Chapter 11

Class Dictionaries

Class dictionaries are more sophisticated propagation pattern customizers than are class dictionary graphs. With a class dictionary we can choose not only the detailed structure of objects, but also an application-specific notation for describing the objects succinctly. This notation allows us to describe "input stories" for propagation patterns. For example, for a restaurant administration program we can write a story that describes today's menu.

Objects are important for object-oriented design and programming, but they are too bulky to look at or to produce by hand. An example of an object is

```
Compound(
  <op> MulSym()
  <args> ArgList{
    Variable(
      <v> DemIdent "a")
    Variable(
      <v> DemIdent "b")})
```

Can we find a more succinct way to describe objects than the textual object notation or, even worse, the statements of a programming language (e.g., constructor calls)? Which information is essential in the objects? We certainly need the values of the atomic objects and some information about how those atomic objects are grouped together into larger objects. In the above example we need to know that `a` and `b` are atomic objects of the expression. This grouping can be expressed with some extra strings that we put between the atomic objects to allow a program to recover an object from a sentence. We use the word sentence simply to mean a sequence of terminals. It can be a proper English or French sentence or a stylized sentence, or it can be anything. To make a complete sentence out of `a b`, we use a few extra terminals: `(* a b)`. Although much shorter than the above object, it conveys the same information if we use a class dictionary to interpret `(* a b)`.

We first study how we can assign a concise sentence to an object. The goal in the back of our mind is to make the sentences expressive enough so that we can recover the objects automatically.

Consider again the meal example. We would like to describe a meal with a sentence such as

```
Appetizer:
  Melon
Entree:
  Steak Potato Carrots Peas
Dessert:
  Cake
```

instead of using the object notation. We can achieve this with the class dictionary in Fig. 11.1. A sentence is like a story about an object. The stories can be concise, like the one above, or verbose, and it is the class dictionary designer who decides. For example, the above meal description could be given by the following sentence:

```
At Hotel Switzerland you will enjoy
  Melon as an appetizer
  Steak Potato Carrots Peas as entree
  and Cake as a delicious dessert.
You will enjoy a splendid view of the Alps during good weather.
```

It is easy to adjust the class dictionary in Fig. 11.1 so that meals are represented in the above verbose form as sentences. All we need is to replace the first class definition by

```
Meal = "At Hotel Switzerland you will enjoy"
  Appetizer "as an appetizer"
  Entree "as entree and"
  Dessert "as a delicious dessert."
  "You will enjoy a splendid view of the Alps during good weather.".
```

If sentences are like stories about objects, then class dictionaries are like templates for stories. A class dictionary prescribes precisely how we have to write the stories about the application objects. Class dictionary design is like designing story templates.

We can also look at a sentence as describing a family of objects. We select a specific object from the family by selecting a class dictionary that is compatible with the sentence. From this point of view a sentence is like a propagation pattern: both are customized by a class dictionary.

Conceptually, a class dictionary is very similar to a class dictionary graph. A class dictionary can be viewed as a class dictionary graph with comments required to define the input language.

Concrete syntax (also known as syntactic sugar) is used to "sweeten" the syntax of the sentences. Below are examples of construction, alternation, and repetition class definitions which show where concrete syntax may be used.[1] We call the concrete syntax elements **tokens**.

[1] Legal class dictionary, page 437 (32).

```
Meal =
  "Appetizer:" Appetizer
  "Entree:" Entree
  "Dessert:" Dessert.
Appetizer : Melon | ShrimpCocktail.
ShrimpCocktail = "Shrimp Cocktail" Shrimps Lettuce [CocktailSauce].
CocktailSauce = Ketchup HorseRadish.
Entree : SteakPlatter | BakedStuffedShrimp.
SteakPlatter = Steak Trimmings.
BakedStuffedShrimp = StuffedShrimp Trimmings.
Trimmings = Potato <veggie1> Vegetable <veggie2> Vegetable.
Vegetable : Carrots | Peas | Corn.
Dessert : Pie | Cake | Jello.
Shrimps ~ Shrimp {Shrimp}.

Shrimp = .
Melon = "Melon".
Lettuce = .
Ketchup = .
Steak = "Steak".
Potato = "Potato".
Carrots = "Carrots".
Peas = "Peas".
Cake = "Cake".
Pie = "Pie".
Jello = "Jello".
Corn = "Corn".
StuffedShrimp = "Stuffed Shrimp".
HorseRadish = .
```

Figure 11.1: Meal language

Construction class:

```
Info =
  "Demeter System" <t> Trademarked
  "followed" "by"
  ["Law of Demeter" NotTrademarked]
  "developed" "at" Northeastern.
```

Each part may have some syntax associated with it that can appear before or after the part.

Alternation class:

```
Fruit: Apple | Orange *common*
  "weight" <weight> DemNumber "end".
```

The alternatives of an alternation class may not contain syntax.

Repetition class:

```
List ~ "begin" "list"
  {"before-each" Element "after-each"}
  "end" "list".

List ~ "first" Element
  {"separator" "prefix" Element "suffix"}
  "terminator".
```

Syntax is not allowed between the first element and the repeated part. To specify the language defined by a class dictionary, we first translate a class dictionary into a class dictionary without common parts; that is, into a flat class dictionary. The class dictionary without common parts[2] is then used as a printing table to print a given object.[3] The collection of all printed legal objects constitutes the language of the class dictionary.

The expansion of common parts is best demonstrated with an example. Consider the class dictionary

```
Basket = <contents> Fruit_List.
Fruit_List ~ {Fruit}.
Fruit : Apple | Orange *common*
  "weight" <weight> DemNumber "end".
Apple = "apple".
Orange = "orange".
```

After expansion of common parts

[2] Class dictionary flattening, page 439 (33).
[3] Printing, page 439 (34).

```
// flat class dictionary
Basket = <contents> Fruit_List.
Fruit_List ~ {Fruit}.
Fruit : Apple | Orange.
Apple = "apple"
  "weight" <weight> DemNumber "end".
Orange = "orange"
  "weight" <weight> DemNumber "end".
```

The common parts are flattened out to all the construction classes; therefore we call the expanded class dictionaries flat. Flat class dictionaries are usually not written by the user but are produced from nonflat class dictionaries by tools. Flat class dictionaries are a useful intermediate form. Notice that the flattening operation is well defined since there can be no cycles of alternation edges in a class dictionary graph.

For flat class dictionaries it is straightforward to define a printing operation[4] that is applicable to any object. We determine the class of the object and look up the class definition. Then we print the object according to the class definition, including the concrete syntax. For example, to print an Apple-object, we first print weight followed by printing a DemNumber-object followed by printing end. The set of all legal objects in printed form for some class dictionary G is the language defined by G. The language defined by G is sometimes called the set of sentences defined by G.

To demonstrate the printing algorithm we use the above class dictionary for baskets. Consider the following Basket-object that we want to print.

```
Basket (
  < contents >  Fruit_List {
      Apple (
        < weight >  DemNumber "2" ) ,
      Orange (
        < weight >  DemNumber "5" ) } )
```

When we print it, we get the following output:

```
// sentence describing a Basket-object
apple weight 2 end
orange weight 5 end
```

If we change the class dictionary to

```
Basket = "basket" <contents> Fruit_List.
Fruit_List ~ "(" {Fruit} ")".
Fruit : Apple | Orange *common*
  "weight" <weight> DemNumber.
Apple = "apple".
Orange = "orange".
```

[4] Printing, page 439 (34).

the same object appears as

```
basket
  ( apple weight 2 orange weight 5 )
```

11.1 PARSING

We know how a class dictionary defines a language by assigning a sentence to each object. An object represents the structure of a given sentence relative to a class dictionary. A class dictionary is closely related to a grammar, the main difference being that a grammar defines only a language and a class dictionary additionally defines classes. (Knowledge of grammars is *not* a prerequisite for understanding this section.) Examples of two grammars, using the Extended Backus-Naur Form (EBNF) notation are in Fig. 11.2.

```
// Grammar 1
Basket = {Apple | Orange}.
Apple = "apple" "weight" DemNumber "end".
Orange = "orange" "weight" DemNumber "end".

// Grammar 2, almost a class dictionary
Basket = Fruit_List.
Fruit_List = {Fruit}.
Fruit = Apple | Orange.
Apple = "apple" "weight" DemNumber "end".
Orange = "orange" "weight" DemNumber "end".
```

Figure 11.2: Two grammars defining the same language

The differences between a grammar and a class dictionary are

- A grammar is usually shorter than a class dictionary since it is not concerned about object structure.
- A grammar does not have labels to name parts.
- A grammar does not have common parts; it is like a flat class dictionary.
- The syntax for grammars and class dictionaries is different but grammars can be written in a form that is close to a class dictionary (see Grammar 2 in Fig. 11.2).

Normally we are interested in defining an object by reading[5] its description from a text file. We call such a description a sentence that defines an object. A special kind of object, called a **tree object**, is defined by a sentence. It is called a tree object since its

[5] Parsing, page 441 (38).

underlying graph structure, given by the reference relationships between the objects, is a tree. Not every object is a tree object for some sentence. There are also circular objects and objects that share subobjects. An object o is said to be a **tree object** for a sentence s if its structure is a tree and not a general graph. Tree objects o have the property that printing o and reading the sentence again returns an object identical to the original object o. Not every object needs to be a tree object since many objects are built under program control, and there is never a need to read them from a text file.

The class dictionary contains all the information that is usually put into a grammar for defining a language. Therefore standard parser generator technology can be used to generate a parser automatically from the class dictionary. The parser takes as input a sentence in some file and returns the corresponding tree object. The grammar given in the class dictionary defines how to build the tree object.

We want to restrict ourselves to a subset of all class dictionaries that promote good "object story writing". We want the stories to be easy to read and write and learn. We also want the stories to be unique so that no two different stories describe the same object.

Therefore we introduce the concept of an ambiguous class dictionary. A class dictionary is **ambiguous** if there exist two distinct objects that map to the same sentence when they are printed. An example of an ambiguous class dictionary is:

```
Basket = <fruits> Fruit_List.
Fruit_List ~ {Fruit}.
Fruit : Apple | Orange.
Apple = "apple".
Orange = "apple".
```

The sentence `"apple apple"` represents four different kinds of baskets:

- A basket with two apples
- A basket with one apple and one orange
- A basket with one orange and one apple
- A basket with two oranges.

Therefore, the class dictionary is ambiguous.

We want to avoid ambiguous class dictionaries; therefore we need an algorithm to check whether a class dictionary is ambiguous. Not all problems are algorithmically solvable and computer scientists have found many computational problems that are provably not algorithmically solvable. Indeed, the class dictionary ambiguity problem cannot be solved by an algorithm. This can be proved by a reduction that shows that if the class dictionary ambiguity problem is solvable, then one of the provably unsolvable problems (Post's correspondence problem) is solvable. This leads to a contradiction and therefore the class dictionary ambiguity problem is not algorithmically solvable.

We need to look for a work-around regarding the checking of a class dictionary for ambiguity. The solution is to restrict our attention to a subset U of all class dictionaries that are useful in practice and for which we can solve the ambiguity problem efficiently.

We also need to find a subset U so that we can efficiently check whether a class dictionary belongs to U or not.

We choose U to be the set of LL(1) class dictionaries. We can efficiently check whether a class dictionary is LL(1), and in fact all LL(1) class dictionaries are not ambiguous. An LL(1) class dictionary has to satisfy two rules. We will learn Rule 1 shortly; Rule 2 is more technical and is explained in the next section and in the theory part of the book.

The LL(1) class dictionaries tend to define languages that are easy to learn and read. LL(1) class dictionaries are therefore very useful in practice especially in an environment where languages change frequently.

We parse class dictionaries by so-called recursive descent parsing, which will be explained next. This explains in detail how an object is constructed from a sentence. Recursive descent parsing is a standard concept from compiler theory; refer to your favorite compiler book (for example, [ASU86]) to learn how recursive descent parsing is used to build compilers.

A sentence is made up of terminals. There are two kinds of terminals, namely terminals with a value and terminals without a value. A number such as 123 is a terminal with a value and it represents an object of terminal class DemNumber. As a rule, terminals with values are representing objects belonging to a terminal class. Terminals without values correspond to the terminals appearing in the class dictionary. For example, apple orange are two terminals that correspond to the two terminals in the following class dictionary:

```
Fruits = Apple Orange.
Apple = "apple".
Orange = "orange".
```

We also call the terminals without value **tokens**. Notice that we overload the token concept since the syntax elements in a class dictionary are also called tokens.

Recursive descent parsing is best explained by mapping class dictionaries into **syntax graphs** (also called **syntax charts** or **syntax diagrams**) which are widely used for defining programming languages. In a syntax graph, classes are shown inside rectangles and tokens inside ovals. For every class there is one syntax graph. The syntax graph of a construction class

$$A = B_1[B_2] \dots \ B_n.$$

is given in Fig. 11.3.

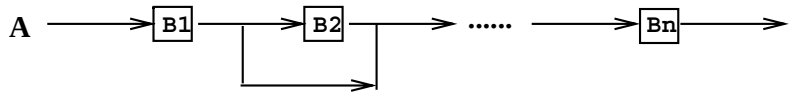

Figure 11.3: Syntax graph construction

The syntax graph of a repetition class

```
A ~ {S ";"}.
```

is given in Fig. 11.4.

The syntax graph of a repetition class

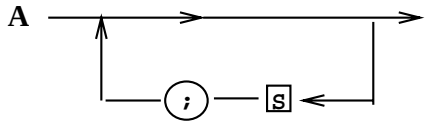

Figure 11.4: Syntax graph repetition

```
A ~ S { ";" S}.
```

is given in Fig. 11.5.

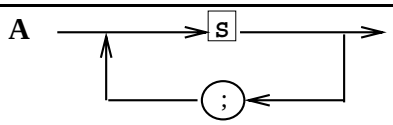

Figure 11.5: Syntax graph repetition (nonempty)

The syntax graph of an alternation class

$$A : B_1|B_2|\ldots\ |B_n.$$

is given in Fig. 11.6.

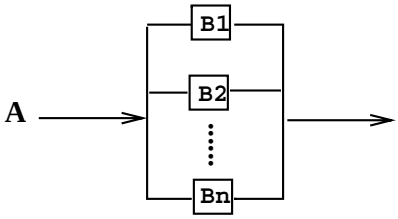

Figure 11.6: Syntax graph alternation

The parser works like a train that is trying to load a sentence while traversing the syntax graphs. (The sentence is broken into a sequence of terminals by a scanner.) The train enters the start syntax chart which corresponds to the start class of the class dictionary. Whenever the train enters a rectangle it moves to the syntax graph of the class in the rectangle. Whenever the train enters an oval it loads the token, provided it is the next terminal in the sentence to be parsed. The train stops and signals a syntax error if there is a different terminal in the input sentence.

The train has to make a decision whenever it comes to an intersection. We assume that the decision is made with a look-ahead of only one terminal and that no backtracking

will be necessary. In a syntax graph that corresponds to a repetition class there is one branching point with a branching factor of 2. In a syntax graph that corresponds to a construction class there are as many branching points with branching factor 2 as there are optional elements. In a syntax graph that corresponds to an alternation class there is one branching point with a factor of n, where n is the number of classes on the right side of the alternation class definition.

To define the decision process more formally we have to define the **first set** $first(S)$ for every class S.[6] $first(S)$ is the set of all terminals that can appear as first terminal in a sentence of S. A branch gets labeled with the set $first(S)$. When S may derive the empty string, we define that $first(S)$ contains *epsilon*.

We give several representative examples of how to compute first sets. We describe first sets as sets of strings between quotes, epsilon, and terminal class names. *terminal-class* DemIdent is used for class DemIdent (similarly for other terminal classes) and epsilon stands for the empty string.

- Construction classes:

```
A = B C.
B = "is".
first(A) = first(B) = "is"

A = [B] [C] "is".
first(A) = first(B) union first(C) union "is"

A = [DemString] [DemNumber] DemIdent.
first(A) =
  {*terminal-class* DemString,
   *terminal-class* DemNumber, *terminal-class* DemIdent}

A = B C.
B = [DemIdent].
C = ["else" DemString].
first(A) = first(B) union first(C) union epsilon
         = {*terminal-class* DemIdent, epsilon, "else"}
```

epsilon is in the first set since the language of A contains the empty string as a legal sentence.

```
A = B C.
B = [DemIdent].
C = "else" DemString.
first(A) = first(B) union first(C) removing epsilon
         = {*terminal-class* DemIdent, "else"}
```

[6] First sets, page 439 (37).

- Repetition classes:

```
A ~ {DemIdent}.
first(A) = {*epsilon*, *terminal-class* DemIdent}

A ~ "is" {DemIdent}.
first(A) = "is"

A ~ {DemIdent} "is".
first(A) = first(DemIdent) union "is"
         = {*terminal-class* DemIdent, "is"}
```

- Alternation classes:

```
A : B | C.
B = "b".
C = "c".
first(A) = first(B) union first(C) = {"b", "c"}

A : B | C.
B ~ {DemIdent}.
C = "c".
first(A) = first(B) union first(C)
         = {*terminal-class* DemIdent, *epsilon*, "c"}
```

To simplify the decision process at a branching point we make the following assumption:

- We require that all the branches at a branching point in an alternation class definition have disjoint first sets (Rule 1).

With this restriction it is easy for the train to make these decisions. At an alternation class branching point, compare the next input terminal in the sentence to be parsed with the first sets of the branches. If the next input terminal is contained in any of those first sets we take that branch. Otherwise an error message is printed unless *epsilon* is in the first set of one branch, in which case this epsilon branch will be taken. According to Rule 1, only one branch may contain *epsilon*.

At a construction class branching point, we check whether the next input terminal is in the first set of what is inside the square brackets. If it is, we take the path that brings us to the optional terminal; otherwise, we take the other branch.

At a repetition class branching point, we check whether the next input terminal is in the first set of what is inside the curly brackets. If it is, we take the path through the loop, else we take the other branch.

This description implies that we have to compute the first function not only for classes, but for classes that might be preceded by terminals. This is a straightforward generalization. If the class is preceded by a string then the first set contains only that string.

We have seen that an error message can be generated at a branching point inside an alternation class definition. At a branching point inside a construction or repetition class definition we will never generate an error message. However, the parser generates an error message at nonbranching points, namely whenever a specific terminal is expected and that terminal is not the next input terminal.

We now extend the parser described above so that it returns a tree object for a given input string. This tree object stores the structural information about the string, but not all the details. None of the strings in the grammar definition will show up in the tree object.

Whenever the train starts a new syntax graph G that corresponds either to a construction or repetition class definition, a class instance is created. If G is defined by a construction class, the values of the parts will be assigned recursively when the syntax graphs of the classes on the right side are traversed. If G is defined by a repetition class, a list of objects will be created. It will be a list as long as the number of repetitions of objects of the class on the right side of the repetition class. Whenever the train starts a new syntax graph G that corresponds to an alternation class, the tree object remains unchanged.

The basket examples at the end of the last section also serve as examples for parsing.

11.2 LL(1) CONDITIONS AND LEFT-RECURSION

The LL(1) conditions for a class dictionary consist of two rules.[7] These conditions exclude ambiguous class dictionaries while being checkable efficiently. We have already discussed the first LL(1) rule since it is needed by the parsing algorithm. The first LL(1) rule requires that the first sets of the alternatives of an alternation class are disjoint. The second LL(1) rule is needed to make class dictionaries nonambiguous. To define Rule 2 we need to introduce follow sets.

$follow(A)$ for some vertex A consists of all terminals that can appear *immediately after* a sentence of $L(A)$. $L(A)$ is the set of all printed A-objects. The follow sets are computed with respect to the start class, which is the first class appearing in the class dictionary. For terminals of terminal sets, the corresponding terminal class name is given. If the end of file can appear after a sentence of $L(A)$, then $follow(A)$ contains `eof`.

Consider the example class dictionary

```
Basket = <contents> SeveralThings.
SeveralThings ~ {Thing}.
Thing : Apple | Orange *common* <weight> DemNumber.
Apple = "apple".
Orange = "orange".
```

Some of the follow sets are

$follow($`Thing`$) =$ { `eof, "apple", "orange"` },
$follow($`SeveralThings`$) =$ { `eof` }.

The follow set of Thing contains `eof` since a Thing-object can be the last thing in a basket. It contains `"apple"` since an apple may appear after a Thing-object.

[7] LL(1) conditions, page 442 (42).

Now we can formulate the second and last rule of the LL(1) conditions
Rule 2:
For all alternation classes

```
A : A1 | ... | An.
```

if an alternative, say $A1$, contains *empty* in its first set $first(A1)$, then $first(A2)$, $first(A3)$, ... have to be disjoint from $follow(A)$.

The following example motivates Rule 2. The class dictionary

```
Example = <l> List <f> Final.
List : Nonempty | Empty.
Nonempty = <first> Element <rest> List.
Empty = .
Element = "c".
Final : Empty | End.
End = "c".
```

violates Rule 2.

We choose A1 to be Empty and A2 to be Nonempty and A to be List. The relevant first and follow sets are

```
first(Empty) = {empty}.
first(NonEmpty) = {"c"}.
follow(List) = {"c", eof}.
```

Now first(NonEmpty) is not disjoint from follow(List) and therefore Rule 2 is violated.

The following two objects have the same corresponding sentence

```
object 1:

:Example(
  <l> :Empty()
  <f> :End())

object 2:

:Example(
  <l> :Nonempty(<first> :Element() <rest> :Empty())
  <f> :Empty())
```

In both cases, the sentence c is printed. For object 1, c is printed by the End-object. For object 2, c is printed by the Element-object.

Violation of the LL(1) conditions does not necessarily imply that the class dictionary is ambiguous. For example, the following class dictionary is not LL(1) but it is not ambiguous.

```
Example = <l> List <f> Final.
List : Nonempty | Empty.
```

```
Nonempty = <first> Element <rest> List.
Empty = .
Element = "c".
Final : End.
End = "c".
```

Rule two of the LL(1) conditions is violated. follow(List) contains "c" and so does first(Nonempty). But we cannot find two distinct objects that are mapped to the same sentence by the printing function *g_print*. This example however shows parsing ambiguity. When the parser sees the "c" terminal in the input while parsing an Example-object, it does not know whether to build a Nonempty or an Empty-object in part 1. The first "c" terminal has two different interpretations. We can represent it as an Element-object or as an End-object.

The LL(1) conditions force the object printing algorithm *g_print*() to have a useful property. We can always retrieve the object from the output of the printing algorithm. The LL(1) conditions are sufficient for *g_print* to be a **bijection** (i.e., onto and one-to-one) between tree objects and sentences. If a flat class dictionary G satisfies the LL(1) Rules 1 and 2 then the function $g\_print(G, \omega)$ is a bijection from C-objects in $TreeObjects(G)$ to $L(C)$. A flat class dictionary is a class dictionary where all common parts and terminals have been pushed down to the construction classes.

The inverse of *g_print* is function $g\_parse : for\ \ all\ \ \omega \in TreeObjects(G)$

$$\omega = g\_parse(G, class\ of\ \omega, g\_print(G, \omega))$$

11.2.1 Left-Recursion

The LL(1) rules exclude a certain kind of left-recursion.

Informally, a class dictionary is left-recursive if it contains paths along which no input is consumed. An example of such a class dictionary is

```
Basket = <contents> Contents.
Contents : Fruit | Basket.
Fruit : Apple | Orange.
Apple = "apple".
Orange = .
```

There is left-recursion that involves the two classes

```
Basket = <contents> Contents .
Contents : Fruit | Basket .
```

We can go through the cycle Basket, Contents any number of times without consuming input.

This kind of left-recursion is a special case of LL(1) condition violation; specifically, a Rule 1 violation. Consider the first sets of the two alternatives of Contents:

```
first(Fruit) = {"apple", empty}.
first{Basket} = {"apple", empty}
```

The two first sets are not disjoint and therefore Rule 1 is violated.

Left-recursion can appear in a second form; consider the class dictionary graph

```
Mother = <has> Child.
Child  = <has> Mother.
```

Here the LL(1) conditions are satisfied but we still have left-recursion. This kind of left-recursion is excluded by the inductiveness axiom, which is discussed in the chapter on class dictionary design techniques (Chapter 12).

11.3 SUMMARY

A class dictionary D defines a language through the following mechanism. We consider all objects defined by the class dictionary graph G contained in D. This set is called $TreeObjects(D)$. We apply the print function which prints each object in $TreeObjects(D)$, and we call the resulting set $Sentences(D)$. This is the language defined by D.

To facilitate the writing, understanding, and learning of sentences, we use a subset of class dictionaries, called LL(1) class dictionaries. An LL(1) class dictionary is not ambiguous, and has other desirable properties. Specifically, different alternatives of an alternation class are introduced by different tokens.

This chapter explained the parsing process in detail, which takes a class dictionary and a sentence and constructs the corresponding object.

The relationships between class dictionaries and class dictionary graphs is summarized in Fig. 11.7. Four properties are considered in the figure: nonambiguous, LL(1), inductive,

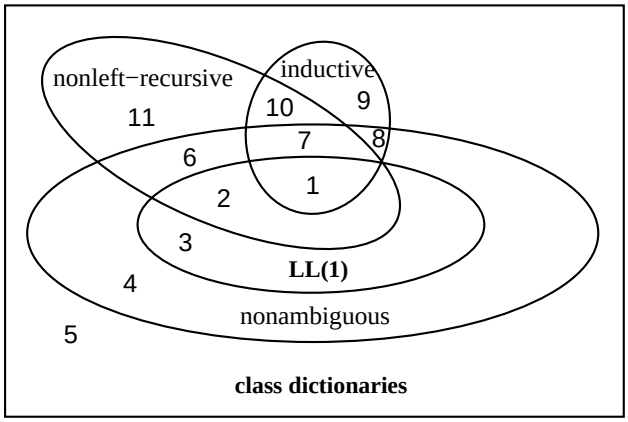

Figure 11.7: Venn diagram for class dictionaries

nonleft-recursive. Inductive class dictionaries are discussed in Chapter 12 but we give here the intuition: A class dictionary is inductive if it contains only good recursions; that is, recursions that terminate. Ideally, a class dictionary should satisfy all four properties. If the properties were independent, sixteen different sets would be defined by the four

properties. However, there are only eleven because of the implication relationships between the properties (LL(1) implies nonambiguous, LL(1) and inductive imply nonleft-recursive).

We show example members for some of the eleven sets.

1. Nonambiguous, nonLL(1), noninductive, and left-recursive

```
A = B C.
B : E | C.
C = "c".
E = E.
```

2. LL(1), left-recursive, noninductive, Fig. 11.8

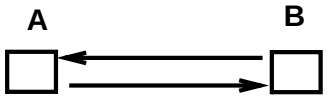

Figure 11.8: LL(1), left-recursive, noninductive

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

3. LL(1), nonleft-recursive, inductive, Fig. 11.9

Figure 11.9: LL(1), nonleft-recursive, inductive

```
A = .
```

4. Nonambiguous, nonLL(1), left-recursive, inductive

```
A = B.
B : A | C.
C = "c".
```

5. Ambiguous, nonLL(1), inductive

```
A = B C.
B : U | V.
C : G | H.
U = "c".
V = .
G = "c".
H = .
```

6. Ambiguous, nonLL(1), noninductive, left-recursive

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

11.4 EXERCISES

Exercise 11.1 (Design and implementation objective)

Write a class dictionary that defines Lisp lists, assuming that the atoms are only identifiers. Your language should handle the following examples:

```
()
(a b c)
(a (a b c) d)
(a (a b () c ( a b)) d)
etc.
```

Write a program for the class dictionary that counts the number of atoms in a Lisp list.
The answer for the above examples should be
0 3 5 7 etc.
Verify that your class dictionary satisfies the LL(1) properties.

Exercise 11.2 Write an adaptive program so that it removes all while statements from a Modula-2 program. We assume that the class dictionary for Modula-2 contains the following class definitions:

```
StatementSeq ~ Statements {";" Statements}.
Statements = [Statement].
Statement : WhileStat | IfStat ...
WhileStat = "while" ...
```

You can assume that Statements is used only in StatementSeq.

Exercise 11.3 (Programming for given class dictionary objective)

The following class dictionary defines the data structures used by company Zeus Inc.

```
CustomerList ~ {Customer}.

Customer =
  <customerNumber> DemNumber <customerName> DemString
  <customerAddress> Address <telephone> DemNumber
  <contracts> ContractList.

Address =
  <street> DemString <city>   DemString
  <state>  DemString <zip>    DemNumber
  <phone>  DemNumber.

ContractList ~ Contract {Contract}.

Contract =
  <contractNumber> DemNumber <deliveryAddress> Address
  <date> DemString <remarks> DemString
  ContractLines.

ContractLines ~ ContractLine {ContractLine}.

ContractLine =
  Part <quantity> DemNumber
  <discount> DemNumber <amount>   DemNumber.

Part =
  <partNumber> DemNumber <description> DemString
  <price> DemNumber.
```

The company Zeus Inc. has to send a letter to all customers who bought part number 4556. Write an adaptive program that prints out the addresses of all customers who ordered part 4556. The format of the addresses is unimportant, as long as the street, city, state, and ZIP code is contained in each address.

Exercise 11.4 (Design and implementation objective)

.

1. Invent a notation for describing any given position on a chess board, write a class dictionary for it, and write a program that prints the number of white pieces on a given board.

2. Give a sample input that describes a board with about five pieces on it.

3. Give the same board position in the object notation.

Exercise 11.5 Consider the following class dictionary:

```
A ~ {B}.
B : C | D .
C = "xxx" A ["if" B] "yyy".
D = .
```

Check the following inputs for syntactical correctness. For each input that is syntactically correct, give the object in the object notation.

- 1 2 3 xxx if 999 yyy
- a b c xxx 1 2 3 yyy
- xxx if yyy

Exercise 11.6 (Programming for given class dictionary objective)

A postfix expression is an expression where the operator comes after the arguments. For example, [3 4 *] is a postfix expression that evaluates to 12.

Consider the following postfix expression language:

```
Example = ExpressionList.
ExpressionList ~ { Expression }.
Expression : Simple | Compound.
Simple = <v> DemNumber.
Compound = "[" <argument1> Expression <argument2> Expression
               Operator "]".
Operator : MulSym | AddSym | SubSym.
MulSym = "*".
AddSym = "+".
SubSym = "-".
```

Write a program that returns the list of evaluations of the postfix expressions. For example, if the input contains 2 3 [3 4 *] then the program returns the list (2 3 12).

Exercise 11.7 (Programming for given class dictionary objective)

Write a program that operates on a grammar that satisfies the following class dictionary

```
Grammar ~ {Rule}.
Rule = <ruleName> DemIdent Body ".".
Body : Construct | Alternat | Repetit.
Construct = "=" <partsAndSyntax> List(AnySymbol).
Alternat = ":" <alternatives> BarList(DemIdent).
List(S) ~ {S}.
BarList(S) ~ S {"|" S}.
SandwichedSymbol = <first> AuxList Symbol <second> AuxList.
Repetit = "~" <first> AuxList [ <nonempty> DemIdent ]
              "{" SandwichedSymbol "}" <second> AuxList.
AnySymbol : Symbol | OptSymbol | Aux.
```

```
Symbol = [ "<" <labelName> DemIdent ">" ] <symbolName> DemIdent.
OptSymbol = "[" SandwichedSymbol "]".
Aux : Token.
Token = <v> DemString.
AuxList ~ { Aux }.
```

Write a program that prints the list of all rules with a Construct body.
Write a program that prints out all label names.
Example:

```
A = <x> B <y> DemIdent.
B = <x> DemIdent.

The output should look like:

with Construct body = (A B)
labels = (x y x)
```

Your algorithm should be linear time and space in the length of the input grammar.

Exercise 11.8 (Programming for given class dictionary objective)
Consider the following class structure:

```
Tree = "proper" <root> DemNumber <left> TreeOrLeaf <right> TreeOrLeaf.
TreeOrLeaf : Tree | Leaf.
Leaf = "leaf" DemNumber.
```

All instances of class Tree are binary search trees. All numbers that occur in the left subtree are smaller than the root, and all numbers that occur in the right subtree are greater than the root.

Write a method search for class Tree that takes as argument a number, and returns 1 if the number is in the tree and 0 otherwise.

Exercise 11.9 (Programming for given class dictionary objective)
Write a translator for the following language:

```
Statement : ForStatement | PrintStatement.
ForStatement = "for" DemIdent ":=" <lower> DemNumber "to" <upper> DemNumber
  "do" Statement.
PrintStatement = "(print" IdentList ")".
IdentList ~ DemIdent { DemIdent}.
```

The purpose of the translator is to expand the for statements and produce a sequence of lists. They reflect the assignments made by the for statements. The following example should make the semantics of this language clear.

Example: The input

```
for i:=1 to 2 do
  for j :=3 to 4 do (print j i)
```

should output

```
(3 1)
(4 1)
(3 2)
(4 2)
```

Exercise 11.10 (Programming for given class dictionary objective)

Write a semantic checker for the language of the last problem. Verify that every variable that occurs in the print statement is assigned within a for statement.

Example:

```
for i:=1 to 3 do (print x)
```

is illegal.

Exercise 11.11 Consider the grammar

```
Person = "name" <name> DemIdent ["bittenBy" <bittenBy> DogList].
DogList ~ {Dog}
Dog =
  "dogName" <dogName> DemIdent
  <owner> Person.
```

Check the following three inputs for syntactical correctness. For those that are correct, draw the object.

```
name Peter
  bittenBy "dogName" Barry name Jeff
           "dogName" Bless name Linda

name Ana
  bittenby


name bittenby
```

Exercise 11.12 Consider the following class dictionary:

```
S = "a" [S] "b".
```

(Input finding objective) Give three distinct elements belonging to the language defined by this class dictionary.

(Language objective) Give a precise definition of the language defined by this class dictionary. Give a proof that the class dictionary defines exactly the described language.

11.5 BIBLIOGRAPHIC REMARKS

The meal example is from [LR88b].

- Compiler theory:

  The concepts of recursive descent parsing, first sets, follow sets, and the LL(1) conditions are reused from compiler theory. See for example [ASU86].

- Grammar-based programming:

  There are few papers about object-oriented programming using a grammar-based approach. An early paper that goes in this direction is [San82], which describes the Lithe language. In Lithe, class names are used as the nonterminal alphabet of a grammar. For manipulating objects, Lithe does not use message passing, but syntax-directed translation.

  A grammar-based approach to meta programming in Pascal has been introduced in [CI84]. [Fra81] uses grammars for defining data structures. [KMMPN85] introduces 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 [GL85b] is the predecessor of Demeter. The EBNF grammar notation is due to [Wir77].

- Program enhancement: [Bal86] proposes a frame-based object model to simplify program enhancement which has some similarities to the Demeter system.

- Knowledge engineering: Many papers in knowledge engineering propose an approach similar to the one used in the Demeter system. Minsky proposed an object-oriented approach to knowledge representation [Min75].

  The language KL-ONE [BS85] is an object-oriented knowledge representation language based on inheritance. KL-ONE was used in the late seventies. A class is called a *concept* in KL-ONE. Concepts are subdivided into primitive and defined concepts. Primitive concepts can be specified by a rich set of necessary conditions. A *role* belongs to a concept and describes potential relationships between instances of the concept and those of other closely associated concepts (i.e., its properties, parts, etc.). Roles are the KL-ONE equivalent to two-place predicates. The components of a KL-ONE concept are its *superconcepts* and the local internal structure expressed in 2.1 *roles* and 2.2 *constraints*, which express the interrelations among the roles. The roles and the constraints of a concept are taken as a set of restrictions applied to the superconcepts. Superconcepts are thought of as approximate descriptions, whereas the local internal structure expresses essential differences.

  There are several different kinds of roles, of which the *role set* is the most important. A role set captures the commonality among a set of individual role players. Role sets themselves have structure. Each role set has a value restriction (given by a type), and number restrictions to express cardinality information. KL-ONE supports role set restrictions that add constraints on the fillers of a role with respect to some concepts. KL-ONE uses a graphical language and the JARGON [Woo79] language to

specify concepts. JARGON is a stylized, restricted, English-like language for describing objects and relationships. KL-ONE has been further developed in NIKL [KBR86], [Mor84], and KL-TWO [Vil84].

Classes defined by predicates (or generators [Bee87]) allow automatic classification of objects. This shifts an important burden from the user to the system (where it surely belongs), and it is very useful in knowledge acquisition and maintenance.

When classes are defined by predicates, it is necessary to study the complexity of the subsumption problem. The subsumption problem consists of deciding whether one class is a subclass of another class. It is well known that the subsumption problem can easily become intractable (for a summary see [PS88]; for the original article see [BL84]).

Frame-based description languages (including KL-ONE; a recent paper is [PS88]) are related to the Demeter system in the following way. A class dictionary defines a concept language that allows us to define concepts in terms of classes defined in the class dictionary and restrictions expressed in terms of instance variables. Such a concept language defines a subsumption algorithm that computes whether one concept is a subconcept of another.

Sheu [She87] proposes to put a logic-programming knowledge base as an interface between the user and an object-oriented system.

Object-oriented knowledge representation for spatial information is proposed in the paper [MK88].

- Object-oriented design:

  A good overview is given in [Weg87].

- Theory of program data:

  The work of Cartwright promotes a constructive approach to data specification, called domain construction, and is a precursor of our work on class dictionaries [Car84]. The idea of domain construction has its roots in the symbolic view of data pioneered by John McCarthy and embodied in the programming language Lisp. The domain construction approach to data specification views a data domain as a set of symbolic objects and associated operations satisfying the following three constraints:

  - Finite constructibility. Every data object is constructed by composing functions, called constructors.
  - Unique constructibility. No two syntactically distinct objects denote identical elements of the domain universe.
  - Explicit definability. Every operation, excluding a small set of primitive functions serving as building blocks, is explicitly defined by a recursive function definition.

  Cartwright uses subset definition to define noncontext-free types like height-balanced binary trees or nonrepeating sequences of integers. Quotient definitions are used to define types containing objects that are not uniquely constructible, such as finite sets and finite maps.

The Demeter approach also falls into the constructive method of data definition. At the moment we do not support subset and quotient definitions since they are difficult to handle at compile-time.

The constructors in the Demeter system come from construction and repetition classes. Alternation classes don't provide constructors.

11.6 SOLUTIONS

Solution to 11.12
3 inputs:

```
a b
a a b b
a a a b b b
```

This class dictionary defines the language $a^n b^n$. We prove this by induction on n:

Base For $n = 1$ it is true. When the optional symbol is missing, we get ab.

Step Induction hypothesis: Assume that the above class dictionary defines the language $a^n b^n$ for all $n = m-1, n > 0$. We want to show this fact for $n = m$. Consider entering the optional symbol `[S]` one additional time. This adds one a and one b to $a^{m-1}b^{m-1}$ which by the induction hypothesis belongs to the language. Therefore we get that $a^m b^m$ also belongs to the language.

Chapter 12

Style Rules for Class Dictionaries

In this chapter we present several style rules related to the structural organization of classes. Defining the class dictionary for an application is a very important and interesting task. The class dictionary determines all the data structures, which in turn 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 that are easier to manage. This topic of modularization will be discussed elsewhere. In this chapter we have collected a set of useful design techniques for those modular pieces of class dictionaries.

The style rules cover several topics: avoiding bad recursion in class structures, optimization of class structures, parameterization, systematic structuring and naming, functional dependency normalization, and notational issues such as viscosity.

12.1 LAW OF DEMETER FOR CLASSES

The class dictionary graphs of object-oriented applications often contain cycles which means that the class definitions are recursive. The goal of the Law of Demeter for classes is to avoid bad recursions in class structures; that is, recursions which cannot terminate.

If a class dictionary graph does not contain any cycle, we can build complex objects from simple objects inductively. The reason is obvious. We can topologically sort any acyclic directed graph, and the topological order tells us in what order to build the objects. As class dictionary graphs become more and more complex, which means there may be more and more cycles, we can still build objects inductively and incrementally as long as every cycle has a way out of cycles. We call such class dictionaries inductive. Otherwise we have to build finite cyclic objects for any vertex on those cycles. We argue that noninductive class dictionary graphs should be avoided most of the time.

Consider the class dictionary graph in Fig. 12.1a. When we construct a class dictionary graph slice anchored at vertex `Nonempty`, vertex `Nonempty` forces all the outgoing construction and inheritance edges to be included in the slice. Vertex `List` must have the only outgoing alternation edge `List`$\Longrightarrow$ `Nonempty`, because it has an incoming construction

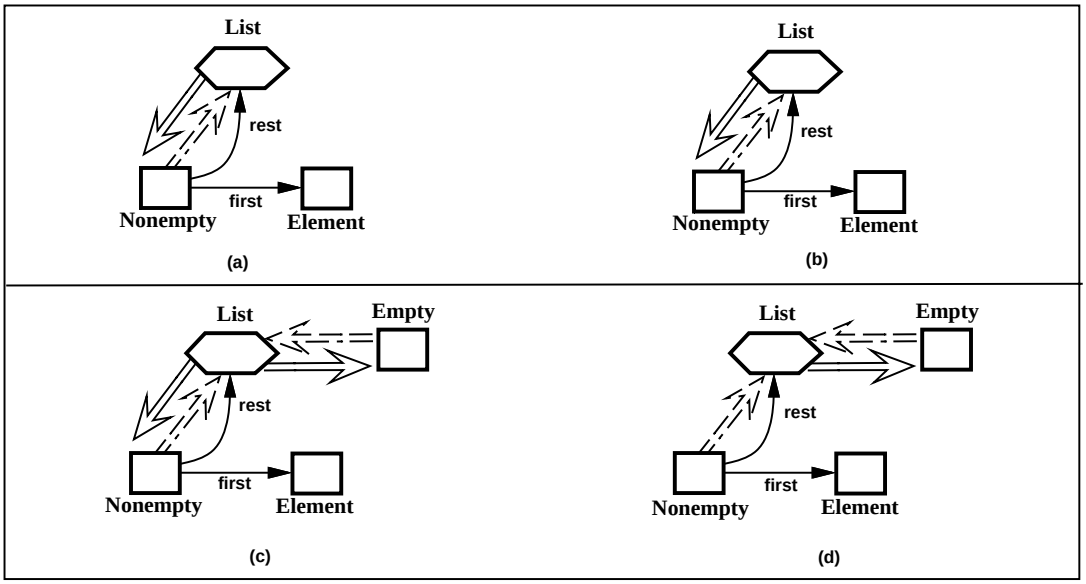

Figure 12.1: Illustration of class dictionary graph slices

edge. Fig. 12.1b shows the only class dictionary graph slice anchored at vertex `Nonempty`.

Consider the class dictionary graph in Fig. 12.1c. In Fig. 12.1d we show one of the class dictionary graph slices anchored at vertex `Nonempty`. The difference from the above case is that we can select alternation edge `List`$\Longrightarrow$ `Empty` instead of taking alternation edge `List`$\Longrightarrow$ `Nonempty`.

In the class dictionary graph of Fig. 12.1a, a Nonempty-object must contain an `Element`-object and a `List`-object. A `List`-object must always be a `Nonempty`-object—an infinite recursion. In Fig. 12.1b, this infinite recursion is expressed by the cycle formed by `Nonempty`$\stackrel{rest}{\longrightarrow}$ `List` and `List`$\Longrightarrow$ `Nonempty`. This cycle is forced to be included.

In the class dictionary graph of Fig. 12.1c, a `Nonempty`-object must contain an `Element`-object and a `List`-object. But a `List`-object can be an `Empty`-object. In this case, we don't have an infinite recursion. We can have a `Nonempty`-object that is a list containing only one element, an `Element`-object. The `Empty`-object is used here for the end of the list.

Comparing the two class dictionary graphs in Fig. 12.1a and 12.1c, we can build only cyclic `Nonempty`-objects from the first class dictionary graph in Fig. 12.1a; but we can build acyclic `Nonempty`-objects of any size based on the `Nonempty`-objects of smaller size for the second class dictionary graph. We call the second class dictionary graph an **inductive** class dictionary graph. The first class dictionary graph is not inductive.

To introduce the Law of Demeter for classes, we reuse reachability concepts and the class dictionary graph slice concept introduced earlier.

A vertex w in a semi-class dictionary graph is said to be **reachable** from a vertex v by a path of length n, if there is a knowledge or an inheritance path of length n from v to w.

A semi-class dictionary graph is **cycle-free** if there is no $v \in V$ such that v is reachable from v by a path of at least length 1.

A semi-class dictionary graph is **inductive** if it satisfies the inductiveness rule. The inductiveness rule is: For all vertices v there exists at least one cycle-free class dictionary graph slice anchored at v.

The purpose of the inductiveness rule is

1. To make each recursion well defined and to guarantee that the inductive definitions of the objects associated with the vertices of the class dictionary graph have a base case. Informally, the rule disallows classes that have only circular objects.

2. To exclude certain useless symbols from the grammar corresponding to a class dictionary graph. There are two kinds of useless symbols: the ones that cannot be reached from the start symbol and the ones that are involved in an infinite recursion. The inductiveness rule excludes useless symbols of the infinite recursion kind.

3. To allow a tool to generate more code for groups of classes that satisfy this rule.

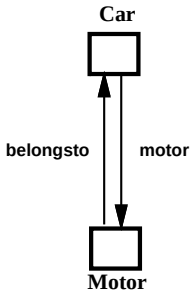

Figure 12.2: Car and motor

Sometimes, people may want to keep their class dictionary graphs noninductive for some purposes, as shown in Fig. 12.2. Every Car-object must have a Motor-object. Every Motor-object must have a Car-object on which it is installed. Therefore we propose an approximation of the inductiveness rule.

The Law of Demeter for Classes is:

Maximize the number of inductive vertices of a class dictionary graph[1].

Maximizing the number of inductive vertices in a class dictionary graph minimizes the complexity of building objects and the software associated with them. Fewer objects are forced to be cyclic. Further motivation for the Law of Demeter for classes includes

[1]The Law of Demeter for classes is different from the Law of Demeter (for functions) in class form discussed in Chapter 8.

- The objects defined by noninductive vertices must all be cyclic. Classes that define only cyclic objects should be used only when absolutely needed. It is harder to reason about them.
- Cyclic objects are harder to manipulate because of the danger of infinite loops.

It is useful to discuss three dimensions of class dictionary design.

- C: number of common parts of abstract classes.
- F: number of vertices that are not inductive.
- L: LL(1) violations. Count the number of different violations of Rule 1 and Rule 2.

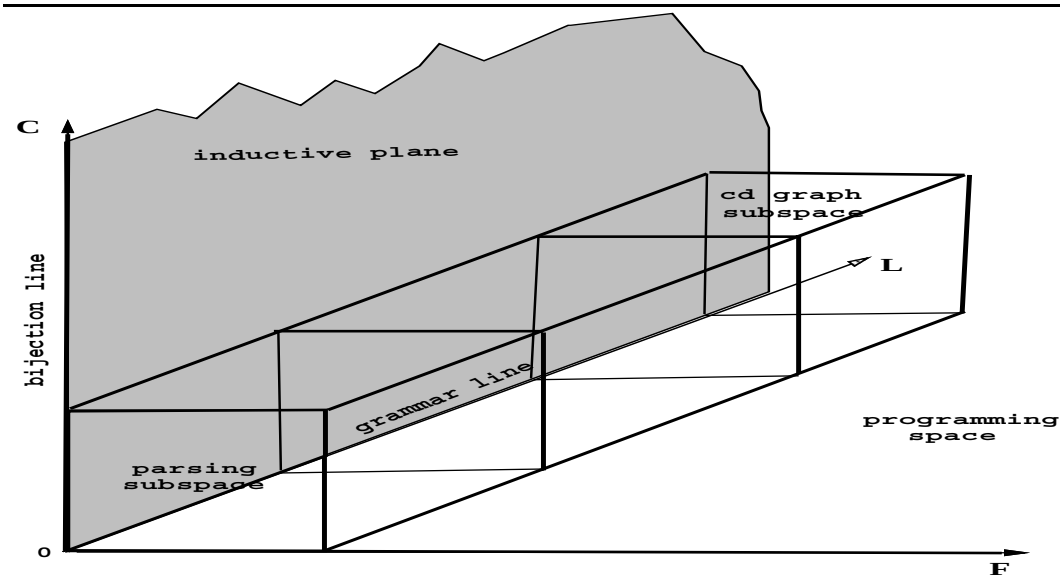

Figure 12.3: Three dimensions of class dictionary design

Figure 12.3 shows the design/programming space in the three dimensions.

- Pure data model subspace, class dictionary graphs (labeled as cd graph subspace in Fig. 12.3)

  Initially, when we develop a class structure, we put it into the class dictionary graph subspace. We will have many LL(1) violations and the class dictionary graph might not be inductive.

- Inductive class dictionaries plane

We improve the class dictionary graph and turn it into a class dictionary graph without noninductive vertices. This brings us into the inductive plane. We also maximize the common parts, which moves us away from the grammar line (traditional grammars don't have common parts).

- Parsing subspace

  We improve the class dictionary graph and turn it into a class dictionary with zero LL(1) violations. This moves us onto the bijection line. For class dictionaries on the bijection line, there is a bijection between sentences and tree objects.

12.2 CLASS DICTIONARY GRAPH OPTIMIZATION

The goal of class dictionary graph optimization is to improve the class organization while keeping the set of objects invariant. This involves "inventing" abstract classes to minimize the total size of the class dictionary graph.[2] Our algorithms are programming-language independent and are useful to programmers who use languages such as C++. Class dictionary graph optimization has applications to design, reverse engineering and optimization of programs.

We formalize the concept that two sets of class definitions define the same set of objects. A class dictionary graph $D1$ is **object-equivalent** to a class dictionary graph $D2$ if

$$Objects(D1) = Objects(D2)$$

The **size** of a class dictionary graph is the number of construction edges plus one quarter the number of alternation edges.

The constant one quarter is arbitrary. All that is important is that this constant is smaller than a half. The reason is that we want the class dictionary in Fig. 12.4a to be smaller than the class dictionary in Fig. 12.4b.

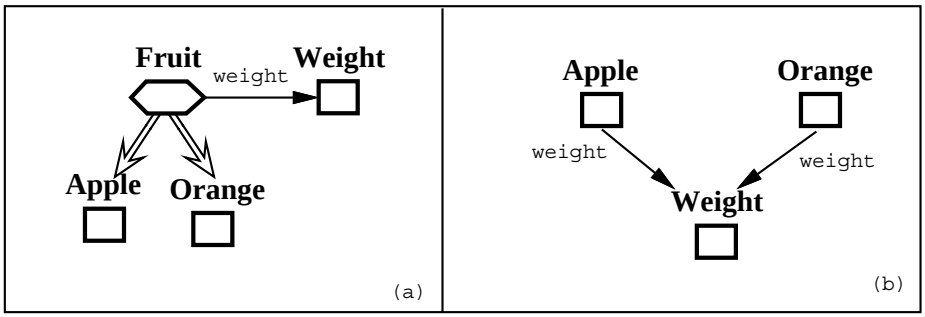

Figure 12.4: a has smaller size than b

[2] Class dictionary graph minimization, page 444 (50).

Anyway, we want alternation edges to be cheaper than construction edges since alternation edges express commonality between classes explicitly. This leads to better software organization through better abstraction and less code duplication.

The class dictionary graph minimization problem is defined as follows. Given a class dictionary graph, find an object-equivalent class dictionary graph of minimal size. Class dictionary graph minimization means more than moving common parts "as high as possible" in the class dictionary graph. It also minimizes the number of alternation edges.

In other words, we propose to minimize the number of edges in a class dictionary graph while keeping the set of objects invariant. Our technique is as good as the input it gets: If the input does not contain the structural key abstractions of the application domain then the optimized hierarchy will not be useful either, following the maxim: garbage in—garbage out.

However if the input uses names consistently to describe a class dictionary graph then our metric is useful in finding good hierarchies. However, we don't intend for our algorithms be used to restructure class hierarchies without human control. We believe that the output of our algorithms makes valuable proposals to the human designer who then makes a final decision.

Our current metric is quite rough: we just minimize the number of edges. We could minimize other criteria, such as the amount of multiple inheritance or the amount of repeated inheritance. A class B has repeated inheritance from class A, if there are two or more edge-disjoint alternation paths from A to B. The study of other metrics is left for future investigations.

12.2.1 Minimizing Construction Edges

Even simple functions cannot be implemented properly if a class dictionary graph does not have a minimal number of construction edges. By properly we mean with resilience to change.

Consider the class dictionary in Fig. 12.5, which is not minimized.

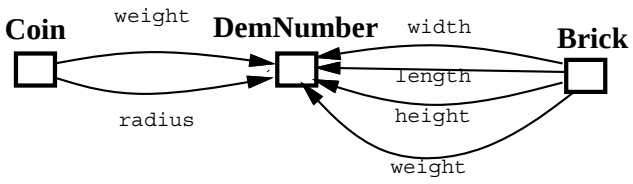

Figure 12.5: Class dictionary to be minimized

Suppose we implement a print function for Coin and Brick. Now assume that several hundred years have passed and that we find ourselves on the moon where the weight has a different composition: a gravity and a mass. We then have to rewrite our print function for both Coin and Brick.

After minimization of the number of construction edges in Fig. 12.5 we get the class dictionary in Fig. 12.6. In this minimized class dictionary we implement the print function

```
Coin = <radius> Number .
Brick = <width> Number <length> Number <height> Number .
Weight_related : Coin | Brick *common* <weight> Number.
```

Figure 12.6: Optimized class dictionary

for Coin with the method:

```
void Coin::print() {
  radius -> print(); Weight_related::print();}
```

The advantage of the optimization is that information about weights is now isolated to one class. If we change information about weights, we have to update only one class. For example, after the change of the weight composition, we get the new class

```
Weight_related : Coin | Brick *common* <mass> Number <gravity> Number.
```

We reimplement the print function for this new class and no change is necessary for classes Brick and Coin.

In summary, if the class dictionary graph has a minimal number of construction edges and the functions are written following the strong Law of Demeter (for functions), the software is more resilient to change. The strong Law of Demeter says that a function f attached to class C should call only functions of the *immediate* part classes of C, of argument classes of f including C, and of classes that are instantiated in f. A disadvantage of construction edge minimization is that it creates multiple inheritance. Therefore, it is not always strictly followed.

12.2.2 Minimizing Alternation Edges

Consider the following nonminimal class dictionary graph.

```
Occupation :
  Undergrad_student | TA | Professor | Adm_assistant
    *common* <ssn> Number.
Student : Undergrad_student | TA *common* <gpa> Real.
Faculty : Professor | TA *common* <course_assigned> Course.
Professor = .
TA = .
Adm_assistant = .
Course = .
Undergrad_student = <major> Area.
Area : Economics | Comp_sci.
```

```
Economics = .
Comp_sci = .
University_employee : TA | Professor | Adm_assistant
                      *common* <salary> Real.
```

Change the class definitions for Occupation and University_employee to

```
Occupation : Student | University_employee *common* <ssn> Number.
University_employee : Faculty | Adm_assistant *common* <salary> Real.
```

We have now reduced the number of alternation edges by three at the expense of adding repeated inheritance. By repeated inheritance we mean that a class is inherited several times in the same class. In the above example, class Occupation is inherited twice in class TA:

```
Occupation -> University_employee -> Faculty -> TA
           -> Student ->                        TA
```

However, not only alternation edges are reduced, but also the amount of multiple inheritance, which we propose as another metric to produce good schemas from the software engineering point of view.

Class dictionary graph minimization consists of two steps.

- Construction edge minimization. This is an easy task: we abstract out the common parts and attach them to an alternation class. If there is no appropriate alternation class, we introduce a new one.
- Alternation edge minimization. Alternation edge minimization is in general a computationally expensive problem (it is known to be NP-hard), but there is a special case, called the tree property[3] case, where there is an efficient algorithm.

To minimize the construction edges, we use the concept of a redundant part. In a first approximation a construction edge with label x and target vertex v is called **redundant** in a class dictionary graph, if there is more than one x-labeled construction edge going into v. This definition of redundant part is adequate for many practical situations. To cover all cases, it needs to be slightly generalized. A construction edge with label x and target vertex v is called **redundant** if there is a second construction edge with label x and target vertex w such that v and w have the same set of associated classes.

A class dictionary graph has a minimal number of construction edges if it does not contain any redundant construction edges.

Alternation edge minimization solves the following problem. Given a class dictionary graph D with a minimal number of construction edges, find a class dictionary graph D_1 such that the total number of alternation edges of D_1 is minimal, and so that D and D_1 are object-equivalent.

Next we consider a special case of the alternation edge minimization problem. This creates an interesting link between single inheritance and a property of class dictionary graphs, called the tree property.

[3] Tree property, page 444 (49).

Definition A class dictionary graph G is called a **single inheritance** class dictionary graph, if for each vertex v in G, v has at most one incoming alternation edge.

A class dictionary graph can become an object-equivalent, single inheritance class dictionary graph if and only if its sets of associated vertices satisfy the tree property. (The set of associated vertices of a vertex is the set of all the concrete subclasses.) The set of associated vertices of a vertex can be regarded as an inheritance cluster. If all inheritance clusters in a class dictionary graph are pairwise disjoint or in a proper subset relationship, then the class dictionary graph can become an object-equivalent single inheritance class dictionary graph. Furthermore, by checking whether the sets of associated vertices of a class dictionary graph satisfy the tree property, we are effectively transforming such a class dictionary graph into a single inheritance class dictionary graph.

Definition A collection of subsets of a set S has the **tree property** if for any pair of subsets of S one element of the pair is completely contained in the other, or if the two subsets are disjoint.

When a collection of subsets has the tree property, the graph having the subsets as vertices and the subset relationships as edges is a tree.

When the tree property is satisfied, it is easy to reorganize the class dictionary graph into a single inheritance class dictionary graph. The set inclusion relationships describe the inheritance structure.

Consider the following example. The class dictionary in Fig. 12.7 satisfies the tree property. The classes associated with ChessPiece are a superset of the classes associated with Officer. Therefore we can transform the class dictionary into the object-equivalent

```
ChessPiece : Queen | King | Rook | Bishop | Knight | Pawn.
Officer : Queen | King | Rook.
```

Figure 12.7: Class dictionary that satisfies tree property

class dictionary in Fig. 12.8, which is single inheritance.

```
ChessPiece : Officer | Bishop | Knight | Pawn.
Officer : Queen | King | Rook.
```

Figure 12.8: Single inheritance class dictionary

12.3 PARAMETERIZATION

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

will have less duplication of functionality. The goal of abstraction is to factor out recurring patterns and to make an instance of the recurring pattern where it is used.

Parameterization uses auxiliary parameterized classes for reinforcing the abstraction mechanism.

Consider the following class dictionary that introduces two classes (Department and Division) by using two parameterized classes (Organization and List). The parameters are used to express the degree of variability of the parameterized class.

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

List(P) ~ {P}.

Division = "Division"
  <org> Organization(Department, Company).
Department = "Department"
  <org> 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 more flexible. 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 following example shows how to define parameterized lists without a repetition class, using the terminology of the Lisp programming language.

```
List(E) : Nil | Cons(E).
Cons(E) = <car> E <cdr> List(E).
Nil = .
```

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

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

is better than (version 2)

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

Both class dictionaries use

```
Instance = Vertex.
LabeledInstance = [<label> Label] Vertex.
```

A sentence for Repetit, version 1 is

```
~ "start" { Family } "end"
```

A sentence for OptionalInstance, version 1 is

```
[ <arg1> Exp ]
```

However, a sentence for Repetit, version 2 is

```
~ "start" { <urban> Family } "end"
```

which is not allowed by version 1.

Although we use abstraction, we cannot precisely formulate the recurring pattern. Therefore the language defined by the second class dictionary is larger. Version 1 is preferred since it defines exactly what we want.

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.

12.4 REGULARITY

Good label names, class names, and parameterized class names significantly 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 lowercase 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.

To facilitate the writing of adaptive programs, it is advisable that terminal classes be buffered by construction classes. Instead of using

```
Order =
  <orderNumber> DemNumber
  <quantity> DemNumber
  <customerNumber> DemNumber
  <price> DemNumber.
```

it is better to use

```
Order =
  <orderNumber> OrderNumber
  <quantity> Quantity
  <customerNumber> CustomerNumber
  <price> Money.
OrderNumber = <v> DemNumber.
Quantity = <v> DemNumber.
CustomerNumber = <v> DemNumber.
Money = <v> DemNumber.
```

This leads to a more regular class structure for which it is easier to write adaptive software.

To summarize this section we propose the following design rule, called **Terminal-Buffer rule**:

> Usually, a terminal class should be used only as the *only* part class of a construction class. The label of the terminal class should be unimportant, for example, it could be always `<v>`. This leads to the desired buffering of terminal classes.

12.4.1 Regular Structures

We use the adjective regular in an informal way. We say that a class dictionary has a regular structure if similar classes are defined similarly. 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.

```
// Part of Modula-2 grammar

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.

```
// Part of 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. Both if-statements and repeat-statements contain statement sequences and this is expressed in the same way for both kinds of statements. In the Pascal class dictionary, however, if-statements and repeat-statements are treated differently. A class, called BeginEnd, is needed, which turns several statements into one. This class is needed in the if-statement through class Statement.

12.5 PREFER ALTERNATION

Alternation classes 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 that needs to be done for an alternation class.

For 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 *common* ".".
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 that processes a clause corresponding to the first definition will contain a conditional statement that tests whether rightSide is non-nil. A program that 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. In this case it was necessary to add the keywords `"fact"` and `"rule"` to the language because of the look-ahead of one symbol requirement.

There are other reasons, besides having shorter programs, for using alternation in a class dictionary:

- 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 that is always nil.
- Ease of adaptive programming.

Consider the following example: `A = B [C] [D].` If C and D are mutually exclusive and exactly one is present, it is better to use `A = B X. X : C | D.`

The object-oriented program for the second version will send a message to the object in instance variable X and the underlying object-oriented system will determine whether we have an instance of C or D. There is no need for an explicit conditional statement to distinguish between the two possible types of X.

However the program for the first version will contain at least one explicit conditional statement.

To compare the class dictionaries further, consider the following programming task: Given a PrologProgram-object, print the list of all the Rule-objects that are contained in the PrologProgram-object.

For the second class dictionary, we can use

```
*operation* void print_rules()
  *traverse*
    *from* PrologProgram *to* Rule
  *wrapper* Rule
    *prefix* (@ cout << this; @)
```

For the first class dictionary, we can use

```
*operation* void print_rules()
  *traverse*
    *from* PrologProgram
```

```
      *via* Clause
    *to LiteralList
  *carry* *in* Clause* cin = (@ this @)
    *along* *from* Clause *to* LiteralList

  *wrapper* LiteralList
    *prefix* (@ cout << cin; @)
```

12.6 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 class dictionaries that describe object-oriented databases. The adopted definitions serve in turn as style rules for class dictionaries. The motivation behind these definitions is to introduce the concept of a normalized class with respect to functional dependencies.

Definition: An instance variable $V1$ of some class C is **functionally dependent** on instance variable $V2$ 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; that is, the intended set of instances.

Consider the class

```
Employee =
  <employeeNumber> DemNumber
  <employeeName> DemString
  <salary> DemNumber
  <projectNumber> DemNumber
  <completionDate> DemString.
```

The key is employeeNumber. Several problems with this class definition are:

- 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> DemNumber
  <employeeName> DemString
  <salary> DemNumber
  <projectNumber> DemNumber.

Project =
  <projectNumber> DemNumber
  <completionDate> DemString.
```

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 that a class is **normalized** if whenever an instance variable is functionally dependent on a set S of instance variables, S contains a key.[4] 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> DemIdent <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.

12.7 COGNITIVE ASPECTS OF NOTATIONS

We want the notations defined by class dictionaries to be easy to read, write and modify. What is important in a notation to make it that way? Here is some advice from cognitive psychology.

- Opportunistic planning (which means to adapt the planning to circumstances without regard to principles): The notation must allow for opportunistic planning rather than require a fixed strategy. It has been repeatedly shown that users prefer opportunistic planning. High-level and low-level decisions are mixed; development in one area is postponed because potential interactions are foreseen; the descriptions are frequently modified.

  However, opportunistic planning can hinder reusability. The use of individual modeling approaches may lead to nontransferable models. A method for system design should provide enough flexibility to allow designers to make full use of their creative resources while guiding them towards uniform descriptions.

- Order independence: The descriptions should be order independent as much as possible. What is needed is to decouple the meaning of the description from the final text order as much as possible.

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

- Viscosity: A viscous notation resists local changes. Correspondingly, a viscous notation contains many dependencies between its parts, so that a small change requires several implied adjustments. The notation should have the right amount of viscosity. Viscous notations cause more work, yet they often have advantages. Their higher redundancy helps to detect certain errors and sweeping accidental changes are less likely. The extra work involved in using viscous notations may encourage users to think about their requirements more carefully.

- Role-expressiveness: The reader of a sentence must discover the role of each component of the sentence. Notations that show their structure clearly are called role-expressive.

  Since the reader of a sentence has to recognize the intentions from the text, the presence of keywords reliably associated with particular intentions is helpful. This implies that each part of a class should be introduced by some keyword. This in turn implies that each alternative of an alternation class should start with a different keyword. Therefore the need for role-expressiveness is a strong motivator to use the LL(1) conditions for class dictionaries. The LL(1) conditions improve readability, a fact that is well known since the early days of Pascal in the late 1960s. Role-expressiveness also implies that keywords should not be overused; each keyword should indicate one intention. A rich set of keywords, however, also has disadvantages. It makes the language less uniform and increases the vocabulary to be learned.

To make a notation easier to use it is often necessary to provide tool support. These tools should keep track of dependencies that are expressed by a sentence, and the tools should make the dependencies easily accessible to the user (for example, by cross-referencing or browsing).

12.8 EXTENDED EXAMPLES

In this section we show some extended examples that have been designed with the techniques explained in this section.

12.8.1 VLSI Architecture Design

The functionality and structure that is put onto a chip is often naturally expressed in parameterized form: n-bit carry-look-ahead adder, n-bit multiplier, n-bit sorter, n-bit bus, n-processor array, etc. It is very natural to define the hardware on a chip in our class dictionary notation and then to express the functionality of the chip as an object-oriented program. The next class dictionary defines the structure of a Batcher sorting network in parameterized form. The structure of a sorting network is simple: The input consists of n numbers that are split into two parts of equal size. Each half is sorted in parallel by a sorting network of half the size. The output of the two half-sized sorting networks is sent through a merging network. The output from the merging network is the the desired sorted sequence. Such recursive structures have many applications. For example, a Batcher odd-even merging network has a similar structure. Therefore we parameterize the structure description and introduce the parameterized classes DivideAndConquerNetwork, Induction, and NonTrivial.

```
Merge = <network> DivideAndConquerNetwork(List(Comparator)).
Sort = <network> DivideAndConquerNetwork(Merge).
DivideAndConquerNetwork(Q) =
   "input" <input> List(DemNumber)
   "output" <output> List(DemNumber)
   "local" <local> Induction(Q).
Induction(Q) : NonTrivial(Q) | Trivial(Q).
NonTrivial(Q) =
  "left" <left> DivideAndConquerNetwork(Q)
  "right" <right> DivideAndConquerNetwork(Q)
  "postProcessing" <postProcessing> Q.
Trivial(Q) = .
List(S) ~ {S}.
Comparator = "c".
```

It is interesting that at this level of abstraction merging and sorting are almost identical. The only difference is that the sorting network uses a merger for post processing and the merging network uses a list of comparators. This class dictionary can be used in several ways for simulating, for example, sorting networks.

The parameterized class DivideAndConquerNetwork will be useful for many other applications.

An example sentence for a Merge-object is

```
input 1 2 3 4
output 5 6 7 8
local
  left
    input 9 10
    output 11 12
    local
  right
    input 13 14
    output 15 16
    local
  postProcessing
    c c
```

Indentation is used to show the recursive structure of the network.

In the next example we define the structure of a Newton-Raphson pipeline. The parameterized classes are: ProcessorArray and List.

```
NR = <array> ProcessorArray(NewtonRaphsonElement).
ProcessorArray(Processor) =
  "input" <input> Ports
  "local" <processors> List(Processor)
  "output" <output> Ports.
```

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

Register = "Register"
  "input" <i> DemReal
  "local" <store> DemReal
  "output" <o> DemReal.

NewtonRaphsonElement = "NewtonRaphsonElement"
  "input" <input> Ports
  "local"
    <argumentSave> Register
    <estimateSave> Register
  "output" <output> Ports.
Ports = <argument> DemReal <estimate> DemReal.
```

The parameterized class ProcessorArray will have many more applications than just defining a Newton Raphson pipeline.

12.8.2 Business Applications

We describe the example from [TYF86] in our class dictionary notation. The constraints are not formulated in the class dictionary. Instead, they are formulated as part of the object-oriented program that works on the data.

```
Company = <divisions> List(Division).
Organization(SubOrganization, SuperOrganization) =
  <contains> List(SubOrganization)
  [<partOf> SuperOrganization]
  <managedBy> Employee.
List(P) ~ {P}.

Division = "Division"
  <org> Organization(Department, Company).
Department = "Department"
  <org> Organization(Employee, Division).
Employee : Manager | Engineer |
        Technician | Secretary
  *common*
    [<belongsTo> Department]
    [<manages> Department]
    [<heads> Division]
    [<marriedTo> Employee]
    <skills> List(Skill)
    <assignedTo> List(Project).
Project =
  <requiredSkills> List(Skill)
```

```
  <location> Location.
Manager = "Manager".
Engineer = "Engineer"
  <hasAllocated> PC
  <belongsToProfAssoc> List(ProfAssoc).
Technician = "Technician".
Secretary = "Secretary".
Skill = "skill".
PC = "pc".
ProfAssoc = "assoc".
Location = "location".
```

Next we describe a class dictionary for an other company. The classes Order, Customer, and Product are normalized. This example shows the buffering of terminal classes.

```
Company =
  "orders" <orders> List(Order)
  "customers" <customers> List(Customer)
  "products" <products> List(Product).

Order = "Order" <orderNumber> OrderNumber
        <orderDate> Date
        <customer> Customer
        <quantityOrdered> DemNumber
        <product> Product.

Customer = "Customer" <customerNumber> CustomerNumber
           <customerName> Name
           <customerAddress> Address.

Product = "Product"
  <productNumber> ProductNumber
  <productName> Name
  <productPrice> Money.
Address = .
OrderNumber = DemNumber.
CustomerNumber = DemNumber.
ProductNumber = DemNumber.
Name = DemString.
Money = DemNumber.
Date = .
List(P) ~ {P}.
```

12.9 SUMMARY

This chapter used to play an important role in the Demeter Method. With the advent of adaptive software, the role of the chapter has diminished somewhat. The rules described

here are still useful since a clean class dictionary is important.

12.10 EXERCISES

Exercise 12.1 What is the relationship between a noninductive vertex and a useless vertex? A vertex is useless, if it cannot be instantiated in a finite, noncyclic object.

12.11 BIBLIOGRAPHIC REMARKS

- Database design:

  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.

  Normalization of relational databases is explained in [Ull82] and [Sal86]. For interesting relationships between relational database design and object-oriented database design see [Kor86].

  Types and subtypes are discussed in [HO87].

- Complexity:

  Whether the language equivalence problem for deterministic context-free grammars is decidable or not is an open problem. Class dictionaries not using recursion define regular expressions of a restricted form (LL(1) restrictions). The equivalence problem for general regular expressions is NP-hard [GJ79].

- Transformations:

  The term "promotion of structure" is from [SB86].

- Predecessor:

  Since 1984 we have designed or participated in the design of numerous class dictionaries of various sizes, ranging from a couple of lines to a few hundred lines. Some of these class dictionaries were written for the predecessor of Demeter: GEM [GL85b]. The class dictionaries were used for applications such as silicon compilation for Zeus [GL85a], translation between intermediate forms for automatic test generation, translation of algebraic specifications into Prolog, programming language implementation, etc.

- Cognitive aspects:

  [Gre89] describes cognitive dimensions of notations.

Chapter 13

Case Study: A Class Structure Comparison Tool

In this chapter we go through the process of developing a simple programming tool for comparing class dictionaries. We use the Demeter Method for adaptive software development that we developed piece by piece in earlier chapters. The Demeter Method allows you to develop adaptive software, which is highly generic software that needs to be instantiated. The Demeter Method is a two-phase software development method. In phase one the adaptive software is developed and in phase two the adaptive software is instantiated by customizers. The phases are used iteratively.

Adaptive software consists of three parts:

- **succinct constraints** C on customizers
- **initial behavior specifications** expressed in terms of C
- **behavior enhancements** expressed in terms of C

The succinct constraints express the set of permissible customizers. A key ingredient to adaptiveness is that the constraints are succinct; that is, they are expressed in terms of partial knowledge about a larger structure. The initial behavior specifications express simple behavior. The behavior enhancements express in terms of the constraints, how the simple behavior is enhanced to get the desired behavior.

The constraints are graph constraints that are expressed, for example, in terms of edge patterns and propagation directives. The initial behavior specifications are propagation patterns, possibly with transportation directives, but without the wrappers. They define traversals and transportations. The enhancements are the vertex and edge wrappers.

13.1 THE DEMETER METHOD

We first give a summary of the method.

13.1.1 The Demeter Method in a Nutshell

The following artifacts are derived from the use cases.

- Derive a class dictionary.

  Start with requirements, written in the form of use cases and a high-level structural object model that describes the structure of application objects. The structural object model provides the vocabulary for expressing the use cases. A use case describes a typical use of the software to be built. From the high-level structural object model we derive a class dictionary to describe the structure of objects. The class dictionary has secondary importance since, after the project is complete, it is replaceable by many other class dictionaries without requiring changes or only a few changes to the rest of the software.

- Derive traversal and transportation patterns without wrappers.

  For each use case, focus on subgraphs of collaborating classes that implement the use case. Focus on how the collaborating classes cluster objects together. Express the clustering in terms of transportation patterns. Express the collaborations as propagation patterns that have minimal dependency on the class dictionary. The propagation patterns give an implicit specification of the group of collaborating classes, focusing on the classes and relationships that are really important for the current use case.

- Derive the wrappers.

  Enhance the propagation patterns by adding specific functionality through wrappers at vertices or at edges of the class dictionary. The wrappers use the object clusters. Derive test inputs from use cases and use them to test the system.

Next we describe the steps taken during adaptive software development and maintenance in more detail.

13.1.2 Design Checklist

We give a summary of the software engineering process for adaptive software. When applying the Demeter Method, the following activities are performed iteratively.

1. Develop/maintain use cases (and the high-level object structure)

   Use cases are used throughout adaptive software development and maintenance. Use cases are often described in English. Sometimes a class dictionary is developed to define a use case notation and a tool is used to test the software after development by driving it with use cases written in the use case notation. In other words, the use cases serve as test scripts.

   Use cases are used to develop and test class dictionaries and propagation patterns.

   Organize use cases into a list where the easy use cases are first and the most complex uses cases are last. Find relationships among the use cases such as when one use case calls another use case or when one use case is a refinement of another use case. The list of use cases you produce should contain a small set of functionally simplified use cases