To appear in: *Abstract State Machines 2003 — Advances in Theory and Applications*, Proc. 10th International Workshop, Taormina, Italy, March 3-7, 2003, eds. Egon Boerger, Angelo Gargantini, Elvinia Riccobene, Springer-Verlag 2003.

# A framework for proving contract-equipped classes

*Bertrand Meyer*
ETH Zürich, Chair of Software Engineering
(also Eiffel Software, Santa Barbara)
se.inf.ethz.ch

**Abstract**. As part of a general effort to provide a new basis for software development through reuse of "Trusted Components", we outline a scheme for proving that classes equipped with contracts in the Eiffel style meet these contracts. The approach takes advantage of the inheritance structure to separate proof obligations between deferred (abstract) classes, to be validated against a model, and their effective implementations, which then must only be proved against the contracts of the deferred ancestors. The testbed for this study is the EiffelBase library of fundamental data structures and algorithms, whose classes include extensive contracts.

## 1 TRUSTED COMPONENTS, LOW ROAD AND HIGH ROAD

The techniques proposed here are part of a general effort to develop *Trusted Components* [8]: reusable software elements with guaranteed properties. We start with an overview of this broader goal and of the present work's place in it.

Work on Trusted Components rests on the observation that one of the most realistic hopes for radical improvements in software quality and productivity is to combine *reuse* and *quality*. Reuse ensures faster time to completion, economies of scale, and the opportunity to turn expertise into concrete assets. But reuse scales up everything, deficiencies included; reusable components should be subjected to quality criteria far more rigorous than ordinary non-reusable software. The effect of scale then becomes a benefit rather than a risk: every reusing application profits from the quality investment made in the reusable components.

There has been widespread advocacy for the idea of reuse, especially of object-oriented classes [3] and binary components [13]. A real market for components has emerged in recent years, although the need to associate quality with reuse doesn't seem to have registered with the software industry at large. Trusted Component efforts attempt to remedy this mistake and to provide a solid basis of high-quality components for software development.

Ideally, a Trusted Component should have a formal specification of its relevant properties and then a proof that its implementation satisfies the specification. This is indeed the approach explored in the following sections. It assumes, however, components that are designed from the outset with this goal in mind; while important for the long term, this view does little to address the concerns of industry developers who have to work with existing components — object-oriented, EJB, COM, .NET … — that have not been built accordingly; if nothing else, most commercial components do not publish their source code, making third-party proofs impossible.

As a result we face a choice between providing full proofs of components produced specifically for this purpose ("*software as it should be*") and attempting partial qualification of existing commercial components built with far more informal techniques ("*software as it is*"). Our work on Trusted Components recognizes the need for both of these approaches, and as a consequence pursues two tracks at once:

- A *high road* for producing components whose correctness can be proved, and indeed proving it.

- A *low road* for assessing properties of existing components, commercial or open-source. It will generally be impossible to include correctness proofs in this case, but we can still define relevant quality criteria — extent of functional and performance specification, examples of use, documentation, evidence of prior reuse, extendibility … — and assess components against them. A *Component Quality Model* is under development for this purpose [10].

The work presented here is on the "high road". Focused on fundamental components covering common data structures and algorithmes, it attempts to produce full proofs of their correctness.

An important practical issue of correctness proofs for object-oriented components is the matter of run-time structures, which typically involves extensive use of *pointers* (or "references"). In parallel with the work described here, a theory has been developed to model pointer-rich object structures and prove the corresponding software properties. Described in a separate paper [11], it is closely connected to the present work, and plays an important part in its application to actual proofs of object-oriented software.

## 2 FORMAL METHODS FOR REUSABLE COMPONENTS

Most applications of formal methods so far have been to systems or subsystems, generally in mission-critical areas where the investment in formality and proofs is justified by the grave consequences that malfunctions could cause. Examples include defense applications, transportation systems such as those built with the B approach [1], and Java Bytecode verification through Abstract State Machines [12]. With components, the economic justification is simply the effect of scale arising from reuse. A reusable component may not be by itself "critical" in the same sense as the system controlling the closing of doors in a train, but the effect of a deficiency could be just as bad given the number and scope of applications that may rely on it.

Proof technology has now advanced to a point where it appear practical enough to go beyond these applications and consider proofs of components. Although not extensively explored until now, this combination of formal methods and reuse offers one of the most interesting outlets for formal development.

In pursuing it we will rely on Eiffel's **Design by Contract**™ [4] [5] [7]. Classes equipped with contracts — routine preconditions, routine postconditions and class invariants — already possess a degree of formal specification.

Although devised from their origin with the ultimate goal of permitting proofs, contracts have been used so far in Eiffel for other purposes [7]: as a design method to obtain correct software; as a documentation technique in connection with the tools of the EiffelStudio environment; as a central part of the testing, debugging and quality assurance process; as a management aid; as a guide for using inheritance properly; and as a basis for exception handling relying on a clear definition of what constitutes a "normal" and "abnormal" case.

## 3 LAYERS OF PROOF

To prove properties of components (and in fact of applications too) it appears desirable, because of the potential complexity of some libraries, to organize the proof process in a hierarchical manner, where the basic (lowermost) levels are very close to mathematical concepts, and each higher level is proved conditional to the correctness of the lower one:

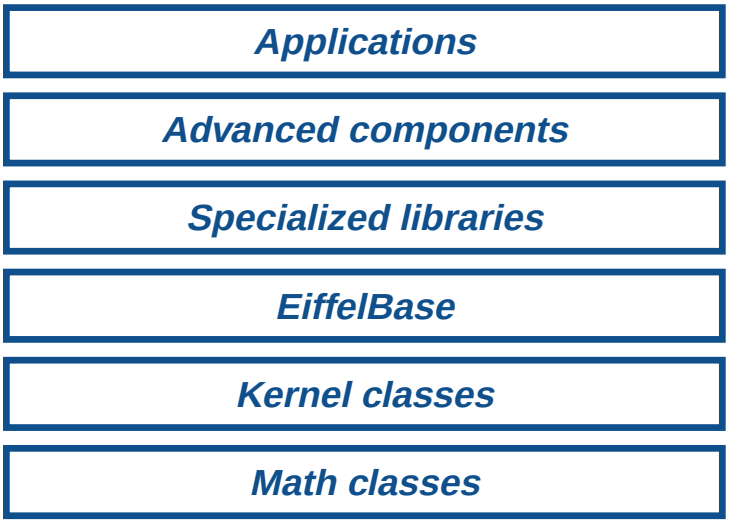


**Figure 1: Application and component layers**

## 4 EIFFELBASE

Of particular interest among the layers of figure 1 is the EiffelBase library, the focus of our first investigation. The library results from a systematic approach, detailed in reference [6], at providing a "Linnaean" taxonomy of the fundamental structures of computing science. EiffelBase takes full advantage of object-oriented mechanisms and is thus a good testbed for the scalability of any techniques developed. Figure 2 describes the hierarchy of its top inheritance levels.

EiffelBase classes are extensively equipped with contracts, as illustrated by the examples reviewed below. This makes them a prime target for proofs, since the properties to prove are already part of the class text.

An obvious objection to the choice of EiffelBase is that industrial users may be more interested in coarser-grain components (EJB, CORBA, COM, .NET) covering — say — print drivers, web services or payroll records rather than stacks and linked lists. But several reasons actually make components such as the EiffelBase classes actually very useful for such a study:

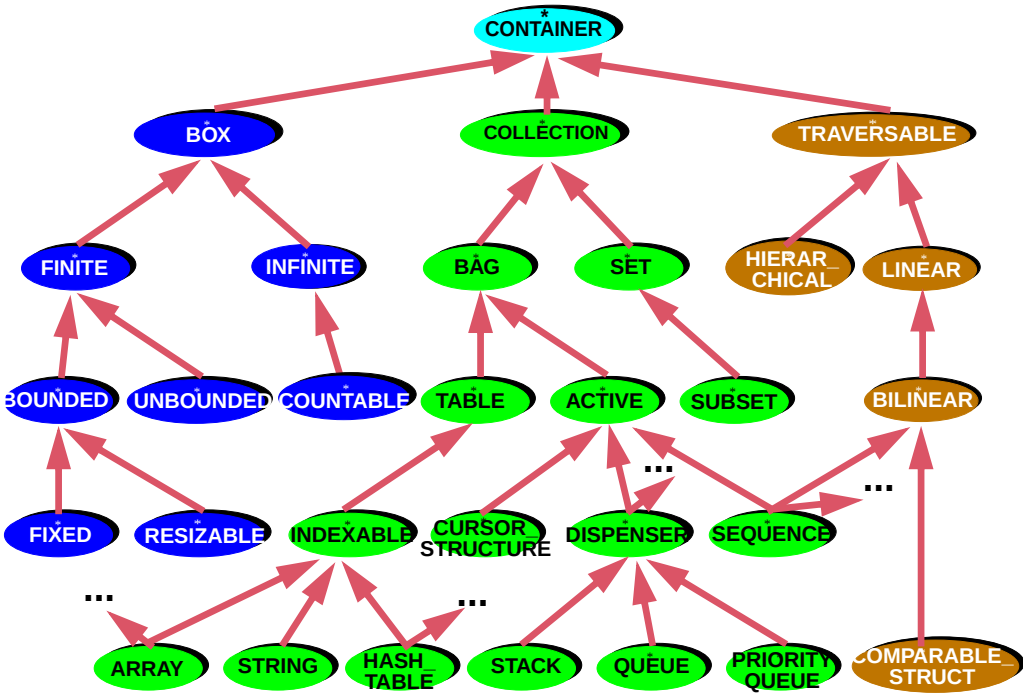


**Figure 2: EiffelBase classes, topmost inheritance hierarchy**

- These components are used extensively in all applications. Bringing them to a high degree of quality is by itself an important achievement.
- They raise difficult problems, providing a microcosm of the more general issues of component-based development.
- The underlying theory has been explored in depth, and so is ready for integration in a proof framework.
- They are a required step: we can hardly hope to provide proofs of more complex, application-specific components until we know how to tackle these. If we can't prove the contracts of a list class, we won't be able to prove a Grid control.
- Finally, as noted, this is not the *whole* approach, simply one of its facets, complemented for more immediate industry needs by the "low road" techniques mentioned above.

So they appear a proper choice for this foundational work, a first step towards the development of general techniques for proving properties of different component kinds, fine-grained and coarse-grained.

## 5 A REUSABLE EFFECTIVE CLASS: *STACK*

Let's take a look at a typical class and see what it would mean to "prove it". *FIXED_STACK* is one of the implementations of *STACK*, through arrays. Along with *ARRAYED_STACK*, *LINKED_STACK* and other implementations, it inherits from the deferred (abstract) class *STACK*:

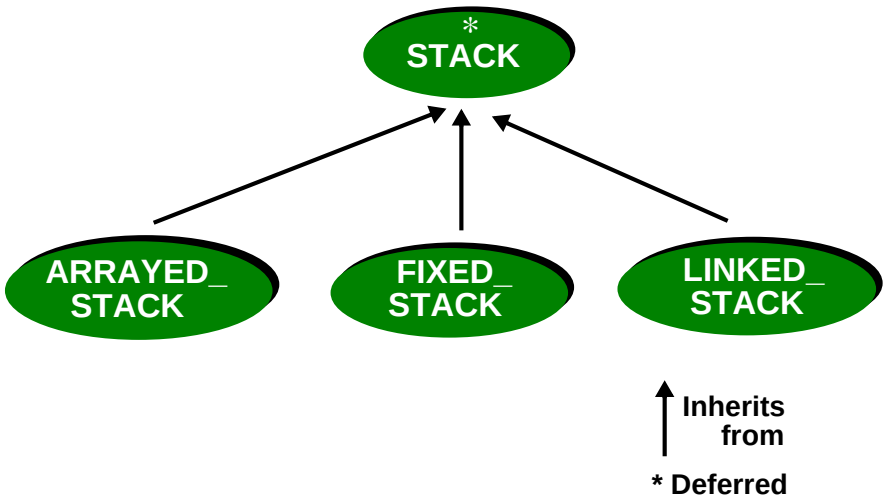


**Figure 3: Multiple implementations through inheritance**

In its general form, including the contracts, *FIXED_STACK* looks like this:

```
class FIXED_STACK [G] inherit
        STACK [G]
        ARRAY [G]
                rename item as array_item, put as array_put,
                        count as capacity
                end
feature -- Access
        count: INTEGER
                        -- Number of stack items
        item: G is
                        -- Top element
                require
                        not_empty: count > 0
                do
                        Result := array_item (count)
                end
```

```
feature -- Element change
    put (x: G) is
            -- Push x to top of stack.
        require
            not_full: count < capacity
        do
            count := count + 1
            array_put (x, count)
        ensure
            pushed: item = x
            one_more: count = old count + 1
        end
    remove is
            -- Pop top element.
        require
            not_empty: count > 0
        do
            count := count – 1
        ensure
            removed: count = old count – 1
        end
invariant
    meaningful_count: count >= 0
    bounded_by_capacity: count <= capacity
end
```

A number of details have been omitted as compared to the actual class in the EiffelBase delivery:

- Assertion inheritance: most of the contracts are actually in the higher-level, deferred class *STACK*; class *FIXED_STACK* inherits them without having to reproduced them. What is shown here is close to its “flat form” rather than its actual text.
- Several features.
- Creation procedures.
- Some inheritance details.
- The Indexing clause.
- Array resizing (found in *ARRAYED_STACK*): in general, EiffelBase classes avoid putting strict limits on size, and instead silently resize the representation if the contents outgrow it. Here we have chosen the fixed-size variant since it leads to a simple and interesting contract.

# 6 INTRODUCING A MODEL

The first question to ask is: What does it mean to "prove" *FIXED_STACK*?

The accurate phrasing is of course that we are interested in proving the *correctness* of the class with respect to particular *specification properties*.

> Like the rest of the literature this discussion casually talks about "proving a software element", but this is just an abbreviation for the notion of proving that the element possesses explicitly specified properties.

Thanks to the presence of contracts, these properties are already present in the text, in the form of preconditions (**require**), postconditions (**ensure**), class invariants (**invariant**). We have to prove that the implementation satisfies these contracts: any routine called with the precondition satisfied will terminate and ensure the precondition; in addition, if it is called from outside the class, it will preserve the invariants.

To talk about such correctness proofs will we use a conceptual *model* of the corresponding structure. We consider, for example, that a stack is conceptually representable as a sequence, where the push operation (*put*) happens at the end:\

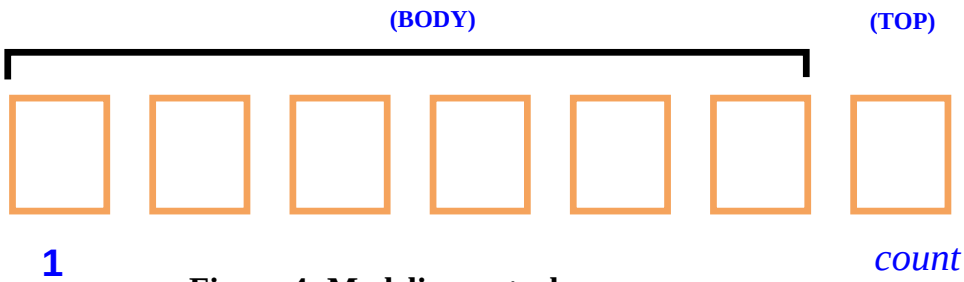


**Figure 4: Modeling a stack as a sequence**

Such a model plays no role in the class implementation; it is also of no significance to users of the class, who still view it in purely abstract terms, defined by official features (*put*, *remove*, *item* etc.) according to the principles of abstract data types. But it will help us for the proofs.

Mathematically, a sequence is a function from an integer interval:

$$SEQUENCE\ [G] \triangleq 1..count \rightarrow G$$

for some *count*. *s* (*i*) is the *i*-th item of *s*.

We may note here that the choice of a model is independent of the choice of implementation, such as *FIXED_STACK* rather than *ARRAYED_STACK* or *LINKED_STACK*. It can be done at the level of the deferred class *STACK*, their common ancestor. This opens up the possibility of doing part of the proof at the deferred level, an interesting prospect since it means that we can use inheritance for

reuse and factorization in the *software proving* process like we do in the *software construction* process. Let us see how the deferred class — itself somewhat simplified too — looks:

```
deferred class
    STACK [G]
feature -- Access
    count: INTEGER
            -- Number of stack items
    item: G is
            -- Top element
        require
            not_empty: count > 0

        end
feature -- Element change
    put (x: G) is
            -- Push x to top of stack.
        require
            not_full: not full
        deferred
        ensure
            pushed: item = x
            one_more: count = old count + 1
        end
    remove is
            -- Pop top element.
        require
            not_empty: count > 0
        deferred
        ensure
            removed: count = old count – 1
        end
invariant
    meaningful_count: count >= 0
end
```

Abstract assertions

Now we can ask again the question “what does it mean to prove a class?” at the level of a deferred class such as *STACK*. The answer will be obtained by introducing the model (a sequence in our case) at that level.

# 7 MATHEMATICAL CLASSES

To proceed we need a better definition of basic model elements such as sequences. They will be described using simple set-theoretical concepts: Sets, Relations, Functions, Sequences (Z or B style).

For simplicity and consistency, it is useful to express such a notion in a form that looks like an Eiffel class:

```
deferred class
    SEQUENCE [G]

feature -- Access
    count: INTEGER
            -- Number of sequence items

    item (i: INTEGER): G is
            -- Element at index i
        require
            in_bounds: i >= 1 and i <= count
        deferred
        end

    last: G is
            -- Element of highest index
        require
            not_empty: count > 0
        deferred
        ensure
            definition: Result = item (count)
        end

    first: G is
            ... Similar ...

    subsequence (i, j: INTEGER): SEQUENCE [G] is
            -- Sequence made of elements from i to j, if any
        require
            first_in_bounds: i >= 1 and i <= count

            second_in_bounds: j >= 1 and j <= count
        deferred
        ensure
            Result.count = max (j – i + 1, 0)
        end
```

```
feature -- Element transformers
        extended (x: G): SEQUENCE [G] is
                -- Sequence extended with x at end.
            deferred
            ensure
                one_more: Result.count = count + 1
                added: Result.last = x
                rest_unchanged:
                   equal (Result.subsequence (1, count), Current),
            end
        head: SEQUENCE [G] is
                -- Sequence minus its first element
            require
                not_empty: count > 0
            deferred
            ensure
                removed: Result.count = count – 1
                rest_unchanged:
                     equal (Result, subsequence (1, count–1))
            end

... Similar declaration for tail ...
invariant
    meaningful_count: count >= 0
end
```

In spite of this programming-like appearance, however, such classes are purely mathematical objects. They correspond to the lowest level of the proof hierarchy:

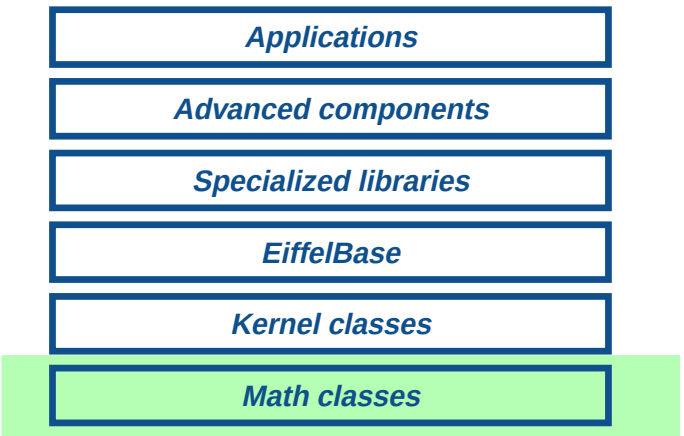


**Figure 5: Place of mathematical classes**

These classes are expressed in a subset of Eiffel called IFL (for Intermediate Functional Language), with strong restrictions: no procedures but only functions; no assignment to attributes; no modifications of any data structure. For example to represent sequence concatenation class *SEQUENCE* above uses a function *extended* that returns a new sequence, not a procedure *extend* that would modify the current sequence. In other words this is a purely applicative language, just a programming-like representation of pure mathematical concepts. We may view IFL as pure mathematics disguised in Eiffel syntax. Using a notation similar to that used for actual classes at higher levels of the hierarchy provides consistency and simplicity.

EiffelBase classes such as *STACK* and its variants rely on these IFL classes, through an intermediate level that we ignore for this discussion:

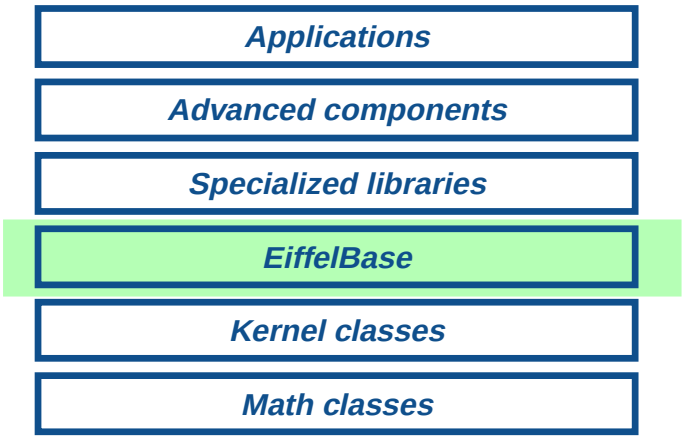


**Figure 6: Place of data structure library**

In the writing of class *STACK* we can now explicitly introduce the model. Using Eiffel techniques for selective export we export the corresponding features — and as a result the corresponding contracts — to a special class *SPECIFICATION* and to that class only. This means that they will not be visible in the official documentation ("contract form" or "short form") of the class.

```
deferred class
    STACK [G]
feature {SPECIFICATION} -- Specification
    model: SEQUENCE [G] is deferred end

feature -- Access
    count: INTEGER is
            -- Number of stack items
        deferred
        ensure
            same_count: Result = model.count
        end
    item: G is
            -- Top element
        require
            not_empty: count > 0
        deferred
        ensure
            is_last: Result = model.last
        end
feature -- Element change
    put (x: G) is
            -- Push x to top of stack.
        require
            not full: not full
        deferred
        ensure
            pushed: item = x
            one_more: count = old count + 1
            extended: model = old model.extended (x)
        end
remove is
            -- Pop top element.
        require
            not_empty: count > 0
        deferred
        ensure
            removed: count = old count – 1
            chopped_off: model = old model.head

        end
invariant
    meaningful_count: count >= 0
    model_exists: model /= Void
end
```

Abstract assertions

Model assertions

Our contracts now have two kinds of assertions, marked in the above extract: the original *abstract assertions*, part of the contracts for the class; and the new *model assertions*, introduced together with the model. We represent the model, in the class, by a new feature *model* (exported to *SPECIFICATION* only and hence not visible by normal clients); the model assertions are expressed in terms of *model*, referring here to features of the IFL class *SEQUENCE* as introduced earlier.

We have the first part of the answer to our general question of "what does it mean to prove a class?":

**Proving a deferred class**

To prove a deferred class is to prove that the model assertions imply the abstract assertions.

The following figure illustrates this process

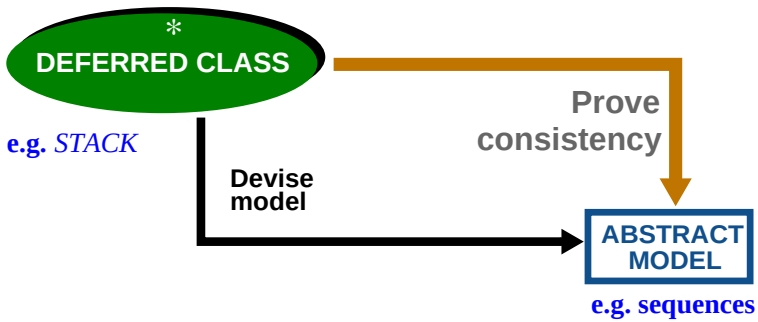


**Figure 7: Proving a deferred class (partial view)**

In the example of *STACK* and its *SEQUENCE* model, the proof is straightforward. For example, considering the postcondition of *put*, we set out to prove the first clause, *item* = *x*. The definition of *item* tells us that *item* is *model*.*last*. The model postcondition tells us that *model* = **old** *model*.*extended* (*x*), so *item* is **old** *model*.*extended* (*x*).*last*. But now the postcondition clause *definition* of *last* in class *SEQUENCE* tells us that this is indeed *x*.

The proof of the second postcondition clause, *count* = **old** *count* + *1*, is similar, using the postcondition of *extended* in *SEQUENCE*.

Note that these proofs of deferred classes really involve mathematical rather than programming properties, since there are no explicit assignment, feature call (the central O-O operation) or control structures.

In practice, it will be convenient to work not directly on the classes, but on mathematical versions that makes the manipulations easier, leading to the following variant of the above figure:

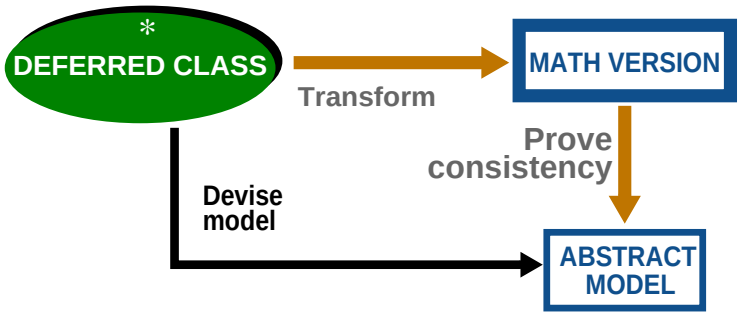


**Figure 8: Proving a deferred class (full view)**

## 8 PROVING THE EFFECTIVE CLASS

We now come back to the effective class *FIXED_STACK*. It provides implementations of the features that were deferred in *STACK*, for example:

```
put (x: G) is
        -- Push x to top of stack.
    require
        not full: not full
    deferred
    ensure
        pushed: item = x
        one_more: count = old count + 1
        extended: model = old model.extended (x)
    end
```

The proof of the model assertions has been taken care of at the level of the deferred class. So what remains to be done is to check the consistency of the implementation vis-à-vis that first model:

**Proving an effective class**

To prove an effective class is to prove that the implementation satisfies the model assertions.

Of course the abstract assertions are those which matter for the clients. But by factoring out the choice of model at the level of the deferred class we have also factored out the proof of the abstract assertions; all that remains of interest at the level of individual effective classes is the model.

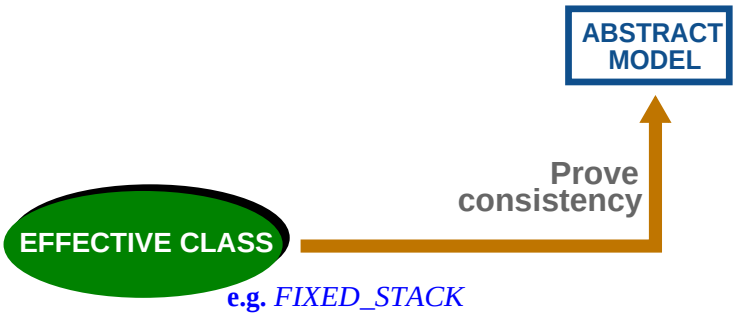


**Figure 9: Proving an effective class (partial view)**

In our example the proof is again straightforward, but requires an axiomatization of programming language constructs such as assignment, feature call and control structures, which the present discussion does not address.

Here too it's more convenient to work on a translation of the class into a mathematical form:

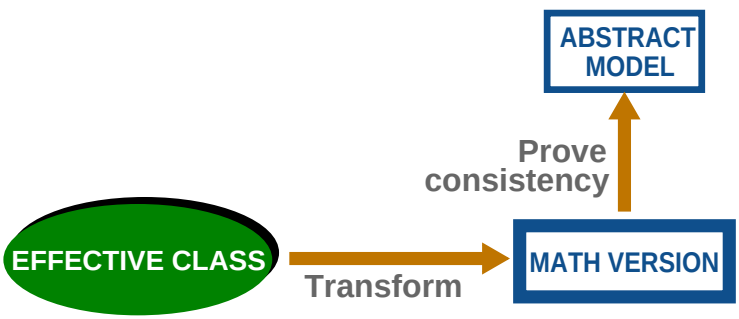


**Figure 10: Proving an effective class (full view)**

# 9 THE OVERALL SCHEME

The following figure combines the preceding ones to present the overall proof scheme:

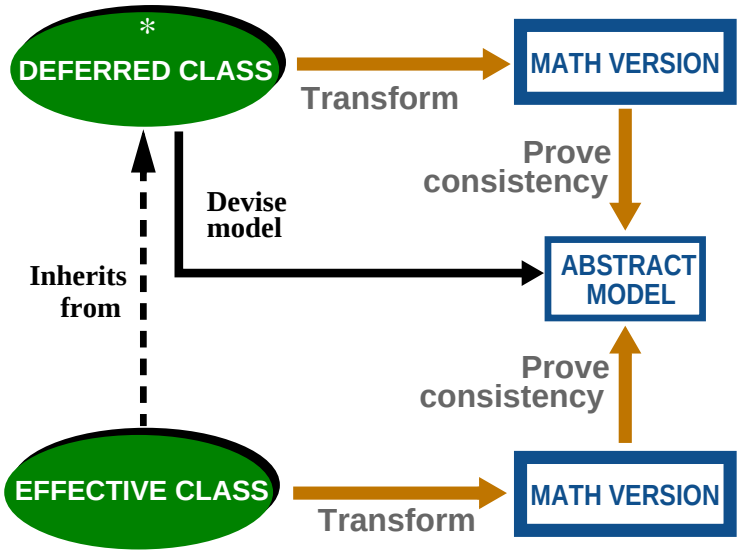


**Figure 11: Scheme for proving classes**

# 10 ISSUES TO BE ADDRESSED

The approach outlined here is only a general scheme. A number of important details remain to be filled in:

- Choice of a mathematical model for proving class properties. Translating into mathematics avoids the symbol manipulation problems that often plague proof efforts.
- Finding the right mix between axiomatic and denotational approaches to modeling the software concepts involved.
- Handling the semantics, in particular (at the level of effective classes) for advanced language constructs.
- Adapting the theory to the form required by the chosen proof engine. Given the amount of proof work involved a fundamental requirement of the approach described here is mechanical support for proofs.
- Scaling up to bigger components.
- Exploring the possibility ot synthesizing the model automatically in some cases, instead of having to invent it for every class.

## 11 STRATEGY

Because of the number of concepts involved in a full-scale library such as EiffelBase (used daily by numerous applications, including large mission-critical systems) we cannot hope to tackle the full current library at once. A related issue is the number of programming language mechanisms that must be modeled.

To avoid an "all-or-nothing" approach, the strategy starts with elementary versions of both the language and the library, working its way up on both sides by introducing advanced language constructs and advanced library concepts one by one, and proving everything correct at each step.

We hope that this strategy will lead us to a fully proved and practically usable version of a contract-equipped library of fundamental computing structures.

## References

[1] Jean-Raymond Abrial: *The B Book*, Cambridge University Press, 2002

[2] Felix Bachmann, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, Kurt Wallnau: *Technical Concepts of Component-Based Software Engineering*, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh (Pennsylvania), Report CMU/SEI-2000-TR-008, available at www.sei.cmu.edu/publications/documents/00.reports/00tr008.html.

[3] Bertrand Meyer, *Reusability*: *the Case for Object-Oriented Design*, in IEEE *Software*, vol. 4, no. 2, March 1987, pages 50-62. Also in: *Selected Reprints in Software*, ed. M. Zelkowitz, IEEE Press, 1987; *Software Reusability*, ed. T. Biggerstaff, Addison-Wesley, 1988; *Object-Oriented Computing*, IEEE Press, 1988.

[4] Bertrand Meyer: *Applying "Design by Contract"*, in IEEE *Computer*, 25, 10, October 1992, pages 40. Also in *Object-Oriented Systems and Applications*, ed. David Rine, IEEE Computer Press, 1994.

[5] Bertrand Meyer: *Eiffel*: *The Language*, Prentice Hall, second printing, 1991.

[6] Bertrand Meyer: *Reusable Software*: *The Base Object-Oriented Libraries*, Prentice Hall, 1994.

[7] Bertrand Meyer: *Object-Oriented Software Construction, second edition*, Prentice Hall, 1997.

[8] Bertrand Meyer, Christine Mingins and Heinz Schmidt: *Providing Trusted Components to the Industry*, in IEEE *Computer*, vol. 31, no. 5, May 1998, pages 104-105.

[9] Bertrand Meyer: *Contracts for Components*, in *Software Development*, vol. 8, no. 7, July 2000, pages 51-53.

[10] Bertrand Meyer: *Towards a Component Quality Model*, working paper at www.inf.ethz.ch/~meyer.

[11] Bertrand Meyer: *Proving Program Pointer Properties*, working papers at www.inf.ethz.ch/~meyer/ongoing/references.

[12] Robert F. Stärk, Egon Borger and Joachim Schmid: *Java and the Java Virtual Machine*: *Definition*, *Verification*, *Validation*, Springer-Verlag, 2001.

[13] Clemens Szyperski: *Component Software*: *Beyond Object Oriented Programming*. Addison-Wesley, 1998.