Bertrand Meyer

Toward More Expressive Contracts

The principles of Design by Contract form the basis of the Eiffel approach and account for a good deal of its appeal. Eiffel's contracts are the result of a design trade-off between the full extent of formal specifications and what is acceptable to practicing software developers. The latter criterion has been critical: The ideas had to be practical; any competent programmer can immediately see their benefits, and start using them. In James McKim's words, "If you can code, you can spec." This is the principle behind Eiffel contracts—not a dream for a perfect world, but a practical tool for solving everyday problems.

The design has proved its value, and Eiffel programmers enjoy it. But the time may have come to revisit the trade-off and see how much more we could express with contracts—how close we could come to the goal of full specification without losing the simplicity and self-evidence of the classic Eiffel mechanism. Developments such as the Object Constraint Language (OCL) addition to UML show that interest in contracts has grown beyond the Eiffel community, together with a search for ways to express higher-level contracts, including, in particular, properties of first-order predicate calculus. Eiffel has the considerable advantage of including contracts as a part of the programming language, intricately woven into the process of conceiving, developing, documenting, debugging, and maintaining software. This is all the more reason to explore whether we can make contracts more expressive. This article explores recent progress in this direction, resulting from work in several areas:

- a language development: the new "agent" language mechanism presented in an earlier column[1] with a new variant (in-line agents);
- new Eiffel library work, especially around EiffelBase, an open-source framework[2–3] covering the fundamental data structures and algorithms; and
- progress in the Trusted Components project,[4] aimed at building a rich set of high-quality reusable components for the industry.

Bertrand Meyer is the author of *Object-Oriented Software Construction*, 2nd ed., and several other books including *Eiffel: The Language* and *Reusable Software*. He is with Interactive Software Engineering (Santa Barbara) and a Professor at Monash University (Melbourne). He may be contacted at Bertrand_Meyer@eiffel.com.

CLASSIC CONTRACTS

Applying Design by Contract to software development yields a number of well-known advantages[5]:

- Contracts make it possible to be serious about reuse by equipping software components with specifications similar in spirit to those of engineering components in electronics and other engineering fields.
- They provide built-in documentation.
- They transform the process of in-depth testing and debugging, making it more focused thanks to the presence of precise definitions of expected behavior.
- They are also a powerful management tool, giving managers a better handle on the progress of their projects by enabling them to concentrate on function rather than implementation.

Contracts in Eiffel are closely connected with the object-oriented structure of systems:

- *Class invariants* express the consistency constraints applicable to all instances of a class, such as, in a class describing car trips, "the average gas mileage since the beginning of the trip is the distance traveled divided by the amount of gas used."
- *Routine preconditions* express the requirements under which an operation is legal, such as "*start_engine* requires *shift_pedal_down*" (on recent cars with shift transmission).
- *Routine postconditions* express properties satisfied as a result of executing an operation, such as "after a successful *resume_cruise_control*, the *speed* is equal to the *previously_set_speed*."

THE LANGUAGE OF CONTRACTS

Contract elements (assertions) are, in their simplest form, boolean expressions, as in the invariant clause

```
invariant
   gas_mileage = (odometer - initial_odometer) /
                  (initial_gauge / gauge)
```

(where the "=" sign of course denotes equality, not assignment), or the precondition and postcondition (**require** and **ensure**) of the following routine:

```
start_engine is
    -- Try to start engine; set started to record
    -- whether successful.
  require
    shift_pedal_down
  do
    ...
  ensure
    started implies engine.running
  end
```

Boolean expressions state properties that may or may not hold at a certain time during execution, such as "is the engine running?" That is exactly what we need for assertions, except that in a correct program we expect that the property *will always* hold when needed. (This is why contracts are so useful for debugging, enabling us to check reality against expectations.)

The operator `implies` is often useful in assertions, as here. Defined from logic, `a implies b` denotes the property that, whenever *a* is true, *b* also is. (In other words, `a implies b` is false in only one case: *a* true and *b* false.) Here we are stating that unless something went wrong, causing `started` to be set to false, executing `start_engine` will leave the engine running.

The assertion sublanguage extends the language of boolean expressions by only one construct, the `old` notation, permitted only in postconditions, as in

```
ensure
  speed >= old speed * 1.1
```

which expresses that the routine must have increased *speed* by at least 10%.

APPLYING BOOLEAN EXPRESSIONS

Although the theme of this article is to look at the limitations of using boolean expressions for contracts, we must first realize how far the humble boolean expressions can already take us. As a typical example of the kind of contract routinely used in Eiffel development, consider the procedure

```
extend (v: ELEMENT_TYPE)
```

which, in various lists and other collection classes of EiffelBase, adds an element at the end of a sequential structure. Its postcondition reads:

```
ensure
  inserted: has (v)
  one_more: count = old count + 1
  one_more_occurrence:
   occurrences (v) = old (occurrences (v)) + 1
```

(The "assertion tags"—*inserted*, *one_more*, and *one_more_occurrence*—document the assertion clauses and provide further debugging support. The postcondition is shown as it appears not in the class texts but in their "flat-short" forms, including postconditions inherited from ancestor versions.)

The first clause, labeled *inserted*, states that if after using *extend* to insert *v* we ask the question "Is *v* in the list?"—by calling the function *has*, which determines whether an element appears in a list—the answer will be yes.

The second clause, relying on the `old` notation, states that any call to *extend* increases the size (number of items) of the list—*count*—by one. The corresponding clause for more complex structures would of course be more sophisticated. For example, after a hash table insertion, the property is that the *count* has been increased if the key was not already used before; this is straightforward to express thanks to the `implies` operator:

```
(not (old has (key))) implies (count = old count + 1)
```

The final clause of the example postcondition, labeled *one_more_occurrence*, states that if after the insertion we ask for the number of occurrences of *v*, then by using the function *occurrences* (applicable, as is *has*, to most container structures) we will get one more than before. Again, this will be a bit less elementary for more complex structures.

THE TRUTH AND THE WHOLE TRUTH

The problem with the postcondition of *extend* is not what it says, which is definitely part of the semantics of appending an item to a list—the appended element will be in the list, with one more occurrence, and the list's size will have grown by one—but what it doesn't say. Appending an item should not affect the items previously present. But nothing in the postcondition expresses that property.

In ordinary cases, we can live with such incompleteness because it would take a devious implementation of *extend* to start modifying the existing element. But in the long term, this is not a valid excuse, because contracts should express all the properties of interest, and no one knows what twists bugs may take. So we must look for ways of expressing all the relevant properties of classes and routines.

This issue has been known for a long time, of course. Many people have suggested addressing it by extending the contract sublanguage beyond boolean expressions (which mathematically correspond to propositional calculus) to predicate calculus, with support for the operators ∀ (for all) and ∃ (there exists), known as quantifiers. But this may be overkill, risking the transformation of Eiffel into a full-fledged mathematical specification language, losing the advantages of the basic Design by Contract mechanism: simplicity, ease of use, closeness to concepts familiar to all developers, executability (to monitor assertions in debugging and testing mode), and ability to write commercial Eiffel compilers of manageable complexity. Although the following approaches do rely on new language facilities (agents, recently implemented in ISE

A Resource Center for Design by Contract

As this article goes to press, the new "Contract Portal," the result of extensive research around the Internet, goes live at http://designbycontract.com, providing a wealth of resources on everything having to do with Design by Contract for software quality and component-based development. Along with introductory articles, links are available to hundreds of pages around the Web, conveniently organized by categories—languages (Eiffel, Java, C++), tools, books, training, component standards, research developments, and others. A feedback form is also available, enabling you to suggest any site that the list may have missed.

The site will continue to evolve as the field develops and more people contribute to advances in Design by Contract.

Eiffel), they leave the contract sublanguage unchanged. You will have access to the power of quantifiers—and, in fact, to even more powerful operators—but not as language constructs; instead, library classes from EiffelBase will provide these mechanisms as functions.

Before moving on to these facilities, we must see what was already available, in classic Eiffel, to address the problems cited, since even there we were not quite helpless.

USING FUNCTIONS

Does the use of boolean expressions for assertions prevent us from writing full-fledged contracts? In fact, no. Boolean expressions can include function calls. Thus we can add to the postcondition of *extend* a clause of the form:

```
same_initial_items (old twin, old count)
```

where *twin* is a function that returns the duplicate of a structure. (*twin* comes from the class *ANY*, ancestor to all classes, and so is available to all classes). The call to *twin* may also be written *clone (Current)*, where *clone*, a variant of *twin*, also defined in *ANY*, returns a duplicate of the object passed as an argument. Note that in current versions of EiffelBase, *twin* (as *clone*) is redefined in the list classes to clone an entire list, not just the list header; this is what we want here. The postcondition assumes that we have added to the enclosing class (or an ancestor) a boolean-valued function *same_initial_items* with two arguments:

```
same_initial_items (other: like Current;
                    i: INTEGER): BOOLEAN
    - Are all items of the current list, if any,
    - in positions 1 to  i, equal to those of list
    - other at the corresponding positions?
  require
    i >= 0
    i <= count
    i <= other.count
```

This function is indeed easy to implement, for example, using a local variable *j*:

```
from
  start ; other.start; j := 1
  Result := True
until (j > i or not Result) loop
  Result := item = other.item
  forth ; other.forth; j := j + 1
end
```

(This follows EiffelBase conventions: *start* brings a list cursor to the first element; *item* is the list's item at the cursor position; *forth* advances the cursor by one position.)

With the addition of function *same_initial_items* to the class, we can express what we want: that the final items at position 1 to **old**_*count* (the original value of *count*, before the execution of *extend* increases it by one) are equal to the corresponding items in a clone ("twin") of the original list.

FUNCTION ISSUES

Using a function such as *same_initial_items* solves the expressiveness problem: We can state the properties of interest about *extend*, but it raises two immediate issues, one theoretical and one practical:

- Assertions are supposed to be nonoperational properties of the software; while code is prescriptive, assertions are descriptive. Reintroducing functions—that is to say, routines—breaks this distinction: *same_initial_items* does not have an obviously different status from the routine it is supposed to document, *extend* (except that one is a function and the other a procedure). For example, a routine may produce a side effect; in a function used in an assertion, this would clearly be unacceptable. (See the sidebar entitled "Pure functions.")
- More pragmatically, equipping classes with lots of special-purpose functions, such as *same_initial_items*, for the sole purpose of providing the routines of primary interest (such as *extend*) with appropriate assertions, is not a very enticing prospect. True, this approach has precedent in the field of program proving, where Owicki and Gries[6] showed the usefulness of adding special auxiliary variables to prove parallel programs; but we may fear that our classes will soon become bloated with numerous contract-oriented features that confuse library users, especially novices.

Let us now see how the agent mechanism alleviates these problems.

INTRODUCING AGENTS

In a first step, short of getting rid of special functions altogether, we can use agents to reduce their scope. Function *same_initial_items* decides whether all items between given positions, in two lists, are equal. We can define a more elementary function

```
same_item (other: like Current; i: INTEGER):
          BOOLEAN is
    - Are items at position i in the current list
    - and other equal?
  require
    i >= 1
    i <= count
    i <= other.count
  do
    Result := (i_th (i) = other. i_th (i))
  End
```

where the function call *i_th (i)* returns the item at position *i*. Then we can use a *for_all* iterator to apply this function to every element. The postcondition clause becomes:

```
(1 |..| old count). for_all (~ same_item (other, ?))
```

This uses the agent mechanism as described in the earlier column.[1] The tilde character "~" is the distinctive mark for agents, so ~ *same_item ((other, ?)* is an agent, that is to say, an object representing a routine ready to be called when requested: the function *same_item*, to be called on two arguments, the first of which is *other* and the second an integer to be provided at the time of each call. This second argument is represented in the agent by a question mark, indicating that the value is not known when we define the agent, only when someone calls it at runtime.

We pass this agent to a *for_all* iterator applicable to integer intervals: *i* |..| *j* is the integer interval from *i* to *j*, using the infix operator |..| defined in class *INTEGER* and yielding a result

Pure Functions

There are two kinds of routines: functions, which return a result, and procedures, which do not. Procedures produce side effects (that is, in fact, their raison d'être), but the Eiffel method advises programmers to keep their functions "pure"—without side effects, that is. The methodological justifications are numerous, and explained in detail in *Object-Oriented Software Construction*.[5] But for all kinds of reasons, also explained in the book, this is a methodological guideline, not a language rule enforced by compilers. You (or at least your functions) can actually be quite impure if you wish.

For contracts, however, there is nothing funny about impurity. An assertion should be a purely descriptive property of the state of the computation. Evaluating it should never change that state! In particular, it is an important rule of Design by Contract that whether you turn assertion monitoring on or off should not affect the semantics of a program (although it obviously affects its execution time); this is why assertion monitoring is so useful for testing and debugging.

More precise rules are needed, then, to define what constitutes a side-effect-free function, suitable for use in assertions. Work is indeed proceeding to add a notion of *pure function* as a language construct. A future column will describe this work in detail, but you may already note that—pending the actual introduction of a proper mechanism—*pure* is already a keyword of the language.

of type `INTERVAL`; in class `INTERVAL`, the function `for_all` returns true if the function agent that it takes as an argument (`~ same_item (?)` in this example) returns true for all values in the interval. This explanation may seem contorted because words are not good at describing mathematical properties, but in fact the meaning is very simple; the Eiffel text is close to a mathematical formulation

```
∀i ∈1 |..| old count • same_item (i)
```

where ∀is "for all" and ∈ is "in."

FIRST PROGRESS ASSESSMENT

How does the new Eiffel formulation, using function `for_all` and an agent built from function `same_item`, address the two issues raised: the danger of introducing routines into assertions, and the complication of the class text?

It has solved neither of them completely. On the first issue, we still have a need for auxiliary functions; it's only a small consolation that `equal_item` is simpler and of a smaller scope (two list items rather than two entire lists) than the original `same _initial_items`. On the second issue, however, we have progressed significantly, in practice if not in theory. In theory, functions such as `for_all` are just as bad as any other routines, side effects and all. But in practice, such functions will be written once and for all in a few high-level classes of EiffelBase, a typical example of "trusted components"; we can hope to make sure, through all available means—careful design, verification by appropriate tools, extensive public scrutiny—that they are of pristine quality and do not include any questionable aspects such as side effects. The notion of a "pure" function, mentioned in a sidebar, can take us closer to this goal.

So in practice we may consider that we have significantly reduced the scope of the problem. We have rid ourselves of the nontrivial part of `same_initial_items`, the algorithm that iterates over a number of list items, by making it a general feature of library classes such as `INTERVAL` rather than a specific and tricky part of every application. But we still have to encumber ourselves with application-specific, contract-only features such as `same_item`, which, however simple, can still clutter our classes, causing especially undesirable complication in library components.

INTRODUCING INLINE AGENTS

Our final technique will result from the preceding observations. We need the semantics of the agent `~ same_item ((other, ?)`, to pass it as an argument to `for_all`; this agent denotes a routine object, which will tell us, given an integer `i`, whether the list item at position `i` is the same as the one in the `other` list. The agent gets this semantics through function `same_item`. Can we have that semantics, and the agent, without the function?

The notion of inline agent provides a solution. A caveat before we see how it works: I have been careful, in earlier articles for this column, to avoid vaporware by talking only about facilities that are available; for example the first mention of agents[1] appeared several months after the inclusion of this facility in ISE Eiffel 4.2. At the time of writing, inline agents are not yet released, although they are slated for a future release. In software, as we all know, nothing is ever certain until it's implemented (and even then...). So, even though at the present stage I expect the mechanism to be delivered exactly as presented below, surprises are always possible.

The best way to present inline agent is to repeat the previous postcondition clause

```
(1 |..| old count) • for_all (~ same_item (other, ?))
```

remembering that the body of function `same_item (other, i)` read

```
Result := (i_th (i) = other• i_th (i))
```

and rewrite the postcondition clause using an inline agent in lieu of the routine-based (noninline) agent. This gives:

```
(1 |..| old count) • for_all
    (i: INTEGER | (i_th (i) = other• i_th (i)))
```

The meaning should be immediately clear: the argument of `for_all` describes the same thing as the original (a function agent) but without the need for a named routine, because the routine's effect is expressed directly where needed—inline.

The inline agent in this example is

```
i: INTEGER | (i_th (i) = other• i_th (i))
```

denoting an implicit function that, for any integer `i`, returns the value of the expression to the right of the vertical bar. The vertical bar will indeed be the mark of inline agents. Another example would be:

```
i, j: INTEGER | condition: BOOLEAN | condition and i > j
```

denoting a boolean-valued function of three arguments—two integers and one boolean—returning true if and only if the boolean is true and the first integer is greater than the second. This example illustrates how to deal with several agent arguments of the same type (as *i* and *j*) and of different types (separated by more vertical bars).

These inline agents all have, after the final vertical bar, an expression, so they denote functions (routines returning results). It is also possible to define inline agents that define procedures (routines with no results), as in

```
 i, j: INTEGER | do i := j + 1 end
```

which can be useful to iterate simple actions over a data structure without having to define a specific procedure representing these actions. It is easy, for example, using an iterator procedure from class `INTERVAL` or a list class, to add every element of a list of numbers to the corresponding element of another list. This leads to a loop-free style of programming that may be attractive in certain situations.

Inline agents are similar to ideas well known in other areas: lambda expressions (in combinatory logic), closures (in functional languages), and blocks (in Smalltalk). They do not replace the noninline agents because, for many applications, agents are based on existing procedures or functions. For any nontrivial computation, you should not write a complex inline agent, but define the appropriate routine and use the tilde notation.

TOWARD FULLY CONTRACTED LIBRARIES

The example discussed illustrates one of the principal applications of inline agents: expressing powerful contracts. With this mechanism and other techniques developed in recent years, it now seems possible to reach a goal that not so long ago would have seemed elusive: a fully contracted version of EiffelBase and other key libraries. A number of people have made contributions to this effort (including Christine Mingins in the Trusted Components project, James McKim and Richard Mitchell in extending the concepts of Design by Contract, Marcel Satchell in his work on EiffelBase, and Jean-Marc Jézéquel in his work on component testing). There is even a glimpse of hope that we could benefit from the newest and most exciting approach to program proving, Abrial's Atelier B,[8] to prove components.

We are still very far from any such goal; things are, in fact, only starting, and the difficulties are numerous. However, the prospect of trusted components is worth the effort, and to advance it we must take every advantage we can from the techniques, old and new, of Design by Contract. ■

References

1. Dubois, P. et al. "Eiffel: From Calls to Agents," *JOOP*, 12(6): 66–69, Oct. 1999.
2. Meyer, B. *Reusable Software: The Base Object-Oriented Libraries*, Prentice–Hall, Englewood Cliffs, NJ, 1994.
3. "EiffelBase: The Ultimate in Reusability," *The Home Page for Object-Oriented Technology and Eiffel*, http://www.eiffel.com/products/base/.
4. *The Trusted Components Initiative*, http://www.trusted-components.org.
5. Meyer, B. *Object-Oriented Software Construction*, 2nd ed., Prentice–Hall, Englewood Cliffs, NJ, 1997.
6. Owicki, S. and D. Gries. "Verifying Properties of Parallel Programs: An Axiomatic Approach," *Communications of the ACM*, 19(5): 279–285, May 1976.
7. *Talkitover—Software: EiffelBase Improvements*, http://talkitover.com/eiffel/base.
8. Abrial, J. *The B Book*, Cambridge University Press, New York, 1996.

LETTER TO EDITOR

continued from page 1

always lead to inclusion of all relevant, or at least most of the natural, features in the base class. Unless the problem domain (in the example) requires handling of some dogs with one, and some with two (or even more) appetites, such an incremental extension seems somewhat unnatural, or reminiscent of poorly designed extension.

Poorly designed class hierarchies are always problematic; this is one reason there is great emphasis on requirement analysis and adequate knowledge of the problem domain in all design methodologies. An adequate understanding of the problem domain should clearly indicate the objects, relationships, and components involved in the problem. Without this knowledge, the ultimate goal of greater software reuse, or even the primary goal of good design, will be out of reach, irrespective of the methodology practiced.

My second observation relates to the choice of appetite methods. This choice of methods suffers from a major drawback—the methods are independent of the second argument in the definition. Each appetite always returns the same constant value. If such is the case, making appetites a data attribute with fixed value should be considered; but this will lead to a situation where no multimethod is necessary since the appetite is no longer a method.

This is not to imply that multimethods are not needed, just that they are not needed in the example selected by Saar.

As to the issue of encapsulation provided by classes (though not fully supported by some OOPLs including C++), it has been strongly advocated in literature for decades and is known to improve the quality of the software products. Breaking this encapsulation must only be considered if the alternative offers tangible advantages in the design of software—something the examples used in the work are unable to substantiate.

Considering the problems caused by the unidimensionality of the method dispatch (p. 13), the author refers to the example in Fig. 1 stating "the methods have to be selected via both dimensions, i.e., predator and prey." This doesn't seem to be the case, since the methods appetite in the example consider, and return the result for, only the predator-to-prey relationship, and never for the prey-to-predator relationship. Thus the mutidimensional aspect is not visible in the example in any clear manner.

Multimethods may turn out to be useful facility to handle interclass relationships, something current research in this area will determine in due course of time.

Masud Malik
Philadelphia University, Jordan, masudmalik@hotmail.com