Demeter: A CASE Study of Software Growth Through Parameterized Classes

Karl J. Lieberherr, Arthur J. Riel
Northeastern University, College of Computer Science
360 Huntington Avenue, Boston MA 02115

December 14, 2004

Abstract

The *Demeter*^{TM} system is a CASE (Computer-Aided Software Engineering) tool designed for the development of large software projects using a new software design methodology which focuses on growing rather than building software. We describe the software development process as one of growth and evolution as opposed to building and rebuilding because most complex objects in the real world are grown and not built. Since software design is obviously a complex process this new paradigm may be helpful in unraveling some of the problems associated with current software design practices. Demeter begins by providing an ideal environment for the sprouting and nuturing of a seed (class dictionary) into a plant (large scale software project). In addition, through the combined use of object-oriented programming technology, and parameterized classes, Demeter provides a facility for the reuse of software which was developed in previous software projects.

A short form of this paper was published in [LR88].

- To appear in *Journal of OBJECT-ORIENTED PROGRAMMING*, Vol.1, No.3, 1988.

1 Introduction

The primary purpose of the *Demeter*^{TM} system is to provide a medium for efficient growth and evolution of large scale software projects. We describe the software development process as one of growth and evolution as opposed to building and rebuilding because most complex objects with few repetitive building blocks are grown and not built [Mil71] [Bro87]. Examples of these are plant and animal life. Software, being a very complex creature, should not be built from bits and pieces of inanimate code which is sewn together and shocked into life with a marathon session of debugging. A creature created in such a fashion is assured of sharing the same fatal flaws as other creatures created in this fashion (e.g. the Frankenstein[1] monster). Instead, the software plant should be grown from a small but animate sprout. At each stage of growth the creature will become more complex but will always exhibit characteristics appropriate for that stage of development.

Demeter, appropriately named for the Greek goddess of the earth, provides an ideal environment for the planting of a small prototype system (sprout) and its nurturing into a fully developed software system. In addition, Demeter provides the necessary facilities for grafting whole or part of previously developed software systems to the new system or even grafting the new system onto a suitably developed root stock (a parameterized class). The Demeter system creates such an environment by marrying object-oriented programming techniques with a common data structure description language and the concept of parameterized class. Through the use of this environment, large software environments can be grown from humble beginnings, borrowing at each stage of development the experience (code) gained in any previously grown projects.

The Demeter system makes extensive use of parameterized classes. Parameterized classes are best thought of as generic software modules which can be used and reused by a large assortment of projects so long as they are of the same species. We consider the relationship between parameterized classes and software projects to be similar to the relationship between root stocks and fruit trees. A root stock is a tree grown solely for its roots. The variety of the tree is typically winter hardy and disease resistant but otherwise uninteresting in fruiting habit. This root stock is used as a strong generic rooting system for trees of the same species which have spectacular fruiting habits but poor immunity to cold and disease. We graft the desired variety onto the root stock and get a superior tree. We could have produced such a tree by years of selective breeding within the desired variety, but by using the root stock we save a significant amount of time. The parameterized classes of Demeter have much in common with root stocks. They provide a base of software with some very desirable properties. We present several clear examples of these constructs in the appropriate section below.

[1] The American Heritage Dictionary defines a Frankenstein as "An agency or creation that slips from the control of and ultimately destroys its creator".

2 Demeter Class Dictionaries (The Seeds)

All large software projects should begin with a description of the underlying data structure. It is the underlying data structure which is the backbone of any software system. A poor design at this level is the shortest road to disaster. In Demeter, the first step in designing a software environment is the construction of a class dictionary. The class dictionary is a natural description of the object hierarchies which make up the underlying data structure. Demeter class dictionaries are defined by an LL(1) language whose description appears in a form similar to EBNF. Any actions which are to be carried out in the "program" are reduced to manipulations of the object hierarchy. The machine readable data structure is automatically generated, along with an abundance of support software, by the Demeter system. When a user describes his or her data structure abstractly, he or she gains the advantage of having a common data structure for all software projects. This allows for the building of a facility for reusing any software previously written in the Demeter system regardless of the nature of the old software project.

The Demeter system is currently implemented in Franz Lisp/Flavors. In this implementation an object hierarchy is built out of Flavor instances and object manipulations are written as method definitions. The Demeter system could be ported to any language which has facilities for defining objects and classes of objects, instance variables attachable to an object or class, and a multiple inheritance mechanism. The porting of the Demeter system to the following languages is now under review: Objective-C, C++, Common Lisp (CLOS), PC Scheme with Scoops and Prolog with an object-oriented extension.

Many complex class dictionaries have been written, tested, and developed into full fledged software systems. We begin by constructing a class dictionary which represents a real world system, namely, a meal. We first construct this class dictionary in English and then transcribe it into the language of Demeter in order to illustrate the naturalness of the system.

At some point in defining a class dictionary we must decide what we consider to be atomic objects. In a meal a steak might be considered atomic since we are not interested in breaking the meat down into its molecular structure. A shrimp cocktail might be considered by some to be atomic, but for the purpose of this example we assume that we are ignorant of the notion of a shrimp cocktail and need to break it down further. Atomic objects are presented as terminal symbols in our grammar (typically strings). We define our meal as follows:

1. A meal is an appetizer, an entree, and a dessert. While a dessert may be considered optional in many households, we consider it a required part of a meal.

2. An appetizer is a melon or a shrimp cocktail.

3. A shrimp cocktail is lettuce, one or more shrimp, and possibly cocktail

sauce.

4. Cocktail sauce is ketchup and horseradish.
5. An entree is a steak platter or a baked stuffed shrimp platter.
6. A steak platter is a steak and the trimmings.
7. A baked stuffed shrimp platter is a stuffed shrimp and the trimmings.
8. The trimmings are a potato and two veggies.
9. A veggie is carrots, peas, or corn.
10. A dessert is pie, cake, or jello.
11. We assume that the following items are atomic and need no further description: melon, shrimp, lettuce, ketchup, horseradish, steak, stuffed-shrimp, potato, carrots, peas, corn, pie, cake, and jello.

The decision as to whether an object is atomic or built up from other objects is not only domain specific but also specific to the person performing the data abstraction.

Each of the eleven descriptive phrases falls into one of three categories. Either we are constructing an object in terms of other objects (a construction production), or we are defining an object as being an ordered collection of zero (one) or more of another object (a repetition production), or we state that an object is one of several possible objects (an alternation production). Examples of these three object descriptions are shown in Demeter notation below.

Construction An element of class A has a first part which is an element of class B followed by an element of class C, possibly an element of class D, and an element of class E.

```
A = B C [D] E.
```

A construction class may inherit from another construction class. An element of class X has a first part which is an element of class X followed by a second part which is an element of class Z followed by all parts of class A.

```
X = Y Z *inherit* A.
```

Repetition An element of class A is a collection of zero (one) or more elements of B.

```
A ~ { B }.
A ~ B { B }.
```

Alternation An element of class A is either an element of classes B or C or D.

```
A : B | C | D.
```

If the alternatives have common properties, we use alternation with implied inheritance. An element of class A is either an element of class B or C and both B and C have two parts, the first being an object of class X and the second an object of class Y. (X and Y are the last two parts of B and C.)

```
A : B | C *common* X Y.
```

Many times we want to add keywords to our object descriptions. These may be necessary to satisfy the LL(1) restriction on the grammar or may be used to "sweeten" the syntax of a class dictionary. In any event they have no effect on the object hierarchy descriptions. The Demeter system guarantees that all user produced code is completely shielded from the concrete syntax. This implies that the input language for an object hierarchy can be modified to the whims of the user with no risk to the reliability of the system. This modification can be done at any point in the system development. The only restriction is that the structure of the underlying object hierarchy must remain static. Below are examples of construction and repetition productions which demonstrate where concrete syntax can be inserted into the object descriptions.

Construction Production .

```
A = "a" B "followed" "by" "a" C
    "inherits-from-Q"
      *inherit* Q
    "end-inherit"
    ["maybe" D "!!"]
    "followed by" E.
```

Repetition Production .

```
A ~ "first" B
    {"prefix" "String" B "suffix"}
    "terminating".
```

Alternation Production .

```
Fruit : Orange | Apple
  *common* "weight" <weight> Number "terminating".
```

Many class dictionaries make use of descriptive labels in their construction productions. A descriptive label is an identifier which is enclosed in angle brackets and preceeds a given nonterminal. The sole purpose of descriptive labels is to rename a sub-object in the class dictionary. This renaming is required if two (or more) of the same nonterminal are identified on the right hand side of a construction production. The following is an example of such a production, it defines an object A which is comprised of two B's.

```
A = <first> B <second> B.
```

The other case in which descriptive labels are often used is with class terminals. When a class terminal is used without a label it is not descriptive. We use labels to make the objects more self describing. The following example illustrates this point.

```
GradeReport = Ident Number.
```

is not self describing. The following is semantically equivalent but makes the class dictionary, and any methods written for it, easier to read.

```
GradeReport =
  <studentName> Ident
  <testScore> Number.
```

It is important to note that the object stored at a symbol position remains the same whether or not it is labeled. The labels are used only for descriptive purposes and are analogous to the renaming of variables. Labels are also used to override instance variable types in connection with inheritance in construction or alternation productions. (We use class and type as synonyms.)

```
A = B <label> K.
X = D *inherit* A *override* <label> L.
M : P | Q *common* *inherit* A *override* <label> L.
```

An element of class X is an element of class D followed by all elements of class A. The type of the instance variable called label is L and not K. An element of class M is either an element of class P or Q. Both P and Q have in common all the instance variables of A, with the type of instance variable label changed to L.

Certain instance variables are inherited by all objects. The instance variables *attribute* and *father* are inherited by all Demeter defined objects. The instance variable *attribute* is an application specific variable which can be used to "decorate" the objects. The latter instance variable is used in constructing backward pointers in an object hierarchy. There are also some instance variables which are inherited by certain classes of objects. All repetition objects inherit the instance variable *child*. This instance variable contains a list of the repeated

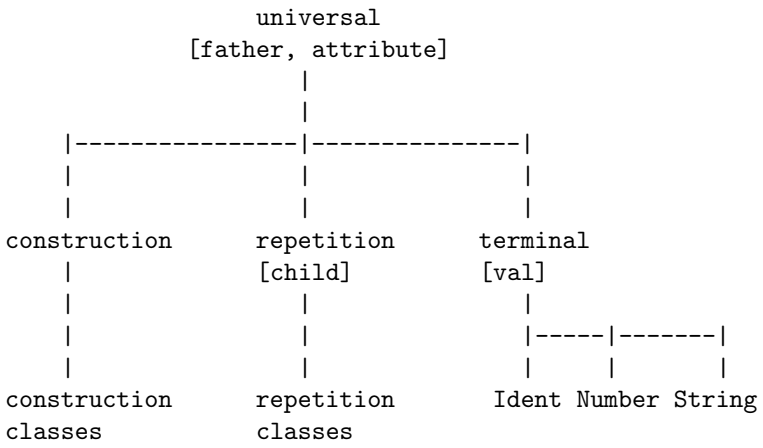


Figure 1: Class hierarchy organization

objects used to define the repetition object. In addition, each class terminal (i.e., Ident, Number, String) inherits the instance variable *val* which contains the value of the identifier, number, or string (respectively).

The instance variables father, attribute, child and val are attached to classes as shown in Fig. "Class hierarchy organization". Each repetition class inherits from class repetition and each construction class inherits from construction. Class terminals (e.g. Ident) inherit from terminal and all three classes: construction, repetition and terminal inherit from universal.
The description of our meal in Demeter notation is as follows:

```
Meal =
  "Appetizer:" Appetizer
  "Entree:" Entree
  "Dessert:" Dessert.
Appetizer : Melon | ShrimpCocktail.
ShrimpCocktail = Shrimps Lettuce
  [CocktailSauce].
CocktailSauce = Ketchup HorseRadish.
Entree : SteakPlatter | BakedStuffedShrimp.
SteakPlatter = Steak Trimmings.
BakedStuffedShrimp = StuffedShrimp
  Trimmings.
Trimmings = Potato
  <veggie1> Veggie <veggie2> Veggie.
Veggie : Carrots | Peas | Corn.
```

```
Dessert : Pie | Cake | Jello.
Shrimps ~ Shrimp {Shrimp}.
Shrimp = "shrimp".  Melon = "melon".  etc.
```

We can now input this class dictionary to Demeter and generate an environment skeleton. We can then create object hierarchies representing meals, e.g. an input of the form

```
Appetizer: melon
Entree: steak potato carrots peas
Dessert: cake
```

defines a meal. This example demonstrates the straight forward translation of our conceptual understanding of a meal to the Demeter notation. The representation of a system by a hierarchy of object descriptions is very natural to people.

3 Sprouting and Fertilizer

Once the seed of a project has been established we need to produce a working system from it. In addition to producing this small working system we would like to provide a large quantity of support code to aid the early system in its first stages of growth. It is known that each plant in the agricultural world grows best when an applied fertilizer was developed specifically for the given plant. Similarly, each system responds best to a set of utilities built specifically for the class dictionary. This set of utilities (fertilizer) is produced by the Demeter system through a series of software generation steps. In these steps the following utilities are generated or provided generically from the information stored in the class dictionary.

- The class definitions for each object defined in the class dictionary. This frees the user from learning the syntax of defining classes and frees Demeter from being totally dependent on its underlying implementation.
- A set of methods for pretty-printing a hierarchy of objects. It is considered advantageous to have data which is self-describing. In Demeter, the type definitions specify the external representation of the objects.
- A set of methods for "drawing" a hierarchy of objects. The current implementation is a simple hierarchical sketch of the object hierarchies. Later implementations will include graphics to give a more aesthetic image.
- A scanner/parser for the LL(1) description of the class dictionary. This software allows the user to have a hierarchy of objects built from an ASCII input file which contains the description of the object hierarchy written in the LL(1) language.

- A structure-directed editor which allows the user to build a hierarchy of objects by answering questions about the hierarchy. This object hierarchy could then be printed to an ASCII file, via the pretty-printing methods, for later reloading by the scanner/parser. This editor does not require the user to know about the concrete syntax of the input language so one could refer to it as a syntax-directed editor as well.
- A skeleton program (sprout) which defines a sample method for each object in the hierarchy. Each method includes a reference to all of the instance variables of the object as well as a comment header describing the object for which the method is defined. This method is the initial working system. Its sole function is to walk through the object hierarchy printing the value of any class terminals in the hierarchy. The Demeter system supports the predefined class terminals Ident, Number, and String as well as a facility for the definition of additional class terminals. The entire system is grown from this simple skeleton program. This simple program is the sprout from which the software plant will be grown.
- A growth plan generator which suggests in which order to grow the plant. The growth plan is determined by the chromosomes (productions) in the class dictionary (seed).

4 The Growth and Evolution of a System (Nurturing)

Once a Demeter environment has been generated from the class dictionary description we want our system to grow and evolve. We will present two methods associated with our Meal environment. The first is to calculate the cost of a meal. This method will walk the object hierarchy and compute the cost of each object in the hierarchy. The method is very modular, thanks to the object-oriented nature of Demeter, so we can use it to compute the cost of any component of a meal as well. We define food costs in the following menu: Melon 0.75, Shrimp Cocktail 0.65 per shrimp, cocktail sauce 0.15 extra, Steak Platter 5.00, Baked Stuffed Shrimp 6.00, Pie 0.60, Cake .65, Jello 0.35.

```
  ; Banquet ~ Meal { Meal }.
(defmethod (Banquet :cost) ()
  (let ((total-cost
    (loop for each-meal in child sum
      (send each-meal ':cost))))
      (format t "cost of ~A meals: ~A.~%"
        (length child) total-cost)))
  ; Meal =
  ;   "Appetizer:" Appetizer
```

```
  ;   "Entree:" Entree
  ;   "Dessert" Dessert.
(defmethod (Meal :cost) ()
  (+ (send Appetizer ':cost)
     (send Entree ':cost)
     (send Dessert ':cost)))
  ; Melon = "melon".
(defmethod (Melon :cost) () 0.75)
  ; ShrimpCocktail = Shrimps Lettuce
  ;   [CocktailSauce].
(defmethod (ShrimpCocktail :cost) ()
  (+ (send Shrimps ':cost)
     (if CocktailSauce
       then (send CocktailSauce ':cost)
       else 0.0)))
  ; Shrimps ~ {Shrimp}.
(defmethod (Shrimps :cost) ()
  (loop for each-shrimp in child sum
    (send each-shrimp ':cost))))
  ; Shrimp = "shrimp".
(defmethod (Shrimp :cost) () 0.65)
  ; CocktailSauce = Ketchup HorseRadish.
(defmethod (CocktailSauce :cost) () 0.15)
  ; SteakPlatter = Steak Trimmings.
(defmethod (SteakPlatter :cost) () 5.00)
  ; BakedStuffedShrimp =
  ;   StuffedShrimp Trimmings.
(defmethod (BakedStuffedShrimp :cost) ()
  6.00)
  ; Pie = "pie".
(defmethod (Pie :cost) () 0.60)
  ; Cake = "cake".
(defmethod (Cake :cost) () 0.70)
  ; Jello = "jello".
(defmethod (Jello :cost) () 0.35)
```

We now examine a situation which is common in software development. Our restaurant owner takes one look at our finished software and informs us that our meal definition is all wrong. The cocktail shrimp should come in three sizes; small, medium, and large costing 0.20, 0.50, and 1.00 (respectively). We kindly ask the irate restaurant owner to have a seat and relax. We then make the following changes to our class dictionary without the least concern of ruining our existing system. We modify the atomic object *Shrimp* to be an alternation production of the following form.

```
Shrimp : SmallShrimp | MediumShrimp | LargeShrimp.
```

We remove the cost method for shrimp and replace it by the following three method definitions (one for each of the new atomic objects).

```
(defmethod (SmallShrimp :cost) () 0.20)
(defmethod (MediumShrimp :cost) () 0.50)
(defmethod (LargeShrimp :cost) () 1.00)
```

Modifications are made at or below the point at which the object hierarchy definitions are changed. This allows the user the freedom of updating his underlying data structure without having to worry about the loss of system integrity.

The next set of methods we will define are more complex. We would like to simulate an action such as eating. When we eat a meal we are left with objects which we refer to as garbage. Each object in the hierarchy either completely disappears or leaves some object behind. For example, a melon leaves a rind, a potato leaves a skin, a steak leaves a bone and possibly some fat, etc. The first step in such a task is to build a class dictionary which will allow us to represent and build a hierarchy of garbage. This class dictionary can be constructed as follows:

```
Gbanquet ~ Gmeal {Gmeal}.
Gmeal =
  "Appetizer Garbage:" Gappetizer
  "Entree Garbage:" Gentree
  "Dessert Garbage:" Gdessert.
Gappetizer : Rind | GShrimpCocktail.
GshrimpCocktail = Shrimptails Lettuce
  [CocktailSauce].
Shrimptails ~ Shrimptail {Shrimptail}.
Gentree : Gsteak | Gshrimp.
Gsteak = [ Bones ] [ Fat ] Gtrimmings.
Gshrimp = Shrimptail Gtrimmings.
Gtrimmings = PotatoSkin.
Gdessert : Crust | Frosting | GJello.
GJello = [ Jello ] [ WhippedCream ].
```

The following symbols in this class dictionary

```
Rind, Shrimptail, Bones, Fat, PotatoSkin,
Crust, Frosting, Jello, WhippedCream.
```

are defined by construction productions each consisting of one or more strings, e.g.,

```
Fat = "fat".
WhippedCream = "whipped" "cream".
etc.
```

In this example steak and jello can return some optional garbage. For these optional choices we flip a coin. For example, some steaks will leave behind bones and fat when eaten but others will not.

```
  ; Banquet ~ Meal { Meal }.
(defmethod (Banquet :eat) ()
  (make-instance 'Gbanquet
    ':child
; initialize Gbanquet slot called child
      (loop for each-meal in child collect
        (send each-meal ':eat))))
  ; Meal = Appetizer Entree Dessert.
(defmethod (Meal :eat) ()
  (make-instance 'Gmeal
    ':Gappetizer (send Appetizer ':eat)
    ':Gentree (send Entree ':eat)
    ':Gdessert (send Dessert ':eat)))
  ; ShrimpCocktail = Shrimps Lettuce
  ;   [CocktailSauce].
(defmethod (ShrimpCocktail :eat) ()
  (make-instance 'GshrimpCocktail
    ':Shrimptails (send Shrimps ':eat)
    ':Lettuce (send Lettuce ':copy)
    ':CocktailSauce
      (send CocktailSauce ':copy)))
    ; copy returns a new object
    ; which is a copy of the argument
  ; Shrimps ~ {Shrimp}.
(defmethod (Shrimps :eat) ()
  (make-instance 'Shrimptails
    ':child (loop for each-shrimp
      in child collect
        (send each-shrimp ':eat))))
  ; Shrimp = "shrimp".
(defmethod (Shrimp :eat) ()
  (make-instance 'ShrimpTail))
  ; SteakPlatter = Steak Trimmings.
(defmethod (SteakPlatter :eat) ()
  (make-instance 'Gsteak
    ':Bones
      (if (eql (flip) 'head)
```

```
       then (make-instance 'Bones)
       else nil)
    ':Fat
      (if (eql (flip) 'head)
        then (make-instance 'Fat)
        else nil)
    ':Gtrimmings (send Trimmings ':eat)))
  ; BakedStuffedShrimp =
  ;   StuffedShrimp Trimmings.
(defmethod (BakedStuffedShrimp :eat) ()
  (make-instance 'Gshrimp
    ':Shrimptail (send StuffedShrimp ':eat)
    ':Gtrimmings (send Trimmings ':eat)))
  ; Trimmings = Potato
  ;   <veggie1> Veggie <veggie2> Veggie.
(defmethod (Trimmings :eat) ()
  (make-instance 'PotatoSkin))
  ; Pie = "pie".
(defmethod (Pie :eat) ()
  (make-instance 'Crust))
  ; Cake = "cake".
(defmethod (Cake :eat) ()
  (make-instance 'Frosting))
  ; Jello = "jello".
(defmethod (Jello :eat) ()
  (make-instance 'GJello
    ':Jello
      (if (eql (flip) 'head)
        then (make-instance 'Jello)
        else nil)
    ':WhippedCream
      (if (eql (flip) 'head)
        then (make-instance 'WhippedCream)
        else nil)))
```

In such a system a meal input file might look like:

```
Appetizer:
  smallshrimp smallshrimp smallshrimp
  lettuce
Entree: steak potato corn peas
Dessert: pie
```

After sending this object hierarchy the :eat message we get a garbage hierarchy returned to us. We can send this returned hierarchy the pretty print

message (a Demeter generated method) and print the following ASCII text.

```
Appetizer Garbage:
  shrimptail shrimptail shrimptail
  lettuce
Entree Garbage: bones fat potato skin
Dessert Garbage: crust
```

This technique is powerful in that you can evolve one object hierarchy into another fairly easily. This easy evolution frees us from the bounds of ineffective programming languages and allows us the liberty of declaring the following rule: "If the language you are working with does not allow you to write a necessary task comfortably, then create your own language and translate the current representation into the new representation." This rule is meaningful only if the task of creating a new language and building a translator is significantly easier than the original task. This is usually the case within the realm of Demeter.

5 Inheritance (Grafting)

There are three ways of acquiring plants. The easiest and most expensive way is to buy mature plants. The major drawback (besides cost) of this method is the unavailability of many varieties. Many plant fanciers decide to grow their favorite varieties from seed. While the cost may be lower, the time involved can be quite substantial. In addition, there are many things which can go wrong as our seed takes the long trip from sprout to mature plant. In the world of software development we find we have the same two choices. Either we buy the necessary software from someone who has already developed it or we develop it ourselves. Again we have the same advantages and drawbacks. There is a fair chance that the available software will not be the specific variety we want or it may be of dubious quality. If we decide to develop it ourselves then we must be willing to wait a long time and there is always the chance of something going wrong during that long development process.

Many horticulturists look to a third alternative called grafting. The process of grafting involves splicing together part of a desired plant with a host plant of a related type. This method has the advantage in that the cost is low and much of the risk of nurturing a seed to maturity is removed. Software developers are usually at a loss when they attempt this third alternative. While plants have a common interface (typically a woody tissue) software systems do not. The chances of two systems using the same underlying data structure, the same language, and being modular enough to take functions for reuse in a new system are very low. The Demeter system avoids these problems by providing a common language for the abstract data structure of each system, a means of easy language translation, and very modular coding. This latter quality is enforced

by the underlying object-oriented technology and the Law of Demeter[2].

The Demeter system grafts one software system onto another through the use of the keywords ∗inherit∗ and ∗override∗ in the class descriptions. If an ∗inherit∗ keyword prefixes a symbol it specifies that the class is to be "copied" into the host system. For the purpose of language definition the copying is explicit. For example,

```
A = Ident Number.
B = String *inherit* A "end".
```

implies that the language B is defined by

```
B = String Ident Number "end".
```

For the purpose of class definition we use the inheritance semantics of the underlying object-oriented system. In the above example we simply specify that class B inherits from class A. We make the assumption that the underlying object-oriented system uses dynamic binding of the first argument, often called self. This implies that the methods attached to superclasses of a class are available at that class. A more detailed discussion of inheritance is given later.

The ∗override∗ keyword is only used in conjunction with the ∗inherit∗ keyword. It specifies that the class is to be inherited with some changes which will always be declared after the ∗override∗ keyword. A similar approach is taken in Trellis/Owl [SCB*86].

Our motivation for using override is as follows: A subclass is made from a number of superclasses by specifying additional properties (instance variables and methods) that elements of the subclass must have compared to elements of the superclasses. It is also necessary to make a subclass in which some of the properties inherited from superclasses are modified. Inherited methods and instance variable types can be modified in a subclass, subject to some restrictions that insure that the inheritance is conceptually a specialization hierarchy.

The ability to modify the type of instance variables in subclasses is restricted in such a way that static type checking of the class definitions is still possible. If this property is violated, a warning will be given. We want the Demeter system to be able to guarantee that if B inherits from A then all objects in class B may safely be treated as A-objects. If B is defined by inheriting from A and adding additional variables, then the B-objects may be treated safely as A-objects. If B is defined by inheriting from A and modifying some of the types of the inherited instance variables (i.e. using an ∗override∗ clause) then the rule that the type must be specialized will insure that B-objects may be treated safely as A-objects. The B-objects will know about all the messages that are known to A-objects.

[2]The Law of Demeter states that messages may only be sent to self and objects which are an instance of a type which is associated with the instance variable types or argument types of the given method. This law is discussed in [LHR88]

There is an alternate way of specifying inheritance in the Demeter system: we can use an alternation production with implied inheritance. Syntactically this is expressed by using the the *common* token after the alternatives. For example,

```
Object : Pen | Brick
         *common* <weight> Number.
Pen = ...
Brick = ...
```

defines an abstract class (one which will never be instantiated) Object which is inherited by both Pen and Brick. Pen and Brick have a common instance variable called weight. It is helpful to think of these classes in terms of like relations. We say that a Pen is like a Brick in that they share a common data definition, namely, a weight. Alternation productions are the preferred way of introducing inheritance since it expresses the generalization principle in a natural way. Of course, for the purposes of grafting from previously developed projects, the explicit ∗inherit∗ is necessary. We assume that the underlying object-oriented system supports transitivity of the inheritance relation.

Our class dictionary notation requires that we give our data type information precisely for instance variables. If we say that an instance variable x is of type A, where A is defined by a construction production then x cannot contain an instance of a subclass of A; it has to be an instance of A. If we want to store a subtype of A in x we cannot define A by a construction production; we have to use an alternation production which specifies the legal subtypes of A.

For the definition of legality of objects and for subtyping, we need to define the set of classes associated with a given class. As an abbreviation of associated(S) we use S'.

Definition 1 *For a construction, repetition or terminal class S,* S' *is S. For an alternation class S,* S' *consists of the union of the classes associated with the alternatives on the right-hand-side of the alternation production.*

For the following class dictionary,

```
Expression : Simple | Compound.
Simple : Ident | Number.
Compound = ...
```

Expression' consists of Ident, Number and Compound.

Definition 2 *A construction object is* legal *with respect to a construction production if every instance variable contains an object which is an instance of a class associated with the instance variable class given in the production. There is an exception for optional instance variables: their value may be nil.*

A repetition object is legal *with respect to a repetition production if every element of the object is an instance of a class associated with the class on the right-hand-side of the repetition production. If the repetition production does not allow an empty list, the repetition object must contain at least one element.*

A terminal object is legal *if it contains a value which satisfies the rules defined in the scanner generator specification (i.e. its lexical definition).*

For defining the subtype concept we need the following definitions:

Definition 3 *A class T directly inherits from class S if the definition of T is of one of the following forms:*

```
T =  ... *inherit* S ...
S : T | ... *common* ...
```

Definition 4 *A class T inherits from class S if there is a sequence of intermediate classes which form direct inheritance links between class T and S.*

For example, the following production implies that Orange inherits from class EatableObject. Fruit is an intermediate class.

```
Fruit : Orange | Apple
  *common* *inherit* EatableObject.
```

Definition 5 *The class T is compatible with class S if the classes associated with the class T are a subset of the classes associated with class S.*

Definition 6 *A class T is a subtype (or subclass) of class S if*

- *T inherits from S or*
- *T is compatible with S.*

Every class is a subtype (or subclass) of itself.

We can summarize the subtype definition by:

$$\boxed{\text{Subtype}} = \boxed{\text{Inheritance}} \text{ or } \boxed{\text{Compatibility}}$$

To illustrate the subtype definition, consider the following example:

```
AutoDealer = <cars> List(Car).
Car :
  GasolineCar *override* <fuel> Gasoline |
  CombustionCar *override* <fuel> GasolineOrCoal
    *common* "speed" <speed> Number *inherit* Machine.
List(S) ~ {S}.
GasolineCar = "gasolinecar".
```

```
CombustionCar = "combustioncar".
Machine =
  "age" <age> Number "fuel" <fuel> FuelType.
FuelType : Coal | Gasoline | Electricity.
GasolineOrCoal : Gasoline | Coal.

Gasoline = "gas".
Coal = "coal".
Electricity = "electricity".
```

We define a class Car which inherits from class Machine. The Car class has two alternatives: GasolineCar and CombustionCar. Both override the inherited instance variable type for fuel. A machine can have any type of "fuel", be it gas, coal or electricity. A combustion car can use only gasoline or coal. Since we want to consider a combustion car as a special kind of car, we have introduced compatibility into our definition of subtype. There are other reasons to have both compatibility and inheritance in the subtype definition: reduction of inheritance links, avoidance of multiple inheritance, and efficiency.

The main idea behind the subtype definition is that if a type knows about a message then all objects belonging to subtypes of that type also know about that message. We want to prove that any combination of compatibility and inheritance will support this condition. The combination of compatibility and inheritance tends to constrain the form of class dictionaries. This is partially due to the fact that we restrict inherited classes to those defined by construction productions. We follow several definitions by a case analysis on the legal structure of class dictionaries for each of the four compatibility/inheritance combinations. We will define several semantic rules to deal with class dictionary structures which violate our definition of subtype.

To support our subtype definition we need need to introduce the concept of a signature. Informally, signature(S) for a class S consists of all messages m to which all classes in S' can react to by having a method attached to them or by inheritance.

To define the concept of signature more formally, we use the following notation:

Definition 7

- *A class S is* responsive *to message m if m is attached to S or there is a class T such that m is attached to T and there are direct inheritance links from S to T.*
- *A class S knows message m if each class in S' is responsive to m.*
- $Signature(S) = \{m | S \text{ knows } m \}$.

The distinction between responsiveness and knowledge is useful; knowledge involves the compatibility relationships while responsiveness relies on inheritance.

We give now the rules for computing the signature of a class. We define *attached*(C) to be the set of methods attached to C. We assume that the subtype graph does not contain cycles and we apply the following rules to a topologically sorted order of the classes. We assume that the signature of the abstract classes universal, construction, repetition, and terminal are given.

- CONSTRUCTION:

```
S = <name> A ...
signature(S) =
  {name, set-name, ...} union
    signature(construction) union
      attached(S).

S = *inherit* A.
signature(S) =
  signature(A) union
    signature(construction) union
      attached(S).

S = <name1> A ... *inherit* B *override* <name2> C ...
signature(S) =
  {name1, set-name1, name2, set-name2, ...} union
    signature(B) union
      signature(construction) union
        attached(S).
```

- REPETITION:

```
S ~ {R}.
signature(S) = signature(repetition) union
  attached(S).
```

- ALTERNATION:

```
   S : A | B.

signature(S) = signature(A) intersection signature(B)
```

- ALTERNATION with implied inheritance:

```
S : A | B *common* X Y.
A = <b> Ident.
B = <c> Ident.

signature(S) =
  (signature(A) intersection signature(B)) union
    {X, Y, set-X, set-Y} union
      attached(S).

signature(A) =
  {b, set-b} union signature(S) union
    signature(construction) union
      attached(S).
signature(B) =
  {c, set-c} union signature(S) union
    signature(construction) union
      attached(S).
```

We are now ready to prove an important property of the subtype predicate.

Theorem 1 Name-Signature-Theorem: *If* T *is a subtype of* S *then* signature(T) *is a superset of* signature(S).

To prove this theorem we need to define semantic rules which are easily checked at compile-time. We state those semantic rules in the proof at the place where they are needed to highlight the exact dependency of of the theorem on the semantic rules.

The proof is divided into the proof of four claims. They state that knowledge of messages and responsiveness to messages are transmitted properly along inheritance and compatibility links.

Claim 1 *If B inherits from A and A is responsive to m, then B is responsive to m.*

This follows immediately from the definition of responsiveness and the transitivity of inheritance.

Claim 2 *If B inherits from A and A knows m then B knows m.*

We have only to discuss the case where A knows m, but is not responsive to m. This implies that A is defined by an alternation production and every class in A' is responsive to m. If B is in A' then B is not an alternation and is responsive to m and therefore knows about m. If B is not in A', we have two cases: In the first case, B inherits from A using a *inherit* link. If we request that A be a construction class, we know that A must be responsive to m (which implies that B is responsive to m and therefore B knows m.) Therefore we use the following semantic rule.

Semantic rule 1 *Inheritance with *inherit* is only allowed from construction classes.*

In the second case, B is a subclass of A through alternation productions with implied inheritance and B is an alternation class itself. The claim follows with the following semantic rule:

Semantic rule 2 *In an alternation class with implied inheritance (i.e. using* ***common*** *) the alternatives which are defined by alternation productions must also use implied inheritance.*

Claim 3 *If B is compatible with A and A is responsive to m, then B is responsive to m.*

A must be defined by an alternation production. If A is responsive to m then A has either m attached or inherits it. If A has m attached, we require that the alternation production has implied inheritance with the following rule:

Semantic rule 3 *If an alternation class has a method attached, it must be defined with implied inheritance (i.e. with *common*).*

As a side-effect, this rule implies that if A is responsive to m then A knows m. Note that an alternation class B can only inherit from another class if B is an alternation class with implied inheritance.

Claim 4 *If B is compatible with A and A knows m then B knows m.*

A must be defined by an alternation production. We have to discuss the case where A knows m but is not responsive to m. All classes in A' are responsive to m. If B is in A' then B is responsive to m and therefore knows about m. If B' is a subset of A', the claim holds by the definition of knowledge of a message.

These four claims allow us to prove the theorem as follows: If T is a subtype of S then there must be a sequence of compatibility and/or inheritance links between T and S. The claims guarantee that the knowledge relation is transitive along those links which completes the proof of the Name-Signature-Theorem.

The subtype concept is used both for checking whether the type of the first actual parameter of a method is a subtype of the first formal parameter type and for checking the specialization hierarchy property for ∗override∗. The above proof guarantees the desirable property of name compatibility for Demeter subtypes. We have proven stronger forms of compatibility, but a discussion of this topic is beyond the scope of this paper. We now prove the transitivity of the subtype relation by examining four cases. This proof is needed since our subtype definition contains a disjunction (or). The subtype transitivity theorem is not essential for object-oriented programming. It is the Name-Signature-Theorem that is important.

Theorem 2 Subtype-Transitivity-Theorem: *The subtype relation is transitive.*

Proof by case analysis.

- If a class C inherits from a class B and the class B inherits from A then the class C inherits from the class A.

  1. True by definition of inheritance.

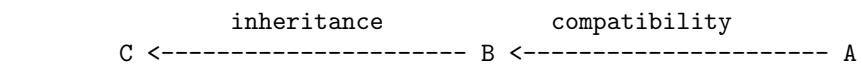
- If a class C is compatible with a class B and the class B is compatible with a class A then the class C is compatible to the class A.

  1. C' $\subseteq$ B' by def. of compatibility
     B' $\subseteq$ A' by def. of compatibility
     C' $\subseteq$ A' by transitivity of subset
     C is compatible to A by definition of compatibility

  For the last two cases we present class dictionary templates which demonstrate the relationship between class dictionary structure and subtype links.

- If a class C inherits from a class B and a class B is compatible to a class A then:

  1. This third case combines inheritance with compatibility such that there exists a class C which inherits from a class B and the class B is compatible with a class A. There are three possible class dictionary templates which will produce this type of subtype link. In the second we use set notation to restrict the classes which appear on the right hand side of B to be a subset of the classes appearing on the right hand side of A. This is to ensure the compatibility relation.




```
case 3a:

   A : B | ..... *common*.
   B : C | ..... *common*.

case 3b:

   A : C | {X1 ... XN}.
   B : C | { y | y is in subset of {X1 .. XN}} *common*.
```

```
   C = ......

case 3c (disallowed by semantic rule 5.4):
  for guaranteeing transitivity, not needed
  for theorem 5.1.

   A : B | ......
   B = .....
   C = *inherit* B.
```

In case 3a we can prove the subtype relation in two ways. C is both compatible to A via the definition of compatibility and C inherits from A via the definition of inheritance. This occurs because B is both compatible and inherits from A and C is both compatible and inherits from B.

2. In case 3b the class C is compatible to A by definition of compatibility. C' is a subset of A' by inspection.
3. We disallow the case 3c with the following semantic rule.

   Semantic rule 4 *A class which is used for inheritance with *inherit* cannot be used as an alternative in an alternation production without implied inheritance (i.e. without *common*).*

   We need this restriction since we cannot reduce the combined subtype links in class dictionaries of this form to either compatibility or inheritance exclusively. Our restriction does not reduce the expressiveness of our notation. We can simply add implied inheritance to the alternation production in case 3c (i.e. we use *common*).

- The fourth and final case is a class hierarchy where a class C is compatible to a class B and the class B inherits from a class A. There are four cases where this can occur. We will analyze these separately.

```
       compatibility            inheritance
C <---------------------- B <---------------------- A

case 4a:

  A = ......
  B : X | Y | Z *common* *inherit* A.
  C : X | Z.
```

```
case 4b (disallowed by semantic rule 5.3):

   A : B | ..... *common*.
   B : X | Y | Z.
   C : X | Y.

case 4c:

   A = ......
   B = ...... *inherit* A.
   C : B.

case 4d:

   A : B | ...... *common*.
   B = ......
   C : B.
```

1. In case 4a C inherits from A since all elements in C' inherit from A.
2. The case 4b has been defined as illegal by a semantic rule.
3. In cases 4c and 4d the class C inherits from class A since all elements in C' inherit from A.

This completes the proof of the Subtype-Transitivity-Theorem.

6 Parameterized Classes (The Root Stocks)

The notion of parameterized classes is related to grafting in that we inherit and/or override classes when we want to use parameterized classes. The difference is that these classes have one or more formal type parameters.

This parameterization adds complexity to the construction of a class dictionary, but the power endowed on such a system more than makes up for the additional complexity. Systems which are built around parameterized class dictionaries are constructed under a different philosophy than ones which are built around classes. Parameterized class systems are typically not designed to be stand alone working systems, although they sometimes are. These systems are built as generic foundations to be inherited by many software systems. By way of analogy, they are grown for their roots and not their fruit. The fruiting portion will be grown in other environments, and each will use the parameterized class system for roots.

For an example, consider a generic library of routines related to graphs. Since all graphs have nodes and edges of different types we would need a class

dictionary for each graph application. We could then exploit the use of inherit and override to reuse the software. However, it would be better to construct a generic graph class dictionary through the use of parameterized classes. Consider the following class dictionary which specifies a generic graph as a list of edges where an edge is defined as two nodes. We have added some concrete syntax, but recall that all applications are completely shielded from this syntactic sugaring of the input language.

```
Graph(Node) = "edgeList"
  <edgeList> List(Edge(Node)).
Edge(Node) =
  "from" <from> Node "to" <to> Node.
List(S) ~ {S}.
```

We can now write a lot of generic graph software which can be used by all graphs which can be represented as a list of edges where edges are represented as a list of two nodes. Let us assume we have made a large investment in this type of graph software and it has served us well. We now find that we have an application which requires labeled edges. This application happens to involve graphs where the nodes are towns or villages and the labels are the distance in kilometers between the towns and/or villages. We would like to write software which uses our previous investment in generic graph software and, if possible, we would like the new software to be generic for any graph with a labeled edge. We accomplish both goals by constructing the following class dictionary.

```
LabeledGraph(Node, EdgeLabel) =
  *inherit* Graph(Node)
    *override* <edgeList>
      List(LabeledEdge(Node, EdgeLabel)).
LabeledEdge(Node, EdgeLabel) =
  *inherit* Edge(Node)
    "label" <label> EdgeLabel.
```

We can use this parameterized class dictionary to define our regional map of towns, villages, and distances by giving actual parameters to the above class dictionary.

```
Map = *inherit*
  LabeledGraph (=> EdgeLabel Road, => Node Community).
Community : Village | Town
  *common* <communityname>, <desc> String.
Village = "village".
Town = "town".
Road = "road" <roadname> String
       <routeNumber>, <distance>Nnumber.
```

This class dictionary is expanded by the Demeter system into the following grammar:

```
map =
  "edgelist" < edgelist > road_community-labelededge-list .
community : village | town .
village =
  "village" < communityname > string < desc > string .
town =
  "town" < communityname > string < desc > string .
road =
  "road" < roadname > string
  < routenumber > number < distance > number .
community-edge =
  "from" < from > community "to" < to > community .
road_community-labelededge =
  "from" < from > community
  "to" < to > community
  "label" < label > road .
road_community-labelededge-list ~
  { road_community-labelededge } .
community-edge-list ~ { community-edge } .
community-graph =
  "edgelist" < edgelist > community-edge-list .
road_community-labeledgraph =
  "edgelist" < edgelist > road_community-labelededge-list .
```

The following shows the inheritance relationships between the generated classes. We use the following notation: We use `!` to introduce an abstract class (i.e. a class which cannot be instantiated) and `!!` to introduce an instantiable class in the underlying object-oriented system. *inherit* followed by several class names determines from where where the class inherits. These productions are generated by the system and can be drawn as a directed acyclic graph.

```
map !!
  *inherit* road_community-labeledgraph,
  construction .
community ! < communityname > , < desc > string .
village !! "village" *inherit* construction .
town !! "town" *inherit* construction .
road !!
  "road" < roadname > string
  < routenumber > , < distance > number
  *inherit* construction .
```

```
labeledgraph ! *inherit* construction .
labelededge !
  "label" < label > anytype *inherit* construction .
graph !
  "edgelist" < edgelist > anytype *inherit* construction .
edge !
  "from" < from > anytype
  "to" < to > anytype *inherit* construction .
list ! *inherit* repetition .
community-edge !! *inherit* edge .
road_community-labelededge !!
  *inherit* labelededge , community-edge .
road_community-labelededge-list !! *inherit* list .
community-edge-list !! *inherit* list .
community-graph !! *inherit* graph .
road_community-labeledgraph !!
  *inherit* labeledgraph , community-graph .
```

This inheritance allows us to write generic, reusable software. A typical input file for this class dictionary might look like:

```
edgelist
  from village "Ebnat-Kappel"  "small"
    to town "Wattwil" "bigger"
      label road "howart" 12 5
  from village "Diepoldsau" "on Austrian side"
    to village "Widnau" "on Swiss side"
      label road "bridge" 34 2
```

For a thorough description of the meanings of a parameterized class dictionary we refer the reader to [Lie88b].

Thus far we have used parameterized classes without restrictions. The following example illustrates the need for restrictions on the usage of parameterized classes. It shows that parameterized context-free grammars can define non-context-free languages.

```
ContextSensitive = <example> A(U,V,W).
A(X,Y,Z) : L(X,Y,Z) | R(X,Y,Z).
L(X,Y,Z) = <first> X <second> Y <third> Z.
R(X,Y,Z) = <u> A(H1(X), H2(Y), H3(Z)).
H1(S) = "u" <t> S.
H2(S) = "v" <t> S.
H3(S) = "w" <t> S.
U = "u". V = "v". W = "w".
```

When we apply the expansion process introduced with the labeled graph example to this class dictionary we get into an infinite loop: the expansion process

would generate infinitely many intermediate classes. Our implementation has to be protected from this condition.

Also, the language defined by this grammar is a well-known context-sensitive language which is known not to be context-free: $u^n v^n w^n, n > 0$. The recursive descent-parsing approach is not capable of parsing such languages. Therefore we will restrict the recursive parameterized class definitions so that any instance defines a context-free language. In applications one does not need the generality of the ContextSensitive example. The following condition places a bound on the parameterization but does not limit the practical applications.

Definition 8 *A parameterized class T is* bounded*, if every recursive use of T satisfies the restriction that every actual parameter of the recursive use is either:*

- *a formal parameter of T*
- *an instance of some parameterized class which does not have a formal parameter of T as an actual parameter.*
- *a class.*

Some interesting examples of bounded parameterized class definitions are:

- Permutation

```
MyT = <ex> T(Ident, Number).
T(X,Y) : E | N(X,Y).
E = "e".
N(X,Y) = "n" <first> X <second> Y <third> T(Y,X).
```

- Exponential

  A parameterized class instantiation can generate an exponential number of parameterized class instances (exponential in the size of the parameterized class definition and the size of the actual parameters). Consider the following example which basically generates all permutations of the symmetric group on 4 elements:

```
MyA = <ex> A(C1,C2,C3,C4).
C1 = "c1".
C2 = "c2".
C3 = "c3".
C4 = "c4".
A(X1,X2,X3,X4) =
  ["t1" <t1> A(X2,X1,X3,X4)]
  ["t2" <t2> A(X1,X3,X2,X4)]
  ["t3" <t3> A(X1,X2,X4,X3)].
```

The instantiation of parameterized class A on the first line generates 24 $= 4 * 3 * 2 * 1$ parameterized class instantiations or in general $n!$ (i.e. the size of the symmetric group on n elements). In practical applications this "permutation feature" is seldom used. One could exclude this exponential behavior by requiring that every parameter of a parameterized class be in the same position throughout a given rule(s).

- Splitting

```
MyT = <ex> T(Ident,Number).
T(X,Y) : E(X) | N(Y).
E(X) = String <first> X .
N(Y) = "n" <erstes> T(Y, List(Number)).
List(S) ~ "list" {S}.
```

A legal input is:

```
n n n n n "ab" list 45
```

The following theorems guarantee that bounded parameterized classes always define context-free languages and that the expansion of parameterized classes is finite.

Theorem 3 *An instance of a bounded parameterized class generates a finite number of classes.*

Theorem 4 *The language defined by an instance of a bounded parameterized class is context-free.*

The concept of subtype is easy to generalize to parameterized classes.

7 Organizing the Garden

We need a structuring mechanism to organize class dictionaries and the associated software. Therefore we introduce the concept of a module which is a collection of (parameterized) class and method definitions. We follow the Modula-2 rules and group our modules into two parts: a definition part and an implementation part. In the definition part we put all the classes which are important for clients and the method headings with type information for arguments and result. In the implementation part we put the implementation related class definitions and the complete methods. The implementation part also contains a list of file names which refer to the files containing the methods written in the language of the underlying object-oriented system. Modules and their organization into pairs have the following advantages:

- Information hiding. We can hide method bodies, the internals of construction classes and auxiliary classes which are only used for implementation.
- Separate environment generation and separate compilation.
- Avoiding conflicts between class names.

8 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 [Lie88b]. 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]. 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. [KMMN85] introduce an algebra of program fragments. The POPART system treats grammars as objects [Wil83]. The synthesizer generator project also uses a grammar based approach [Tei84]. 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 [DHK*75], [DHKL80] 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: [Lie88a] introduces the formal syntax for parameterized class dictionaries and discusses class dictionary design techniques.

Conclusions We have presented a new software design methodology based on growing and reusing software. We find growing software to be a better paradigm than building software. The philosophy of growing software requires that a running system be in place from the earliest stages of project development. This methodology has been implemented and tested in the prototype system "Demeter" which uses this growth paradigm to provide an ideal environment for the development and reuse of large software systems.

The Demeter system has been used by over 200 graduate and undergraduate students over the past 18 months. The feedback from these students has been very favorable and has indicated that the class dictionary is a very natural way of describing an abstract data structure. In addition, the learning curve for Demeter is short. For example, a group of 35 CS sophomore students have used Demeter within the framework of a software design and development course. Most of the students were beginning work on fairly complex systems in three weeks. All of these students had no previous experience with either Demeter or object-oriented programming, however, they did have some experience with Franz Lisp (one course).

In addition to the pedagogical uses of Demeter we have implemented many applications. Some of the more interesting ones include all the Demeter code generators and the generic programs, a simulator for the parallel language Zeus-2, simulators for subsets of Scheme and Prolog, a Fortran-77 to Pascal Translator, a Pascal parser generator, an expert system shell for modelling transportation systems, a generic adventure game generator, and a prototype environment for the modelling and simulation of arbitrary worlds of objects and rules. The latter five projects were developed by groups of five sophomore students in an eleven week course. This time includes the learning curve for Demeter and object-oriented programming.

We are continuing our work with Demeter and the philosophy of growing software both for teaching and research. We will investigate many exiting areas of research: Object-oriented specification, abstraction by example, class dictionary design from examples, expert system for class dictionary design.

Acknowledgments The design and implementation of the predecessor system of Demeter (called GEM, Pascal based) was done at GTE Laboratories in collaboration with Andrew Goldberg and with management support from Sandy Hirschhorn and Bill Griffin. At Northeastern, the Demeter project is supported by numerous dedicated graduate students, including (in alphabetical order): Bill Brown, Ian Holland, Gar-Lin Lee, Kathy Lee, David Lincoln, Benoît Menendez, Jing Na, Robert Paul, Johannes Sujendro and Carl Woolf.

Mitchell Wand taught us that a parameterized context-free grammar does not necessarily define a context-free language. He brought the example, which we called ContextSensitive in this paper, to our attention. This example appeared originally in Michael Fischer's thesis at MIT. Mitchell's seminar on semantics of programming languages was very useful for learning about several key contributions to the field.

Carl Woolf has improved the proof of theorem 5.1. He proposed the terminology of responsiveness to a message and knowledge of a message.

Professors Brown, Bugrara, Casey and Rasala provided valuable feedback on this paper.

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.

[Bro87] Frederick P. Brooks. No silver bullet, essence and accidents of software engineering. *IEEE Computer Magazine*, 10–19, April 1987.

[CI84] R.D. Cameron and M.R. 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.

[DHK*75] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J.J. Levy. A structure oriented program editor: a first step towards computer assisted programming. In *Proceedings of International Computing Symposium 1975*, page , 1975.

[DHKL80] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. 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. *Publ. N. S-22, Norwegian Computing Center*, Oct. 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*, pages 963–968, 1986.

[Fra81] C.W. Fraser. Syntax-directed editing of general data structures. In *Proc. ACM SIGPLAN/SIGOA Conference on Text Manipulation*, pages 17–21, Portland, Oregon, 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] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison Wesley, 1983.

[KMMN85] 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*, SIGPLAN, Seattle, Washington, 1985.

[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.

[LHR88] Karl J. Lieberherr, Ian Holland, and Arthur J. Riel. Object-oriented programming: an objective sense of style. In *Object-Oriented Programming Systems, Languages and Applications Conference*, San Diego, CA., September 1988. A short version of this paper appears in IEEE Computer, June 88, Open Channel section, pages 78-79.

[Lie88a] Karl Lieberherr. Object-oriented programming with class dictionaries. *Journal on Lisp and Symbolic Computation*, 1(2):pages unknown, 1988.

[Lie88b] Karl J. Lieberherr. *The Art of Growing Software: An object-oriented CASE approach.* publisher to be determined, January 1988. book in preparation, approximately 400 pages.

[LR88] 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*, pages 391–405, 1986.

[Mil71] H.D. Mills. Top-down programming in large systems. In R. Ruskin, editor, *Debugging Techniques in Large Systems*, Prentice Hall, 1971.

[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*, pages 1–8, Portland, Oregon, 1986.

[PS83] H. Partsch and R. Steinbrueggen. Program transformation systems. *ACM Computing Surveys*, 15(3):199–236, September 1983.

[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*, pages 9–16, 1986.

[Sol78] M. Solomon. Type definitions with parameters. In *Principles of Programming Languages*, Tucson, Arizona, 1978.

[SS77] J.M. Smith and D.C.P. Smith. Data base abstractions: aggregation and generalization. *ACM Transactions on Database Systems*, 2(2), June 1977.

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

[Tei84] T. Reps T. Teitelbaum. The synthesizer generator. *SIGPLAN*, 19(5), 1984.

[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*, 711–732, 1982.

[Wil83] D. Wile. Program developments: formal explanations of implementations. *Communications of the ACM*, 26(11), 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, Englewood Cliffs, N. J., 1976.

[Wir77] N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Comm. ACM*, 20(11):822–823, 1977.

[Wir84] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, 1984.