression" and "statement" are grammar units which are specified in the same format as "while"; the string in the square bracket is a method which will perform semantic operation when an "expression" has been successfully parsed, here, it will check it that expression returns a boolean type.

After the PDL syntax has been specified, the main task is to design and implement the semantic actions validate operation based on the rules discussed in section 7.2. During the implementation of PDL compiler, we also reuse the error/confirmation message reporting mechanism from Smalltalk compiler.

<module >	::=	**module** <identifier> (<formal parameter list>) <declaration> <statement list> **end .**
<formal parameter list>	::=	<empty> | <parameter list>
<parameter list>	::=	<parameter> <rest of parameter list>
<parameter>	::=	<identifier> : <in out attribute> <identifier>
<in out attribute>	::=	**in** | **out** | **inout**
<rest of parameter list>	::=	<empty> | ; <parameter list>
<declaration>	::=	<empty > | <local variable list> | <global variable list>
<local variable list>	:=	**local** <identifier list> : <identifier> ;
<identifier list>	::=	<identifier> <rest of identifier list>
<rest of identifier list>	::=	<empty> | , <identifier list>
<global variable list> <identifier> ;	::=	**global** <identifier list> : <identifier> **importedfrom**
<statement list>	::=	<statement> ; <rest of statement>
<rest of statement list>	::=	<empty> | <statement list>
<statement>	::=	<assignment> | <condition> | <while> | <for> | <block> | <call>
<assignment>	::=	<identifier> := <expression>
<condition>	::=	**if** <expression> **then** <statement> <rest of condition>
<rest of condition>	::=	<empty> | **else** <statement>
<while>	::=	**while** <expression> **do** <statement>
<for>	::=	**for** <expression> **do** <statement>
<block>	::=	**begin** <statement list> **end**
<call>	::=	<identifier> (<argument list>) <rest of call>
<argument list>	::=	<empty> | <expression> <rest of argument list>
<rest of argument list>	::=	<empty> | , <argument list>
<rest of call>	::=	<empty> | <block>
<expression>	::=	<subexpression> <rest of expression>
<rest of expression>	::=	<empty> | <relop> <subexpression>
<subexpression>	::=	<term> <rest of subexpression>
<rest of subexpression>	::=	<empty> | <addop> <term>
<term>	::=	<factor> <rest of term>
<rest of term>	::=	<empty> | <mulopp> <factor>
<factor>	::=	<identifier> | <character> | <string> | <number> | <call> | (expression) | **true** | **false**
<relop>	::=	< | <= | = | <> | > | >=
<addop>	::=	+ | -
<mulopp>	::=	* | /

Figure 7.6: Grammar Syntax of PDL in SoftClass

Chapter 8

Conclusion

8.1 Summary

Three of the much-vaunted advantages of OO modeling and design turned out to be, at times, at odds: 1) building application models that users can understand and "relate to", 2) a seamless transition between analysis and design, and 3) achieving greater code reuse. Existing OO modeling methodologies prescribe notations, processes, and guidelines which, if followed, do ensure that analysis-level OO models reflect application semantics. As we move into design, implementation-level considerations such as performance and (greedy) code-reuse may distort analysis level models, and the transition is seamless no more. OO language theories have long grappled with the distinctions between classes and types. In an attempt to reconcile the three advantages/objectives mentioned above, the data abstraction mechanism in this thesis addresses similar issues, but for the case of OO analysis and design. This research results in the explicit representation of design models as a modern day "free association" between application (analysis) models and implementation/programming models (generic data structures); neither model has to please/satisfy the other's ancestors, but when they do, greater code reuse take place. One of the advantages of our handling of data has to do with the support of an easy to use, yet verifiable algorithm design language.

8.2 Future Directions

The work presented in this thesis can be extended in several directions. First, we need to test our approach by investigating some real software products and get the quantitative result of the degree of reuse using our approach. Second, we need further exploration of how to distinguish precisely the application-dependent versus data structure-dependent operation, i.e. AIO, LLASO, HLASO, discussed in Section 6.4.3. The third direction has to do with code-generation for target implementation languages. This involves: 1) translating already defined operations from the design language into the implementation language, and 2) generating code, directly or via an intermediary design-level representation, for those operations that weren't explicitly defined by developers but whose appropriateness was validated by the design language compiler (see Chapter 7). Belkhouche presented a prototyping system that automatically generates compilable prototypes by transforming an abstract data type into a program [Bel91]. We should have no problem translating the pseudo-code into complete programs of the target language thanks to the expressiveness of the PDL and to the inherent data abstraction mechanisms, which make it language-independent.

Bibliography

[Agr86] William W. Agresti. The conventional software life-cycle model: Its evolution and assumptions. New Paradigm for Software Development, 1986.

[Ara86] G. Arango, I. Baxter, P. Freeman, and C. Pidgeon. TMM: Software Maintenance by Transformation. *IEEE Software*, pp. 27-39, May 1986.

[Bas87] Paul G. Bassett. Frame-Based Software Engineering. *IEEE Software*, pp. 9-16, July 1987.

[Bel91] Boumediene Belkhouche. Generation of ada and pl/1 prototypes from abstract data type specifications. *Systems Software*, pages 255-264, 1991.

[Bla89] G. Blaschek, G. Pomberger, and A. Strizinger. A comparison of object-oriented programming languages. *Structured Programming*, pages 187 197, October 1989.

[Boo91] Grady Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings Publishing Company. In., 1991.

[Coo92] William Cook. Interfaces and specifications for smalltalk-80 collection classes. In *OOPSLA '92*, pages 1–15, October 1992.

[Cox87] Brad J. Cox. *Object-Oriented Programming: An Evolutionary Approach.* Addison Wesley, 1987.

[Cox90] Brad J. Cox. Planning the software revolution. *IEEE Software*, 7(11):25–35, November 1990.

[Car85] L. Cardelli and P. Wegner. On understanding types, data abstractions, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Coa91] Peter Coad and Edward Yourdon. *Object-Oriented Analysis.* Prentice Hall, 1991.

[Coa91a] Peter Coad and Edward Yourdon. *Object-Oriented Design.* Prentice Hall, 1991.

[Cha92] Dennis de Champeaux, Doug Lea, and Penelope Faure. The process of object-oriented design. In *OOPSLA '92*, number 45-62, October 1992.

[Cha92a] Dennis de Champeaux. Toward an object-oriented software development process. Technical Report HPL-92-149, Hewlett-Packard Laboratories, November 1992.

[DOD83] DOD U.S. Department of Defense. *Ada Reference Manual. ANSI/MIS-STD 1815 (Jan.).* U.S. Govt. Printing Office, 1983.

[Dah66] O. J. Dahl and K. Nygaard. Simula- an algol-based simulation language. *Communications of the ACM*, 9:671–678, September 1966.

[Dan88] Scott Danforth and Chris Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–72, March 1988.

[Duc92] R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *OOPSLA '92*, pages 16–24, October 1992.

[Els82] Bob Elschlager and Jorge Philipps. Automatic Programming. in *Handbook of Artificial Intelligence*, ed. Ed. Feigenbaum, vol. II, pp. 295-380, William Kaufmann Inc., 1982.

[Emb87] David W. Embley and Scott N. Woodfield. A knowledge structure for reusing abstract data types. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 360–368. ACM, 1987.

[Fis87] Gerhard Fisher. Cognitive View of Reuse and Design. *IEEE Software*, pp. 66-72, July 1987.

[Fre87] Peter Freeman. Reusable Software Engineering: COncepts and Research Directions. in *Tutorial: Software Reusability*, ed. Peter Freeman, pp. 10-23, 1987.

[Fut87] Kokichi Futatsugi, Joseph Goguen, and Jose Meseguer. Parameterized programming in obj2. In *Proceedings of the Ninth International Conference on Software Engineering*, number 337-346, 1987.

[Gog84] Joseph A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineer*, pages 528–543, September 1984.

[Gog86] J. Goguen. Reusing and Interconnecting Software Components. *Computer*, pp. 16-28, February 1986.

[Gru88] Galen Gruman. Early Reuse Practice Lives up to its Promise. *IEEE Software*, pp. 87-91, November 1988.

[Hel91] R. Helm and Y. S. Maarek. Integrating Information Retrieval and Domain Specific Approaches for Browsing and Retrieval in Object-Oriented Class Libraries. in *Proceedings of OOPSLA'91*, pp. 47-61, ACM Press, Phoenix, Arizona, 6-11 October, 1991.

[Hor84] Ellis Horowotz and John B. Munson. An Expansive View of Reusable Software. *IEEE Transactions on Software Engineering*, vol. 10(5), pp. 477-487, 1984.

[Kai87] G. E. Kaiser and D. Garlan. Melding Software System from Reusable Building Blocks. *IEEE Software*, p. 17-24, July 1987.

[Kor90] Tim Korson and John D. McGregor. Understanding object-oriented: A unifying paradigm. *Communications of the ACM*, 33(9):40–60, October 1990.

[Lie86] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *OOPSLA '86*, October 1986.

[Lip92] Stanley B. Lippman. *C++ primer*. Addison-Wesley, 1992.

[Mar91] James Martin and James J. Odell. *Object-Oriented Analysis and Design*. Prentice Hall, 1991.

[Mey87] Bertrand Meyer. Reusability: The case for object-oriented design. *IEEE Software*, pages 50–64, March 1987.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, Englewood Cliffs, N.J., 1992.

[Mil84] R. Milner. A Proposal for Standard ML. In *Proceedings of the Symposium on LISP and Functional Programmings (Austin, Tex.)*, number 184-197, August 1984.

[Mil90] Hafedh Mili, John Siber, and Yoav Intrator. An Object-Oriented Model Based on Relations. *Journal of Systems and Software*, vol. 12, pp. 139-155, 1990.

[Mil91] Hafedh Mili. Handling Design and Code in SoftClass. Department of Computer Science, University of Quebec at Montreal, May 1991.

[Mil93] Hafedh Mili and Haitao Li. Data abstraction in softclass, an oo case tool for software reuse. In *Proceedings of TOOLS USA '93*, August 1993.

[Mil93a] Hafedh Mili, Mustapha Friha, Francois Gros d'Aillon, Pascal Lagasse, Haitao Li, and Abdu Errahman El Wahidi. Enhancing software reuse for distributed systems management: Final report. Technical report, Department of Maths and Computer Science, University of Quebec at Montreal, January 1993.

[Mil93b] H. Mili, R. Rada, W. Wang, K. Strickland, C. Boldyreff, L. Olsen, J. Witt, J. Heger, W. Scherr, and P. Elzer. Practitioner and SoftClass: A Comparative Study of Two Softwrae Reuse Research Projects. *Journal of Systems and Software*, 1993 To Appear.

[Mil93c] H. Mili, O. Marcotte, and A. Kabbaj. *Intelligent Component Retrieval for Software Reuse*, July 1993. Submitted to the Third Maghrebian Conference on Artificial Intelligence and Software Engineering.

[Moo86] D. A. Moon. Object-Oriented Programming with Flavors. *OOSPLA '86 Conference Proceedings*, pp. 1-8, Portland, Oregon, 1986.

[Pet87] Lawrence Peters. in *Advanced Structured Analysis and Design*. ed. Randall W. Jensen. Prentice Hall, 1987.

[Pri87] Ruben Prieto-Diaz. Domain Analysis for Reusability. in *Proceedings of COMPSAC'87*, pp. 23-29. IEEE Press, 1987.

[Rum91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Ste87] Lynn Andrea Stein. Delegation is inheritance. In *OOPSLA '87 Proceedings*, pages 138–146, October 1987.

[Str67] C. Strachey. Fundamental concepts in programming languages. Lecture Notes for International Summer School in Computer Programming, Copenhagen, August 1967.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Menlo Park, California, 1986.

[Str88] Bjarne Stroustrup. What is object-oriented programming? *IEEE Software*, pages 10–20, May 1988.

[Tay88] David A. Taylor. *Object-Oriented Technology: A Manager's Guide.* Addison-Wesley, 1988.

[Tha81] R. H. Thayer and A. B. Pyster. Major Issues in Software Engineering Project Management. *IEEE Transactions on Software Engineering*, vol. 7(4), p. 333-342, July 1981.

[Wir90] Rebecca Wirfs-Brock, Brian Wiljerson, and Lauren Wiener. *Design Object-Oriented Software.* Prentice Hall, 1990.

[Wir83] N. Wirth. *Programming in Modula-2.* Spinger-Verlag, New York, 1983.

[You79] E. Yourdon and L. Constantine. in *Structured Design: Fundamentals of Discipline of Computer Programs and System Design.* Prentice-Hall, Englewood Cliffs, N.J., 1979.

[Zlo75] M. M. Zloof. Query by Example. in *Proc NCC*, vol. 44, 1975.