How to Edit "It"; or: A "Black-box" Constraint-Based Framework for User-Interaction with Arbitrary Structures

Tim Menzies
Artificial Intelligence Laboratory,
School of Computer Science and Engineering,
University of New South Wales, PO Box 1, Kensington, NSW, Australia, 2033
timm@cs.unsw.oz.au

Richard Spurrett
Object-Oriented Pty. Ltd.
23 Ben Boyd Rd, Neutral Bay, Sydney, Australia
oorient@extro.ucc.su.oz.au

ABSTRACT: We long for the day when interfaces automatically configure themselves without requiring tedious and time-consuming programming. Such an automatic configuration capability would require knowledge of the objects they are editing. Our current generation of OO interface tools lack a general protocol for exploring the objects they are processing. Therefore, they are unsuitable for auto-configuration. Here we explore a surprisingly simple syntactic model of objects-to-be-edited which a general-purpose interface hierarchy can use to manage the auto-configuration of interfaces. The model relies on "black-box" constraints: methods that when messaged may add error strings to a list of known errors. These constraints are not "glass-box"; i.e. our configuration tools cannot query them to discover their dependency information. Without such dependency information, our black-box system cannot optimise the processing of its constraints. Nevertheless, we demonstrate that black-box constraints can support the automatic configuration and processing of a surprisingly wide variety of interfaces. The open research issue is how far our black-box-based interfaces can be extended without requiring glass-box knowledge.

1. INTRODUCTION

Modern software development requires multiple interfaces to a wide-variety of complex structures. While it is possible to obtain ready-built editor classes for simple objects like dates or pick-one-from-a-list, there are many cases where special purpose editors have to be constructed for domain-specific objects (e.g. risky-loan-applicant or space-shuttle-main-booster).

Consider the architecture of a standard application that utilises a graphical interface (see figure 1). The logical (or business) model may use external services such as a mainframe database. The logical model is presented to the user in an application program that "glues" together collections of display widgets such as text, list, graph panes, buttons, sliders, etc, into a visual unit that has some meaning to the user. The "glue" defines a set of operations and reports that the user can access.

The construction of the "glue", and the customisation of the GUI support code such that the glue works, is a time-consuming process. This is unfortunate since we expect the interface requirements of software in general to increase:

- As users educate themselves in multi-windowed event-driven interfaces (e.g. Microsoft Windows and the Macintosh look-and-feel), we see increasingly more sophisticated interface requirements.
- In projects that emphasis evolutionary development, there will be a requirement for developers and user groups to study the program's execution at many points during the development. That is, software developers should have to build

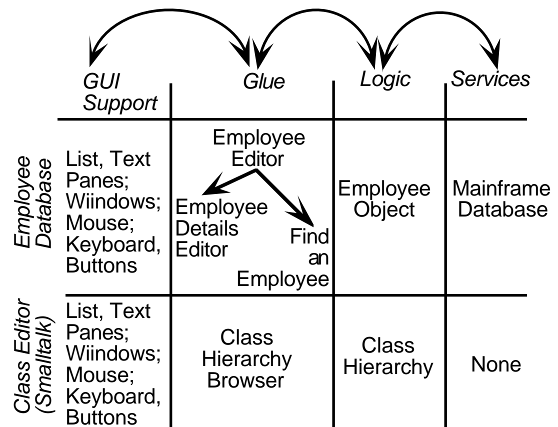

Figure 1: Standard application architecture = GUI support + glue + logic + services. Two applications are described in this framework: (i) a database application (ii) a Smalltalk class editor.

multiple throw-away intermediary interfaces prior to delivery.

In an ideal world, we should be able to develop a clean logical model of an application, press a button, and the interface configures itself automatically. We do not live in an ideal world. Current OO graphical-user-interface (GUI) toolkits (e.g. the MVC triad (Urlocker 1989)) isolate the developer from many low-level details such as paging up and down, window management, displaying characters, tracking mouse movements, etc. Screen painting tools such as WindowBuilder™ and VisualWorks™ simplify layout. However, these tools fall well-short of the goal of automatic configuration since they only work on the GUI support classes or on the very outer-layer of the glue class(es).

Our thesis is that missing from these toolkits is knowledge of the logical model. Without such knowledge interface tools will remain low-level support tools for high-level business applications. Such low-level tools will require extensive and time-consuming customisation before they are useful in an application. Here, we explore methods of collecting that logical knowledge in an OO paradigm.

How should we give our interfaces knowledge of the logical model? One possibility is to demand that every object in the system responds to a "glass-box" protocol that supplies detailed knowledge of the entities used within that object and how these entities are dependant on other entities. This knowledge could include a "dependency diagram": a directed-graph that indicates what entities a method requires to function (i.e. its inputs) and what state changes result from this methods operations (i.e. its outputs). Such a diagram could be used for (i) automatic program validation (Menzies 1993), (ii) rapidly detecting constraint violation, (iii) optimising constraint execution, develop simulation programs (Borning 1981), etc (for a longer list, see the related work section of (Freeman-Benson, Maloney et al. 1990)). Indeed, a program with knowledge of these dependencies can control its execution via an intelligent traversal of its dependency network[1] .

This approach, however, runs contrary to the spirit of the OO paradigm where objects are meant to hide their inner-processing from the outside world. Further, it may not be possible to use our current generation of OO languages for such a glass-box approach. In principle, it is possible that parts of the dependency diagram could be automatically generated from the parse trees of our current object systems. However, the information thus collected automatically may be insufficient and have to be augmented by hand-coded sections. That is, to implement the glass-box approach would require an extensive re-design of exist class libraries and impose stringent requirements on future developments of any class that could be used in an interface.

Here we seek an method for auto-configuration that is an *add-on* to our existing objects and does not imply a re-write of our class libraries. We propose a simple syntactic structure for logical models augmented by black-box constraints: a method that can write error messages if some invariant is violated. The limits to this approach is that constraint application has to be applied in a heuristic rather than an optimal manner. Nevertheless, we demonstrate that it is possible to characterise the interfaces to a surprisingly large number of applications according to this simple structure and a generic black-box constraint protocol.

We hasten to add that our framework does not exist in any single program. For several years now, we have been grappling with the problems of interface design (Menzies 1990; Menzies 1991; Menzies 1991). This paper resulted from pooling the first and second author's practical experience and a reverse engineering of the systems we have developed over the last 5 years. Hence, the framework presented here exists partially in several systems, but not all in a single program. We believe that if a single program had the entire framework, then a wide-range of interface tools would be possible that satisfy the auto-configuration requirement ranging from simple screen painters to arbitrary graph editors as well as expert systems, report generators, and intelligent CASE tools.

This rest of this paper describes the framework (called *MYLE*[2]) and presents our design for these tools in a Smalltalk context. Section 2 describes some over-all design considerations. Section 3 gives details of the framework and section 4 describes its applications. The appendix describes a generic hook into the Smalltalk error handling routines that intercept the generation of error strings and passes them to the active glue instance.

2. DESIGN PRINCIPLES

Our framework may not strike the reader as the way that they would approach this problem. In order to understand why we adopted out particular approach, we list in this section three principles that guided our thinking.

2.1. Dialogue Independence

The standard application architecture of figure 1 is a realisation of the principle of dialogue independence. This principle states that a program's computational component should be kept separate from its interface component. This permits the modification of either component without having to re-configure the other. This is a useful design principle, particularly for programs developed in a prototyping environment (Hartson and Hicks 1989).

Put more crudely, dialogue independence demands that *MYLE* should not pollute a clean logical model (e.g. that of an employee) with interfacing details (e.g. checking for mouse events while editing an employee).

2.2. Ease of Use

The overhead associated with turning a logical model into something that a user/programmer can browse and change should be minimal. Ideally, after creating an object, it should be possible to create a user-friendly, editor of that object that validates user entry by sending one message to it: "specify".

In practice, there is more to it than that:

- Validation methods have to be created for the logical object such that it can check the validity of its contents. Such validation methods do not change

[1] At which point we have left the object paradigm. Such a program can be best described as a rule-based / constraint-based/ logic-based system.

[2] Short for "**MY** **L**ast **E**ditor." The running gag of our research is that we are currently up to *MYLE*, version 3.

state: they merely report errors. We argue that such validation methods are part of the logical definition of any concept and so belong in an object anyway. Many such validation methods already exist in current class hierarchies. For example, the Smalltalk *Date* object calls the error handler if it detects an inappropriate number of days in a month specification. *MYLE* includes a hook into the error handler such that the generated errors are handled by the user-friendly edit screens rather than dropping the user into a user-hostile general-purpose code debugger (see appendix). For an example of the use of validation methods, see figure 7 below.

- A "clean" validation method checks for constraint violation prior to state update. Further, if a violation is detected, the "clean" validator aborts the processing. A "dirty" validation method is not clean. Many validation methods in existing classes are dirty. For example, many of the *error:* methods in Smalltalk/V's Date class do not abort the processing after the call to *error*:. They tacitly assume that the error handler will drop the sender chain and spawn another window. For our purposes, this is unacceptable since our error handler traps the error string, diverts it to the current editor, then continues on with the processing. As to state update, it may be pragmatically difficult to forbid state update prior to validation. Consider a process-control application where one tank instance connects to many other tanks on the plant. In order to check that a newly proposed tank is valid, it may have to update (e.g.) output pipe variables from other tanks then check that the input from these other tanks is compatible. We handle dirty validation methods using *zombie* (see below).
- The logical model has to support one "dataDictionary" method that includes the information needed for a user-friendly edit of the model (e.g. help text explaining each instance variable). The "dataDictionary" method is one of the two violations of dialogue independence that we permit (the other is the commit protocol: see below).

Once the "dataDictionary" and validation methods are installed, then the rest of the edit-related processing can be defined in objects remote to the logical model; e.g.

```
! Object methods!
specify
    self dataDictionary edit ! !
```

2.3. OO Paradigm

We believe in the principle of designing independent stand-alone code modules that can be plugged-together in many ways as a method for the rapid development of applications. Each object should have a clear "mission statement" and should not be required to exhibit behaviour that deviates from its mission statement.

This principle leads us to reject certain design alternatives such as:

- A logic-programming approach. The meta-level predicates of (e.g.) Prolog give us access to the internal structure of our programs. However, the *MYLE* framework is not an academic exercise: it addresses problems that we face in our consultancy work. A solution to the auto-configuration problem in an OO paradigm would be more immediately applicable than a logic-based solution. We similarly reject compromise solutions based on hybrid OO-logic programming languages since such languages are not currently commercial viable[3].
- Adding "connection pointers" to each object in the logical model. For example, consider the employment start-dates and employment end-dates contained in a employee instance. An invariant of these objects is that the start-date should be before the end-date. This could be implemented via connection pointers that link the start-date instance to the end-date instance. However, we would argue against confusing the re-usable concept of Date with methods/instance variables that refer to "Date-as-used-in-an-employee."
- Adding "reset" variables to each object. A reset variable is a copy of the variable at some prior time. That can be used to implement (e.g.) "undo": the current value is replaced with the last value (*lastValue* being one of the reset variables). Reset variables imply that some logical concept will now be extended to include methods for handling edit behaviour. This violates dialogue independence and we so we reject this option.

3. FRAMEWORK

This section describes the *MYLE* framework in some detail. Rather than confuse our logical models, we create a new hierarchy for generic glue objects. It is the glue that stores the connection knowledge such as inter-item constraints as well as the editing functions (reset values, entry-validation, etc).

3.1 Uniformity

The generality of the MYLE system comes from the uniformity of the objects it edits. We burrow an idea from logic programming for our core representation. Part of the BNF of structures in a logic program are shown in figure 2.

```
TERM      ::=  FUNCTION_SYMBOL {THINGS}4
THINGS    ::=  LOGICAL_VARIABLE
             | ATOM
             | LIST
             | TERM
LIST      ::=  {TERMS} nil
```

Figure 2: Partial BNF of logic programs. Note that list is usually implemented as a set of recursive terms.

If a structure is asserted, then the outer-most function symbol is a special (and is called the principle functor). Note the recursive pattern; structures may contain structures.: things can contains terms which can contain things. Much of the power of logic programming comes from the uniformity of the structures that it process. When a program is processing the internals of a structure, there are only a few “things” that can be found there.

---

3 Least the reader accuse of language bigotry, we note that we have developed and field logic-programming-based applications (Menzies, Black et al. 1992). One of us (the first author) is a logic-programming enthusiast but concedes that the current focus of the commercial world is object-oriented.

4 $\{X\}$ denotes zero of more Xs. $\{X\}^+$ denotes 1 or more Xs

We characterise editing as a recursive descent through nested structures and adapt the above BNF to a object model (see figure 3).

```
STRUCTURE      ::=  CLASS_NAME {THINGS}
                    INVARIANT
                    {ERRORS}
                    SERVICE_METHODS
THINGS         ::=  ATOM
               |    LIST
               |    STRUCTURE
ATOM           ::=  STRING
               |    NUMBER
LIST           ::=  FIXED
               |    DYNAMIC
DYNAMIC        ::=  DICTIONARY
               |    DYNAMIC_LIST
STRING         ::=  ONE_LINE
               |    N_LINES
DICTIONARY     ::=  {STRUCTURES}
DYNAMIC_LIST   ::=  {STRUCTURES}
FIXED          ::=  {STRUCTURES}
```

Figure 3: BNF of MYLE-able structures.

Like the internal structures of logic programs, our definition is recursive. OO has no concept of a logical variable (hence, no atoms). Our atoms' are strings or numbers and we replace functor with class_name. Note that we have added an invariant and a set of error messages to each structure. These will be discussed below (see *Semantics*). Service methods are a list of services that the logical model provides which are useful for editing (e.g. a get and put block for accessing and storing values).

Object-oriented lists are of at either fixed or variable length and its elements may be accessed either by an numeric offset or some symbolic reference. For example, a hash table (or Dictionary in Smalltalk-speak) is a variable length symbolic access list. A forms-entry screen for an employee could be described as a fixed-length symbolic access list with accessors defined for (e.g.) name, age, etc.

The main-loop of *MYLE* is (1) ask a data dictionary method to describe it logical model in terms of nested structures using the BNF of figure 3 then (2) pass these nested structures to a suite of classes that know how to edit such recursive structures. Editing then commences via a recursive descent of these structures. On arrival at an atom, a primitive edit function is called (e.g. pick-one-from-a-list).

```
Ed
   ECollection
      EAbstractDynamic
         EDictionary
         EDynamicList
      EFixedList
      EOneThing
   EItem
      EAbstractNof
         EAbstractOneOf
            EBoolean
            EOneOf
         ENof
      EAbstractString
         EOneLineString
         EString
      EMagnitude
         EDate
         ENumber
      ESmalltalkThing
```

Figure 4: The *MYLE* "glue" hierarchy.

3.1. SYNTAX

In our approach, the glue class(es) presents to the user either an editable thing, or a collection of editable things. Both a single thing or a collection of things respond to the same protocol so that one thing in a collection of things can be another collection of things and so on recursively ad infinitum. Editable items contain "container" pointers back to the thing that they are stored inside. If an object has no container, it is a *global object*.

For example, our current glue hierarchy is shown in figure 4. Figure 4 mimics figure 3 except that certain classes are added for pragmatic reasons (e.g. *EDate* and *EBoolean*). Editable items permit the editing of strings (*EAbstractString*) , editing a number (*EMagnitude*), editing a piece of Smalltalk code (*ESmalltalkThing*), or selecting n items from a list (*EAbstractNof*). List selection is further divided into selecting one item or "n" items. Booelan editors are a special kind of single-selection editors: there are only two options: one associated with true and one with false.

Editable collections are divided into editors of fixed sizes (e.g. an employee has a fixed number of fields) and editors of lists of dynamic size. *EDictionary* editors support the editing of lists of items where each item has a symbolic id (e.g. a hash table) while *EDynamicList* editors use a numeric offset to address their contents. The *EOneThing* class is a special service class used when a single item wants to edit itself (which it does by created a fixed list editor of size 1 and installing itself as the first item).

3.2. SEMANTICS

3.2.1. Black-Box Constraints

An editable item is removed from the global name space by a sequence of containers. The errors of an item is defined to be its own error strings plus the error strings of its containers (see figure 5).

These error strings are written by invariants stored by each class in the glue hierarchy. When a value is changed in an item the invariants of it and its container are

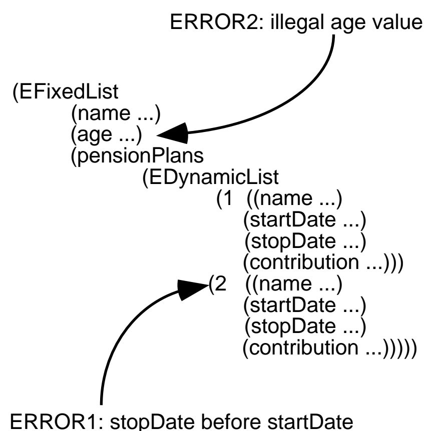

Figure 5: MYLE glue for an employee instance with error strings for age and pension plan #2. When editing pension plan #2's name, the displayed error messages would be "illegal age value" and "stop date before start date".

excepted. As a side-effect of this execution, extra error strings may be generated.

For example, figure 6 shows a *EFixedList* editing date details. The screen is in two halves: the display top section and the help bottom-section. The bottom section shows 3 lines. The currently active item is the "Day" line on line 1 of the top-section. Its errors are defined to be its own error strings, plus the error strings of the containing date editor. These errors are displayed as the help text for the active item (shown in the bottom portion of the screen in Figure 6). This text comprises the help text associated with the active item (see the line "The day of month") and is followed by (i) the error messages generated by the invariant of this *EFixedList* then (ii) the error messages associated with the currently active item. In this case, the *EFixedList* invariant has realised that May does not have 35 days and the currently active item has realised that the current value of 35 violates the valid range for its numeric entries (1 to 31).

3.2.2. Disadvantages of Black-Box Constraints

In a glass-box system, the system would have knowledge about the invariant of figure 3 (in a logic programming paradigm, the constraint would be expressed like everything else, i.e. the BNF of figure 2). *MYLE* has no knowledge of the internal structure of the constraints: they are black boxes that can be executed and which may generate new error strings. We can not query them and ask (e.g.) "if I was to message you, what other information would you gather?". This has some drawbacks. For example, we lack sufficient information to optimise constraint testing. Currently, if *X* is edited and *X* is contained by *Y*, then when *X*'s state is updated, we fire constraints for *X* and *Y*. This may be inadequate to detect certain constraints. For example, suppose we are editing a database definition contained within some global database definition library. Imagine that our constraint on *jobName* insists that the set of all jobs performed by employees should be stored in the *legalJobs* list. When can we delete a job name? Obviously, when no employee is performing that job. However, this could be a very expensive constraint to test since it implies a search through all the employees stored on the database. Our generic processor of "black-box" constraints lacks sufficient meta-knowledge of this constraint's scope to recognise this problem and (e.g.) suggest to the user that this particular edit be performed last thing at night so that the system can check it overnight.

Glass-box knowledge of the structure of the dependencies can be used to intelligently schedule constraint application. Reps & Teitelbaum extended conventional compiler technology of procedural language to add a "attribute equations" to symbols in a BNF. Further, they developed a generic syntax-directed editor (the Synthesiser Generator) that inputs a BNF and auto-configures itself. As a background process, the editor repeatedly applies the attribute equations to perform such tasks as type checking and incremental compilation. Nearly half the effort of their work (6 man-years) was spent evolving a generic control mechanism for the scheduling of the evaluation of the attribute equations. A poor controller could send the system into infinite loops as it chases circular state updates. Their solution was to pre-compute and cache "plans" of how to apply the attribute equations. Such a computation was only possible due to the glass-box nature of the system: the planner had access to the parse as well as some knowledge of the attribute equations (Reps and Teitelbaum 1989).

MYLE only has half the problem of the Synthesiser Generator. *MYLE*'s "attribute equations" (read invariants) only test values; they should not update state (exception: see the section on *zombies* below). Nevertheless, the lesson is clear: a generic control mechanisms for the intelligent application of invariants is a non-trivial problem.

3.2.3. Advantages of Black-box Constraints

Our case is not that glass-box constraints can be replaced by black-box constraints. There are many areas where glass-box constraints are essential; e.g. very intricate screen layout (such as placing the notes of a musical score on a display) require glass-box constraints (Freeman-Benson, Maloney et al. 1990). However, our introduction noted that a glass-box solution to interface auto-configuration (e.g. the Synthesiser Generator) implies changing our existing class libraries and imposing additional requirements on the development of any future classes.

The black-box solution proposed here is less restrictive and is an add-on to our current practice. Despite its comparative lack of meta-knowledge, our seemingly-simplistic framework permits the implementation of a surprisingly wide-variety of interfaces (see below). Further, our black-box constraints can be used in several generic and useful manner:

- If an editor contains error strings, we cannot save it.
- If an editor contains no errors, and a state change results in error strings being added to the editor, then the editor should pop-up an undo dialogue urging the user to reverse their last action.
- To perform a batch test of all the instances in the system, we ask their dataDictionary method for a description of their editor, then ask that editor to write alls it values back to itself. If this process generates any error strings, then the instance is corrupted in some way.
- If an instance is corrupted, then we should try and avoid it during processing of the application.
- Simple screen lay-out (e.g. a forms entry for defining an employee, 2-D displays, etc- see below).

Figure 6: A Date instance editor with errors. The cross on the left-hand-side of the title bar indicates that the contents of this editor have been changed, but not saved.

```
! Date methods !
dataDictionary
1. ^(EFixedList new: 'Date';
2.   add:  ((ENumber new: 'Day')
3.            help:   'The day of month.'
4.            id: #day min: 1 max: 31
5.            get:     [self dayOfMonth]);
6.   add: ((EOneOf new:  'Month')
7.            help: 'The month of the year.'
8.            id: #month
9.            options: [self class
                            monthNames]
10.           get:      [self monthName]);
11.  add:  ((ENumber new:  'Year')
12.           help:   'The current year.'
13.           id: #year min: 1900 max: 2100
14.           get:     [self year]);
15.  temporary: [:ed|
16.        Date newDay:(ed at: #day) value
17.             month:(ed at: #month) value
18.             year:(ed at: #year) value
19.             ];
20.  rule:      [:ed :value | ed temporary];
21.  putBlock:  [:ed| self day:
                     ed temporary day] ! !
```

Figure 7: Defining glue for editing a Date instance.

3.2.4. *Zombies*

Ideally, the glue editors utilise clean validation methods defined within the logical model. Since we cannot guarantee that all of the validation method in existing class libraries are not dirty, we have to handle the case where validation updates state prematurely (i.e. prior to completing a check that the state change proposed by the user is correct).

A *zombie* is an instance of the same class as the portion of the logical model being edited. It is called a zombie because it is not a live section of the logical model and is doomed for destruction when the editing process is finished. We only create it to access services of that class which (i) are useful to the editing and (ii) change state inside the instance.

For example, suppose the creation methods of an employee initialise defaults which we wish to use when prompting the user for input. Suppose also that the set methods for that class fill out other values as side-effects. We create a zombie employee to access the defaults and fill out the instance variables created by side-effect. We destroy the zombie after editing since we don't want a permanent copy of the state of this used and abused slave instance.

For another example, consider the "dataDictionary" method for Date shown in figure 7. Lines 2-14 define the three edit fields of a Date and give them unique identifiers (#day, #month, #year). Zombie construction is defined in lines 15-19 which use the symbolic names to access the values of the edit fields. The zombie is used by the invariant on line 20. As a side-effect of testing the invariant, the Date class tries to create a new Date instance using the current values in the editor. Validation methods inside the Date instance creation methods can generate error strings (which are trapped using the error handler defined in the appendix). Note that this method is the only intrusion of the *MYLE* system into the Date class. The zombie facility allowed us to access many of the services of the Date class without having to add multiple methods to Date.

3.2.5. *Libraries*

Often users process collections of instances, which we call *libraries*. For example, all the preferences for an application or all the employees known to the system could be stored together in a library. A library cached to disc is a database.

The ability to create more than one type of *X* implies a need to search through all *X* looking for a particular one. Hence, libraries support indexed searches. A "finder" object is created containing some/all of the fields of the goal object. We run over the library comparing values between library members and the finder. All matching library members are collected together in a dynamic list. The user can then select one or more and edit them using the standard edit process described above.

Note that the finder object will require many of the same services as the instance that it is searching for. For example, a finder for employee needs to ask the user for valid entries for age, postcode, and all other fields the user wants to search for. Note that in figure 1, our employee editor was subclassed into a employee details editor and a glue object for finding an employee. Common functions were stored in the super class and used by both subclasses.

A more general approach allows us to generate arbitrary finders without having to define a new class. The "dataDictionary" method is used to configure both an editor of an instance of the logical model but also a finder for these instances in a library. The finder is a list that stores one entry for each field the user is interested in. The full query is the and of all the fields. If a field is mentioned twice, then this is an or query. Figure 8 shows the Employee dataDictionary providing details for the configuration of an *EFixedList* that is an employee details editor and a *EDynamicList* that it is employee finder.

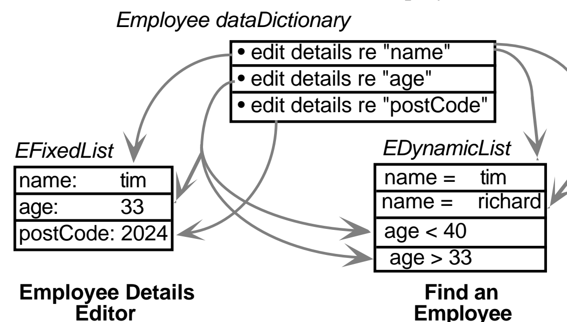

Figure 8: Using a dataDictionary method to configure both an editor of employees and a finder of employees in a library. The query in the finder would find both authors and reads as follows: `find employee with (name = tim or name = Richard) and (age < 40 or age > 33)`

3.2.6. *The Commit Protocol*

It is an error to store a zombie in a library since they are only temporary instances created for some specific purpose. This error can occur if (e.g.) an instance creation method calls a commit as part of creation.

In order to avoid this problem, we propose a commit protocol. The lowest-level commit routines should check that, prior to commit, the instance to be

stored has the appropriate commit permission. By default, commit is disabled for all objects.

```
!Object methods!
canCommit
    ^false ! !
```

If an editor wishes to commit an instance, then the instance has to give permission for the commit. We define a class hierarchy whose root is *GlobalObject* and whose subclasses store *canCommit* methods. An instance variable of *GlobalObject* is a boolean *isZoombie* value.

```
!GlobalObject methods!
canCommit
    ^isZoombie not ! !
```

Non-global objects cannot be committed unless they are contained in a *GlobalObject* instance. This matches our experience with application building. A Date instance by itself is not stored in a library. However, a Date within an Employee (e.g. pension-plan-start-date) can be committed as a side -effect of storing an Employee who contains this Date.

3.3. EDITING

3.3.1. Recursive vs Primitive Edits

Internally, the things-being-edited is a tree structure. Each node can contain multiple sub-trees. The user is presented with a visual representation of the node and portions of the sub-tree. The user can click on portions of the screen. Double-clicking starts a recursive edit:

1) If the item is another collection of editable items then *MYLE* spawns another window and edits the collection in that window. The close action of the parent window is disabled until all its child windows are closed.

2) If not (1) then we call a *primitive edit* action. Primitive edit actions collect a simple value and validate it. Figure 9 shows a primitive edit action for a *NOneOf* item. The legal items are thrown up in a list and allow the user to pick one of them. Other primitive edit actions throw up dialogue boxes and allow the user to enter in values.

Figure 9: Primitive edit action of *NOneOf* item. Note that the current value is ticked.

The primitive edit protocol is defined at the *EItem* level (see figure 10). If the editor has no current value, it collects it from the logical model. The current value is stored in the last value variable and a *basicEdit* is called to collect the new value (see figure 10). One pragmatic note: a flag *editCancelled* is set to true if the user hits the "Cancel" button. On editCancelled, the previous state of the instance is restored.

```
! EItem methods !
edit
     "Save the last value in the lastValue
      method. Call basic Edit."
     |temp tempLastValue|

     currentValue isNil
       ifTrue: [currentValue := self value].
     tempLastValue := lastValue.
     lastValue := currentValue copy.
     editCancelled := false.
     temp := self basicEdit.
     editCancelled
       ifFalse: [
         modified := true.
          ^self accept: temp].
     lastValue := tempLastValue.
     ^false! !
```

Figure 10: Primitive edits

Subclasses define the basicEdit action. For example, figure 11 shows an *EOneLineString* method that collects a single line of text and the *editCancelledAction*.

```
! EOneLineString methods !
basicEdit
     "Initiate the basic editing process. If
      the user cancels the edit, return the
     current value. Otherwise, return the
     new value."
     |temp|
     temp := Prompter prompt: prompt
                      default: currentValue.
     (temp isNil or: [
      temp trimBlanks size = 0])
         ifTrue: [ ^self editCancelAction ].
     ^temp! !

! EItem methods !
editCancelAction
     "Perform the appropriate actions for a
     cancelled edit."
     editCancelled := true.
     ^currentValue! !
```

Figure 11: Basic Edit for one line strings.

The protocol for accepting new values is generic to all items. Figure 12 shows that protocol. A quick pre-processor (line 2) tests that the input does not contain obvious syntactic errors (e.g. letters in a string representing an integer). The input is then converted into Smalltalk object (line 4) and tested for validity (line 6).

```
! EItem methods !
accept: input
    "If the input is valid, and we can
     compile it, then compile it and set
     current value to this result. Collect
     all errors generated the owner. Return
     true if the value was accepted and
     updated and false otherwise."

    |result|
1.     errors := Set new.
2.     (self canCompile: input)
3.         ifTrue: [result :=
4.                self compile: input.
5.                self currentValue: result.
6.                (self valid: result)
7.                     ifTrue: [^true ]].
8.    ^false! !
```

Figure 12: The accept protocol.

Subclasses implement the details of this protocol. For example:

ENof The user can only pick from one value from a list of legal values. Hence, "canCompile:" and "valid:" is always true and "self compile: input" just "input.

EMagnitude The user's input is a string. After "canCompile" tests for non-numerics, "self compile:" calls methods in the Number hierarchy to turn a string into a number. "Valid:" then tests that the number is within the valid ranges of min/max.

3.3.2. *Layout*

The "dataDictionary" method supplies enough information to automatically configure the data validation routines and the cross-item edits. It also gives us enough information to automatically layout the screen in several different ways

3.3.2.1. *Simple List Display*

For each item in the top-most container, assign one line to the item and display the lines in a list box. For example, figure 6 showed a simple list display of the figure 7 data dictionary. The width of the screen is computed from the size of the prompt strings and the maximum size of the legal edit values. The user can click on lines in the screen to initiate an edit.

3.3.2.2. *Table Display*

Simple list editors display the top-level container, one item per line. This is undesirable in an application with multiple many-levelled structures. Each edit at any level forks another window and soon the screen would be cluttered and unmanageable.

A table display permits N (N >= 0) items per line. If the nth item on each line occupies the same width, then the display looks like a 2-D spreadsheet. Table displays augment the list display by adding a formatting prefix and suffix to each item. Adding a (e.g.) line break then become a matter of insert the ascii value for carriage return into the formatting suffix. The table display tracks the mouse movements and when the user single clicks, the simple list display edit protocol defined above is performed.

Visual clues are added to the table display to indicate the container of the currently active item. The user can spawn a new edit window on that container.

3.3.2.3. *General Screen Display*

General screen displays take table displays one step further. No longer is the display restricted to ascii text. Knowledge of graphic widgets is added to allow the system to add (e.g.) radio buttons to store one-of actions. Consider the table layout screen shown in figure 13. We could have defined it using the data dictionaries of figure 14.

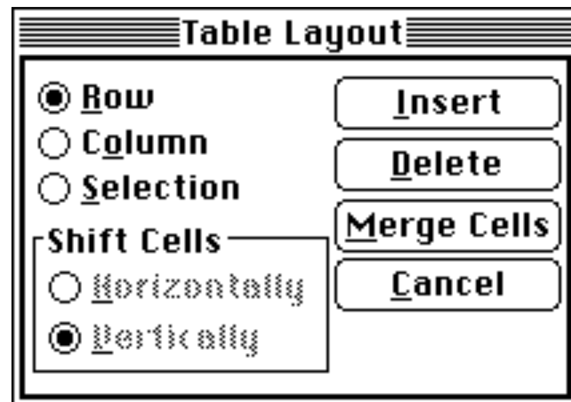

Figure 13: A typical screen from a commercial application.

```
! WordProcessorToolBox methods !
dataDictionary
^(EFixedList new: 'Table Layout')
    add: ((EOneOf new: 'Shift Cells')
          help:    'Cell movement direction'
          id:      #shiftCells
          options:[#(Horizontally
                     Vertically)]
          get:      [#Vertically]
          enabled: [:ed| ed shiftEnabled]);
    add: ((EOneOf new:  '')
          help:     'Selection scope'
          id:       #scope
          options: [#(Row Column
                      Selection)]
          get:      [#Selection]);
    add: ((EOneOf new: ''
          help:     'Perform this command'
          id:        #command
          options: [#(Insert Delete
                     'Merge Cells' Cancel)]
          get:      [#Cancel];
      putBlock:  [:ed| self
              tableLayOut:(ed at: #command)
              scope:  (ed at: #scope)
       shift:      (ed   at:   #shiftCells)]
! !
```

Figure 14: Data dictionary for figure 11.

3.3.2.4. *Network Display*

How would we edit a 2-D network such as that in Figure 15? We could represent the graph as the nested lists of Figure 16. We could then define a set of simple list displays that process the contents of each node.

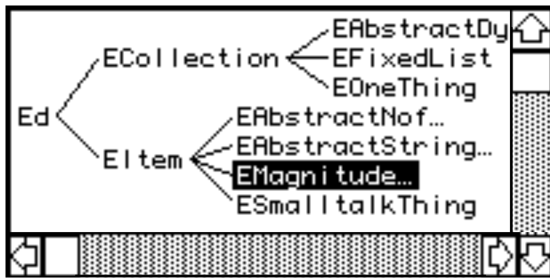

Figure 15: A 2-D graph display.

Alternatively, after defining the list displays, we could present the structure in a graphical editor exactly like figure 15. An automatic lay-out program and the ability to select a node would be the only additional essential requirements for converting the nested list displays into a graphical editor. All the rest of the logic (such as nodes with sub-trees can not be deleted without deleted the sub-tree elements first) would be available from the validation methods of the logical model.

The user could browse the network, fork editors for individual nodes using the simple-list displays, spawn editors of sub-trees, etc, etc using mostly the services provided by the simple-list glue.

3.3.3. *Reset Variables*

If the user wishes to cancel an edit, then any changes they made must disappear. The simplest way to do this is to take a temporary copy of the portion of the logical model being edited, and edit the contents of the logical model in temporary isolation to the model.

In practice, 2 copies are made. These become (i) the current value and (ii) the last value. Whenever the user edits an item, the old value is cached in the last value (see figure 10). If the user requests an "undo" then the last value is retrieved and assigned to the current value. On "save", the current value is written back to the container object. If the container object is a global object, then it is written back to the logical model. On "cancel", the editor quits without saving.

Note that the users changes are only really committed on the save of a global object. Consider one scenario of an edit of pension plan #2 in figure 5. The employee glue shows the user three fields: name, age, and pensionPlans. If the user selects pensionPlans, a *EDynamicList* editor is forked up on the pension plans. When the user quits and saves the pension plans editor, any changes are written back to the employee editor. If the user then cancels the employee edit, then the intuition is that the changes to the pension plans should be discarded. Is this a reasonable assumption? If we permit the pension plans editor to commit its contents back to the actual employee instance, then the "cancel" action of employee editor loses its meaning. However, if we forbid lower-level editors such as the pension plan editor to commit, then the one action "save" has two meanings: (i) commit outer-level edits and (ii) return lower-level editor contents to a higher-level editor contents. Currently, we have no resolution for this inconsistency.

3.3.4. *Temporary Errors*

If the user requests an action that is impossible, the state of the edited items is left unchanged and a *temporary error* string is generated.

```
(Ed
        (ECollection
                (EAbstractDynamic
                        EDictionary
                        EDynamicList)
                EFixedList
                EOneThing)
        (EItem
                (EAbstractNof
                        (EAbstractOneOf
                                EBoolen
                                EOneOf)
                        ENof)
                (EAbstractString
                        EOneLineString
                        EString)
                (EMagnitue
                        EDate
                        ENumber)
                ESmalltalkThing))
```

Figure 16: Figure 15 as nested lists.

Temporary errors are errors displayed in the help-section, but not cached in an errors list. Consequently, the user sees them only until the next screen update (e.g. when the user changes the currently active item or they try a re-edit of the currently active item) after which time, they disappear.

4. APPLICATIONS

This section examines the issue of what can be done with the framework. This is a theoretical discussion: none of the following applications exist. The claim being made by this section is that a wide-variety of seemingly diverse and complex applications could be built as extensions to the current framework. Some of the proposed applications are dependant on other applications (see figure 17).

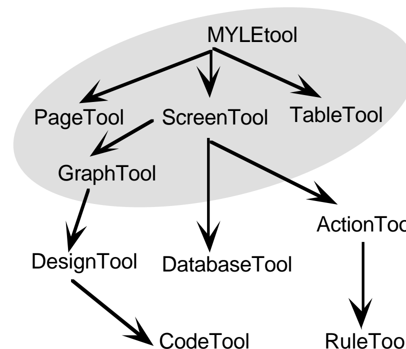

Figure 17: Applications using the *MYLE* framework. The ones in the shaded region represent the basic kit.

4.1. ScreenTool

At the lowest level, with no extension to the above framework, *MYLE* could simplify interface design. Existing screen painters could be augmented with the glue single-item hierarchy and use the services of that hierarchy to augment existing field and inter-item validation routines. The screen painters could have a "suggest" button which calls the MYLE screen layout routines. The output of those routines could be mapped back into the screen painters' internal structures and presented to the user as a first-pass screen design.

4.2. MyleTool

The dataDictionary structure has certain repeated features (e.g. the *add:* methods are very similar). A *MYLE* editor could be defined to edit data dictionaries.

4.3. PageTool

Often the screens of an application can be viewed as "N" pages. *PageTool* defines a "page" to be a simple list display with a header line. One window could be shown with the current page as a simple list. All the other pages could be available via a list selection. When another page is selected, the header on the window could change to the new page header and the list display change to the new page.

Their are two advantages of the *PageTool*:

- Instead of having 1 window per screen, the user sees only needs on window to be on screen at any time.
- Page switching logic is simpler (since it is all local to one window). This could be used to implement (e.g.) an intelligent questionnaire that (i) only allows page switching when certain fields are completed and (ii) only presents the relevant pages to the user (i.e. don't display the visa details page to Australian citizens).

4.4. TableTool

Spreadsheets have proved a useful tool for many business modelling purposes. *TableTool* is an extension of the table display that supports the processing of large tables with user-defined dependencies between cells (e.g. calculations).

4.5. GraphTool

Editors could be created for arbitrary graphs and for simulation programs. User defined node positioning would have to be added.

4.6. ActionTool

Internal to the finder object would be a parse tree representing a query. For example, the query of figure 8 would look something like figure 18; i.e. a tree of nested commands. We could execute this as a program in a top-down, left-to-right manner; i.e. our trees have become a structure chart that can process user-specified commands. Combined with the libraries, we then have the basis for a generalised report writer for arbitrary objects.

ActionTool would need to define "action nodes" (if, repeat, etc) and a name space convention for variables in the structure chart.

```
(and  (E.name = X)
      (or      (X = tim)
               (X = richard))
         (E.age = Y)
         (or   (Y < 40)
         (Y > 33)))
```

Figure 18: A code-tree. Code trees are nested *EDynamicLists* whose first item is a procedure that filters the output of the other items (e.g. negates, sums, averages, performs a disjunction or conjunction, etc).

4.7. DatabaseTool

A report writer, a finder for instances, and user-friendly editors for collection of items represent some of the tools needed to a generalised object-oriented database. If we add a disc-storage mechanism then we can rapidly build single-user databases. Our "dataDictionary" methods serve as DB schema definitions and the *ActionTool* could serve as a DB trigger language

Naturally, additional features would be required such as multi-user login, record locking, instance indexing, rollback and audit trials in order to build an industrial strength DB system. However applications that require simple disc-storage, could use a small extension to *MYLE* rather than suffer the overheads of a full OO database.

4.8. RuleTool

Object-oriented expert systems require a search mechanism over collections of instances. The *ActionTool* provides such a mechanism. Rules could be written that permit a user to customise the high-level processing of the system in a declarative manner. Rule conditions and actions could call arbitrary methods. Intricate processing could be hidden from the user behind high-level object interfaces to complex processing.

In figure 18, the operator was any message symbol that could be sent the LHS operand. We could limit this to the high-level interfaces of each object by adding *publicInterface* method to each object. By default, all methods are in the public dictionary and all parameters could be of any type (a protocol defined at the Object level). Subclasses could specialise this interface to (i) include only the high-level methods for the objects and (ii) the valid types for the parameters. Once defined, the *RuleTool* editor could define the LHS operand first, then use the publicInterface method of that operand to control ht editing of the rest of the structure.

A major issue in expert systems is search. *RuleTool* would require an indexing/search schema over the libraries.

4.9. DesignTool

A pressing need in contemporary OO practice is better design tools. Consider the graphical notation used by many of the current methodologies. If we characterise such notations as a graph editor with an expert system attached to criticise the design, then a combination of *RuleTool* and *GraphTool* could be used to implement a generic OO design tool.

The icons of the nodes in the networks could be customised to reflect the various whims of the different authorities. Rules could include heuristics such as Myer's one method rule[5] or Wirfs-Broch's sub-system connection rule[6].

4.10. CodeTool

Code generation could be characterised as a report written using *ActionTool* that runs over the *DesignTool* networks. If *MYLE* is written in language X, then code

---

5 Beware the object with only one method. A procedural programmer has been here and has tried to create a sub-routine.

6 Sub-systems with too many contracts with other sub-systems complicate the design. Review the design and decrease the dependency between sub-systems.

generation for language Y may be restricted to class and method headers. However, code generation for language X code include methods as well.

Code generation could also be achieved by using *MYLE* as a syntax-directed editor. Suppose that the BNF of a program was defined as a nested set of *EDynamicList* editors. Constraints would insist that the structure being edited conforms to the BNF of the program (e.g. a procedure must have a "begin" and "end" statement). Such a structure could be presented in a table display. Code generation could be achieved by a generic recursive-descent report that works from the outer container of the "program" to the lowest levels.

5. CONCLUSION

A general framework for user-interaction with objects of arbitrary complexity has been proposed. "Editing" is a process of interacting with glue classes who understand how to process things and lists of things (which may contain nested glue). Editable items are augmented with invariants that, when executed can write error strings into the editor. Much of the processing can be characterised in terms of recursive processing of the glue structures or querying the number of error messages contained in an glue class. Class specific processing can be isolated in validation methods. The additional code needed in objects to utilise this system is:

- a "dataDictionary" method
- a canCommit protocol
- a publicInterface method (*RuleTool* only).

A set of applications were proposed that represent minor extensions of the framework and cover a wide-variety of software systems including interactive design tools, code generators, and expert systems.

Note every application interface would be achievable using *MYLE*. The claim being made here is that a 80% solution could be achieved in minimal time. If further extensions are required beyond the *MYLE* framework, then the modularity of the system implies that the building blocks would be available for further customisation.

A test of the framework would be to build the applications listed in section 4, then use these applications to attempt arbitrary applications. The framework would be considered a failure if the time taken to develop these arbitrary applications was not significantly shorter (say, by one order of magnitude) than time estimates for their development based on conventional programming techniques.

The limit of the framework is the nature of the invariants. The system has no meta-knowledge of these invariants: they are merely black-box code blocks that the framework activates at certain points in the processing. We believe that it would require a major re-write of existing class libraries and the imposing of strict coding standards on future class development to make this knowledge available. We offer our compromise design as an alternative: we give the editors a little knowledge and define some generic processing loops based on our recursive syntactic structure and black-box constraints. The open research issue is how far we can extend this formalism. Our examples presented here suggest that it may be further than one would first believe.

6. REFERENCES

Borning, A. (1981). *The Programming Language Aspects of ThingLab: a Constraint-Oriented Simulation Laboratory.* **ACM Transactions on Programming Languages and Systems 3**(4 (October)): 353-387.

Freeman-Benson, B. N., J. Maloney, et al. (1990). *An Incremental Constraint Solver.* **Communications of the ACM 33**(1): 54-63.

Hartson, H. R. and D. Hicks (1989). *Human-Computer Interface Development: Concepts and Systems for its Management.* **ACM Computing Surveys 21**(1, March): 5-92.

Menzies, T. J. (1990). *Isa Object Part-Of Knowledge Representation?* **Proceedings of AI '90**, November, Perth, Australia,

Menzies, T. J. (1991). ***Beyond the MVC Triad: Quality Assurance via Interactive Specification Editors.*** **Tools 3: Proceedings of the third International Technology of Object-Oriented Languages & Systems conference**, Sydney, Australia,

Menzies, T. J. (1991). *Isa Object Part-Of Knowledge Representation (Part Two)?* **Tools 4: Proceedings of the third International Technology of Object-Oriented Languages & Systems conference**, Sydney, Australia.,

Menzies, T. J. (1993). *The Complexity of Model Review.* Dx -93: **The International Workshop on Principles on Model-Based Diagnosis,** Absersywath, Wales, UK,

Menzies, T. J., J. Black, et al. (1992). *An Expert System for Raising Pigs.* **The first Conference on Practical Applications of Prolog,** London, UK.,

Reps, T. W. and T. Teitelbaum (1989). **The Synthesiser Generator: A System for Constructing Language-Based Editors.** Springer-Verlag.

Urlocker, Z. (1989). *Abstracting the User-Interface*. **Journal of Object-Oriented Programming** (Nov/Dec): 68-74.

7. APPENDIX: PATIENT ERROR HANDLING

In a Smalltalk context, validation methods are written without knowledge of the *MYLE* system. When an invariant violation is detected, the methods just call

```
^self error: message
```

where message is the error string. In standard Smalltalk, the "error:" method is defined in the root Object. Here, we modify that method such that, on error, the user is not dropped into the user-hostile debugger (where they can, potentially, destroy the entire system).

```
! Object methods !
error: aString
   "On error, call the friendly debugger"
   ^Friend current
        friendlyWarning: aString
        from: self!

basicError: aString
      "New call to the standard debugger."
     Process queueWalkback: aString
         makeUserIF:CurrentProcess isUserIF
         resumable: false!

friendlyWarning: error from: aFriend
     "By default, whenever an object gets a
      friendly warning,  it calls the main
      Smalltalk debugger. Subclasses that
      handle  user-friendly  dialogues  re-
      define this method for a more user-
      palatable error behaviour."
     ^aFriend basicError: error! !
```

When testing invariants, *MYLE* subclasses pass a code block representing the test to a friendlier debugger.

```
! Ed methods !
test
     "Remove all existing errors.
      Execute the invariant. Now, patiently
        execute the invariant. Any   errors
        re now current errors
        |test|
        test   :=   [invariant   value:   self
                                value: nil]
        errors := OrderedCollection new.
        self patientlyTry: test !!

! Object methods !
patientlyTry: action
     Friend current do: action for: self.! !
```

The user-friendly debugger is the currently active instance of the "Friend" class. Friends store a "noticeBoard" variable. On error, the Friend tries to write a friendly warning to the noticeBoard. *Ed* subclasses, on receiving such a warning, add it to its list of error strings.

```
! Ed methods !
friendlyWarning: warning from: aFriend
     "Handle  a  friendly  warning  from  a
      friend. Add the warning to the list of
      errors."
     errors add: warning ! !
```

Other classes could try some error recover action. For example, a dictionary could be stored whose keys are the various error strings that can be generated and whose values are fix-methods that could repair that error. Note that the error recovery could also crash leading to recursive calls to the error handler. The friend has limited patience. If it is repeatedly called more than "self patience" number of times, it despairs and calls the standard Smalltalk debugger.

```
!Object subclass: #Friend
   instanceVariableNames:
      'nErrors noticeBoard '
   classVariableNames:
      'One '
   poolDictionaries: '' !

! Friend class methods !
current
        One isNil ifTrue: [One := self new].
        ^One! !
new
     ^super new initialize! !!

patience
     "Return the number of times that Friend
      will issue friendly warnings prior to
      calling the Smalltalk debugger."
     ^1! !

! Friend methods !
do: performAction for: aFriend
     "Patiently   try   to   execute   the
      performAction, sending any complaints
      back to aFriend."
     self noticeBoard: aFriend.
     self morePatience.
     performAction value.
     self release.
     self morePatience.!

friendlyWarning: message from: object
     "Send the error message to the notice
      board  (if  it  exists)  unless  my
      patience has been exhausted."
     noticeBoard isNil
         ifTrue:[
            ^object basicError: message].
     nErrors := nErrors + 1.
     nErrors >  self class patience
         ifTrue: [
        "we have run out of patience
         with this notice board. It
         must be untrustworthy. Get
         rid of it."
              self release.
              ^object basicError: message].
     ^noticeBoard  friendlyWarning:  message
                               from: self!

initialize
        nErrors := 0.!

morePatience
     "Tell your friend to have more patience
      when   dealing   with   errors."
      nErrors := 0.!

noticeBoard: newValue
     noticeBoard := newValue !

release
    "Forget the value of noticeBoard, thus
     deactivating friendly warnings."
     noticeBoard := nil! !
```

Note that, once installed, this scheme works transparently in the case of normal calls to the error handler. Such calls are not made in the context of a *patientlyTry:*. Hence the *noticeBoard* of the current *Friend* is nil and *basicError:* will be used during error handling.

The advantage of this approach is that validation methods can be coded as per normal in the classes of the logical model and used for either black-box constraints or as per normal error calls.