Eiffel Software

Simple. Elegant. Powerful.

Downloads
Search
Purchase

This site contains older material on Eiffel. For the main Eiffel page, see http://www.eiffel.com.

Building bug-free O-O software: An introduction to Design by Contract(TM)

Eiffel Software is the pioneer of Design by Contract and the Component Revolution. For a more detailed look at Design by Contract and how it can make your code more reliable read this document or watch the presentations.

Overview

The notion of Design by Contract is central in the systematic approach to object-oriented software construction, as embodied in the Eiffel method. This article presents the key ideas.

In our opinion the techniques outlined below are as important as the rest of object technology -- as important as classes, objects, inheritance, polymorphism and dynamic binding, which they complement -- although only a subset of the O-O literature has so far devoted its attention to it. (See the references at the end of this paper.)

To go beyond the theoretical understanding provided by this paper and experience the practical power of its ideas, take a look at the Eiffel environment, which is their direct implementation.

NOTE: For a sobering reminder of the practical consequences of not applying the principles described in this article, see Put it in the Contract: The Lessons of Ariane, originally published in *IEEE Computer* and now available in these Web pages too.

1 - Introduction

When thinking of new software development methods and tools, many people tend to view *productivity* as the major expected benefit. In object technology and especially in Eiffel, productivity benefits follow not just from the immediate benefits of the approach but from its emphasis on quality. In the words of K. Fujino, Vice President of NEC Corporation's C&C Software Development Group:

When quality is pursued, productivity follows

(Quoted in Carlo Ghezzi, Dino Mandrioli and Mehdi Jazayeri, *SoftwareEngineering*, Prentice Hall 1991.)

A major component of quality in software is reliability: a system's ability to perform its job according to the specification (*correctness*) and to handle abnormal situations (*robustness*). Put more simply, reliability is the absence of bugs.

Reliability, although desirable in software construction regardless of the approach, is particularly important in the object-oriented method because of the special role given by the method to reusability: unless we can obtain reusable software components whose correctness we can trust much more than we trust the correctness of usual run-of-the-mill software, reusability is a losing proposition.

How can we build reliable object-oriented software? The answer has several components. Static typing, for example, is a major help for catching inconsistencies before they have had time to become bugs. Such a technique as garbage collection, although sometimes dismissed as an implementation detail, is actually essential too, removing the specter of devious memory management errors. By itself, reusability also helps: if you are able to reuse component libraries produced and (presumably) validated by a reputable outside source, rather than developing your own solution for every single problem you encounter, you can start trusting part of the software no less than you trust the machine on which it runs. In effect, the reusable libraries become part of the "hardware-software machine" (hardware, operating system, compiler).

But this is not enough. To be sure that our object-oriented software will perform properly, we need a systematic approach to specifying and implementing object-oriented software elements and their relations in a software system. This article introduces such a method, known as Design by Contract. Under the Design by Contract theory, a software system is viewed as a set of communicating components whose interaction is based on precisely defined specifications of the mutual obligations -- contracts.

The benefits of Design by Contract include the following:

- A better understanding of the object-oriented method and, more generally, of software construction.

A systematic approach to building bug-free object-oriented systems.

An effective framework for debugging, testing and, more generally, quality assurance.

A method for documenting software components.

Better understanding and control of the inheritance mechanism.

A technique for dealing with abnormal cases, leading to a safe and effective language construct for exception handling.

The ideas developed below are part of Eiffel [1, 3] which the reader is urged to view here not so much as a programming language but rather as a software development method. A longer exposition of the approach may be found in a recent article [2].

2 - Specification and debugging

To improve software reliability, the first and perhaps most difficult problem is to define as precisely as possible, for each software element, what it is supposed to do. The immediate objection is that specifying a module's purpose will not ensure that it will achieve that specification; this is obviously true, but:

- One may reverse this proposition and note that it we don't state what a module should do, there is little likelihood that it will do it. (The law of excluded miracles.)
- In practice, it is amazing to see how far just stating what a module should do goes towards helping to ensure that it does it.

As will be seen below, the presence of a specification, even if it does not fully guarantee the module's correctness, is a good basis for systematic testing and debugging.

The Design by Contract theory, then, suggests associating a specification with every software element. These specifications (or contracts) govern the interaction of the element with the rest of the world.

This presentation will not, however, advocate the use of full formal specifications. Although the work on formal specifications in general is attractive, we settle for an approach in which specifications are not necessarily exhaustive. This has the advantage that the specification language is embedded in the design and programming language (here Eiffel), whereas formal specification languages are typically non-executable or, if they are executable, can only be used for prototypes. Here our criteria are more demanding: we want our language to be used for practical commercial development and hence to yield efficient implementation. This preserves a key property of a well-understood object-oriented process: its **seamlessness**, which makes it possible to use a single notation and a single set of concepts throughout the software lifecycle, from analysis to implementation and maintenance, ensuring better mapping from solution to problem and hence, among other benefits, smoother evolution.

3 - The notion of contract

In human affairs, contracts are written between two parties when one of them (the *supplier*) performs some task for the other (the *client*). Each party expects some benefits from the contract, and accepts some obligations in return. Usually, what one of the parties sees as an obligation is a benefit for the other. The aim of the contract document is to spell out these benefits and obligations.

A tabular form such as the following (illustrating a contract between an airline and a customer) is often convenient for expressing the terms of such a contract:

	Obligations	Benefits
Client	*(Must ensure precondition)* Be at the Santa Barbara airport at least 5 minutes before scheduled departure time. Bring only acceptable baggage. Pay ticket price.	*(May benefit from postcondition)* Reach Chicago.
Supplier	*(Must ensure postcondition)* Bring customer to Chicago.	*(May assume precondition)* No need to carry passenger who is late, has unacceptable baggage, or has not paid ticket price.

A contract document protects both the client, by specifying how much should be done, and the supplier, by stating that the supplier is not liable for failing to carry out tasks outside of the specified scope.

The same ideas apply to software. Consider a software element *E*. To achieve its purpose (fulfil its own contract), *E* uses a certain strategy, which involves a number of subtasks, $t_1, \ldots t_n$. If subtask t_i is non-trivial, it will be achieved by calling a certain routine *R*. In other words, *E* contracts out the subtask to *R*. Such a situation should be governed by a well-defined roster of obligations and benefits -- a contract.

Assume for example that t_i is the task of inserting a certain element into a dictionary (a table where each element is identified by a certain character string used as key) of bounded capacity. The contract will be:

	Obligations	Benefits
Client	*(Must ensure precondition)* Make sure table is not full and key is a non-empty string.	*(May benefit from postcondition)* Get updated table where the given element now appears, associated with the given key.
Supplier	*(Must ensure postcondition)* Record given element in table, associated with given key.	*(May assume precondition)* No need to do anything if table is full, or key is empty string.

This contract governs the relations between the routine and any potential caller. It contains the most important information that can be given about the routine: what each party in the contract must guarantee for a correct call, and what each party is entitled to in return.

So important indeed is this information that we cannot remain satisfied with an informal specification of the contract as above. In the spirit of seamlessness (encouraging us to include every relevant information, at all levels, in a single software text), we should equip the routine text with a listing of the appropriate conditions. Assuming the routine is called *put*, it will look as follows in Eiffel syntax, as part of a generic class *DICTIONARY* [*ELEMENT*]:

```
put (x: ELEMENT; key: STRING) is
                -- Insert x so that it will be retrievable through key.
        require
                count <= capacity
                not key.empty
        do
                ... Some insertion algorithm ...
        ensure
                has (x)
                item (key) = x
                count = old count + 1
        end
```

The ***require*** clause introduces an input condition, or precondition; the ***ensure*** clause introduces an output condition, or postcondition. Both of these conditions are examples of assertions, or logical conditions (contract clauses) associated with software elements. In the precondition, *count* is the current number of elements and *capacity* is the maximum number; in the postcondition, *has* is the boolean query which tells whether a certain element is present, and *item* returns the element associated with a certain key. The notation ***old*** *count* refers to the value of *count* on entry to the routine.

4 - Contracts in analysis

The above example is extracted from a routine describing an implementation (although the notion of dictionary is in fact meaningful independently of any implementation concern). But the concepts are just as interesting at the analysis level. Imagine for example a model of a chemical plant, with classes such as *TANK*, *PIPE*, *VALVE*, *CONTROL_ROOM*. Each one of these classes describes a certain data abstraction -- a certain type of real-world objects, characterized by the applicable features (operations). For example, *TANK* may have the following features:

- Yes/no queries: *is_empty, is_full...*
- Other queries: *in_valve, out_valve* (both of type *VALVE*), *gauge_reading*, *capacity...*
- Commands: *fill, empty, ...*

Then to characterize a command such as *fill* we may use a precondition and postcondition as above:

```
fill is
                -- Fill tank with liquid
        require
                in_valve.open
                out_valve.closed

        deferred             -- i.e., no implementation

        ensure
                in_valve.closed
                out_valve.closed
                is_full
        end
```

This style of analysis avoids a classic dilemma of analysis and specification: either you use a programming notation and run the risk of making premature implementation commitments; or you stick with a higher-level notation ("bubbles and arrows") and you must remain vague, forsaking one of the major benefit of the analysis process, the ability to state and clarify delicate properties of the system. Here the notation is precise (thanks to the assertion mechanism, which may be used to capture the semantics of various operations) but avoids any implementation commitment. (There is no danger of such a commitment in the above example, since what it describes includes no software and indeed no computer yet! Here we are using the notation just as a modeling tool.)

The Business Object Notation, as described by Waldén and Nerson [5], the only O-O method that fully integrates these ideas at the analysis and design level, providing graphical notation for the ideas developed in the present article.

5 - Invariants

Preconditions and postconditions apply to individual routines. Other kinds of assertions will characterize a class as a whole, rather than its individual routines. An assertion describing a property which holds of all instances of a class is called a **class invariant**. For example, the invariant of *DICTIONARY* could state

```
invariant
        0 <= count
        count <= capacity
```

and the invariant of *TANK* could state that *is_full* really means "is approximately full":

```
invariant
        is_full = (0.97 * capacity <= gauge) and
                          gauge <= 1.03 * capacity)
   ... Other clauses ...
```

Class invariants are consistency constraints characterizing the semantics of a class. This notion is important for configuration management and regression testing, since it describes the deeper properties of a class: not just the characteristics it has at at a certain moment of its evolution, but the constraints which must also apply to subsequent changes.

Viewed from the contract theory, an invariant is a general clause which applies to the entire set of contracts defining a class.

6 - Documentation

Another key application of contracts is to provide a standard way to document software elements -- classes. To provide client programmers with a proper description of the interface properties of a class, it suffices to give them a version of the class, known as the **short form**, which is stripped of all implementation information but retains the essential usage information: the contract.

In the EiffelBench environment, you obtain the short form interactively by clicking on the Short button of the Class Tool. The output can be plain text or can be converted to any text processing format (Microsoft's RTF, HTML for Web publishing, MIF or MML for FrameMaker, TEX, troff, Postscript etc.) through one of the environment's predefined filters -- to which you can add any of your own filters since the mechanism is completely open.

The short form retains headers and assertions of exported features, as well as invariants, but discards everything else. For example:

```
class interface DICTIONARY [ELEMENT] feature

 put (x: ELEMENT; key: STRING) is
          -- Insert x so that it will be retrievable
                         -- through key.
                require
                        count <= capacity
                        not key.empty
                 ensure
                        has (x)
                        item (key) = x
                        count = old count + 1

              ... Interface specifications of other features ...

invariant

              0 <= count
              count <= capacity

end -- class interface DICTIONARY
```

This short form serves as the basic tool for documenting libraries and other software elements. It also serves as a central communication tool between developers. We have learned from our customers and from our own experience that emphasis on the short form facilitates software design and project management, as it encourages developers and managers to discuss the key issues (interface, specification, inter-module protocols) rather than internal details.

7 - Testing, debugging, quality assurance

Given a class text equipped with assertions, we should ideally be able to prove mathematically that the routine implementations are consistent with the assertions. In the absence of realistic tools to do this, we can settle for the next best thing, which is to use assertions for testing.

Compilation options enable the developers, class by class, what effect assertions should have if any: no assertion checking (under which assertions have no effect at all, serving as a form of standardized comments), preconditions only (the default), preconditions and postconditions, all of the above plus class invariants, all assertions.

These mechanisms provide a powerful tool for finding mistakes. Assertion monitoring is a way to check what the software does against what its author thinks it does. This yields a productive approach to debugging, testing and quality assurance, in which the search for errors is not blind but based on consistency conditions provided by the developers themselves.

The availability of these mechanisms is in my experience one of the most significant consequences of moving to this technology. It causes a dramatic drop in the number of bugs, and a new attitude of developers towards software reliability.

8 - Contracts and inheritance

An important consequence of the Design by Contract theory is to yield a better understanding of the central object-oriented notions of inheritance, polymorphism, redefinition and dynamic binding.

A class *B* which inherits from a class *A* may provide a new declaration for a certain inherited feature *r* of *A*. For example a specialized implementation of *DICTIONARY* might redefine the algorithm for *put*. Such redefinitions are potentially dangerous, however, as the redefined version could in principle have a completely different semantics. This is particularly worrisome in the presence of polymorphism, which means that in the call

```
a.r
```

the target *a* of the call, although declared statically of type *A*, could in fact be attached at run time to an object of type *B*. Then **dynamic binding** implies that the *B* version of *r* will be called in such a case.

This is a form of subcontracting: *A* subcontracts *r* to *B* for targets of the corresponding type. But a subcontractor must be bound by the original contract. A client which executes a call under the form

```
    if a.pre then
            a.r
    end
```

must be guaranteed the contractually promised result: the call will be correctly executed since the precondition is satisfied (assuming that *pre* implies the precondition of *r*); and on exit *a.post* will be true, where *post* is the postcondition of *r*.

The **principle of subcontracting** follows from these observations: a redefined version of *r* may keep or weaken the precondition; it may keep or strengthen the postcondition. Strengthening the precondition, or weakening the postcondition, would be a case of "dishonest subcontracting" and could lead to disaster. The Eiffel language rules for assertion redefinition [3] support the principle of subcontracting.

These observations shed light on the true significance of inheritance: not just a reuse, subtyping and classification mechanism, but a way to ensure compatible semantics by other means. They also provide useful guidance as to how to use inheritance properly.

9 - Exception handling

Among the many other applications of the contract theory we may note that the theory leads naturally to a systematic approach to the thorny problem of exception handling -- handling abnormal cases.

A software element is always a way to fulfil a certain contract, explicit or not. An exception is the element's inability to fulfil its contract, for any reason: a hardware failure has occurred, a called routine has failed, a software bug makes it impossible to satisfy the contract.

In such cases only three responses make sense:

1. Retrying: an alternative strategy is available. The routine will restore the invariant and and make another attempt, using the new strategy.

2. Organized panic: no such alternative is available. Restore the invariant, terminate, and report failure to the caller by triggering a new exception. (The caller will itself have to choose between the same three responses.)

3. False alarm: it is in fact possible to continue, perhaps after taking some corrective measures. This case seldom occurs (regrettably, since it is the easiest to implement!).

The exception mechanism follows directly from this analysis. It is based on the notion of "rescue clause" associated with a routine, and of "retry instruction", which implements retrying. This is similar to clauses that occur in human contracts, to allow for exceptional, unplanned circumstances. If there is a Rescue clause, any exception occurring during the routine's execution will interrupt the execution of the body (the **do** clause) and start execution of the Rescue clause. The clause contains one or more instructions; one of them is a **retry**, which will cause re-execution of the routine's body (the **do** clause). An integer local entity such as *failure* is always initialized to zero on routine entry (but not, of course, after a **retry**).

Here is an example illustrating the mechanism (see [2, 3] for details). We assume a low-level procedure *unsafe_transmit* for transmitting a message over a network. We have no control over that procedure but know that it may fail, in which case we want to try again, although after 100 unsuccessful attempts we will give up, passing on the exception to our caller. The Rescue/Retry mechanism supports this simply and directly:

```
attempt_transmission (message: STRING) is
                -- Attempt to transmit message over a communication line
                -- using the low-level (e.g. C) procedure unsafe_transmit,
                -- which may fail, triggering an exception.
                -- After 100 unsuccessful attempts, give up (triggering
                -- an exception in the caller).
        local
                failures: INTEGER
        do
                unsafe_transmit (message)

        rescue
                failures := failures + 1
                if failures < 100 then
                   retry
                end
        end
```

10 - Further developments

This article has provided an overview of the basic ideas of Design by Contract. This is a very active area of application and further research, with several books in preparation. Two areas of development are:

- **Concurrency and distribution**: the principles of Design by Contract yield a fascinating solution, described elsewhere in these Web pages, to the problem of concurrent and distribution object-oriented programming (avoiding the so-called "inheritance anomaly" and other non-issues of O-O concurrent computation, resulting from a misunderstanding of object technology). An article [4] describes in detail the Eiffel approach to concurrent computation, based on the Design by Contract concepts and currently being implemented for ISE Eiffel 4.2. (See the new edition of [1] for the most up-to-date description.)

- **An extended specification language**, allowing the expression of a richer set of assertions.

Design by Contract has already been widely applied; the theory provides a powerful thread throughout the object-oriented method, and addresses many of the issues that many people are encountering as they start applying O-O techniques and languages seriously: what kind of "methodology" to apply, on what concepts to base the analysis step, how to specify components, how to document object-oriented software, how to guide the testing process and, most importantly, how to build software so that bugs do not show up in the first place.

In software development, reliability should be built-in, not an afterthought.

Bibliography

[1] Bertrand Meyer: *Object-Oriented Software Construction*, Prentice Hall, 1988. Extensively revised second edition now out.

[2] Bertrand Meyer: *Applying "Design by Contract*, in *Computer (IEEE)*, vol. 25, no. 10, October 1992, pages 40-51.

[3] Bertrand Meyer: *Eiffel: The Language*, Prentice Hall, 1992.

[4] Bertrand Meyer: *Systematic Concurrent Object-Oriented Programming*, in *Communications of the ACM*, vol. 36, no. 9, September 1993, pages 56-80.

[5] Kim Waldén and Jean-Marc Nerson, *Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems*, Prentice Hall, 1995.

Note: an earlier version of the present article appeared in the *Hotline on Object Technology*.