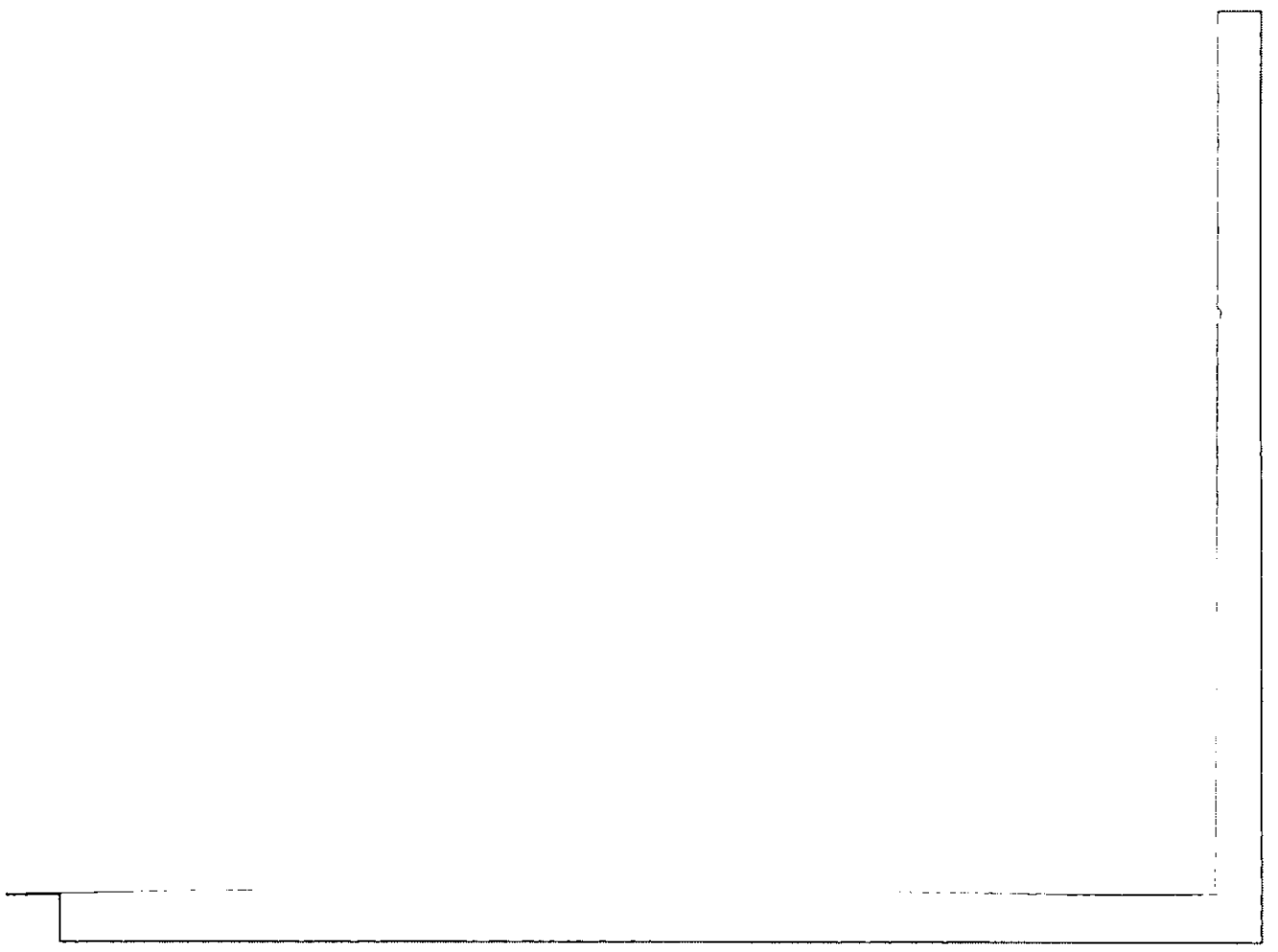
SERIES BU1



BRANZ

THE RESOURCE CENTRE FOR BUILDING EXCELLENCE

REPRINT

NO. 107 (1991)

Integrating Functional and Object-Oriented Programming

J G Hosking, J Hamer & W B Mugridge

Reprinted from Proceedings of TOOLS Pacific
Sydney, pp 345 - 355

ISSN - 0111-7459

INTEGRATING FUNCTIONAL AND OBJECT-ORIENTED PROGRAMMING

J.G. Hosking, J. Hamer,
W.B. Mugridge
Department of Computer Science
University of Auckland
Auckland, New Zealand
john@cs.aukuni.ac.nz

Abstract

We describe *Class*, an object-oriented functional language, the development of which was motivated by applications for representing and applying codes of practice for the building industry. *Class* is strongly typed, with information hiding. Class types encapsulate functions and procedures. Functions are executed lazily providing, amongst other things, lazy object creation. Classes may inherit and/or override functions and procedures. A particular novelty is the classification mechanism, which both constrains class membership and permits objects to dynamically extend their class membership in a controlled fashion.

1. Introduction

Since 1985, the authors have collaborated with the Building Research Association of New Zealand (BRANZ) in the development of systems and tools for representing and applying codes of practice of relevance to the building industry (Hosking et al, 1990). An important component of this work is an object-oriented model for code of practice representation (Hamer et al 1989), and a realisation of that model as the programming language *Class*. This language provides functional programming facilities within a strongly typed, object-oriented framework.

Both functional and object-oriented approaches have been advocated for the productive construction of reliable software. Advantages for object-oriented programming include modularity, design based on abstract data types, and reusability through inheritance (Meyer, 1988). Advantages of modern functional languages include referential transparency and improved modularity via higher-order function abstraction and lazy evaluation (Hughes, 1989). Work integrating the benefits of both of these approaches includes: ML modules (Tofte, 1989), providing class-like encapsulation of ML functions; Cardelli and Wegner (1985), who explore issues of data types and modularity; and Goguen and Meseuger (1987), who provide a comprehensive discussion of integrating functional, object-oriented and logic programming. Our approach provides a seamless integration of the two paradigms, and, in addition, introduces a novel object classification mechanism.

We introduce the non object-oriented features of the language first and follow with a discussion of the object oriented extensions, including the classification mechanism. Introduction of each feature is justified by the problem of code of practice representation. We finish with a review of our current research directions.

2. Non object-oriented features

2.1 Functional Representation

Class is a single assignment or functional language. It is intended to model a consistent state of a complex object like a building and its components, such as in programs which check for code conformance. A simple program consists of a set of typed function definitions, which can alternatively be viewed as a set of typed *instances* (or single-valued variables) with associated expressions. The expression associated with an instance is evaluated

when a value is required, such as in the evaluation of another expression.

The motivation for adopting a functional basis to *Class* was the problem of representing code of practice provisions. Fig. 1 shows a typical provision, adapted from Hay (1984), which defines the *occupant load* of a portion of a storey. A functional, as opposed to a multi-direction constraint, is the accepted interpretation of such provisions (Fenves et al, 1987).

Fig. 2 shows the *Class* representation of the provision of Fig.1 Predefined integer and text (string) types are used, together with an enumerated type. A conditional expression is used in the expression for *occupant_load* (read "|" as "else"). The function *ask* collects data from the user, who provides the function value by responding to argument of the *ask* function.

The single-assignment part of the language allows for expressions to be defined, but does not provide any need for evaluation to occur. Evaluation is initiated by the procedural component of *Class*, introduced in the next section.

2.2 Procedural representation and an execution model

Class is also a procedural language. Procedures direct the flow of control, causing evaluation of functions as needed. Importantly, there is no procedural assignment statement; procedures do not specify how to evaluate anything, only what to evaluate. Thus the functional aspect of the language remains pure; procedural code is an outer-layer over the functional part. Conditional control constructs are provided; the *display* statement produces output.

This limited form of procedural representation was motivated by the need to construct reports to output on line-oriented terminals, and to explicitly represent sequential aspects of code of practice applications (Hosking et al, 1990).

```
The occupant load of an area of a storey is, for an assembly
area, the number of occupants for which the area is designed
and, for a sleeping area, the number of bedspaces.
```

Figure 1: A provision adapated from Hay (1984).

```
occupant_load: integer
    := occupant_count if area_type = assembly
    |  bedspace_count if area_type = sleeping.
area_type: [assembly, sleeping]
    := ask('Is area ', area_name ' a'&
           '[ 1 ] Assembly area'&
           '[ 2 ] Sleeping area'&
           'Enter 1 or 2 ?: ').
occupant_count :integer
   := ask('How many occupants are in area ',area_name,' :?').
bedspace_count: integer
   := ask('How many beds are in area ', area_name,' :? ').
area_name: text
   := ask('What is the name of this area :?').
```

Figure 2: Functional encoding of the provisions in Fig. 1

```
procedure main.
   display('Occupant load of area ',area_name,
           ' = ', occupant_load).
end main.
```

Figure 3: An example procedure

Fig. 3 shows a procedure *main* which uses the *area_name* and *occupant_load* functions of Fig.2. Execution of a *Class* program commences with procedure *main*, resulting, in the above example, in execution of the display statement, causing in turn execution of the *area_name* and *occupant_load* functions. The resulting dialogue is shown in Fig. 4.

```
CLASS> run
-1- What is the name of
this area :? dormitory
-2- Is area dormitory a
    [ 1 ] Assembly area
    [ 2 ] Sleeping area
    Enter 1 or 2 ?: 2
-3- How many beds are in
area dormitory :? 10
Occupant load of area
dormitory = 10
```

Figure 4: Dialogue with program of Figs 2 and 3

2.3 Caching function results and change

Class caches input data (such as results of ask function invocations). This is necessary to avoid reprompting users for the same information. For example, in Fig. 4 the area name is prompted for once, even though it is used twice. Caching of other function results, such as *occupant_load* is not necessary, as the referentially transparent nature of the functions means that they can always be recomputed. However, for efficiency reasons, results of parameterless functions are cached. This contributes to a dichotomy of views of the functional component: functions or single assignment data items with lazily evaluated expressions.

The single assignment nature of the language is important for code of practice application. Here it is vital to maintain consistency between input and results, something which is not guaranteed with reassignment. Another requirement for such applications, though, is to allow users to modify answers to explore a design space or correct errors. To support this, a change facility is provided. If a user changes an input value, a dependency graph is used to flush cached values dependent on that input, and affected output is redisplayed. Fig. 5 shows change facility usage.

```
CLASS> run
-1- What is the name of
this area :? dormitory
-2- Is area dormitory a
    [ 1 ] Assembly area
    [ 2 ] Sleeping area
    Enter 1 or 2 ?: 2
-3- How many beds are in
area dormitory :? change 2
-2- Is area dormitory a
    [ 1 ] Assembly area
    [ 2 ] Sleeping area
    Enter 1 or 2 ?: 1
-4- How many occupants are
in area dormitory :? 15
Occupant load of area
dormitory = 15
```

Figure 5: Dialogue with program of Figs 2 and 3 using change.

3. Classes and objects

Class inherits from the object-oriented paradigm notions of information-hiding, abstract data types, and multiple inheritance. It introduces the novel notions of lazy object parameters and dynamic classification. Unlike most object-oriented languages, *Class* is a functional language, and does not allow state-change, only state elaboration. Here, we examine classes and their components. In the following sections we examine inheritance and dynamic classification.

3.1 Classes

In addition to functions and procedures, a *Class* program can include class definitions. Classes contain features, in the Eiffel (Meyer, 1988) sense, which may be functions or procedures. For a feature to be visible outside a class it must be declared *public*. Classes can include parameter features, used to pass data to a newly created instance of a class from its creation context.

Fig. 6 is an example program using classes. Class *Storey* has a public procedure, *report*, and three private functions, *occupant_load*, *areas*, and *area_count*. Class *Area* has a public function, *occupant_load*, a parameter *seq_no*, and three private functions, *area_type*, *occupant_count*, and *bedspace_count*. The program elaborates the example of the previous section, calculating the occupant load of a storey as the sum of the occupant loads of its constituent areas.

```
the_storey : Storey := new.
procedure main.
     display('Occupant load calulation').
     the_storey^report.
end main.

class Storey.
     public report.

     occupant_load: integer
           := sum(collect(a in areas, a^occupant_load)).
     areas: list Area
           := collect(i in 1..area_count, new Area(seq_no :=i)).
     area_count: integer
           := ask('How many areas are there:? ').

     procedure report
           display('Occupant load = ', occupant_load).
     end report.
end Storey.

class Area.
     public occupant_load.
     parameter seq_no: integer.
     occupant_load: integer
         := occupant_count if area_type = assembly
         |  bedspace_count if area_type = sleeping.
     area_type: [assembly, sleeping]
           := ask('Is area #', seq_no ' a'&
                  '[ 1 ] Assembly area'&
                  '[ 2 ] Sleeping area'&
                  'Enter 1 or 2 ?: ').
     occupant_count :integer
        := ask('How many occupants are in area #',seq_no,' :?').
     bedspace_count: integer
        := ask('How many beds are in area #', seq_no,' :? ').
end Area.
```

Figure 6: An example program using classes

3.2 Objects and lazy object creation

Classes extend the available types. The function *the_storey*, for example, evaluates to an *object* which is of type *Storey*. New objects are created with the pseudo-function *new*.

With the dichotomy of views, *the_storey* can also be viewed as an object reference. Features of objects may be referred to using the carat ("^") operator between an object reference and the feature name. For example, in *main* is the procedure invocation *storey^report*. This invokes the *report* procedure of the object referred to by *the_storey*. The object reference, and the object, are constructed lazily; executing *storey^report*, firstly causes the expression associated with *the_storey* to be executed (if its value has not been cached previously). This expression creates a new object of class *Storey*. Having created the object, and made *the_storey* a reference to it, the *report* procedure of that object executes. Any subsequent references to *the_storey* (none in this example) will refer to the same object. For now we assume this object reference view. In 3.5 we explore a functional view of objects in more detail.

3.3 Object parameters

The object creation function *new* may be qualified by the class of the object to create (if it differs from but conforms to the feature type, or where it may be ambiguous) and expressions to be associated with parameters of the class. An example is in the expression for the *areas* feature of *Storey* class (explained in detail below): the expression "*new Area(seq_no := i)*" creates an object of class *Area*, with parameter *seq_no* associated with the expression "*i*".

Parameter expressions are *not* evaluated at object creation, but, like other features, lazily, when a value is needed for the parameter (or in the functional view, the parameter passed is a function body, invoked as required from within the object).

3.4 Lists

Lists (ordered collections) of objects or values are available, illustrated by the feature *areas* in *Storey*, a reference to a list of *Area* objects. A number of list manipulation operators and functions are provided. For example the subrange constructor ".." makes a list of integers ranging between its two arguments.

The *collect* function (*map* in conventional functional languages) takes a list and produces a new list by applying a function to each element . The expression for *areas* in Fig. 6 uses *collect* to make a list of new *Area* objects, one for each item in the list *"1..area_count"*, each being passed a parameter corresponding to the subrange element value. The expression for *occupant_load* uses *collect* to create a list of the *occupant_load*'s of each object in *areas*.

The *sum* function returns the sum of a list of numbers. The expression for *occupant_load* in Fig. 6 uses *sum* to sum the list of occupant load values generated by the *collect* expression.

3.5 Objects as functions

The previous discussion has viewed objects and object references fairly traditionally. However, another view, relating to the functional nature of the language, is also possible. In this view objects are seen to be limited types of partially evaluated, or *curried*, functions (Bird and Wadler, 1988). Creating an object is then equivalent to partial function application, the object parameters and creation context being the parameters to that partial evaluation. Seen in this way, there is very little difference between object creation and function invocation in *Class*.

Further currying occurs by composing the object with one or other of its public features to produce another object (another curried function) or a simple value. The currying is restricted as the partial evaluation can only be extended by the public features of the object's class.

Hence, the expression "*the_storey^report*" can be interpreted as an invocation of the function *the_storey*, which returns a curried function (a *Storey* object). This function (object) is then applied to the feature *report* (which happens to be a procedure, and which is executed).

4. Inheritance

A class may be declared as a subclass of another class. For example, it may be useful to split the Area class into one general class, with two subclasses corresponding to *Assembly* and *Sleeping* areas (Fig. 7.). This would certainly be the case if there were additional provisions that made distinctions between the two sorts of area.

A subclass inherits the public interface (including parameters) and implementation of the superclass. It may add to either, as well as supply new bodies for inherited functions and procedures. For example, the feature *occupant_load* is declared in class *Area* in Fig. 7 (with an empty body - like an Eiffel deferred feature) and is provided with a body in both subclasses.

4.1 Type conformance and polymorphism

The generalisation relationships between classes define a (partial) type ordering. For example, the type *Assembly* is a subtype of *Area*. Hence a feature of type *Area* may be assigned an object of type *Assembly*, as shown in Fig. 8. Functions and procedures are dynamically bound; in Fig. 8, the body for *occupant_load* from class Assembly is used.

```
class Area.
     public occupant_load.
     parameter seq_no: integer.
     occupant_load: integer.
end Area.

class Assembly.
     generalisation Area.
     occupant_count :integer
          := ask('How many occupants are in area #',seq_no,':?').
     occupant_load
          := occupant_count.
end Assembly.

class Sleeping.
     generalisation Area.
     bedspace_count: integer
          := ask('How many beds are in area #',seq_no,':? ').
     occupant_load := bedspace_count.
end Sleeping.
```

Figure 7: Subclasses

```
instance
     area1: Area := assembly_area.
     area2: Assembly := new(seq_no := 1).
procedure poly.
     display('Occupant_load = ',area1^occupant_load).
end poly.
```

Figure 8: Polymorphic assignment

In general, an object of type *C* may be assigned to a feature of type *P* (i.e. *C* conforms to *P*), when *P* inherits from *C* or from a class that *C* conforms to. There is no notion of inheritance without a type relationship, as compared to Smalltalk (Goldberg and Robson, 1983) and other untyped object-oriented languages.

4.2 Multiple inheritance

A class may inherit multiply. The inherited public interface of the class is the union of the public interfaces of its superclasses. Name clashes between independent superclasses are illegal, but if several inheritance paths lead to one superclass one copy only of its features is inherited.

The order of classes is significant when subclasses override an inherited function or procedure; code in the lowest class in the inheritance hierarchy is used before code in classes further up. However, multiple inheritance leads to a partial order of classes, so the order of execution is not defined completely. Snyder (1986) discusses ways of making the partial order a total order.

5. Dynamic Classification

Dynamic classification of objects is a novel and powerful feature of *Class*, which allows the class membership of an object to be elaborated at runtime. As information is gathered about an object of a general class, it can be classified as also belonging to one or more subclasses. Other work, mostly in AI, aims to provide such a mechanism, (Bylander & Mittal, 1986; Chandrasekaren, 1986), but none has the integration and power provided by that of *Class*.

5.1 Motivation

As motivation for this mechanism, consider incorporation of the *Area* subclasses of Fig. 7 in the example of Fig. 6. Here, we would like the *areas* of class *Storey* to be instantiated with a mixed list of *Assembly* and *Sleeping* objects, depending on the type of each area. Polymorphic assignment allows this, but the resulting "case creation" code (Fig. 9) is a little tortuous.[1]

[1] Case creation code like this is common in OO languages; see e.g. the Eiffel example libraries.

```
areas: list Area
      := collect(i in 1..area_count,
                    new Assembly(seq_no :=i )
                       if ask('Is area # ',i,' assembly ?')
                    | new Sleeping(seq_no := i)
                 ).
```

Figure 9: Case creation of a list of objects of mixed classes

```
class Area.
     public occupant_load.
     parameter seq_no: integer.
     classification
          area_type: [Assembly, Sleeping]
               := ask('Is area #', seq_no ' a'&
                         '[ 1 ] Assembly area'&
                         '[ 2 ] Sleeping area'&
                         'Enter 1 or 2 ?: ').
     occupant_load: integer.
end Area.
```

Figure 10: Addition of a classification feature to class Area

Several points can be made about this approach. Firstly, *Storey* had to be modified, just because of a simple internal reorganisation of the "area"concept. A central principle of object-oriented programming is that object behaviour is defined internally. In the situation above, information had to be included in *Storey* concerning the different types of *Area*, and how to classify them. It should not be necessary for *Storey* objects to know this, it is information rightly belonging to the *Area* classes. Additionally, code like this is tolerable only if case creation occurs very few times (ie: once) in a program. If several such lists are needed, the case creation code must be repeated for each. The classification mechanism eliminates these problems by including classification information within the appropriate classes.

5.2 Classification features and dynamic classification

In *dynamic classification* the *name* of the class to which an object is classified is the *value* of a feature of the object. We call these features *classification features*, but apart from their special role they appear in all respects as ordinary features. Smith and Smith (1977), use a similar technique to express the static class membership of an object in a relational database extension.

Fig. 10 shows incorporation of a classification feature into class *Area*. The classification feature *area_type* can take values of *Assembly* or *Sleeping*. The associated expression, in this case an *ask* call, is executed when a value for *area_type* is required, as for any other feature. However, when the *area_type* of an *Area* object is evaluated, the class membership of the object is extended to include the resulting value. Hence, if *area_type* is *Assembly*, the object becomes to all intents an *Assembly* object. This *dynamic classification* can be likened to an internal polymorphic assignment.

Boolean classification features permit optional classification. If one evaluates to *true* classification to the class named by the feature results, if *false* no classification occurs.

5.3 Classification is lazy and automatic

As described above, classification occurs when a classification feature is evaluated. This can be explicitly done, by referencing the classification property. However, if this is the only way, it may result in classification to a class *after* a feature with a body defined in that class has been evaluated, leading to inconsistency between the feature value and the value the new body would return. Rather, classification is best done automatically on demand. Whenever a possible classification can provide additional feature bodies that could affect the current function or procedure evaluation, evaluation of the appropriate classification feature occurs.

```
class Assembly.
   generalisation Area.
   classification
       Fixed_seating: boolean
         := ask('Does area #', seq_no,' have fixed seats :?').
   occupant_count :integer
       := ask('How many occupants are in area #',seq_no,':?').
   occupant_load
       := occupant_count.
end Assembly.

class Fixed_seating.
   generalisation Assembly.
   fixed_seat_count: integer
      := ask('How many fixed seats are in area #',seq_no,':? ').
   occupant_load := bedspace_count.
end Fixed_seating.
```

Figure 11: Incremental classification

For example, if the *occupant_load* of an *Area* (Fig. 10) object is required, *Class* knows that bodies for the feature are available in classes *Assembly* and *Sleeping*, both of which are able to be classified to via the *area_type* classification feature. Hence, before evaluating *occupant_load*, *area_type* is evaluated and classification to an appropriate subclass results.

There is one other time when classification is in a sense automatic. If an object of a class which can be classified to by one of its superclasses is directly created, *Class* ensures evaluation of the corresponding classification feature is consistent with the object's class. For example, if an *Assembly* object is created, evaluating its *area_type* is guaranteed to return the value *Assembly*.

5.4 Classification is incremental

Chains of classifications can occur in an incremental fashion. For example, a special class of assembly area is one with fixed seating, where the occupant load is the number of fixed seats in the area. This can be handled by creating *Fixed_seating* as a subclass of *Assembly*, and including a boolean classification feature in class *Assembly* permitting optional classification of *Assembly* objects to *Fixed_seating* (Fig. 11).

Now if the *occupant_load* of an *Area* object is required, classification of the object to class *Assembly* may result, through evaluation of *area_type*, followed incrementally by classification to *Fixed_seating*, by evaluating the boolean *Fixed_seating*.

Lazy classification means that incremental classification may occur through evaluation of more than one feature. For example, suppose an additional feature of class *Area* is *maximum_area*, and a body for this function exists in *Assembly*, but not in *Fixed_seating*. If the *maximum_area* of an *Area* object is needed, this may result in classification of the object to *Assembly*. If, later, the *occupant_load* of the object is required, further classification to *Fixed_seating* may result.

5.5 Multiple classification

A class may include or inherit more than one classification feature. This allows an object of that class to be classified along several independent paths, and to thus occupy more than one point in the class inheritance structure. Thus, the set of classes to which objects belong is no longer fully specified by inheritance; a distinct *classification* relationship between classes is available.

Multiple classifiers have proved useful in the code of practice applications. For example, Hay (1984) includes two almost independent classifications of building areas, one being the usage (assembly, sleeping, etc) and the other being whether they have one (deadend) or more exits. All combinations of the two classifications (deadend-assembly, deadend-sleeping, etc) must be provided. In a traditional OO approach this requires a combinatorially explosive number of subclasses (Hendler, 1986), many mix-ins, and ghastly case creation code. In *Class* all that is needed is a *Deadend* class, a class for each usage category, and two classification features: a

boolean for optional classification to *Deadend*, and one to classify to the appropriate usage class. Hamer (1990) discusses benefits of multiple classification as an alternative to mix-ins.

5.6 Classification and inheritance are separate

The distinction between classification and inheritance is subtle but important. Most classification systems treat classification as the inverse of inheritance (e.g. Bylander and Mittal 1986); thus, an object can only be classified to an immediate subclass. While this is appropriate in most situations, there are good reasons for separating classification from inheritance.

One reason for using inheritance is to abstract features common to several classes. In Fig. 12, for example, the taxonomy (broad arrows) abstracts features common to gable end and hip roofs in a *RidgedRoof* class, and abstracts ridged, lean-to and other roofs into a *SlopingRoof* class (Hamer et al 1989). Classification of a roof, though, does not need the *SlopingRoof* and *RidgedRoof* classes; it can proceed in one step, to the appropriate roof type (narrow arrows).

Ther are two constraints between classification and inheritance: if a class can be classified to another, they must both inherit from a common ancestor; and classes that a classification feature can classify to are mutually exclusive, and hence no subclass can inherit from more than one of them (Hamer, 1990).

6. Applications

Class has not developed in a vacuum. It has been applied to and influenced by the development of several large applications for checking code of practice conformance of buildings.

FireCode (Hosking et al, 1987; Mugridge et al, 1988) is a system which helps check building plans against requirements of Parts 2 (use classification) and 6 (means of escape) of *DZ4226: Code of Practice for Design for Fire Safety* (Hay, 1984), a draft New Zealand Standard. *FireCode* allows a designer to check a building or individual components such as one storey.

Three projects followed *FireCode*: *Seismic* (Hosking et al, 1988) and *WallBrace* (Mugridge and Hosking, 1988), developed by the authors, and *SubFloorBrace* (Dechapunya and Whitney, 1988), developed by BRANZ. *Seismic* helps designers meet the seismic loading provisions of *DZ4203: 1986 General Structural Design and Design Loadings* (SANZ, 1986). *WallBrace* and *SubFloorBrace* assist designers meet the wall and subfloor bracing provisions respectively of *NZS3604: 1984 Code of Practice for Light Timber Frame Buildings* (SANZ, 1984).

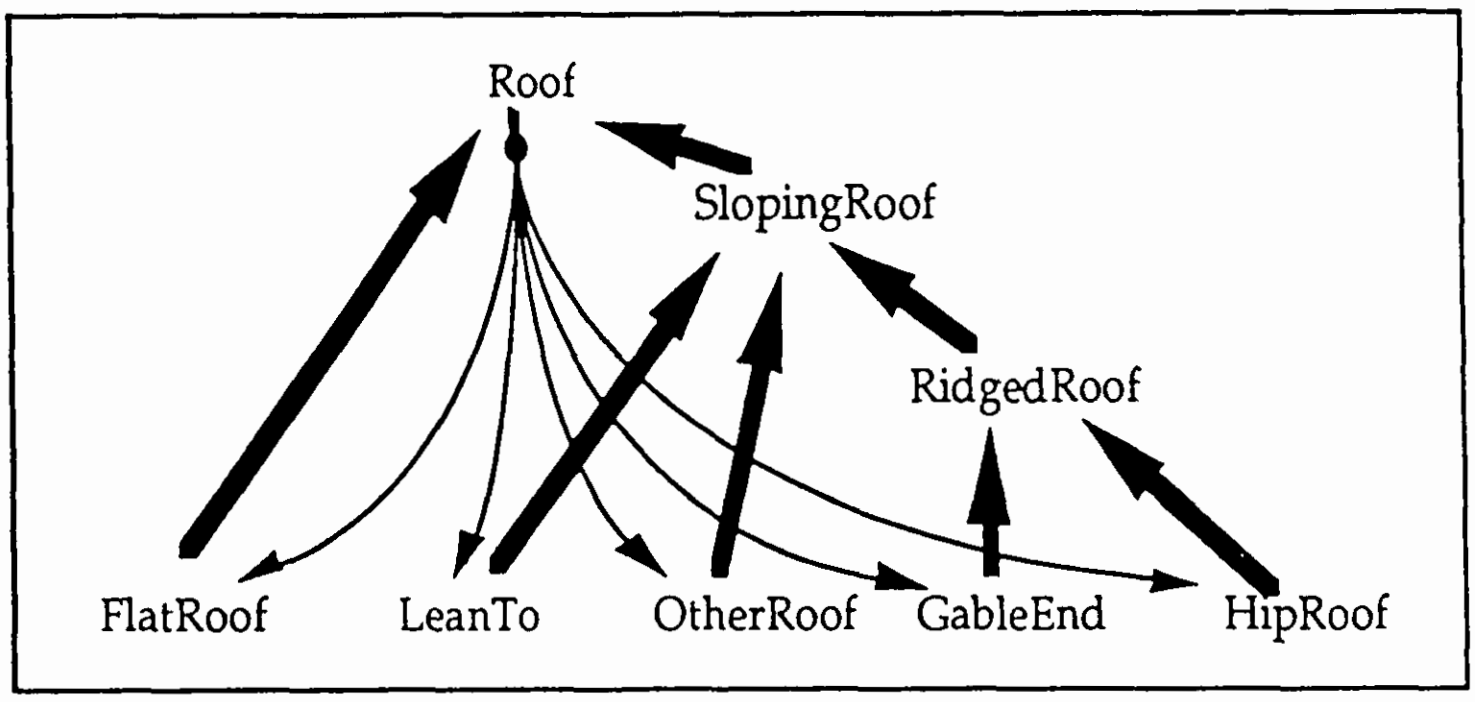


Figure 12: Separate classification and inheritance in a roof taxonomy

7. Conclusions and future work

The motivation for the development of *Class* was the needs of the code of practice applications. To support these a variety of techniques seemed desirable: functional, for provision representation; procedural, for sequence; object-oriented, for representing building structures and categories; and classification, for representing category selection. Rather than integrate these representational mechanisms in an ad-hoc fashion, we chose to tightly integrate them in as seamless a way as possible. The result is a general object-oriented functional programming language, applicable to a wide variety of application domains.

A particular novelty is the classification mechanism. The lazy, incremental nature of this mechanism, based as it is on the lazy-functional nature of the language as a whole, provides considerable power and elegance. An open issue is how much of this elegance would be lost in implementing some form of classification mechanism in an imperative OO language, such as Eiffel (Meyer, 1988) or Trellis/Owl (Schaffert et al, 1986). A simple approach is to automatically perform classification on object creation, rather than lazily. This would, at least, eliminate the case creation problem, and would, in addition, be simple to implement. Multiple classification raises interesting issues of supporting, in effect, dynamic union types.

While we have an interest in exploring these issues, we also have an interest in extending the power of *Class* itself. The functional component is a little weak and restrictive. For example, parameters to features (functions) have only just been added to the language, and user defined higher order functions (other than the ersatz object version) are not yet supported. The addition of such features raises interesting typing issues, which Mugridge (1990) has started to explore.

Other developments include an X-Window based form definition and graphical interface system (Mugridge 1989; Hosking, 1989), a visual programming environment, database extensions, and an application for assisting in the design of thermal insulation for houses.

Acknowledgements

The authors gratefully acknowledge the financial support of the Building Research Association of New Zealand, the University of Auckland Research Committee, and the New Zealand University Grants Committee, and the commitment to our work by the Building Research Association of New Zealand, and Dr Haris Dechapunya in particular.

References

Bird, R. and Wadler, P. (1988). *Introduction to Functional Programming*, Prentice-Hall, UK.

Bylander, T. and Mittal, S. (1986). CRSL: A language for classificatory problem solving and uncertainty handling, *AI Magazine*, August 1986.

Cardelli L., Wegner P. (1985). On understanding types, data abstraction, and polymorphism, *Computing Surveys*, **17**(4), pp471-522.

Chandrasekaren, B. (1986). Knowledge based reasoning - high level building blocks for expert system design, *IEEE Expert* Fall.

Dechapunya, A. H. and Whitney, R.S. (1988) Knowledge-based systems for building technology in New Zealand, *Proc Symposium on knowledge-based systems in civil engineering*, Monash, August.

Fenves, S.J., Wright, R.N., Stahl, F.I., Reed, K.A. (1987). *Introduction to SASE:Standards Analysis, Synthesis, and Expression*, U.S. Department of Commerce, May 1987.

Goguen, J.A. and Meseguer, J., (1987). Unifying functional, object-oriented and relational programming with logical semantics, in Shriver B., Wegner P. (Eds), *Research Directions in Object-Oriented Programming*, MIT Press.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley.

Hamer, J. (1990). *Expert Systems for Codes of Practice*, PhD Thesis, University of Auckland Department of Computer Science.

Hamer, J., Mugridge, W.B., Hosking, J.G. (1989). Object-oriented representation of codes of practice. in Gero, J.S. and Sudweeks, F. (eds) *Proc of the Australasian Conf. on Expert Systems in Eng. Arch. and Construction*, Sydney University, December, 209-222.

Hay, R.D. (Ed), (1984). *DZ4226: Code of Practice for Design for Fire Safety*, Standards Association of New Zealand, Wellington, New Zealand.

Hendler, J. (1986). Enhancements for multiple inheritance. *SIGPLAN Notices*, **21(10)**.

Hosking J. G. (1989). *A plan entry package for ThermalDesign*, BRANZ Contract 85-024 Technical Report No. 7, Department of Computer Science, University of Auckland.

Hosking, J.G., Hamer, J., Mugridge, W.B. (1990). From FireCode to ThermalDesign: KBS for the building industry, to be presented to the 4th New Zealand Expert Systems Conference, Palmerston North, New Zealand, December.

Hosking J.G., Lomas S., Mugridge W. B. (1988). The development of an expert system for seismic loading, *Proc Knowledge-Based systems in Civil Engineering Symposium*, Monash University, Melbourne, Australia.

Hosking, J.G., Mugridge, W.B. and Buis, M., (1987). FireCode: a case study in the application of expert system techniques to a design code, *Environment Planning and Design B* **14**, 267-280.

Hughes, J. (1989). Why functional programming matters, *Computer Journal,* 32(2), 98-107.

Meyer, B. (1988). *Object-Oriented Software Construction*, Prentice Hall.

Mugridge, W.B. (1989). *An External Interface for Class Language*, BRANZ Research Contract 85-024 Technical Report No. 8, 30 pp.

Mugridge, W.B. (1990). *Enhancements to an object-oriented language*, PhD Thesis, University of Auckland Department of Computer Science.

Mugridge, W.B., Hosking, J.G., and Buis M. (1988). FIRECODE: An expert system to aid building design, in Newton, P.W., Taylor, M.A.P., and Sharpe, R. (Eds) *Desktop Planning* Hargreen, Melbourne.

Mugridge, W.B. and Hosking, J.G., (1988). The development of an expert system for wall bracing design, *Proc The 3rd New Zealand Expert Systems Conference*, Wellington, New Zealand, 10-27.

SANZ, (1984). NZS3604:1984 (1984). *Code of Practice for Light Timber Frame Buildings*, Standards Association of New Zealand, Wellington, New Zealand.

SANZ, (1986). DZ4203:1986 (1986). *General Structural Design and Design Loadings for Buildings*, Standards Association of New Zealand, Wellington, New Zealand.

Schaffert, G., Cooper, T., Bullis, B., Kilian, M., Wilpori, C. (1986) An introduction to Trellis/Owl, *OOPLSA 86*, pp9-16.

Smith, J.M and Smith, D.C.P. (1977): Database abstractions: aggregation and generalisation, *ACM Trans. on Database Systems* **2** (2).

Snyder A, (1986). Encapsulation and inheritance in object-oriented programming, *OOPSLA 86*, 38-45.

Tofte, M. (1989). Four lectures on Standard ML, Laboratory for Foundations of Computer Science, university of Edinburgh, report ECS-LFCS-89-73.

BRANZ

THE RESOURCE CENTRE FOR BUILDING EXCELLENCE

BRANZ MISSION

To promote better building through
the application of acquired knowledge,
technology and expertise.

HEAD OFFICE AND RESEARCH CENTRE

Moonshine Road, Judgeford
Postal Address - Private Bag, Porirua
Telephone - (04) 357-600, FAX - (04) 356-070

REGIONAL ADVISORY OFFICES

AUCKLAND
Telephone - (09) 5247-018
FAX - (09) 5247-069
290 Great South Road
PO Box 17-214
Greenlane

WELLINGTON
Telephone - (04) 357-600
FAX - (04) 356-070
Moonshine Road, Judgeford

CHRISTCHURCH
Telephone - (03) 663-435
FAX - (03) 668-552
GRE Building
79-83 Hereford Street
PO Box 496