JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2002**

Vol. 1, no. 4, September-October 2002

Eiffel Assertions and the External Structure of Classes and Objects

Peter Horan, School of Information Technology, Deakin University, Australia

Abstract

The "external structure" in an object oriented system refers here to the graphs of objects and classes. The class structure graph or class model is derived from the object structure graph or object model, and in this operation structural information is lost, or never made explicit. Although object oriented programming languages capture the class model as declarations, contradictory assumptions about object model properties may be made introducing faults into the design. Consistent assumptions about the object model can be specified in the code using assertions such as Eiffel's invariants, preconditions and postconditions. Three examples specifying the external structure are considered.

1 INTRODUCTION

I have been challenged on occasions by my colleagues for using the words "higher level" as distinct from "lower level" in relation to code and design. By these terms, I was trying to convey the fact that in the former case, I was talking about structure between classes and objects, and in the latter, about the internal structure of code implementing routines. I was strongly challenged for my usage because my listeners were hearing that external structure was more important than internal, rather than that it frequently determined details of the code to be developed.

On casting around for a better term, I lighted upon the terms "internal structure" and "external structure". "Internal structure" is the concept of code structure that has been with us for many years, the result of methods of structured programming. "External structure" actually refers to two kinds of structure, the object model and the class model. The external structure of the object model refers to the graph of links (the arcs) between objects (the nodes) that are established at run-time when the system executes. This structure is dynamic. The external structure of the class model refers to the graph of associations (the arcs) between classes (the nodes) and is static.

It is the purpose of this paper to explore the use of external structure in design by examining the role it plays in several examples. The assertion mechanism of Eiffel [Meyer92] is used in these examples to specify structural properties of the system of objects, so that in mapping models to code, these properties can be verified as the models develop. It is shown that assertions about structure remove ambiguities and contribute to the seamlessness of code and object model.

Cite this article as follows: Peter Horan: *Eiffel Assertions and the External Structure of Classes and Objects*, in Journal of Object Technology, vol. 1, no. 4, September-October 2002, pages 105-118. http://www.jot.fm/issues/issue_2002_09/article1

2 OBJECT ORIENTED METHODS

Object oriented methods start by modeling underlying data structure, partitioning the data into objects, mapping this structure into classes and then building a superstructure of routines, each belonging to the class of the data it manipulates. This work begins with possible systems of objects, seeking appropriate external structure for the object model, and mapping this to the class model when the system is properly understood. Issues addressed by modeling include packaging of data, interconnection of data and setting up of objects for later use. In reasoning backwards from the system of objects, eventually to the code, the danger is that the code depends on assumptions about the structure of the system which are forgotten. Methods of object oriented design will be more powerful if assumptions about the external structure of a system of objects are specified as properties of the system and recorded as assertions in the code.

Three examples are presented here to illustrate the role of external structure in software design. The first shows the replacement of code with the building of external structure, the second compares procedural, recursive and structural solutions and how efficiency can be regained by use of appropriate structure, and the last is an example of specifying code to develop the dynamic structure required by the object model.

3 MODELS AND THEIR RELATIONSHIPS

We can model an object as a memory structure containing simple data values, references to other objects and references to routine entry points. With this knowledge, the class which creates such an object can be discerned. A class is required for each distinct kind or type of object.

The class is derived from an object as follows: for each simple data value, a class has an entity (a name in the software text that denotes a run-time value) of a suitable type; for example, for integer values, entities of the integer type; for references to objects, a class has entities of references of a type from which the target object inherits. This information is sufficient to encode the object model and represents the first lines of code to be written. For example, the object structure in Figure 1[1] implies the class structure in Figure 2 which is mapped to the Eiffel code in Listing 1[2].

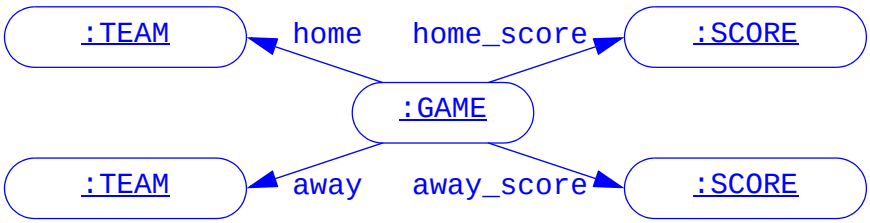


Figure 1. Some objects in a system

1. I have not used the rectangular notation of UML for objects but used rounded rectangles, after OMT, to distinguish them visually from classes.
2. This analysis ignores class hierarchy and will lead to the flat form of the class [Meyer96].

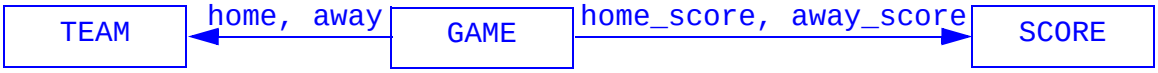


Figure 2. Classes which define the objects

```
class GAME
feature
   home, away: TEAM
   home_score, away_score: SCORE
end
```

Listing 1. The GAME class and its neighbourhood encoded in Eiffel

These three views, object model, class model, code, all need to be considered in design. None conveys the whole story. Note that the object model in Figure 1, a graph of objects and links, is mapped precisely to one class model in Figure 2, a graph of classes and associations, whereas the class model can be mapped back to many object models. For example, nothing in the class model or the code, as it stands, specifies the existence of any of the SCORE or TEAM objects. There is nothing to indicate precisely what system of objects is meant to exist at any particular time. For example, a routine used early may depend on the absence of an object, whereas one used later may depend on its presence. The class model is silent in such cases.

The mapping from a class and its associations to code is not lossy because the class model can be recovered from the code, but information about the clients of a class is less accessible in the code than it is in the class model. The arrow leaving a class in Figure 2 is a declaration in the code in Listing 1. An arrow tail and a matching declaration represent the same information, the supplier of a service. However, the arrow head, representing a reference by a client to a class, is filtered out and does not appear in the code. Information about clients is encoded indirectly as references to suppliers.

As the system of objects is what we require at run-time, how can we specify our models to match that system more accurately? How can we capture structural information that the system depends on? The assertion mechanisms in Eiffel [Meyer96] can be used to capture structural information, which may be otherwise invisible. For example, in Eiffel we can use an invariant clause to specify the existence of objects. The invariant clause

```
invariant
  scores_exist: home_score /= Void and away_score /= Void
```

in the GAME class ensures the existence of two SCORE objects attached to every GAME object, a static property of the class. Invariants can also specify dynamic structure as in

```
consistent: home /= Void implies home.score = home_score
```

It may also be appropriate to specify structure by using postconditions when routines are defined.

Example 1: Building a system at run-time

The first example is provided to show that there are gains to be made from considering the external object and class structure, and capturing its properties in the form of assertions. Three solutions, a procedural one and two closely related structural solutions are considered.

When a system starts, the first task is to build the system of objects, based on information in the classes. As each object is allocated memory, variables are initialized and cross references to other objects are set up. In the simplest case, the run-time system creates a root object, allocating memory space for it. An initialization routine gives initial values to any simple variables. Cross reference values are initialized by creating new objects and allocating references to them. This can proceed, growing a tree of objects from the root. For example, when the `GAME` object in Figure 1 is created, it can be initialized by creating the two `SCORE` objects at the same time.

The `TEAM` objects are not built at the same time because they may exist independently of `GAME` objects whereas `SCORE` objects do not. Precisely how and when a team is selected in a game is outside the scope of this discussion, but the mechanism `select_home_team` belongs in class `GAME` and is specified by Eiffel code in Listing 2. The feature `select_away_team` is similarly specified.

```
select_home_team(t: TEAM) is
  require
    argument: t /= Void
  ensure
    linked: home = t
  end

my_score(t: TEAM): SCORE is
  require
    argument: t /= Void and then (t = home or t = away)
  ensure
    consistent: (t = home implies Result = home_score) and
                (t = away implies Result = away_score)
  end
```

Listing 2. Specification of `select_home_team` and `my_score`

Specifications written in Eiffel shows the preconditions which are assumed in the require clause and the postconditions which are guaranteed in the ensure clause.

Later, we wish one team to be able to access its own score in a game and also the score of its opponent. This information is not contained in the structure shown in Figure 1. One solution is procedural (that is, relationships between data are maintained by routines and not in structure) and depends on the `GAME` object to associate the home team with the home score and the away team with the away score. We can determine which score is which by specifying a query `my_score` (a query is a function with no side effect) in the `GAME` object (Listing 2).

Similarly, `opposing_score(t: TEAM): SCORE` returns the other result. This code depends on `t` referring to the `home` or `away` team being allocated to the game as the precondition indicates.

A second solution is to initialize links between objects (Figure 3). When a team is selected into a game, these links to the scores are also set by using `set_scores` in class `TEAM` (Listing 3).

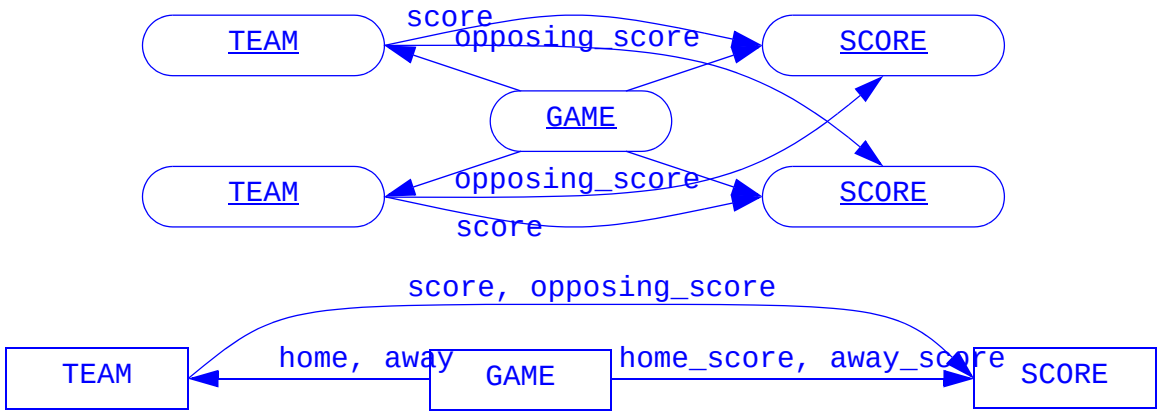


Figure 3. Access from `TEAM` to `SCORE`

```
score, opposing_score: SCORE
set_scores(s, o_s: SCORE) is
  require
    arguments: s /= Void and o_s /= Void
  ensure
    linked: score = s and opposing_score = o_s
  end
```

Listing 3. Specification of `set_scores` in class `TEAM`

The requirement that a team is linked to the correct score can be specified in `GAME` by modifying the postcondition of `select_home_team` and `select_away_team`. Listing 4 shows the specification of `select_home_team`.

A third solution, a variation of the second, is to set up links between the two scores at the time of creation so that each score is linked to its opposing score without involving the game or team object (Figure 4) This change means that the feature `opposing_score` is moved from `TEAM` to `SCORE`, that the postcondition of `set_home_team` in `GAME` is simplified accordingly and that the invariant

```
cross_linked: opposing_score /= Void implies
                  opposing_score.opposing_score = Current
```

is added to the `SCORE` class.

```
select_home_team(t: TEAM) is
  require
    argument: t /= Void
  ensure
    linked: home = t
    consistent_score: home.score = home_score
    consistent_opposing_score: home.opposing_score = away_score
  end
```

Listing 4. Specification of `select_home_team` in class `GAME`

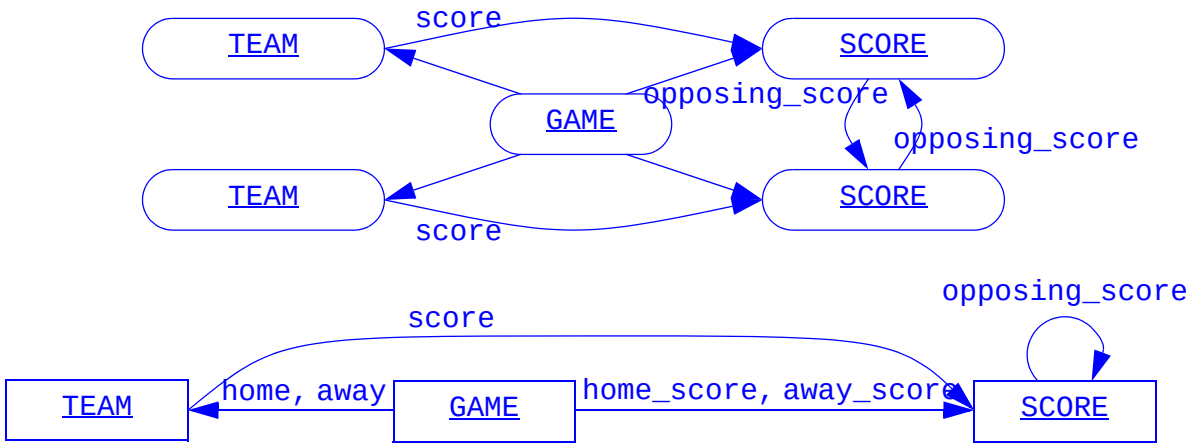
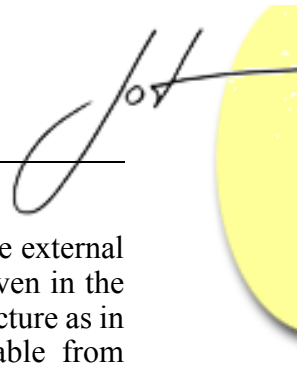


Figure 4. Revised access from `TEAM` to `SCORE`

What are the advantages and disadvantages of each solution? The procedural solution (Listing 2) avoids making the object model complex, but the code itself may be complex as the postcondition shows, and underlying assumptions may not be made explicit. The developer has the choice of recomputation each time the query is called, perhaps sacrificing performance, or of caching the result, perhaps leading to inconsistency if the result goes out of date. If there are many like queries, the maintenance cost may be large. In any case, no assertions have been made about the object model upon which the developer can rely. Perhaps this is the greatest disadvantage of the procedural approach, yet, not obvious for its omission.

The two structural solutions make the object model more complex by adding links leading to more complex initialization code, but dependent code is usually simplified. Identifying and specifying structure is a major gain and avoids the need for recomputing associated queries. Making the links explicit also allow the external structure to be specified as postconditions and invariants affording protection from conflicting modifications. Lastly, linkages between objects are naturally local – what you see in a declaration is what you get – leading to stability and maintainability. As for the choice between the two structural solutions (i.e., as shown in Figure 3 and Figure 4), the latter is to be preferred in my opinion, because more structure is built early.



The point of the example is that there are gains to be made from considering the external object and class structure, and capturing its properties in the form of assertions – even in the procedural case. Working solutions can be obtained without considering external structure as in the first solution above. But, the two structural solutions – not readily obtainable from procedural thinking – allow the capture and protection of structural assumptions upon which later behavior depends.

Example 2: Structure and efficiency

In this example, a polymorphic system is used to regain performance lost by conversion from a procedural to a recursive system. Three versions are compared, nested loop, recursive and chain of polymorphic objects.

Consider the problem of computing the distribution of total runs of a team of eleven batsmen in the game of cricket from the distributions of runs scored by each batsman in the previous (say) ten innings. So, every possible total, computed by taking one score from each batsman, is used to build a histogram. In effect, we are convolving eleven individual distributions together. To evaluate the result in a procedural manner requires eleven nested loops to compute all the possible totals. The two innermost loops appear in Listing 5.

```
from
  i_10 := 1
until
  i_10 > batsman_10.scores.count
loop
  sum_10 := sum_9 + batsman_10.score.item(i_10)
  from
    i_11 := 1
  until
    i_11 > batsman_11.scores.count
  loop
    sum_11 := sum_10 + batsman_11.scores.item(i_11)
    distribution.put(distribution.item(sum_11) + 1, sum_11)
    i_11 := i_11 + 1
  end
  i_10 := i_10 + 1
end
```

Listing 5. Innermost two of eleven loops for computing a distribution

Alternatively, recursion could be used, as shown in Listing 6. The recursive structure may be modelled as a chain of `BATSMAN` objects, all with access to the `DISTRIBUTION` object, as in Figure 5.

Lastly, one can build a chain of `BATSMAN` objects terminated by a `DISTRIBUTION` object (Figure 6), which eliminates the need to test during the computation by replacing the test with a polymorphic call. This is achieved by making classes `BATSMAN` and `DISTRIBUTION` inherit from a common ancestor, `ELEMENT` as shown in Listing 7.

```
class BATSMAN
  feature
    next: BATSMAN
    distribution: DISTRIBUTION
    make is
      ensure
          distribution /= Void
      end
    build(s: INTEGER) is
      local
          i, sum: INTEGER
      do
        from
          i := 1
        until
          i > scores.count
        loop
          sum := s + scores.item(i)
          if next /= Void
          then
            next.build(sum)
          else
            distribution.build(sum)
          end
          i := i + 1
        end
      end
end

class DISTRIBUTION
  creation make
  feature
    distribution: ARRAY[INTEGER]
    make is
      ensure
        distribution /= Void
      end
    build(s: INTEGER) is
      require
        distribution /= Void
      ensure
        updated: distribution.item(s) = old distribution.item(s) + 1
      end
end
```

Listing 6. Computing a distribution using recursion

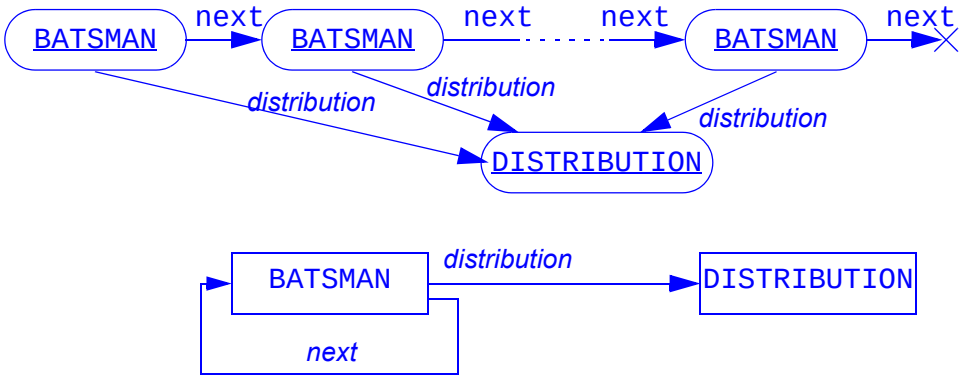


Figure 5. Structure for computing a distribution recursively

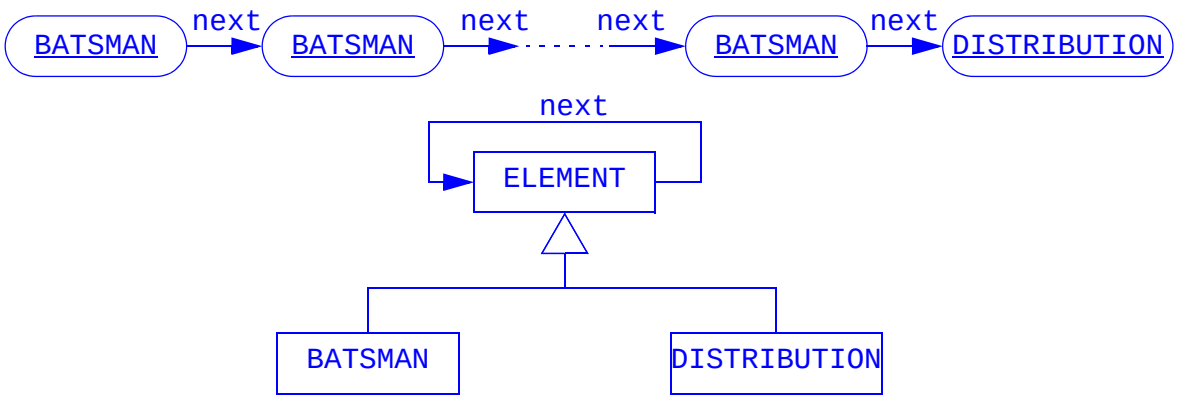


Figure 6. Computing a distribution using polymorphism

This example shows the factoring of multiple loops into a chain of data structures or objects using firstly, recursion and secondly, polymorphism. The multiple loops version in Listing 5 requires many variables to hold the batsmen data and index over their scores and the number of batsman is fixed by the code. The recursive version in Listing 6 requires access to a chain of `BATSMAN` objects, but introduces a test for the last element in the chain, which must be tested at every level of recursion. In the polymorphic case, (Listing 7), computation passes from object to object without the test used in the recursive solution. Instead, the test has been moved into the construction code in the feature `make` and the object linkages play an active role in determing which code is run.

The difference in the structures is reflected in the postconditions of the `make` features in the recursive and polymorphic versions. It does not appear to be possible to say more about the structure of the recursive case than that `distribution` is not `Void`, because the object that `next` references is optional. However, in the polymorphic case, the postcondition specifies the required presence of a `DISTRIBUTION` object at the end of the chain. In neither case, however, do we have a ready means of specifying a finite chain using a postcondition.

```
deferred class ELEMENT
  feature
    make is do end
    build is deferred end
    next: ELEMENT
end

class BATSMAN
  inherit
    ELEMENT
      redefine make end
  creation
    make
  feature
    scores: ARRAY[INTEGER]
    make is
      ensure
        last_element: next = Void
        continuation: next.next /= Void implies
                      next.generator.is_equal("BATSMAN")
        termination: next.next = Void implies
                      next.generator.is_equal("DISTRIBUTION")
      end
    build(s: INTEGER) is
      require
        next /= Void
      local
        i, sum: INTEGER
      do
        from
          i := 1
        until
          i > scores.count
        loop
          sum := s + scores.item(i)
          next.build(sum)
          i := i + 1
        end
      end
end
```

Listing 7. Polymorphic version for computing a distribution

```
class DISTRIBUTION
  inherit
    ELEMENT
  creation
    make
  feature
    distribution: ARRAY[INTEGER]
    make is
      ensure
        distribution /= Void
      end
    build(s: INTEGER) is
      ensure
        updated: distribution.item(s) =
                    old distribution.item(s) + 1
      end
end
```

Listing 7. (cont.) Polymorphic version for computing a distribution

This example has been implemented in the three versions using ISE Eiffel version 4.5. Comparative measurements of the three versions using an in-lining depth of 9, shows that the multiple loop version is fastest, the object oriented version being about 17% slower and the recursive version 48% slower. This quantifies the losses and gains using procedural and object oriented methods.

Example 3: Dynamic structures

In this example, a correct specification for a dynamic structure is captured. Consider the problem of dynamic linking of a model object to an interface object, such as a dialog box, providing the user with means of viewing and modifying the model data. This structure is related to the subject-observer pattern [Jézéquel99], but in which the subject changes dynamically. In this situation, only one of each dialog box is available. (The dialog box object is a singleton). Given many `MODEL` objects, only one can be displayed and modified at one time. Figure 7 shows several objects, instances of the `MODEL` class, accessible via a `CONTAINER` class. The `DIALOG` object is known to all the `MODEL` objects.

To display the selected `MODEL` object, the `DIALOG` object requires access to it. This can be done by providing the `MODEL` class with a command `display` specified in Listing 8.

The postcondition of `display` ensures three things. First, the `DIALOG` object has a correct reference to the `MODEL` object being displayed. This is required to synchronize the correct `MODEL` object when the data is changed. Secondly, the fact, that the `MODEL` object is displayed, is recorded. This makes it possible to assert, thirdly, that if a *different* `MODEL` object was displayed previously, it becomes undisplayed. Note that the precondition does two things: it excludes re-displaying the currently displayed object simplifying the postcondition (and also the code), and it requires the `DIALOG` object to be synchronized with the previously displayed `MODEL` object.

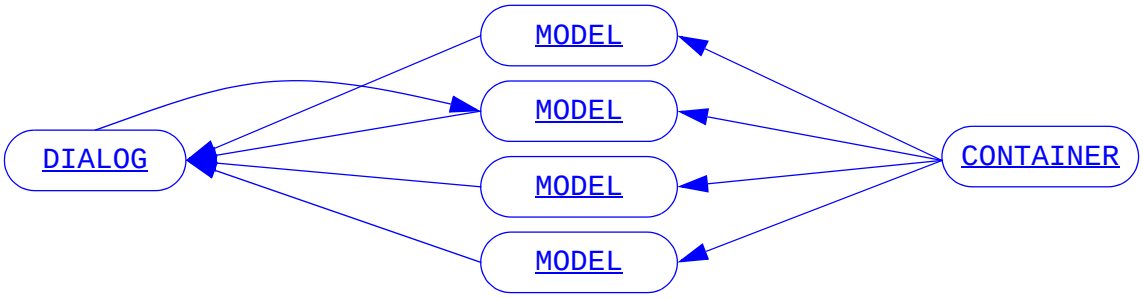


Figure 7. Linking a dialog to a model

```
display is
  require
    not_displayed: not displayed
    dialog_synchronized: dialog.synchronized
  ensure
    notified: dialog.selection = Current
    displayed: displayed
    other_undisplayed: old dialog.selection /= Void implies
                           not (old dialog.selection).displayed
  end
```

Listing 8. Specification of command *display* in class *MODEL*

The postcondition includes the term `dialog.selection`, and implies that this is changed. It follows that the `DIALOG` class must supply the command specified in Listing 9 which will be called from the body of `display` with the argument `Current`. The fact that a selection should not be changed without saving data is specified by the precondition tagged `synchronised`.

```
set_selection(m: MODEL) is
  require
    argument: m /= Void
    synchronized: not modified
  ensure
    selection: selection = m
end
```

Listing 9. Specification of command *set_selection* in class *DIALOG*

It is sometimes suggested that such reflexive coupling between objects should be avoided. However, a correct version can be specified without writing a line of the actual code, by considering the external structure between the objects in the system and capturing its properties as assertions in the specification of routines. A legitimate objection may also be raised that the model and dialog are strongly coupled [Jézéquel99], but this is a design issue and outside the scope of this discussion.

4 CONCLUSIONS

The three examples show the importance of considering the external structure of a system and specifying its important aspects as assertions. In the first example, links are added between objects to make access explicit, allowing the structure to be defined and protected by assertions. Clients can rely on the structure and do not need to compute access paths. In the second example, replacing procedural code with structure using recursion sacrifices some performance most of which can be recovered by a polymorphic structure. In the third example, structural constraints on dynamic links are specified as postconditions. Frequently, assumptions made by the developer about the external structure are hidden in the code. Developing the model to expose implicit assumptions and capturing them as assertions serves to make the model closer to the required system.

When all is said and done, it is the behavior of the system of objects created by executing code that we require. It helps to build and manipulate the object model seeking more effective structures. But, it is also helpful to record assumptions made about structure in order to define the code more completely. If the external structure of the object model is undefined or dynamic, developers and maintainers may make contradictory assumptions. As the examples show, Eiffel invariants and postconditions can be used to define and protect the original assumptions. Such mechanisms allow the developer to claim that the code both implements the required system and is represented accurately by the object model.

5 ACKNOWLEDGEMENTS

I am indebted to Andrzej Goscinski not only for challenging discussions about object oriented methods which have resulted in this paper, but also for commenting on it near completion. I am also indebted to Justin Rough for comments that influenced early drafts.

REFERENCES

[Meyer92] Meyer, B., Eiffel, the Language, Prentice Hall, 1992, pp. 117–134.

[Meyer96] Meyer, B., Object Oriented Software Construction, 2nd edition, Prentice-Hall, 1996.

[Jézéquel99] Jézéquel, J-M., Train, M. and Mingins, C., Design Patterns and Contracts, Addison Wesley, 1999.

About the author

Peter Horan is a Senior Lecturer in the School of Information Technology, Deakin University, Geelong, Victoria, Australia. He lectures in Software Engineering and Software Development. He is interested in the specification, design, development and testing of software.
Peter can be reached at peter@deakin.edu.au