

Object-Oriented Programming With Class Dictionaries

Karl J. Lieberherr, Northeastern University

April 27, 2004

Copyright ©1988 Karl Lieberherr

Abstract

A class dictionary defines all data structures which appear in a program as well as a language for describing data specified by the data structures. We demonstrate that class dictionaries are ideal for simplifying object-oriented programming. Our class dictionary based approach to object-oriented programming is independent of any particular programming language, so it is applicable to a large variety of object-oriented systems. The experience in designing and using over one hundred class dictionaries has resulted in a set of useful design techniques. This novel approach to object-oriented programming makes interesting links between language design, data structure design and data base design.

Appeared in: *LISP AND SYMBOLIC COMPUTATION: An International Journal*, vol. 1, no. 2, 1988.

1 Introduction

An important part of any object-oriented program is the definition of its data structures. When the data structure definitions are collected in a systematic manner separate from the program, they are called a *class dictionary*.

We have used a novel class dictionary based approach to object-oriented programming since winter 1986. Our technique has been implemented as a prototype generator, called DemeterTM, which has been applied by over 250 students to their course and research work. The purpose of this paper is to report on our positive experience with the class dictionary based approach to object-oriented programming. In this paper we will focus on explaining our abstraction mechanisms and on developing class dictionaries.

We assume that the reader of this paper is familiar with the basic concepts of object-oriented programming. The primary objective of our approach is to improve programmer productivity. We achieve this objective by

- generating useful code automatically from a class dictionary.
- using object-oriented programming which is well known for its many advantages, such as shorter programs, modularity, flexibility etc.
- providing a uniform interface for dealing with parameterized “context-free” data structures and languages.
- using parameterized classes which improve code and data structure reusability.

A class dictionary can be viewed as an interface between the user and object-oriented systems. The interface provides programming language independent access to a large number of object-oriented systems and the software provided by the interface removes many of the accidental difficulties of programming. A class dictionary contains definitions of all data structures appearing in a program. A class dictionary has a second important function: it can be interpreted as a language definition which introduces an application specific notation. This notation is used for describing data defined by the data structures. These data descriptions are translated automatically into the internal representation of the data structures. A class dictionary can also be viewed as a database definition. Therefore class dictionary design draws on earlier work on data structure, language and data base design.

To describe class dictionaries we use an EBNF like notation and Entity-Relationship diagrams. To make it easier to navigate through the EBNF, we apply a cross reference generator to our class dictionaries. The Entity-Relationship diagrams represent the cross reference table in two dimensional form.

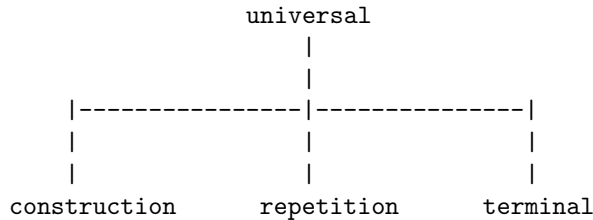


Figure 1: Class hierarchy

2 Class dictionaries

A class dictionary contains definitions of all data appearing in a program. It not only defines all the data structures, but also their attributes, such as concrete syntax (i.e. the detailed textual representation of objects) and type information (we use class and type as synonyms). The term class dictionary is related to the term data dictionary which is widely used in conjunction with data-base management systems. Both types of dictionaries contain meta-data: data about data. In our class dictionaries we deal with meta-data about meta-data which we call parameterized classes.

Our class dictionaries look like grammars and are more powerful than the conventional data dictionaries. We add the following concepts which improve the abstraction capability.

- Inheritance
- Parameterized data structures and languages and
- Modules, for partitioning projects and for abstracting from implementation details.

Parameterized classes will be explained later; modules are not discussed in this paper.

Our class dictionaries store definitions of object classes. The class dictionary defines all classes needed in an application as well as their hierarchical inheritance relationships. The class *universal* is the superclass of all classes which can be instantiated. This class has exactly three subclasses: *construction*, *repetition* and *terminal* as shown in Fig. Class hierarchy.

The class **construction** contains all hierarchical objects which have a finite number of named components. By hierarchical we mean that the components may be non-primitive objects. The components may be of different types. A *construction class* is a class which is defined by a construction production. A construction class inherits from class **construction**. A member of a construction

class is similar to a record which has fields or slots. An example of a definition of a construction class is

```
Division =  
  <contains> DepartmentList [<partOf> Company].
```

A construction class is introduced with =. The two named components are labeled “contains” and “partOf”. “contains” and “partOf” are *instance variables* or *slots* or *local state variables* of class `Division`. The brackets indicate an optional component. If a component is not labeled, the name of the component is used for referring to the component. Labeled components are referred to by the label name. If several components have the same type, we can use an abbreviation. Instead of writing $\langle x \rangle \top \langle y \rangle \top \langle z \rangle \top$ we can use $\langle x, y, z \rangle \top$. Notice the difference between a construction class and the class `construction`. A construction class is a subclass of class `construction`.

The class `repetition` contains all objects having an ordered sequence of components which are all of the same type. The components are not named individually; they are numbered (zero-based, i.e., the first one has number 0).

A *repetition class* is a class which is defined by a `repetition` production. A repetition class inherits from class `repetition`. A member of a repetition class is similar to a list. An example of a repetition class definition is

```
DepartmentList ~ {Department}.
```

A repetition class is introduced with ~ to remind you of a repeating wave. Another reason for not using = is to simplify the reading (by man and machine) of class definitions. The `DepartmentList` class consists of a list of departments which might be subdivided into payroll department, marketing department etc.

The class `terminal` contains all objects which are not further decomposed. It has several subclasses such as `Ident`, `Number`, and `String`.

The class `universal` is defined by the following production:

```
universal ! <attribute> anytype <father> anytype.
```

We use an exclamation point (!) for defining a class which is abstract and which has no implicit inheritance from another class. On the right of ! we list instance variables with their types and we give the classes from which this class inherits. The class `universal` is inherited into all other classes which can be instantiated and therefore all objects have the instance variables attached to class `universal`.

The instance variable called `attribute` is used for attaching additional information to hierarchical objects. The instance variable called `father` can be used to store the object of which the current object is a component. This feature allows bottom-up traversal of an object’s hierarchy. As an example, consider boxes which contain pencils. In every pencil we store in instance variable `father`

the box the pencil is located in. The type `anytype` specifies that any object can be stored independent of its type.

The class `construction` is defined by the production:

```
construction ! *inherit* universal.
```

This class inherits from the class `universal`. We use the reserved word `*inherit*` followed by a list of classes to specify inheritance.

Although the class `construction` does not have its own instance variables, it is a useful class for attaching methods which are defined for all hierarchical objects defined by `construction` productions. The class `repetition` is defined by the production:

```
repetition ! <child> anytype *inherit* universal.
```

The instance variable `child` stores the elements of the ordered sequence represented by the `repetition` class instance. It is a sequence of instances.

The class `terminal` is defined by the production:

```
terminal ! <val> anytype *inherit* universal.
```

The `val` instance variable stores the value of the terminal.

This class organization is class dictionary independent. The four classes `universal`, `construction`, `repetition` and `terminal` are predefined in all class dictionaries. The class dictionary defines additional relationships between classes and subclasses. A class can have several superclasses; that is, multiple inheritance is supported.

An alternation production defines an alternation class which is an abstract class. An abstract class is a class which cannot be instantiated. It is only used through inheritance. As a synonym to *alternation* class we use *union* class. Note that there is *no* predefined class `alternation` as might be expected with analogy to the `construction` and `repetition` productions. The reason is that alternation classes are abstract.

An alternation production does not necessarily imply inheritance. The reason is that there are many situations where we don't take advantage of inheritance and in this case we would like to keep the inheritance relationships minimal. This simplifies object-oriented programming especially in the presence of multiple inheritance.

An example of a union class is:

```
Employee : Manager | Secretary | Engineer.
```

The class `Employee` can be used for attaching methods to it and it can be inherited in other classes, e.g. the `Engineer` class might inherit from the `Employee` class. In this example it makes sense that all three classes: `Manager`, `Secretary`, and `Engineer` inherit from the class `Employee`. This can be achieved by

```
Employee : Manager | Secretary | Engineer *common*.
```

When the keyword `*common*` is used it is assumed that the alternatives are defined by construction productions or alternation productions which are defined in terms of other alternation or construction productions. The alternatives are inheriting from the alternation class.

The above definition is from a class definition point of view equivalent to:

```
Employee ! .
Manager !! ... *inherit* Employee, construction.
Secretary !! ... *inherit* Employee, construction.
Engineer !! ... *inherit* Employee, construction.
```

We use `!!` and `!` to describe classes at the lowest level, just before the level of the class notation of the underlying object-oriented system. This notation is an intermediate notation which Demeter generates and which can be easily translated into the class notation of the underlying object-oriented system. Abstract classes are introduced with `!` and non-abstract classes with `!!`. The `...` are replaced by the instance variables defined in the three productions for `Manager`, `Secretary`, `Engineer`. This example shows that alternation productions which use the `*common*` keyword require that the underlying object-oriented system supports multiple inheritance.

Sometimes we would like to define instance variables for an alternation class which are also defined for all the alternatives. For example, all employees have a salary and a manager (with one exception). This can be expressed by:

```
Employee : Manager | Secretary | Engineer
*common*
  <salary> Number
  [<managedBy> Manager].
```

The common instance variables can be thought of as being added at the end of the three class definitions. The above definition is from a class definition point of view equivalent to:

```
Employee ! <salary> Number <managedBy> Manager.
Manager !! ... *inherit* Employee, construction.
Secretary !! ... *inherit* Employee, construction.
Engineer !! ... *inherit* Employee, construction.
```

Notice the relationship between `*common*` and `*inherit*`. The use of `*common*` implies inheritance.

We can now use the alternation production facility to define the class dictionary independent class organization:

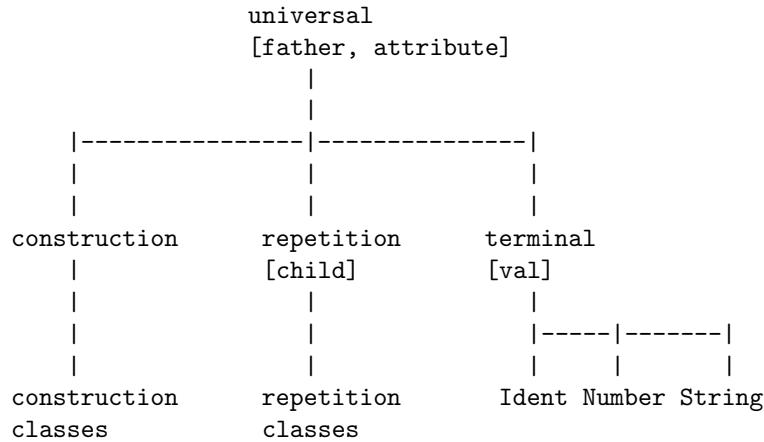


Figure 2: Class hierarchy organization

```

universal :
  construction |
  repetition |
  terminal
  *common*
    <attribute> anytype
    <father> anytype.
construction: ... *common*.
repetition : ... *common* <child> anytype.
terminal : ... *common* <val> anytype.
  
```

The dots are filled in by a specific class dictionary. Fig. “Class hierarchy organization” shows a picture of the class hierarchy. The instance variables are shown in square brackets.

The concrete syntax that is defined in a class dictionary makes it easy to read an object from a file or to print an object in readable form. For example, for a class whose instances represent customers, the class dictionary not only stores the number, name and address fields, but also how to label those fields and how to print them with proper indentation and spacing. For every class the class dictionary defines a print method. However the definition of a read routine is optional because not every class may be readable. The class definitions have to satisfy certain rules for the parsing methods to be generated.

For the purpose of type checking, the class dictionary stores the type for each component of an object in a construction class. Also, the type of the unnamed components of an object in the repetition class is stored in the class dictionary.

3 Inheritance and Parameterization

We have already introduced the concept of inheritance into our class dictionaries. We now contrast the concepts of inheritance and parameterization. The discussion of parameterization is kept at a high level and is incomplete. For other issues related to parameterized classes we refer the reader to [LR88a] or [LR88b].

Both concepts, inheritance and parameterization, serve to introduce abstractions.

3.1 Abstraction

We use the keyword `*inherit*` to express inheritance. For example, we can define a class

```
Organization =  
  <contains> anytype  
  <partOf> anytype  
  <managedBy> anytype.
```

This class can be used to define two similar organizations:

```
Division    = *inherit* Organization.  
Department = *inherit* Organization.
```

The `Division` and `Department` class will have all the functionality of the `Organization` class because of inheritance.

3.2 Parameterization

We have seen how inheritance can be used for abstraction. Unfortunately, inheritance alone is a poor mechanism for flexible abstraction. The following example explains why. We define two classes as follows:

```
Department =  
  <contains> EmployeeList  
  <partOf> Division.  
  
College =  
  <contains> AcadDepList  
  <partOf> University.
```

These two classes have a lot of similarities which cannot be expressed with inheritance. The problem is that inheritance does not allow us to talk about the different types of the instance variables. Therefore we define two parameterized classes:

```
Organization(SubOrganization, SuperOrganization) =  
  <contains> List(SubOrganization)  
  <partOf> SuperOrganization.
```

```
List(P) ~ {P}.
```

SubOrganization and SuperOrganization are formal type parameters. Actual parameters are supplied when the parameterized class Organization is used to create a class. These two parameterized classes can be used to define the two similar organizations succinctly:

```
Department =  
  *inherit* Organization(Employee, Division).  
College =  
  *inherit* Organization (AcadDep, University).
```

With parameterization it is also possible to define classes with a flexible inheritance mechanism. Consider the production

```
SandwichedInstance(P) =  
  <left> StringList *inherit* P <right> StringList.
```

It defines the parameterized class SandwichedInstance. Only when we make an instance of this parameterized class do we have to specify from which class to inherit by giving an appropriate actual parameter for P.

The parameters of a parameterized class may range over classes. When a parameterized class is fully instantiated, i.e. each formal parameter is assigned a class, the parameterized class becomes a regular class.

Parameterized classes provide a powerful abstraction mechanism for defining classes. The main advantages of parameterized classes are:

1. Similar class definitions can be combined into one. This makes class dictionaries more concise.
2. Common methods don't have to be duplicated several times: they can be attached to parameterized classes. This makes programs shorter and more reusable.

3.3 Preprocessing of parameterization

Parameterized classes support the development of reusable software. This software is attached to classes which are generated from the parameterized class definitions. We are using an expansion technique which we explain here with two representative examples.

For programming with parameterized classes it is necessary that factory objects for making instances are available or that the underlying object-oriented

system provides a macro facility. For methods which create instances that depend on the type parameters of a parameterized class it will be necessary to either make the instances through factory objects or to provide a macro which generates a method which makes the right instances.

Our expansion of parameterized classes proceeds in two different steps: we separately translate the class dictionary for the purpose of class definitions and for the purpose of parsing and everything else (pretty printing, type checking, etc.).

Classes The instantiation of parameterized classes may trigger several class definitions. For example, the class dictionary

```
Organization(SubOrganization, SuperOrganization) =
  <contains> List(SubOrganization)
  [<partOf> SuperOrganization].
List(SubOrganization) ~ {SubOrganization}.
```

```
Department =
  *inherit* Organization(Employee, Division).
College =
  *inherit* Organization(AcadDep, University).
```

generates the following class definitions (we use lower case for the classes which are defined in the underlying object-oriented system):

```
organization !
  <contains> anytype
  [<partOf> anytype] *inherit* construction.
list ! *inherit* repetition.

department !! *inherit* division_employee-organization.
division_employee-organization !! *inherit* organization.
employee-list !! *inherit* list.

college ! *inherit* university_acaddep-organization.
university_acaddep-organization !! *inherit* organization.
acaddep-list !! *inherit* list.
```

This will create the class `organization` as shown in Fig. “Organization”. The auxiliary classes are named by concatenating the actual parameters with the parameterized class name.

This hierarchy shown in Fig. “Organization” has been constructed to facilitate reusability of the parameterized class `Organization`. The supplier of the parameterized class `Organization` can define functionality which holds for all organizations. The consumer of `Organization` gets all this functionality for free and can override or expand it.

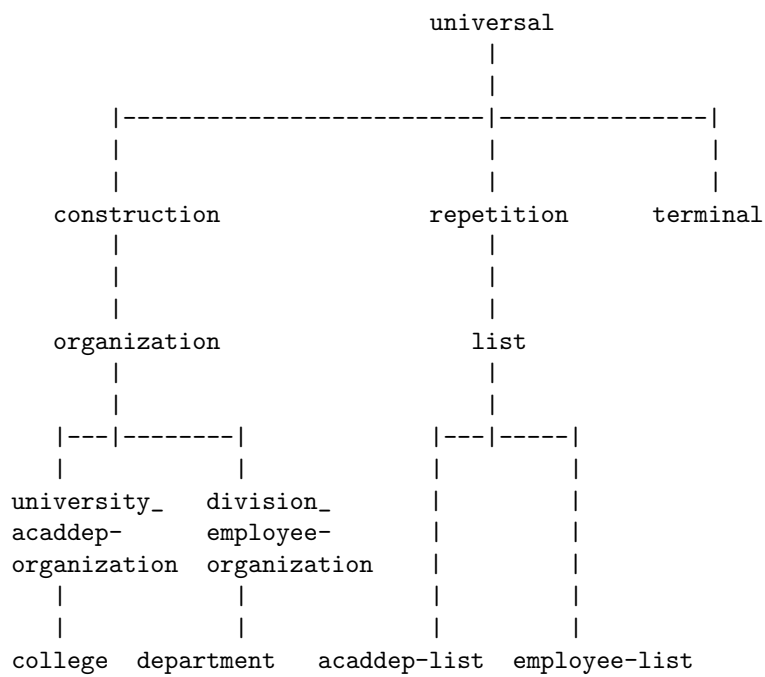


Figure 3: Organization

Parsing, type checking, etc. For the purpose of parsing, type checking etc., the translation behaves like a macro expansion. We consider the **Organization** example from above. It is translated into

```
Department =
  <contains> Employee-List
  [<partOf> Division].
Employee-List = {Employee}.
```

```
College =
  <contains> AcadDep-List
  [<partOf> University].
AcadDep-List = {AcadDep}.
```

by expanding the parameterized definitions and all inheritance. The translation is given to a parser generator, pretty printer generator, type checker generator, etc. This translation mechanism for parameterized classes in combination with the type checker generator will make sure that the user of the **Organization** parameterized class puts a suborganization list into the **contains** instance variable. For example, the generated type checker will check that the user of the **Department** class puts an instance of class **Employee-List** into the **contains** instance variable.

The expansion of inheritance has a special rule for the predefined classes **universal**, **construction**, **repetition**, **terminal** and all subclasses of **terminal** such as **Ident** and **Number**: they are not expanded at all. In other words, the keyword ***inherit*** is without effect, e.g., ***inherit* Ident** expands into **Ident**.

This concludes our informal introduction to class dictionaries. The formal syntax of class dictionaries is defined by the following class dictionary which is central to the implementation of Demeter:

```
Grammar = <rules> List( Rule) .
Rule = <ruleName> Ident [FormalParameters] Body ".".
FormalParameters =
  "(" <formals> CommaList( FormalParameter) ")".
FormalParameter =
  <parameterName> Ident.

; Bodies
Body : ConstructBody | RepetitBody |
      AlternatBody | NeutralBody .
ConstructBodyCore= <elements> List( AnyInstance) .
ConstructBody = "=" *inherit* ConstructBodyCore.
RepetitBody = "~"
              <first> List( StringOrPP)
```

```

    [ <nonempty> BasicInstance ]
    <repeated> RepeatedInstance
    <second> List(StringOrPP).
AlternatBody = ":"
    <alternatives> BarList(OptStarredBasicInstance)
    ["*common*" <common> List(AnyInstance)].
NeutralBody : UnInstBody | InstBody.
UnInstBody = "!" *inherit* ConstructBodyCore.
InstBody = "!!" *inherit* ConstructBodyCore.

; Instances
StarredInstance = "*" *inherit* BasicInstance.
OptStarredBasicInstance :
    BasicInstance | StarredInstance.

RepeatedInstance =
    "{" *inherit* Sandwiched(BasicInstance) }".

AnyInstance : Instance |
    InheritInstance | StringOrPP *common*.
Instance :
    *OptLabeledInstance(CommaList(Label)) |
    *OptionalInstance
    *common*
    <instanceName> Ident
    [<actualParameters>
        CommaList(ParAssoc(BasicInstance))].
OptLabeledInstance(L) :
    BasicInstance | LabeledInstance(L)
    *common*.
BasicInstance =
    <instanceName> Ident
    ["(" <actualParameters>
        CommaList(ParAssoc(BasicInstance)) ")"].

LabeledInstance(L) = <labels> L
    *inherit* BasicInstance.
Label = "<" <labelName> Ident ">".
InheritInstance =
    "*inherit*" <inheritsFrom> CommaList(BasicInstance)
    ["*override*" <overrides>
        NonEmptyList(LabeledInstance(CommaList(Label))) "*end*"].
OptionalInstance =
    "[" *inherit* Sandwiched(OptLabeledInstance(Label)) "]".

```

```

StringOrPP : String | PPCommands *common*.
PPCommands : PPindent | PPunindent |
  PPskip | PPspace *common*.
PPindent = "+".
PPunindent = "-".
PPskip = "*l".
PPspace = "*s".

; auxiliary parameterized classes
ParAssoc(Exp) =
  ["=>" <formalParameter> Ident]
  <actualOrFormalParameter> Exp.

Sandwiched(P) =
  <left> List(StringOrPP) <inner> P
  <right> List(StringOrPP).

; lists
List(S) ~ {S}.
NonEmptyList(S) ~ S {S}.
BarList(S) ~ S {"|" S}.
CommaList(S) ~ S {"," S}.

```

4 Environment definition

The class dictionaries define many useful utilities for presenting, constructing or manipulating objects. These utilities are either automatically generated from the class dictionaries or provided as generic programs which interpret the class dictionaries. All this software is provided by the Demeter system. It is a meta-system for object-oriented programming which abstracts from a large number of object-oriented systems. Our current prototype works with Lisp and Flavors.

A meta-system for object-oriented programming has many advantages:

- It removes many accidental difficulties of programming. When we write programs we model an aspect of the real world. The mapping of objects in the the outside world to the objects in our programs and vice-versa should be as easy as possible. This task is simplified considerably by Demeter since it automatically provides object construction and object presentation utilities.
- It allows for a uniform interface. Demeter provides a uniform interface to a large class of object-oriented systems since the class dictionary notation is system independent. This makes it easier to learn several object-oriented

systems. For each system one has to learn how to write the methods; the class definitions are handled by the uniform Demeter notation.

- It provides communication between different systems. Demeter facilitates the communication between different object-oriented systems since the object construction and presentation facilities provide the software for communication through text files.
- It simplifies implementation of new object-oriented programming systems. Object-oriented programming systems are relatively young and there are numerous new proposals for new programming systems which support object-oriented programming. Unless these new systems are radically different from existing ones, they don't have to provide all the facilities which are already implemented in the meta system (Demeter).

We call the collection of software that is generated from the class dictionary a *programming environment skeleton* for the class dictionary. The idea behind a programming environment skeleton is that we make all the information which is given in a class dictionary readily available to the programmer.

Here is a summary of the software which a class dictionary defines:

- Object construction utilities. There are at least three ways for constructing objects. In the first we read an object description from a text file and we have a parser which checks for correctness and constructs the object. In the second, the object is built under human control by giving directives to an object editor. This approach frees the user from knowing the concrete syntax. In the third, the object is built under machine control following a systematic pattern. In this last approach, the class dictionary defines a program skeleton which can be tailored for building parameterized objects.
- Object presentation utilities. Objects can be presented in at least two ways. The drawing program shows all the parts of an object, including the types. The pretty printer shows only the class terminal instances plus the concrete syntax.
- Class definitions. A class dictionary defines all the object classes for the underlying object-oriented system.
- Scanner. The class terminals which are used in a class dictionary are defined by an associated scanner specification. The scanner specification defines a scanner which works in harmony with the parser defined by the class dictionary.
- Consistency checking utilities. A class dictionary defines a type system. There are three ways of taking advantage of the type system: At the class

dictionary level we use the subtype concept to check for legal inheritance. This provides a safe framework for reusing software. At the object-oriented programming language level we use it for type inference and type checking at compile time which serves to catch possible run-time type errors at compile-time. This eliminates many conceptual programming problems. If the type checking is not successful at compile time, the types will be checked at run-time. This will enforce that only objects can be built which are legal with respect to the class dictionary.

- Object traversers. When algorithms perform their computations they traverse objects. Therefore it is convenient to start with an object traversing program which is nicely documented, instead of implementing an algorithm from scratch. The object traversing program can be viewed as a program skeleton which will be tailored for the specific application. The use of the skeleton not only reduces the typing work but also promotes a good object-oriented programming style.
- Growth plan. There are many ways of “growing” the program skeleton in to the full application program. The growth plan which is generated from a class dictionary suggests in which order we best develop the methods so that we can test our growing program frequently only after minimal updates.
- Translation by example tool. An interesting class of translation tasks is simple enough that it can be specified by examples in a similar way as macros are specified in a macro-by-example extension to Scheme. We have generalized the Scheme idea so that it works with any language definable in the Demeter system.

5 Class dictionary design techniques

Defining the class dictionary for an application is a very important and interesting task. The class dictionary will determine all the data structures which in turn will determine the efficiency of the algorithms. The class dictionary also influences the reusability of the resulting code.

There is a need to break large class dictionaries into modular pieces which are easier to manage. This topic of modularization will be discussed elsewhere. In this section we have collected a set of useful design techniques for those modular pieces of class dictionaries.

5.1 Abstraction

Good abstractions in a class dictionary have numerous benefits. The class dictionary usually becomes cleaner and shorter, and an object-oriented program

which uses the class dictionary will have a cleaner structure which avoids duplication of functionality. The goal of abstraction is to factor out recurring patterns and making an “instance” of the recurring pattern where it is used. First we discuss abstraction by auxiliary classes, often called mixins. A *mix*in is a class designed to augment the description of its subclasses. For example, the mixin `QuadrangularElement` allocates three instance variables for width, height and length. This mixin is included in object definitions which define objects having the characteristic of a quadrangular object. Mixins are often abstract classes. Abstract classes are introduced with `!`.

5.1.1 Mixins and promotion

Suppose we want to define `Coin`, `Brick` and `Box` classes. Suppose that `Element` is the union of these three classes. A coin has a weight and a position. A brick has a color, a width, height, length, a weight and a position. A box has contents, a width, height and length, a weight and a position. In the following class dictionary (version 1) we use two auxiliary classes (mixins): `QuadrangularElement` and `AnyElement`.

```

QuadrangularElement !
  <width> Number <height> Number <length> Number.
AnyElement !
  <weight> Number <position> Vector.

Element : Coin | Brick | Box.
Coin = *inherit* AnyElement.
Brick = <color> Ident
  *inherit* QuadrangularElement, AnyElement.
Box = <contents> ElementList
  *inherit* QuadrangularElement, AnyElement.

```

The `Coin` class inherits all functionality and instance variables from the `AnyElement` class. The `Brick` and `Box` class both inherit from the `QuadrangularElement` and the `AnyElement` class. Recall that all three classes also inherit from the class `construction`. Thus this class dictionary makes use of multiple inheritance.

Multiple inheritance is useful because the functionality we define for the `QuadrangularElement` class and the `AnyElement` class is automatically available to the `Brick` and `Box` classes. The `Coin` class has the functionality of the `AnyElement` class available. For example, if we attach a method `m` to the `AnyElement` class, we can send the `m` message to an instance of class `Box`.

In version 2 we are using the same auxiliary classes but without multiple inheritance.

```

Element : Coin | Brick | Box.

```

```

Coin = AnyElement.
Brick = <color> Ident
      QuadrangularElement AnyElement.
Box = <contents> ElementList
     QuadrangularElement AnyElement.

```

Version 1 is better than version 2 because in version 1, all the functionality which is defined for the `AnyElement` class is available in the `Coin`, `Brick` and `Box` classes. In version 2 this functionality is available only indirectly: In order to activate a method attached to the `AnyElement` class it is necessary to send a message to the instance variable `AnyElement`. This holds for all three classes. The disadvantage is that the programs get longer and the hierarchical objects get bigger but don't store more information.

The third and last version is even worse since it does not factor out the recurring pattern.

```
Element : Coin | Brick | Box.
```

```

Coin = <weight> Number <position> Vector.
Brick = <color> Ident
      <width>, <height>, <length> Number
      <weight> Number <position> Vector.
Box = <contents> ElementList
     <width>, <height>, <length>,
     <weight> Number <position> Vector.

```

The recognition that a particular distinction arising in a subclass can be generalized to other classes is a common occurrence in object-oriented programming. Often there is a motivation to move structure up in the class lattice to increase the amount of sharing. We call this *promotion of structure* [SB86]. The step from version 3 to version 1 can be accomplished by promotion of structure. Version 1 can be improved by using `*common*`:

```

QuadrangularElement : Brick | Box
  *common* <width> Number <height> Number <length> Number.
AnyElement : Coin | Brick | Box
  *common* <weight> Number <position> Vector.
Coin = .
Brick = <color> Ident.
Box = <contents> ElementList.

```

Notice that in the last version, the order of the first two productions will determine the order of the instance variables in the `coin`, `brick` and `box` classes. This is important if objects are read in from files. The `*common*` clause allows a distributed definition of the structure of classes.

As a general rule it is best to avoid to define abstract classes explicitly with `!`. It is best to use an alternation production with implied inheritance (i.e. with `*common*`) to define an abstract class.

5.1.2 Parameterization

The parameterization approach to abstraction extends the auxiliary classes approach. Parameterization uses auxiliary parameterized classes for reinforcing the abstraction mechanism.

Consider the following class dictionary which introduces two classes (`Department` and `Division`) by using two parameterized classes (`Organization` and `List`).

```
Organization(SubOrganization, SuperOrganization) !
  <contains> List(SubOrganization)
  [<partOf> SuperOrganization]
  <managedBy> Employee.
```

```
List(P) ~ {P}.
```

```
Division = "Division"
  *inherit* Organization(Department, Company).
Department = "Department"
  *inherit* Organization(Employee, Division).
```

This class dictionary is much better than the following one which does not use parameterized classes:

```
Division = "Division"
  <contains> DepartmentList
  [<partOf> Company]
  <managedBy> Employee.
DepartmentList ~ {Department}.
```

```
Department = "Department"
  <contains> EmployeeList
  [<partOf> Division]
  <managedBy> Employee.
EmployeeList ~ {Employee}.
```

The parameterized version is better because it invests in the future. It is a well-known principle that solving a more general problem than the one under consideration often yields a better solution for the given problem. It is likely that the insight gained from the generalized problem will be of future benefit.

The class `Organization` is likely to be useful in many other applications. It improves the reusability of the class dictionary and associated software. To

demonstrate this, consider the problem of counting the number of employees in an organization.

With parameterization, the class dictionary builder might add the following two methods:

```
(defmethod (Organization :total) ()
  (send contains ':total))

(defmethod (List :total) ()
  (loop for s in child
    sum (send s ':total)))
```

Anyone who wants to use the classes `Organization` and `SubOrganization` with the method `:total` has to provide a method called `:total` attached to the class which is used as actual parameter for `SubOrganization`. Without abstraction we have to write the above two methods twice; once for `Division` and once for `Department`.

In the next example we use parameterization to personalize a language. We define a language `Sandwiched` which defines an instance “sandwiched” between two lists of strings. The following class dictionary (version 1)

```
Sandwiched(P) !
  <left> StringList *inherit* P <right> StringList.
Repetit = "~"
  <first> StringList [ <nonempty> Instance ]
  "{" *inherit* Sandwiched(Instance) }"
  <second> StringList.
OptionalInstance =
  "[" *inherit* Sandwiched(LabeledInstance) "]".
```

is better than (version 2)

```
SandwichedLabeledInstance =
  <left> StringList LabeledInstance <right> StringList.
Repetit = "~"
  <first> StringList [ <nonempty> Instance ]
  "{" SandwichedLabeledInstance }"
  <second> StringList.
OptionalInstance =
  "[" SandwichedLabeledInstance "]".
```

Although we use abstraction, we cannot precisely formulate the recurring pattern. Therefore the language defined by the second class dictionary is larger. The version 2 class dictionary is still better than (version 3)

```

SandwichedInstance =
  <left> StringList Instance <right> StringList.
SandwichedLabeledInstance =
  <left> StringList LabeledInstance <right> StringList.
Repetit = "~"
  <first> StringList [ <nonempty> Instance ]
  "{" SandwichedInstance "}
  <second> StringList.
OptionalInstance =
  "[" SandwichedLabeledInstance "]".

```

It is inconvenient to write programs for the version 3 class dictionary. The conclusion is: It is acceptable to make the language larger if you can introduce a nice abstraction. It is much better to parameterize the abstraction and avoid enlarging the language. The right abstraction simplifies programming considerably.

5.2 Naming

Good label names, class names and parameterized class names improve the readability of the associated object-oriented programs. We have adopted the following conventions: class and parameterized class names always start with a capital letter. Label names start with a small letter.

It is important that the instance variable names have a succinct mnemonic interpretation. Therefore it is often advisable to introduce labels for the purpose of better naming only.

5.3 Regularity

Regular definitions are without exception easier to learn, use, describe, and implement. They also make a class dictionary more reusable.

As an example we consider a fragment of the Modula-2 grammar, compare it with the corresponding fragment of the Pascal grammar and demonstrate that the Modula-2 grammar is more regular:

```

Statement = [Statements].
Statements : IfStatement | RepeatStatement.
StatementSequence ~ Statement {";" Statement}.
IfStatement =
  "if" <condition> Expression
  "then" <thenPart> StatementSequence
  "end".
RepeatStatement =
  "repeat"

```

```
StatementSequence
"until" <condition> Expression.
```

This Modula-2 grammar is better than the corresponding fragment of the Pascal grammar:

```
Statement : BeginEnd | IfStatement | RepeatStatement.
StatementSequence ~ Statement {";" Statement}.
BeginEnd = "begin" StatementSequence "end".
IfStatement =
  "if" <condition> Expression
  "then" <thenPart> Statement.
RepeatStatement =
  "repeat"
  StatementSequence
  "until" <condition> Expression.
```

Notice how the Modula-2 grammar is more systematic. Each statement has an explicit terminator avoiding the introduction of a construct which turns several statements into one.

5.4 Prefer alternation

Alternation productions should be used whenever possible. The reason is that a well designed object-oriented program will not contain an explicit conditional statement for the case analysis which needs to be done for an alternation production.

Example:

One way to define a Prolog clause is:

```
Clause = <head> Literal
[":-" <rightSide> LiteralList] ".".
```

However, the following definition will give a cleaner object-oriented program:

```
Clause : Fact | Rule.
Fact = "fact" <head> Literal ".".
Rule = "rule" <head> Literal ":-"
      <rightSide> LiteralList ".".
```

Although the concrete syntax is slightly different, both definitions of a Prolog clause store the same information. A program which processes a clause corresponding to the first definition will contain a conditional statement which tests whether `rightSide` is non-nil. A program which processes a clause corresponding to the second definition will delegate the conditional check to the underlying object-oriented system and it will not be explicitly contained in the program.

There are other reasons, besides having shorter programs, for the class dictionary which uses alternation:

- Modularity. The class dictionary is more modular. If we change the definition of a rule we don't have to change the definition of `Clause`.
- Space. The objects can be represented with less space since a fact will not have an instance variable `rightSide` which is always `nil`.

5.5 Normalization

When defining the class dictionary for database type applications, the theory of normal forms is relevant. The class dictionary should be written in normalized form. Normalization will make it easier to extend the class dictionary and it enforces a more systematic and clean organization. The normalization is based on the concepts of *key* and *functional dependency*.

In the following we adopt definitions from the relational database field to our class dictionaries which describe hierarchical data bases. The motivation behind these definitions is to introduce the concept of a normalized class.

Definition: An instance variable $V1$ of some class C is *functionally dependent* on instance variable $V2$ of C if for all instances of class C each value of $V2$ has no more than one value of $V1$ associated with it. In other words, the value of the instance variable $V2$ determines the value of instance variable $V1$. We also use the terminology: $V2$ *functionally determines* $V1$. The concept of functional dependency is easily extended to sets of instance variables.

Definition: A *key* for a class C is a collection of instance variables that (1) functionally determines all instance variables of C and (2) no proper subset has this property.

The concept of the key of a class is not a property of the class definition but rather a fact about an intended use of a class, i.e. the intended set of instances.

Consider the class

```
Employee =  
  <employeeNumber> Number  
  <employeeName> String  
  <salary> Number  
  <projectNumber> Number  
  <completionDate> String.
```

The key is `employeeNumber`. There are several problems with this class definition:

- Before any employees are recruited for a project, the completion date of a project can be stored only in a strange way: by making an instance of class `Employee` with dummy employee number, name and salary.

- If all employees should leave the project, all instances containing the completion date would be deleted.
- If the completion date of a project is changed, it will be necessary to search through all instances of class `Employee`.

Therefore it is better to split the above class definition into two:

```
Employee =
  <employeeNumber> Number
  <employeeName> String
  <salary> Number
  <projectNumber> Number.
```

```
Project =
  <projectNumber> Number
  <completionDate> String.
```

The key for `Employee` is `employeeNumber` and for `Project` it is `projectNumber`.

The reason why the first `Employee` class has problems is that the project number determines the completion date, but `projectNumber` is *not* a part of the key of the `Employee` class. Therefore we define: A class is *normalized* if whenever an instance variable is functionally dependent on a set S of instance variables, S contains a key.¹

We recommend that classes be normalized.

It is often the case that there are no functional dependencies among the instance variables of a class. For example, the class `Assignment` which is defined by

```
Assignment = <variable> Ident <assignedValue> Expression.
```

does not have a functional dependency among its two instance variables. In such classes all instance variables are a part of the key and the concept of normalization is trivial.

5.6 Tokens

The class dictionary information may be used for reading in objects from a text file. This is the case in many applications and therefore the Demeter class dictionary designer has to be concerned with how a parser is generated from the class dictionary definition. We have adopted the most simple and widely used parsing scheme available: the parsing is done by a recursive descent look-ahead-one parser. This parsing technique is powerful enough to handle such languages

¹This definition is a derivative of the Boyce-Codd normal form from relational database theory.

as Pascal or Modula-2. The simplicity of the parsing technique makes it easier to recognize whether a class dictionary is LL(1). This property is important to the programmer who develops many class dictionaries in a short period of time. We use a tool, called an LL(1) enforcer, which automatically translates class dictionaries into LL(1) form by adding concrete syntax. The wording of the concrete syntax is left to the programmer.

5.7 Class dictionary types

A class dictionary can be used to define an input language and/or an output language and/or data structures. Any of the seven possibilities, which chooses at least one of the three features, makes sense. We give some examples:

- input language. An example is a prefix expression interpreter. Since such an interpreter is very simple it does not need additional data structures or an output language grammar.

- input language and exclusive data structures. A Scheme interpreter requires an input language definition as well as internal data structures used during the interpretation process. These internal data structures are used for storing objects such as environments and closures.

- output language. Applications of class dictionaries in this class include definitions of parameterized objects. For example, a Batcher sorting network structure can be specified with a class dictionary, but the details of the network will be generated under program control. The class dictionary is mainly used for defining the structure of the network and simultaneously an output language which allows printing a sorting network in readable form.

5.8 Simple improvements

We list some class dictionaries written by beginners and how those dictionaries can be improved:

Th class dictionary

```
A = B.  
B = Ident.
```

should be written as

```
A = <B> Ident.
```

The class dictionary

```
A = B C.  
B = ";"
```

should be written as

A = ";" C.

or

A = B C.

B : Semicolon.

Semicolon = ";".

The class dictionary

A = <x> List(B) <y> List(C).

List(S) ~ {S}.

is often better written as

A = <z> List(BC).

BC : B | C.

List(S) ~ {S}.

In the second version the ordering in A is considered unimportant.

6 Related Work

The Demeter system has benefitted from work done in three areas: Programming languages, Artificial Intelligence and Data Bases. A detailed comparison exceeds the space requirements of this paper and can be found in the forthcoming book [Lie96]. We give here a few pointers:

- Object-oriented programming: Object-oriented programming is promoted by Simula-67 [DMN70], Smalltalk-80 [GR83], Flavors [Moo86], Objective-C [Cox86], C++ [Str86], Eiffel [Mey86], CLOS [BMG⁺88]. Our prototype implementation uses Flavors and Franz Lisp. Demeter extends the object-oriented languages mentioned above in a useful way.

The override mechanism of Demeter exists in similar form in Galileo [ACO85] and Taxis [MBW80] and in [Tho86].

- Language design and parsing: We use the ideas promoted by N. Wirth on language design [Wir74], [Wir84], data structure and parser design [Wir76], grammar notation [Wir77] and type checking [Wir71].
- Grammar-based programming: A grammar-based approach to meta programming in Pascal has been introduced in [CI84]. [Fra81] uses grammars for defining data structures. [KMMPN85] introduce an algebra of program fragments. The POPART system treats grammars as objects [Wil83]. The synthesizer generator project also uses a grammar based approach [RT84]. GEM described in [GL85] is the predecessor of Demeter.

Our work is novel in that it combines language definition and data structure definition for object-oriented programming into one coherent framework. Such an integration has not been attempted before. It supports a succinct, clear, modular programming style.

- Artificial intelligence: Many papers in knowledge engineering propose a similar approach as Demeter. One which comes close is [FAS⁺86].
- Parameterized data types: An excellent survey is in [CW85]. Solomon introduces the concept of bounded parameterized classes [Sol78, page 25]. Parameterized data types are provided in several programming languages, including CLU [LSAS77], Trellis/Owl [SCB⁺86] and Ada. Parameterization of data type specifications is used in the algebraic approach to data types [Kre87] and [TWW82]. [Mac85], [Mac86] discusses many of the important issues related to generic software design.
- Program transformation systems: Our development of Demeter has been motivated by work on program transformation systems [PS83],[CI84], [BM84], [KW87]. The MENTOR system [DGHK⁺75], [DGHKL80] is designed for manipulating structured data represented by abstract syntax trees. The Mjølner project uses a Demeter-like approach, but without parameterization and multiple inheritance [MN88].
- Data base design: Our class dictionaries share some properties with data dictionaries. The data dictionary approach in the data base field is described in the survey [ALM82]. A paper by John and Diane Smith [SS77] outlines some of the features of the Demeter system. Their aggregation/generalization concepts correspond to our construction/alternation concepts.
- Program enhancement: [Bal86] proposes a frame-based object model to simplify program enhancement which has some similarities to the Demeter system.
- Exchange of structured data: The programming language independent data structure language of Demeter supports the exchange of structured data between different programming languages. The work on IDL [Lam87] is related.
- Demeter project: [LR88b] introduces the gardening analogy with the “meal” example and defines the following concepts: legality of objects, subtype and boundedness for parameterized classes (to exclude context-sensitive languages). [LR88a] is an expanded version of [LR88b] which contains proofs for the subtype transitivity theorem and the name compatibility theorem for subtypes.

7 Conclusions

We are successfully using a class dictionary based approach to object-oriented programming in our course and research work. We strongly recommend our approach to anyone who likes to write reusable, modular software efficiently. Our approach strongly supports the old but very useful idea that it is worth developing an elegant notation for your application before you start programming.

The crucial ingredient to our approach are efficient algorithms which transform class dictionaries into programming environments that are tailored for a specific application. We have implemented such algorithms inside the Demeter system, using its own technology. Our future work will concentrate on adding graphical programming to Demeter and providing tools which assist the user with class dictionary design and programming, e.g. a parameterization by example tool and a software retrieval tool.

Acknowledgements

This work started initially at GTE Laboratories as part of a silicon compilation project. Andrew Goldberg and I designed and implemented a dictionary based generator called GEM [GL85]. Gerry Jones was internal consultant for the GEM project.

GTE Laboratories gave me permission to continue this work at Northeastern, where the project is supported by numerous dedicated graduate and undergraduate students, including (in alphabetical order) Bill Brown, Ian Holland, Gar-Lin Lee, Kathy Lee, David Lincoln, Robert MacDonald, Benoît Menendez, Jing Na, Robert Paul, Arthur Riel and Johannes Sujendro.

This paper benefitted from detailed feedback and many stimulating discussions with Mitchell Wand. Betty Salzberg has contributed to the data base aspects of class dictionary design. Abbas Birjandi, Richard Rasala and Bill Brown have given me detailed feedback on an earlier version. Their feedback has considerably improved the paper.

I would like to thank Richard Gabriel for his feedback and for personally editing my paper to improve readability.

References

- [ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly-typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230 – 260, June, 1985.
- [ALM82] F. Allen, M. Loomis, and M. Mannino. The Integrated Dictionary/ Directory System. *ACM Computing Surveys*, 14(2), 1982.
- [Bal86] R.N. Balzer. Program enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4):66, 1986.

- [BM84] J.M. Boyle and M.N. Muralidharan. Program reusability through program transformation. *IEEE Transactions on Software Engineering*, SE-10(5), September 1984.
- [BMG⁺88] Daniel Bobrow, Linda G. De Michiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification. Draft submitted to X3J13, March 1988.
- [CI84] Robert D. Cameron and M. Robert Ito. Grammar-based definition of metaprogramming systems. *ACM Transactions on Programming Languages and Systems*, 6(1):20–54, January 1984.
- [Cox86] Brad J. Cox. *Object-Oriented Programming, An evolutionary approach*. Addison-Wesley, 1986.
- [CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471, December 1985.
- [DGHK⁺75] Veronique Donzeau-Gouge, Gérard Huet, Gilles Kahn, Bernard Lang, and J.J. Lévy. A structure oriented program editor: A first step towards computer assisted programming. In *Proceedings of International Computing Symposium 1975*, 1975.
- [DGHKL80] Veronique Donzeau-Gouge, Gérard Huet, Gilles Kahn, and Bernard Lang. Programming environments based on structured editors: The MENTOR experience. Technical report, Res. Rep. 26 INRIA, 1980.
- [DMN70] O.J. Dahl, B. Myhrhaug, and K. Nygaard. SIMULA 67 Common Base Language. *Publication Number S-22, Norwegian Computing Center*, October 1970.
- [FAS⁺86] M.J. Freiling, J.H. Alexander, S.J. Shulman, J.L. Staley, S. Rehfuss, and S.L. Messick. Knowledge level engineering: Ontological analysis. In *Proceedings AAAI*, volume 2, pages 963–968, 1986.
- [Fra81] C.W. Fraser. Syntax-directed editing of general data structures. In *Proceedings ACM SIGPLAN/SIGOA Conference on Text Manipulation*, pages 17–21, Portland, OR, 1981.
- [GL85] A.V. Goldberg and K.J. Lieberherr. GEM: A generator of environments for metaprogramming. In *SOFTFAIR II, ACM/IEEE Conference on Software Tools*, pages 86–95, San Francisco, 1985.
- [GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

- [KMMPN85] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pederson, and Kristen Nygaard. An algebra for program fragments. In *ACM SIGPLAN 85 Symposium on Programming Languages and Programming Environments*, volume 20, Seattle, WA, 1985. SIGPLAN.
- [Kre87] H.J. Kreowski. Some initial sections of the algebraic specification tale. *Bulletin of the European Association for Theoretical Computer Science*, (31):55–78, February 1987.
- [KW87] E. Kohlbecker and Mitchell Wand. Macro by example. In *ACM Symposium on Principles of Programming Languages*. ACM, 1987.
- [Lam87] D.A. Lamb. IDL: Sharing Intermediate Representations. *ACM Transactions on Programming Languages and Systems*, 9(3):297–318, July 1987.
- [Lie96] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. 616 pages, web book at www.ccs.neu.edu/research/demeter.
- [LR88a] Karl J. Lieberherr and Arthur J. Riel. Demeter: A CASE study of software growth through parameterized classes. *Journal of Object-Oriented Programming*, 1(3):8–22, August, September 1988. A shorter version of this paper was presented at the *10th International Conference on Software Engineering, Singapore, April 1988*, IEEE Press, pages 254–264.
- [LR88b] Karl J. Lieberherr and Arthur J. Riel. Demeter: A CASE study of software growth through parameterized classes. In *International Conference on Software Engineering*, pages 254–264, Raffles City, Singapore, 1988.
- [LSAS77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.
- [Mac85] D. MacQueen. Modules for standard ML. *Polymorphism Newsletter*, 2(2), October 1985.
- [Mac86] D. MacQueen. Using dependent types to express modular structure. In *ACM Symposium on Principles of Programming Languages*, January 1986.
- [MBW80] J. Mylopoulos, P. A. Bernstein, and H.K.T. Wong. A language facility for designing interactive database intensive systems. *ACM Transactions on Database Systems*, 5(2):185 – 207, June, 1980.

- [Mey86] B. Meyer. Genericity versus inheritance. In *Object-Oriented Programming Systems, Languages and Applications Conference*, in *Special Issue of SIGPLAN Notices*, pages 391–405, 1986.
- [MN88] Ole Lehrmann Madsen and Claus Nørgaard. An object-oriented metaprogramming system. In *Proceedings of the Annual Hawaii International Conference on System Sciences*, pages 406–415, 1988.
- [Moo86] David A. Moon. Object-Oriented Programming with Flavors. In *Object-Oriented Programming Systems, Languages and Applications Conference*, in *Special Issue of SIGPLAN Notices*, pages 1–8, Portland, OR, 1986.
- [PS83] Helmut A. Partsch and R. Steinbrueggen. Program transformation systems. *ACM Computing Surveys*, 15(3):199–236, September 1983.
- [RT84] T. Reps and T. Teitelbaum. The synthesizer generator. *SIGPLAN*, 19(5), 1984.
- [SB86] Mark Stefik and Daniel G. Bobrow. Object-oriented programming: Themes and variations. *The AI Magazine*, pages 40–62, January 1986.
- [SCB+86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *Object-Oriented Programming Systems, Languages and Applications Conference*, in *Special Issue of SIGPLAN Notices*, pages 9–16, 1986.
- [Sol78] M. Solomon. Type definitions with parameters. In *Principles of Programming Languages*, pages 31–38, Tucson, Arizona, 1978.
- [SS77] J.M. Smith and D.C.P. Smith. Database abstractions: Aggregation and generalization. *ACM Transactions on Database Systems*, 2(2), June 1977.
- [Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.
- [Tho86] K. S. Thomsen. Multiple inheritance, a structuring mechanism for data, processes and procedures. Technical Report DAIMI PB - 209, University of Aarhus, Denmark, April 1986.
- [TWW82] J.W. Thatcher, E.G. Wagner, and J.B. Wright. Data type specification: Parameterization and the power of specification techniques. *ACM Transactions on Programming Languages and Systems*, pages 711–732, 1982.

- [Wil83] David S. Wile. Program developments: Formal explanations of implementations. *Communications of the ACM*, 26(11):902–911, 1983.
- [Wir71] Niklaus Wirth. The Programming Language Pascal. *Acta Informatica*, 1:35–63, 1971.
- [Wir74] Niklaus Wirth. On the design of programming languages. In *IFIP, Amsterdam*, pages 386–393. North-Holland, 1974.
- [Wir76] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.
- [Wir77] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823, 1977.
- [Wir84] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, 1984.

Contents

1	Introduction	2
2	Class dictionaries	3
3	Inheritance and Parameterization	8
3.1	Abstraction	8
3.2	Parameterization	8
3.3	Preprocessing of parameterization	9
4	Environment definition	14
5	Class dictionary design techniques	16
5.1	Abstraction	16
5.1.1	Mixins and promotion	17
5.1.2	Parameterization	19
5.2	Naming	21
5.3	Regularity	21
5.4	Prefer alternation	22
5.5	Normalization	23
5.6	Tokens	24
5.7	Class dictionary types	25
5.8	Simple improvements	25
6	Related Work	26

