NO.75
(1988)

BUILDING RESEARCH ASSOCIATION OF NEW ZEALAND

REPRINT

Cl/SfB (Agn)
UDC 681.3.06

The Evolution of Class Language

J.Hamer, J.G.Hosking and W.B.Mugridge

This report originated in work commissioned by BRANZ as part of its ongoing research into knowledge-based systems. The views represented are not necessarily those of the Association

Reprinted from ***Proceedings of the Third New Zealand Conference on Expert Systems,*** Wellington, 11-13 May,1988.

BUILDING RESEARCH ASSOCIATION OF NEW ZEALAND

The Evolution of Class Language

J. Hamer, J.G. Hosking, W.B. Mugridge
Department of Computer Science
University of Auckland

Abstract

Class Language is a programming language for developing expert systems. The language has evolved through the application of knowledge abstraction principles to meet the needs of a succession of applications. In this paper, the stages in the evolution of the language are described, with specific reference to the applications that focused its direction.

1. Introduction

Class Language has been developed as part of our collaborative research program with the Building Research Association of New Zealand (BRANZ). The language has been used to implement a wide range of expert system applications, including FireCode [Buis et. al., 1987], Seismic [Lomas, 1988], DAMP [Trethowen, 1987], Adhesive [Fromont and Watkinson, 1988], and WallBrace [Mugridge and Hosking, 1988]. A chronology of significant events involving Class Language is given below.

Early 1985	First Martin Language system
July 1985	First Class Language system developed Concurrent development of FireCode
Early 1986	Class Language version 1 (interpreted) FireCode installed at BRANZ
1986	BRANZ re-implement DAMP in Class Language BRANZ develop Adhesive and Sealant
September 1986	DAMP reaches its present form
April 1987	Class Language version 2 (compiler) BRANZ convert Adhesive and Sealant to Class, v2
Early 1987	WallBrace started
Mid 1987	Seismic started
Late 1987	Class Language version 2.6 BRANZ start on SubfloorBrace

Many of the major applications to which Class Language has been applied have centered around regulations and codes of practice. This focus has shaped the broad development of Class Language in a number of ways. For example, Class Language does not support any form of probabilistic reasoning since no application has yet required such a form

of control. However, representation of complex structures has been a major issue in many applications, and accordingly this has become a major issue in the language itself.

2. The Precursor: Martin Language

2.1. Introduction

The first version of Class Language was an extension of a simple backward chaining inference engine known as the Martin Language [Buis, 1986].

The Martin Language supports a flat data representation consisting of simple (integer, boolean, etc.) typed properties. Martin Language programs consist of a set of "if-then" rules for each property, with each rule defining a value for the property that can be used when the "if" condition of the rule is valid. As a backward chaining language, the value of a property is sought only when the property is first referred to from another rule. The value found is recorded for use whenever the property is referred to subsequently, thus ensuring that the same value is used for the property where ever it appears.

2.2. How it Works

The way in which Martin Language works is illustrated with the following example.

```
if    financial_situation = wealthy
and   personal_image = glamorous
then  car_to_buy := porsche.

if    financial_situation > limited
and   transportation_needs = shopping
then  car_to_buy := hatch_back.

if    financial_situation >= limited
and   transportation_needs = commutor
then  car_to_buy := mini.

if    true
then  car_to_buy := no_suitable_car.
```

The purpose of this set of rules is to determine the value of the property *car_to_buy*, which can be a *porche*, a *mini*, a *hatch back*, or perhaps *no suitable car*. The value is influenced by three properties relating to the prospective purchasor: their *financial situation*, *personal image*, and *transportation needs*.

To find a value for *car_to_buy*, Martin Language interprets the rules sequentially until a rule with a valid "if" condition is found. Thus it will always start by determining the value of *financial_situation* and checking to see if the person is *wealthy*. If this is indeed the case, the value of *personal_image* is determined to see if

the person is *glamorous*. However, if the person is not *wealthy*, their *personal image* is left undetermined. Similarly, if the *financial situation* of the person does not even reach the *limited* category, a conclusion of *no suitable car* will be reached without enquiring as to their *transportation needs*.

2.3. Limitations

The Martin Language is, and was always intended to be, a toy language. Its most important role is as a teaching language, and it has been used extensively in introductory AI courses. Most of the limitations of the language have therefore been self-imposed in the interests of clarity.

The principle weakness of Martin Language is that it is ill-suited to the construction of large programs. The language does not support any structure larger than a rule, and relies entirely on programming conventions to enforce order. The rules themselves are particularly weak; for example, there is no means of writing rules that apply to a range of similar properties. The control structure is inflexible; for example, the order in which rules are evaluated cannot be altered.

2.4. Principles carried over to Class Language

Although the Martin Language lacks sophistication, it does exhibit a distinctive character. In particular, rules in the language can be read without regard to the state of the program at the time they are invoked. It simply doesn't matter what happens before the rules for determining a property are invoked - the effect of the code is the same. The result is a natural, declarative style of programming. Implicit in the backward chaining control is the use of lazy evaluation. Martin Language side-steps issues of ensuring the relevance of questions by only asking questions neccesary for the immediate computation. These characteristics, along with the strong data typing, were deliberately carried over into Class Language.

2.5. Summary

Martin Language vividly illustrates both the strengths and weaknesses of using simple backward chaining reasoning. The reasoning process is simple to understand and natural to use when small numbers of objects are involved, but for larger applications additional representation and control facilities are necessary.

3. FireCode

FireCode was our first significant expert system application. It motivated much of the early development of Class Language and has since continued to provide a test bed for ideas. An overview of the application is provided here. For a more thorough account of the system, see [Buis, 1986] or [Buis et. al., 1987].

3.1. *Introduction to the Fire Code*

The draft fire safety code DZ4226, on which FireCode is based, aims to: restrict the design of buildings so as to provide adequate means of escape for the occupants of a building; stop the spread of fire to other buildings; restrict fires to controllable areas by limiting floor areas; to avoid premature building collapse by specifying certain structural standards.

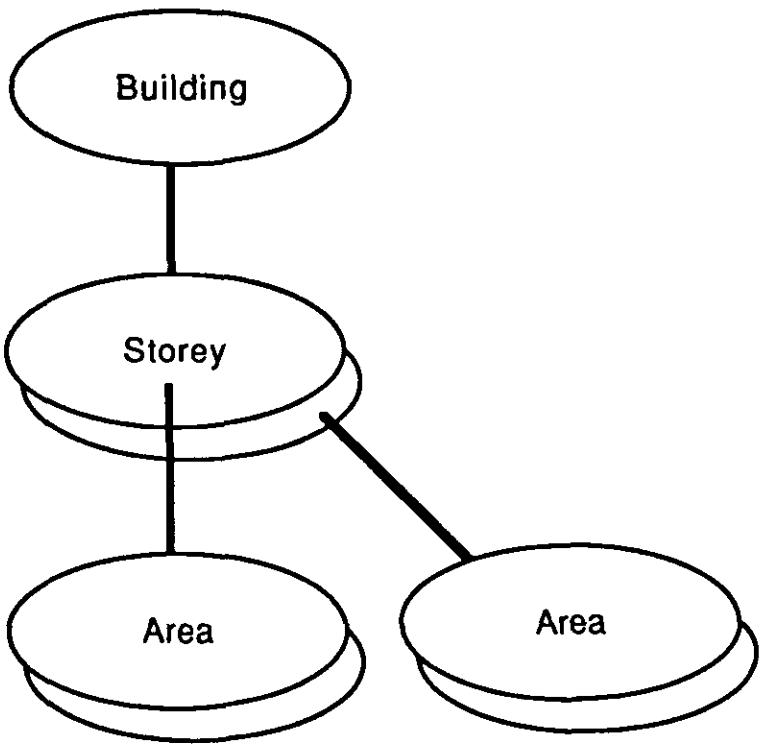
The application of DZ4226 to a building plan involves several interacting activities:

- o determining the building structure;
- o classifying the components according to various criteria, such as the *use* of the areas;
- o checking that clauses corresponding to the classifications made are satisfied;
- o modifying the plan on the basis of any errors found.

3.2 *Structures*

In order to determine the fire safety criteria, a building is divided into a number of structural components: a *fire compartment*, a number of *storeys*, and the distinct *areas* within a storey. Each of these components plays a different role in the event of a fire. The *fire compartment* is the technical term for the building as a whole; it is expected to contain a severe fire by being structurally and/or physically isolated from surrounding buildings. The *storeys* within the building need to isolate a fire, at least until the occupants have had time to escape. An *area* is a section of a storey that presents a characteristic risk situation.

These structural components form a hierarchy, which is pictured below:

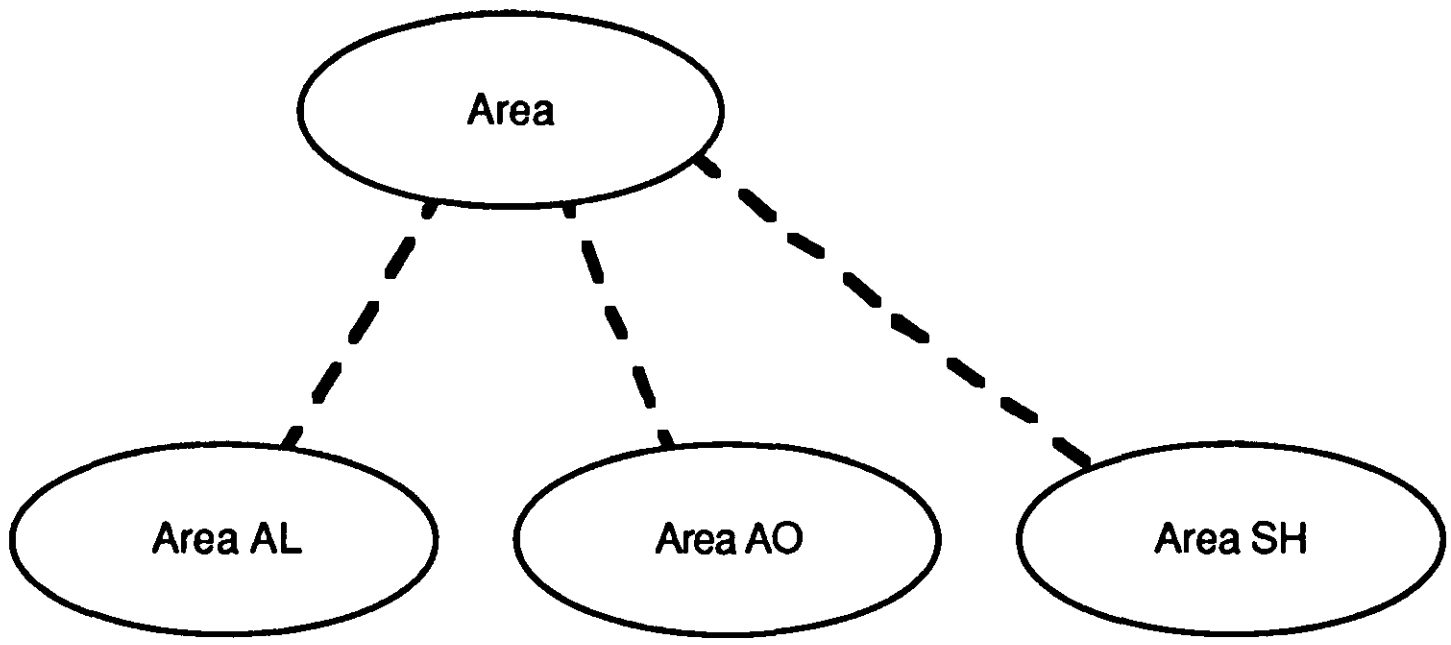


3.3 Categories

Categories are used extensively in DZ4226. For example, each area in each storey of the building is identified as having a *use group* and a *use class*. The *use group* broadly classifies the area as one of the following:

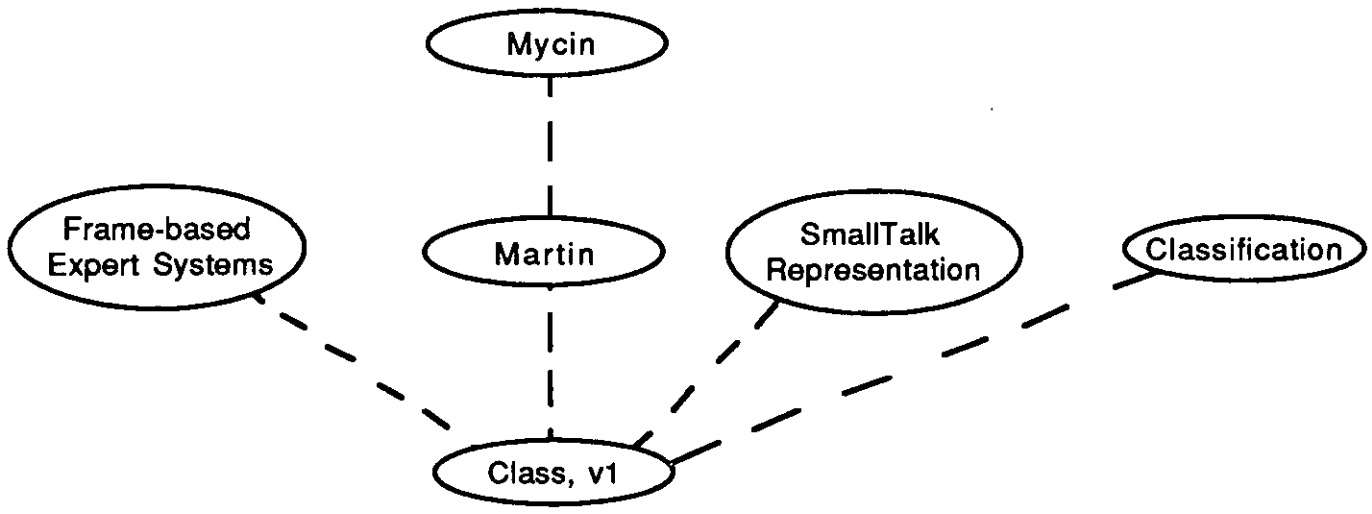
- o assembly area, such as enclosed rooms or suites, or open air seating
- o sleeping area, such as private dwellings, hospitals, prisons, etc.
- o commercial area, such as offices and display areas
- o manufacturing area
- o general area

This broad classification is further categorised into a *use class*. For example, an assembly area used for open air seating will (usually) have a use class of **AO**; a sleeping area used for domestic accomodation has use class **SH**. These categories of *area* are pictured below:



4. Class Language, Version 1

Class Language version 1 retained the flavour of Martin Language, while including ideas on representation and classification. Some of these influences are shown in the sketch below:



4.1. Classes

The concept of a "class" was partially derived from SmallTalk [Goldberg and Robson, 1983], with ideas from a variety of other sources influencing its immediate development, eg: [Smith and Smith, 1977]. A "class" collects together a group of related properties and their associated rules and gives the whole thing a name. The name is important, because one of the reasons for using a "class" is to avoid having to hold the names of all the component properties in your head when you think about it. For the same reason, rules and any other information about the properties are contained within the class. For flexibility, a "class" is treated as a description of an object rather than as the object itself. This means that several objects can share the description of a single class.

The relationship between the class and its components is called *aggregation*. Throughout this paper, aggregation is pictured as a solid line connecting two classes.

4.2. Generalisation and Inheritance

"Classes" gave Class Language the ability to organise quite complicated structures in a natural way. This representational ability was soon extended by allowing classes to be connected in "generalisation" relationships. This relationship is used in languages like SmallTalk [Goldberg and Robson, 1983] for two purposes. First, it allows information common to several classes to be represented once, and thus eliminates some forms of redundancy that can otherwise occur. Second, generalisation relationships can be used to represent control information by making use of the (partial) ordering imposed by the relationship. This was used by SmallTalk to great effect with its inheritance of "methods", and the concept was directly adapted to the inheritance of "rules" in Class Language.

Throughout this paper, generalisations have been pictured as a dotted line connecting two classes.

4.3. Classification

The use of class generalisations to implicitly represent control information was further extended when the concept of "classification" was introduced. At the time, this concept had not been clearly identified in the literature, although it has appeared in related forms in the KL-ONE language [Brachman and Schmolze, 1985] and in the KEE system [Fikes and Kehler, 1985].

Classification is a control mechanism that acts in the opposite direction to rule inheritance, allowing an object to adopt the appearance of a more specialised class during execution. The conditions under which classification can occur are, naturally enough, specified using a form of "if-then" rule. These rules are invoked immediately before determining a property value if there is a possibility of further rules for the property being included should the classification succeed. An example of a classification rule is given below:

```
class area.
...
if      use_group = 'M' and
        fireRating = extraLow
then classify as areaML.
```

Any property that has a rule in the *areaML* class will cause this classification to be invoked before it is evaluated. If the classification rule succeeds, the rules in the *areaML* class will be examined before any rules in the *area* class. The rules in the *areaML* class are thus able to assume that the *use group* is *ML* and that the *fireRating* is *extraLow*.

4.4. Comparison with Frames

An alternative structuring mechanism to the "class" is the "frame" [Fikes and Kehler, 1985; Brachman and Schmolze, 1985]. "Frames" and "classes" are quite similar, and are often used interchangeably in the literature. This is regretable, since the differences that do exist between them are important ones.

The most important difference from our point of view is that a "frame" may be controlled by rules from outside the frame. This is illustrated in the following rule, which appears in a frame-based system example [IntelliCorp, 1984]

```
if    SOME_PLANT is in class plants
and   a reactor of SOME_PLANT is SOME_REACTOR
and   the ph_state of SOME_REACTOR is low
and   a feed_stock_input of SOME_REACTOR is SOME_BIN
and   the constituent of SOME_BIN is alkali
then  the new_constituent_bin of SOME_PLANT is SOME_BIN.
```

This rule is making a statement about a *plant* (viz. the *new_constituent_bin* is given a value), but the rule exists outside the *plants* class (hence the need to screen this condition explicitly).

In a "class", any rules or procedures that effect its behaviour are always included within the class itself. For this reason, "frames" tend to be more flexible structures to work with, but at the cost of weakening the abstractions provided by classes in the first place. In the rule above, it is not at all clear whether *SOME_REACTOR* is directly related to the *plant* class, or an independent relation that is only used to find the required *feed_stock_input*.

4.5. Example: Classes in FireCode

The structural hierarchy of the fire code can be easily represented in Class Language using three main classes:

- a *fire compartment* class, that implements the regulations for the fire compartment as a whole and contains the set of storeys for the building;
- a *storey* class, that implements the regulations concerning an individual storey and contains the set of areas for the storey;
- an *area* class, that implements the regulations concerning a particular area.

The variable number of *storeys* and *areas* required the use of an additional Class Language construct called the **multiple**. The **multiple** only existed in Class Language version 1, having since been superceded by a more general mechanism. Briefly, the **multiple** was used to declare a set of objects. The elements of the **multiple** could not be referred to individually. Special operators were provided for performing tasks such as testing if a condition held for every/some element, etc.

An outline of the Class Language structure is shown below (the syntax has been simplified):

```
class compartment.
      public
            the_storeys : multiple(n_storey) storey.
            n_storey : integer.          The number of storeys in this building

class storey.
      public
            the_areas : multiple(n_area) area.
            n_area : integer.            The number of areas in this storey

class area.
      public
            use_group : [assembly, sleeping, commercial, manufacturing,
                                                      general].
```

Many of the fire code regulations are concerned with the *area* class This made it necessary to further refine the structure of this class to prevent it becoming too complex.

The regulations that apply to a particular *area* are broadly determined by the *use class.* If the *area* class was left as a single class, most of the rules would need a screening clause on the "use_class" property. Such clauses place extra overhead on the evaluation of rule, and make the code difficult to read. Furthermore, many properties of the *area* class are only meaningful when the *use class* has a particular value. For example, *seating_type* is not relevant unless the area is an assembly area of some sort. This is illustrated in the following rule:

```
if    area_class = AS
and   seating_type = bleacher
then  path_width := occupantLoad / 312.5.
```

A better structure for the *area* class is to divide it into a number of separate classes, where each of these new classes handles a particular *use class*. For example, a new class called *areaAS* can contain all the rules that depend on the condition "area_class = AS"; these rules no longer need the screening clause since this is made into a classification rule. The appearance of the *area* class under this structure is illustrated below:

```
class area.

    Classification rule giving the condition under which this area is to assume
    the appearance of the 'areaAS' class

    if area_class = AS then
    classify as areaAS.


class areaAS.

    Properties that are not relevant elsewhere can be declared locally

    public
        seating_type : ...

    Rules do not need to test 'area_class=AS' as this is an implicit condition
    imposed on this class

    if    seating_type = bleacher
    then  path_width := occupantLoad / 312.5.
```

5. Early Evolution

5.1. Parameters

A problem arose with FireCode when it became apparent that each *area* class needed to know the compartment height, number of exitways, and other data belonging to the *storey* class. Although the *storey* class is able to refer to properties of the *area* class, the reverse is not true.

The simplest solution would have been to allow the references to be made both ways, but this solution restricts the ways in which classes could be combined. For example, it would not be possible to consider the *area* class in isolation, as was sometimes desirable.

A better solution to the reference problem was found in **parameters**. Instead of referring to the properties of the *storey* class directly, the *area* class refers to a local **parameter** property that is evaluated in the context of the *storey* class. This is the standard solution used in most traditional programming languages. The only significant difference is that in Class Language the **parameter** is subject to the same lazy evaluation as any other property.

Parameters make it very easy to "plug" a class in anywhere, as the class makes no assumptions about its context. This is extremely useful in testing parts of an application independently, and furthermore simplifies changing the organisation of classes.

5.2. Procedures

In version 1, Class Language used a predicate called **display** for producing output. This predicate could appear anywhere within a rule; it always evaluated to true, and had the effect of writing the value of an expression to the screen. This predicate allowed considerable flexibility in producing output, since there were no restrictions on where it could appear. It also had some appeal in so far as it worked within the existing mechanisms of the language. However, when the amount of written output required by FireCode began to grow, the appeal of **display** rapidly diminished. Many rules were being added to the system whose sole purpose was to cause a sequence of **displays** to be evaluated. As well as being clumsy, "display" rules did not sit easily with our conception of a rule as an inference mechanism. We reasoned that an inference mechanism should really contribute something to the problem solving process, and not just direct a sequence of actions.

Where once there was just a "rule", we had now identified two distinct concepts. "Rules" were to be responsible for inference only: the **display** predicate was removed. A new "procedure" mechanism was formulated which had as its purpose the generation of output and directing sequences of actions, but with no inference capability. This new "procedure" mechanism provided the familiar **display** predicate as its basic operation, together with some simple flow of control constructs. Procedures were intentionally limited in power; in particular, no assignment operation was included.

This created a nice partition between rules and procedures, with rules being solely concerned with determining the values of properties, and procedures being responsible for producing output and controlling the overall flow of a consultation.

For example, the introductory preamble to FireCode was originally coded in the following style:

```
if
        display('Welcome to the FireCode Expert System')
and     help_if_necessary
then    welcomed := true.                    Welcomed is always true; the condition
                                             cannot fail
```

The intent of this code becomes much clearer when coded as procedures, as shown below:

```
procedure welcome.
        display('Welcome to the FireCode Expert System').
        if ask('Would you like some help on how to run the system? ') then
                display('...').
        endif.
end welcome.
```

6. Interlude: Class in Class

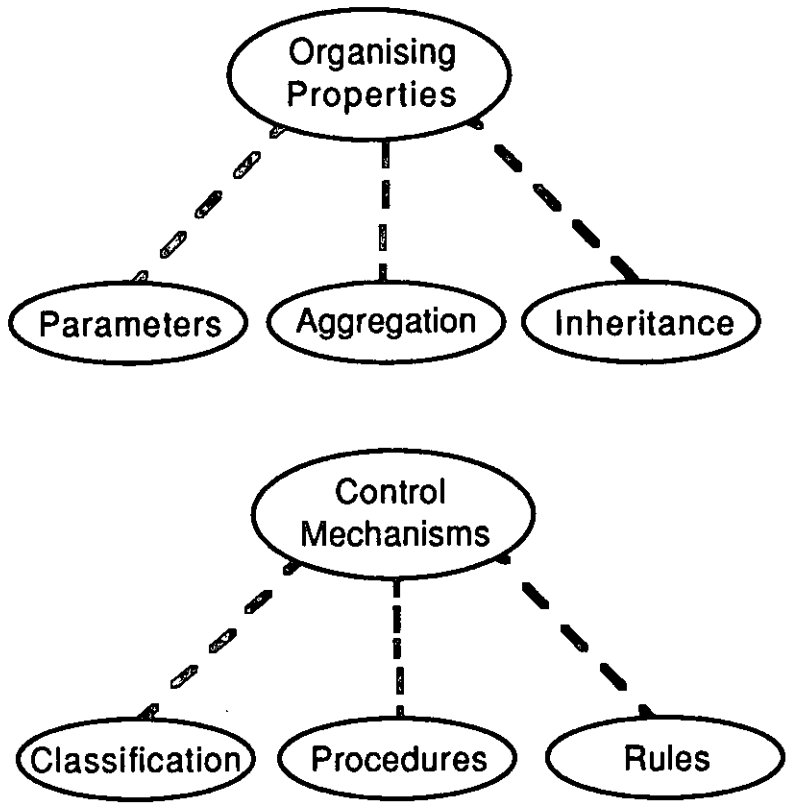
6.1. Reflecting on the Design Process

At about this stage in the design of the language it was discovered that many of the mechanisms that had been introduced could be described using concepts from Class Language itself. For example, procedures can be thought of as a specialised class of control mechanism, derived from the more general rule-based control mechanism class that existed before them. The separation of this general class of control mechanism into two classes could be justified by the existence of a 'classification' rule that effectively partitioned the concepts. An informal classification rule can select which control mechanism - rules or procedures - to use: it is a simple question based on the need for sequencing and output.

By applying Class Language concepts to the language design process we gained a convenient, clearly defined terminology with which to describe the components of the language and the situations in which their use was appropriate. Ultimately, this may be used as the basis for an expert system for aiding an applications programmer, but even without such a formal description the design process benefits in having an impartial model of how the language fits together. This model could be used to "filter" new language features - if it was difficult to adjust the model to include the new feature, the new feature (and the model) would be closely scrutinised. In some cases the model itself suggested new features by directing our attention to areas in which it appeared deficient.

6.2. An Initial Model

An initial model of the Class Language identified two class hierarchies: one concerned with organising properties, and the other concerned with control. The first of these hierarchies contained three specialisations: aggregation, parameters, and generalisation. The latter class contained specialisations for procedures, ordinary rules, and classification rules. This model is shown below:



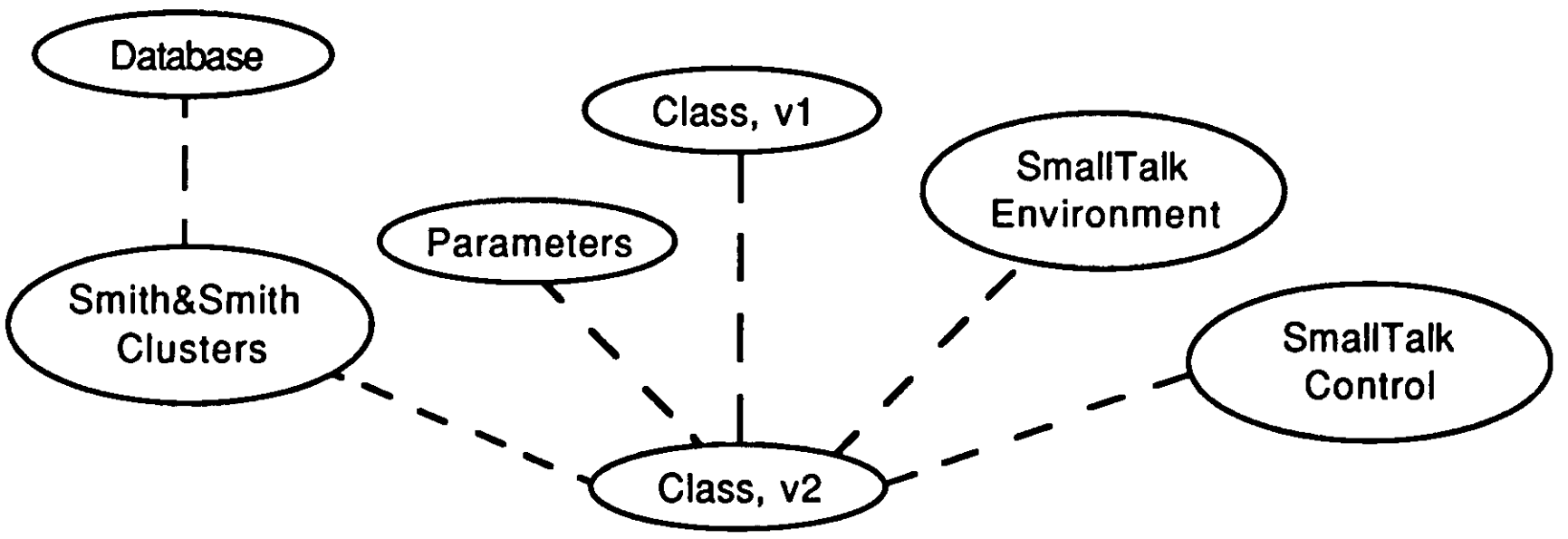
6.3. Using the Model

As we noted earlier, the model should be able to suggest "classification rules" that characterise the differences between the alternative specialisations. Intuitively this is the case with most of the model, since we do not have any problems in practice in distinguishing between rules and procedures, or between any of the property representations. However, the model does have problems in expressing the conditions under which "classification rules" are appropriate to use. This anomoly suggests that either the model or the language, or indeed both, could be improved to clarify the status of classification rules.

While the decision process for choosing between procedures and rules is automatic, the choice between using backward chaining rules and classification rules depends on many factors. A classification rule can be used to factor out a common rule precondition from a set of rules. The reasons for doing this include improving performance or, more importantly, identifying and organising groups of related rules. Often, classification rules are introduced for *representational* reasons: a class that contains a group of properties that only "exist" under certain conditions is a signal for creating a specialised class that is connected to the original class through a classification rule. In this latter case, classification rules are playing an important representational role.

7. Class, version 2

Version 2 of Class Language generalised the classification mechanism of version 1 and added a number of more specialised control structures. The sketch below shows some of these influences:



7.1. Clusters

The dual role played by classification rules was clearly identified by the language model, and was subsequently made explicit with the introduction of "classification properties". The inspiration for this extension came from a classic paper on data base abstractions [Smith and Smith, 1977], in which the specialisations of a class were not restricted to a *single* mutually exclusive group. Instead, the classes of Smith and Smith's conception were able to define any number of independent groups of sub-classes called "clusters". Each cluster is described by a property, the name of the property identifying the particular feature that the cluster represented. The value of a "cluster property" is the name of the class to which the associated cluster has been classified.

Although clusters were not present in the language at the time FireCode was built, the following example illustrates how they could have been used in the *area* class. The *area* class was organised as a general class that could be classified to one of a number of *use classifications*. An alternative organisation would have been to classify according to whether the area was a dead end (having only one exitway) or not, since a significant proportion of the fire code regulations are affected by this issue. Without clustering, only one of these two independent categories could be used as the basis for classification. Clustering allows them both to be used. The Class Language code for the *area* class using clusters is illustrated below:

```
class area.
classification
        escape_access : [ deadEndArea, multipleExitArea ].
        use_group : [ assemblyArea, sleepingArea, commercialArea,
                manufacturingArea, generalArea ].
```

With this code, *area* is classified according to its *use group*, as usual, but may also be independently classified as a *dead end*. The seven classes - *deadEndArea*, *multipleExitArea*, *assemblyArea*, *sleepingArea*, *commercialArea*, *manufacturingArea*, and *generalArea* - therefore provide 14 classes through their combination.

The classification mechanism supported by Class Language version 1 was clearly a degenerate form of the Smith and Smith concept, in which only a single, anonymous cluster was permitted. Adopting the Smith and Smith concept enhanced the expressive power of classification, allowing it to be used in many more situations than was previously possible. In terms of the Class Language model, the extension was a generalisation of the old form of classification rule, and simultaneously a specialisation of the class of representational constructs. As an incidental note, our language model was able to apply this new concept to itself: one classification of the model's control class is needed to cover classification as a control mechanism, with a second independent classification covering classification as a form of representation.

7.2. Beyond The Hierarchy

The Class Language aggregation hierarchy is capable of representing one-to-many relationships between objects, such as a *storey* to its component *areas*, but is not so well suited to more general relationships. This limitation was partially removed in version 2 by allowing objects to take part in more than one hierarchy.

A problem with the FireCode system, that was never properly resolved in Class Language version 1, arose with the representation of exitways out of a building. Exitways are by nature non-hierarchical: a single exitway can serve many areas, and an area will usually have more than one exitway available to it.

Modifying Class Language to use a relational representation would have run completely across the grain of the language: aggregation *is* hierarchial. In an object-centered representation each object is only interested in those relationships in which it is personally involved; knowledge of any relationship not involving the object would violate the principle of information hiding.

The solution for the *exitways* problem is to represent two hierarchies that look at the *same objects* in different ways. Instead of representing a single "connects" relation common to both *areas* and *exitways*, each *area* has a set of "exit routes" and each *exitway* has a set of "areas serviced". An example of these two representations is shown below:

a) The flat relational view
exit1 connects area1
exit1 connects area3
exit2 connects area3
exit3 connects area1
exit3 connects area2

b) The relation viewed from each of the individual objects
exit1 has areas served {area1, area3}
exit2 has areas served {area3}
exit3 has areas served {area1, area2}

area1 has exit routes {exit1, exit3}
area2 has exit routes {exit3}
area3 has exit routes {exit1, exit2}

7.3 Recursion

During the implementation of Class Language, the behaviour of the language in extreme or unusual cases was periodically examined and defined. This was usually motivated from the need to make the language software as robust as possible, but on a number of occasions potentially useful extensions to the language were uncovered. The most significant of these extensions centered around situations in which a class is defined recursively. Our desire to avoid arbitrary restrictions resulted in several extensions being included in the language, even though no application made use of them at the time. This policy was eventually rewarded when the Seismic [Lomas, 1988] application and then WallBrace [Mugridge, 1988] found a genuine use for one of the extensions.

Three of the representational constructs supported by Class Language - inheritance, classification and aggregation - define relationships between classes. Inheritance defines a relationship in which one class is extended to include the definition of another. Classification relationships are a form of dynamic inheritance, in which the decision as to whether one class is to inherit from another is left until runtime. Aggregation is a relationship between a collection of classes that together form a composite class. For each of these relationships, we need to define what happens when a class is directly or indirectly related to itself.

A class that inherits from itself directly is making a trivial statement: how could a class not include its own definition? If such a construct was to appear in a Class Language program it would most likely be the result of a typing error or a serious mis-understanding by the programmer. A more interesting situation arises when a class inherits from itself indirectly. In the simplest case, class "a" inherits from class "b", where class "b" inherits from class "a". In such a structure the appearance of classes "a" and "b" will be the same, since they will both contain their own properties and the properties of the other. However, we are not able to boldly assert that the program is in error because the behaviour of the two classes will be subtly different. For class "a", the order in which rules are applied will start with class "a" and then proceed to class "b", while for class "b" the order is "b" then "a".

Perhaps a clever programmer might one day find this behaviour useful. On the other hand, perhaps this feature is never useful, or worse that it leads to confusion or erroneous programs going undetected. This raises an important question as to the limitations of our language model. The feature is encouraged by the model, since the model is more consistent when the feature is included than when it is excluded, but in actual practice the feature may turn out to be either useless or harmful. Even if a

genuine case is found for using recursive inheritance, the likelihood of it being used in error will remain. The compromise reached was to support recursive inheritance, but to issue a warning message whenever it is encountered.

Like direct inheritance, classification can never be sensibly applied from one class back onto itself, and we will not consider it further.

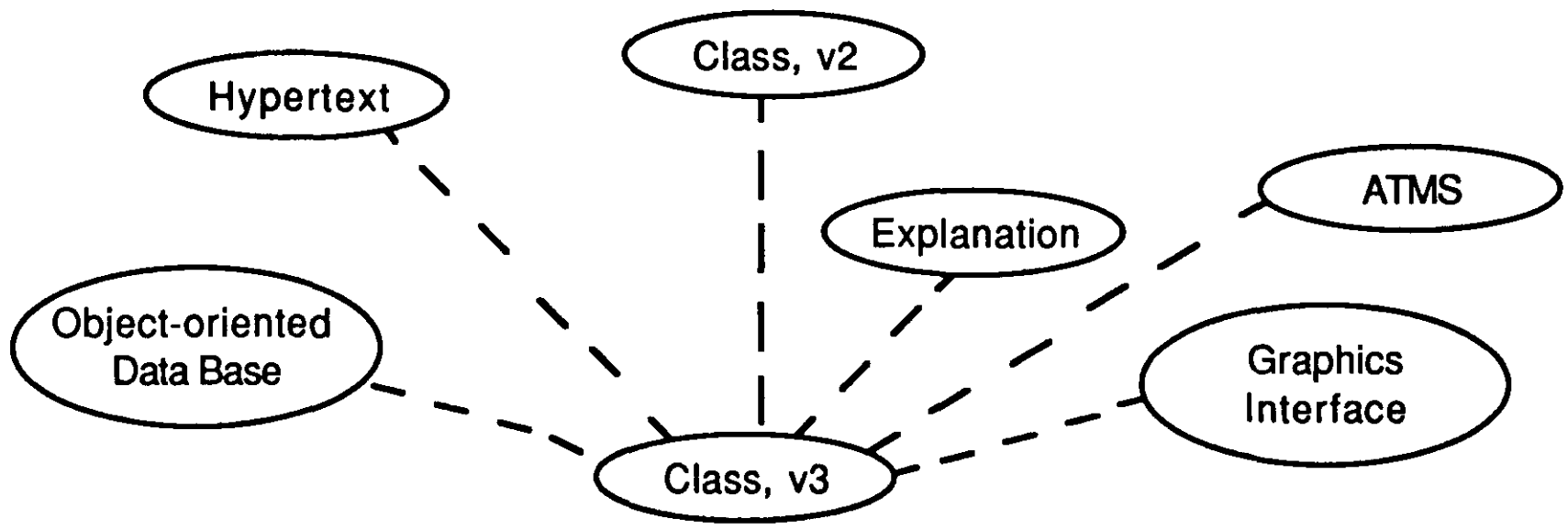
We initially felt that a class that uses *itself* to describe a component could have no real-world counterpart. The mathematical expression for the size of such an object is an abomination: size(object) = size(component) + ... + size(object). An object that is bigger than itself! Interestingly, this "contradiction" has been exploited by novelties like the toy "Russian dolls", in which a wooden doll opens up to reveal yet another doll, and so on. Ultimately, of course, the "Russian doll" degenerates into an ordinary wooden doll that cannot be opened. Needless to say, as computer scientists we were more attracted to the beauty of such novelties than the mathematical contradictions, and "recursive classes" were implemented in Class Language. In fact, implementation was relatively painless, due to the use of lazy evaluation techniques.

After recursive aggregation was implemented it was more or less forgotten about until a difficulty with the Seismic application prompted its reevaluation. Seismic is based on an earthquake loading code that used a design cycle to successively refine a model of the building under consideration. The design cycle starts with some initial parameter estimates. Each iteration of the cycle uses parameters from the previous cycle as the basis for refining the model. The process continuing until a suitable accuracy was obtained. Although the design cycle is expressed in an iterative form, intermediate results are usually retained for future reference. A recursive representation of the design process was clearly appropriate.

Recursive classes are also used in the WallBrace expert system to implement structures such as lists. In this case, however, the construct is being used as a low level mechanism to emulate more sophisticated control structures than those provided by the language. Experimenting with the language in such ways inevitably furthers the design process by providing a starting point for more appropriate control structures.

8. Class, Version 3 (The Future)

Class Language has been the object of continuing change since its inception. While there are still many areas in which the language itself can be strengthened, we are now also looking to enhance the environment for both the application developer and the end-user. Some of these influences are shown in the sketch below:



8.1. Interface Issues

The Class Language interface for the end user currently assumes a simple ascii terminal. This constraint is necessary for applications to be made available over PACNET [Whitney, 1987], but it is a severe restriction to users with access to graphic workstations. Many of the expert systems that have been developed to date would benefit from a graphical interface in conveying spacial information. The ability to display a diagram can greatly simplify the questions that need to be asked of a user.

Class Language currently provides a mechanism for providing simple textual explanations of the reasoning process to the user. This approach is often inappropriate, since it is difficult to cater for the range of information that a user may require. A more flexible approach is provided by hypertext systems [Conklin, 1987], which allow a structure of text and graphics to be explored under user control. We imagine that much of the user interface could be integrated into a hypertext environment.

8.2. Language Issues

A recent expert system, WallBrace, has applied Class Language to a simple design task [Mugridge, 1988]. This exercise has suggested that a non-monotonic reasoning system, such as ATMS [deKleer, 1986], could be integrated into the language to allow design alternatives to be explored. Such a system is described in [Struss, 1986].

An issue that arose in the Adhesive system [Fromont and Watkinson, 1988] was the need to access to information stored in external databases. Such data must currently be compiled into a Class Language program, and this makes updating difficult. Ideally, an external interface should be provided that has the same semantics as an ordinary Class Language class.

9. Conclusions

The major stages in the development of Class Language and the influences behind them have been described. The design of the language was focused by the needs of applications, while constantly referring back to knowledge abstraction principles. As our experience grew, design criteria beyond the needs of any immediate applications were identified. This ultimately lead to the construction of a model of the design process itself. This model proved useful in generalising language concepts, and in imposing criteria for the inclusion of more specialised constructs. We noticed that parts of the design model could be expressed in Class Language itself, and this helped in providing a consistent terminology for the model.

Of the lessons learned from our experience in designing a language, the need to avoid adding features without good reason was perhaps the most important. Class Language illustrates that it is possible to support the needs of a specific class of applications in a principled way.

Acknowledgements

The authors gratefully acknowledge the long term collaboration and financial support of BRANZ that has made this research possible.

References

Brachman and Schmolze, 1985
"*An Overview of the KL-ONE Knowledge Representation System*"
Journal of Cognitive Science, Vol. 9 No. 2, April-June 1985.

Buis, M., 1986
"*The Construction of Expert Systems*"
MSc Thesis 1986, University of Auckland.

Buis, M., Hamer, J., Hosking, J.G., Mugridge, W.B., 1987
"*An Expert System for a Fire Safety Code*"
in "*Applications of Expert Systems*", Quinlan,J.R.[ed] Addison-Wesley, Gt. Britain.

Conklin, J., 1987
"*Hypertext: An Introduction and Survey*"
IEEE Computer, September 1987.

deKleer, J., 1986
"*An Assumption-Based Truth Maintenance System*"
Artificial Intelligence, Vol 28, No 2, 1986.

Fikes, R. and Kehler, T., 1985
"*The Role of Frame-Based Representation in Reasoning*"
Communications of ACM, Vol. 28 No. 9, September 1985.

Fromont, C.J., and Watkinson, P.J., 1988
"*A Knowledge-Based System on the Selection of Adhesives for Wood or Wood-Based Substrates*"
Proc. 3'rd NZES Conference 1988, Wellington.

Goldberg, A. and Robson, D., 1983
"*Smalltalk-80: The Language and its Implementation*"
Addison-Wesley, Melno Park, 714pp.

IntelliCorp, 1984
"*The Knowledge Engineering Environment*"
Technical Article, IntelliCorp™.

Lomas, S., 1988
"An Expert System for a Seismic Loadings Code"
MSc Thesis 1988, University of Auckland.

Mugridge, W.B. and Hosking, J.G., 1988
"*The Development of an Expert System for Wall Bracing Design*"
Proc. 3'rd NZES Conference 1988, Wellington.

Smith, J.M. and Smith, D.C.P., 1977
"*Database Abstractions: Aggregation and Generalisation*"
ACM Transactions on Database Systems, 2, No. 2, 1977, pp. 105-133.

Struss, P., 1987
"*Multiple Representation of Structure and Function*"
Proc. Working Conference on Expert Systems in Computer-Aided Design, Australia 1987.

BUILDING RESEARCH ASSOCIATION OF NEW ZEALAND INC.
HEAD OFFICE AND LIBRARY, MOONSHINE ROAD, JUDGEFORD.

The Building Research Association of New Zealand is an industry-backed, independent research and testing organisation set up to acquire, apply and distribute knowledge about building which will benefit the industry and through it the community at large.

Postal Address BRANZ, Private Bag, Porirua

ISSN 0111–7459