DESIGN BY CONTRACT TM , ASSERTIONS, EXCEPTIONS

Eiffel directly implements the ideas of Design by Contract TM , which enhance software reliability and provide a sound basis for software specification, documentation and testing, as well as exception handling and the proper use of inheritance.

Design by Contract basics

A system — a software system in particular, but the ideas are more general — is made of a number of cooperating components. Design by Contract states that their cooperation should be based on precise specifications — contracts — describing each party's expectations and guarantees.

In DbC, we identify three different kinds of expressions:

- Preconditions
- Postconditions
- Invariants

Let's examine each in more detail.

Preconditions

Preconditions specify conditions that must hold before a method can execute. As such, they are evaluated just before a method executes. Preconditions involve the system state and the arguments passed into the method.

Preconditions specify obligations that a client of a software component must meet before it may invoke a particular method of the component. If a precondition fails, a bug is in a software component's client.

Postconditions

In contrast, postconditions specify conditions that must hold after a method completes. Consequently, postconditions are executed after a method completes. Postconditions involve the old system state, the new system state, the method arguments, and the method's return value.

Postconditions specify guarantees that a software component makes to its clients. If a postcondition is violated, the software component has a bug.

Invariants

An invariant specifies a condition that must hold anytime a client could invoke an object's method. Invariants are defined as part of a class definition. In practice, invariants are evaluated anytime before and after a method on any class instance executes. A violation of an invariant may indicate a bug in either the client or the software component.

Assertions, inheritance, and interfaces

All assertions specified for a class and its methods apply to all subclasses as well. You can also specify assertions for interfaces. As such, all assertions of an interface must hold for all classes that implement the interface.

An Eiffel contract is similar to a real-life contract between two people or two companies, which it is convenient to express in the form of tables listing the expectations and guarantees. Here for example is how we could sketch the contract between a homeowner and the telephone company:

provide_service	**OBLIGATIONS**	**BENEFITS**
Client	(Satisfy precondition:) Pay bill	(From postcondition:) Get telephone service
Supplier	(Satisfy postcondition:) Provide telephone service	(From precondition:) No need to provide anything if bill not paid

Note how the obligation for each of the parties maps onto a benefit for the other. This will be a general pattern.

The client's obligation, which protects the supplier, is called a *precondition*. It states what the client must satisfy before requesting a certain service. The client's benefit, which describes what the supplier must do (assuming the precondition was satisfied), is called a *postcondition*.

In addition to preconditions and postconditions, contract clauses include class invariants, which apply to a class as a whole. More precisely a class invariant must be ensured by every creation procedure (or by the default initialization if there is no creation procedure), and maintained by every exported routine of the class.

Expressing assertions

Eiffel provides syntax for expressing preconditions (*require*), postconditions (***ensure***) and class invariants (***invariant***), as well as other assertion constructs studied later : loop invariants and variants, check instructions.

Here is a partial update of class *ACCOUNT* with more assertions:

```
indexing
        description: "Simple bank accounts"
class
        ACCOUNT
feature -- Access
        balance: INTEGER
                -- Current balance
deposit_count: INTEGER is
                -- Number of deposits made since opening
do
... As before ...
end
```

and

```
feature -- Element change
        deposit (sum: INTEGER) is
                -- Add sum to account.
require
        non_negative: sum >= 0
do
... As before ...
ensure
        one_more_deposit:
                deposit_count = old deposit_count + 1
        updated: balance = old balance + sum
end

feature {NONE} -- Implementation
        all_deposits: DEPOSIT_LIST
                -- List of deposits since account's opening.
invariant
        consistent_balance: (all_deposits /= Void) implies
                                        (balance = all_deposits total)
        zero_if_no_deposits: (all_deposits = Void) implies
```

```
                                        (balance = 0)
end -- class ACCOUNT
```

Using contracts for built-in reliability

What are contracts good for? Their first use is purely methodological. By applying a discipline of expressing, as precisely as possible, the logical assumptions behind software elements, you can write software whose reliability is built-in: software that is developed hand-in-hand with the rationale for its correctness.
This simple observation — usually not clear to people until they have practiced Design by Contract thoroughly on a large-scale project — brings as much change to software practices and quality as the rest of object technology.

Run-time assertion monitoring

Contracts in Eiffel are not just wishful thinking. They can be monitored at run time under the control of compilation options.

It should be clear from the preceding discussion that contracts are not a mechanism to test for special conditions, for example erroneous user input. For that purpose, the usual control structures (if deposit_sum >= 0 then ...) are available, complemented in applicable cases by the exception handling mechanism reviewed next. An assertion is instead a *correctness condition* governing the relationship between two software modules (not a software module and a human, or a software module and an external device). If *sum* is negative on entry to *deposit* , violating the precondition, the culprit is some other software element, whose author was not careful enough to observe the terms of the deal. Bluntly:

Assertion Violation rule
A run-time assertion violation is the manifestation of a bug.

To be more precise:

• A precondition violation signals a bug in the client, which did not observe its part of the deal.

• A postcondition (or invariant) violation signals a bug in the supplier — the routine
— which did not do its job

That violations indicate bugs explains why it is legitimate to enable or disable assertion monitoring through mere compilation options: for a correct system — one without bugs — assertions will always hold, so the compilation option makes no difference to the semantics of the system.

But of course for an incorrect system the best way to find out where the bug is
—
or just that there is a bug — is often to monitor the assertions during development and testing. Hence the presence of the compilation options, which ISE's EiffelStudio lets you set separately for each class, with defaults at the system and cluster levels:

• **no**: assertions have no run-time effect.

• **require**: monitor preconditions only, on routine entry.

• **ensure**: preconditions on entry, postconditions on exit.

• **invariant**: like **ensure**, plus class invariant on both entry and exit for qualified calls.

• **all**: like **invariant**, plus **check** instructions, loop invariants and loop variants

An assertion violation, if detected at run time under one of these options other than the first, will cause an exception ("Exception handling"). Unless the software has an explicit "retry" plan as explained in the discussion of exceptions, the violation will cause produce an exception trace and cause termination (or, in EiffelStudio, a return to the environment's browsing and debugging facilities at the point of failure). If present, the label of the violated subclause will be displayed, to help identify the problem.

The default is **require**. This is particularly interesting in connection with the Eiffel method's insistence on reuse: with libraries such as EiffelBase, richly equipped with preconditions expressing terms of use, an error in the *client software* will often lead, for example through an incorrect argument, to violating one of these preconditions. A somewhat paradoxical consequence is that even an application developer who does not apply the method too well (out of carelessness, haste, indifference or ignorance) will still benefit from the presence of contracts in someone else's library code.

During development and testing, assertion monitoring should be turned on at the highest possible level. Combined with static typing and the immediate feedback of compilation techniques such as the Melting Ice Technology, this permits the development process mentioned in the section "Quality and functionality" where errors are exterminated at birth. No one who has not practiced the method in a real project can imagine how many mistakes are found in this way; surprisingly often, a violation will turn out to affect an assertion that was just included for goodness' sake, the developer being convinced that it could never "possibly" fail to be satisfied.

By providing a precise reference (the description of what the software is supposed to do) against which to assess the reality (what the software actually does), Design by Contract profoundly transforms the activities of debugging, testing and quality assurance.
When releasing the final version of a system, it is usually appropriate to turn off assertion monitoring, or bring it down to the ***require*** level. The exact policy depends on the circumstances; it is a tradeoff between efficiencyconsiderations, the potential cost of mistakes, and how much the developers and quality assurance team trust the product. When developing the software, however, you should always assume — to avoid loosening your guard — that in the end monitoring will be turned off.

The contract form of a class

Another application of assertions governs documentation. Environment mechanisms, such as clicking the *Contract Form* icon in EifffelStudio, will produce, from a class text, an abstracted version which only includes the information relevant for client authors. Here is the contract form of class ***ACCOUNT*** in the latest version given:

```
indexing
        description: "Simple bank accounts"
class interface
        ACCOUNT
feature -- Access
        balance: INTEGER
                -- Current balance
        deposit_count: INTEGER
                -- Number of deposits made since opening
feature -- Element change
        deposit (sum: INTEGER)
                -- Add sum to account.
        require
                non_negative: sum >= 0
        ensure
                one_more_deposit: deposit_count = old deposit_count + 1
                updated: balance = old balance + sum
invariant
                consistent_balance: balance = all_deposits total
        end -- class interface ACCOUNT
```

.

The words class interface are used instead of just class to avoid any confusion with actual Eiffel text, since this is documentation, not executable software. (It is in fact possible to generate a compilable variant of the Contract Form in the form of a deferred class, a notion defined later.)

Compared to the full text, the Contract Form of a class (also called its “short form”) retains all its interface properties, relevant to client authors:

• Names and signatures (argument and result type information) for exported features.

• Header comments of these features, which carry informal descriptions of their purpose. (Hence the importance, mentioned in section 4, of always including such comments and writing them carefully.)

• Preconditions and postconditions of these features (at least the subclauses involving only exported features).

• Class invariant (same observation).

The following elements, however, are not in the Contract Form: any information about non-exported features; all the routine bodies (do clauses, or the external and once variants s; assertion subclauses involving non-exported features; and some keywords not useful in the documentation, such as ***is*** for a routine.
In accordance with the Uniform Access principle , the Contract Form does not distinguish between attributes and argument-less queries. In the above example, ***balance*** could be one or the other, as it makes no difference to clients, except possibly for performance.

The Contract Form is the fundamental tool for using supplier classes in the Eiffel method. It enables client authors to reuse software elements without having to read their source code. This is a crucial requirement in large-scale industrial developments.

The Contract Form satisfies two key requirements of good software documentation:

• It is truly abstract, free from the implementation details of what it describes and concentrating instead on its functionality.

• Rather than being developed separately — an unrealistic requirement, hard toimpose on developers initially and becoming impossible in practice if we expect the documentation to remain up to date as the software evolves — the documentation is extracted from the software itself. It is not a separate product but a different view of the same product. This prolongs the **Single Product** principle that lies at the basis of Eiffel's seamless development model .

The Contract Form is only one of the relevant views. EiffelStudio, for example, generates graphical representations of system structures, to show classes and their relations — client, inheritance — according to the conventions of BON (the Business Object Notation). In accordance with the principles of seamlessness and reversibility, EiffelStudio lets you both work on the text, producing the graphics on the fly, or work on the graphics, updating the text on the fly; you can alternate as you wish between these two modes. The resulting process is quite different from more traditional approaches based on separate tools: an analysis and CASE workbench, often based on UML, to deal with an initial "bubble-and-arrow" description; and a separate programming environment, to deal with implementation aspects only. In Eiffel the environment provides consistent, seamless support from beginning to end.

The Contract Form — or its variant the Flat-Contract Form, which takes account of inheritance ("Flat and Flat-Contract Forms", are the standard form of library documentation, used extensively, for example, in the book Reusable Software . Assertions play a central role in such documentation by expressing the terms of the contract. As demonstrated a contrario by the widely publicized $500- million crash of the Ariane-5 rocket launcher in June of 1996, due to the incorrect reuse of a software module from the Ariane-4 project, reuse without a contract documentation is the path to disaster. Non-reuse would, in fact, be preferable.

Exception handling

Another application of Design by Contract governs the handling of unexpected cases.
The vagueness of many discussions of this topic follows from the lack of a precise definition of terms such as "exception". With Design by Contract we are in a position to be specific:

• Any routine has a contract to achieve.

Its body defines a strategy to achieve it — a sequence of operations, or some other control structure involving operations. Some of these operations are calls to routines, with their own contracts; but even an atomic operation, such as the computation of an arithmetic operation, has an implicit contract, stating that the result will be representable.

• Any one of these operations may ***fail,*** that is to say be unable to meet its contract;for example an arithmetic operation may produce an overflow (a non-representable result).

• The failure of an operation is an **exception** for the routine that needed the operation.

• As a result the routine may fail too — causing an exception in its own caller. Note the precise definitions of the two key concepts, failure and exception. Although failure is the more basic one — since it is defined for atomic, non-routine operations —the definitions are mutually recursive, since an exception may cause a failure of the recipient routine, and a routine's failure causes an exception in its own caller.

Why state that an exception "may" cause a failure? It is indeed possible to "rescue" a routine from failure in the case of an exception, by equipping it with a clause labeled **rescue**, as in:

```
read_next_character (f: FILE) is
        -- Make next character available in last_character ;
        -- if impossible, set failed to True.
    require
        readable: file readable
    local
        impossible: BOOLEAN
    do
        if impossible then
            failed := True
        else
            last_character := low_level_read_function (f)
        end
    rescue
        impossible := True
        retry
    end
```
.
This example includes the only two constructs needed for exception handling: **rescue** and **retry.** A **retry** instruction is only permitted in a rescue clause; its effect is to start again the execution of the routine, without repeating the initialization of local entities (such as *impossible* in the example, which was

initialized to *False* on first entry). Features *failed* and *last_character* are assumed to be attributes of the enclosing class.

This example is typical of the use of exceptions: as a last resort, for situations that should not occur. The routine has a precondition, *file readable* , which ascertains that the file exists and is accessible for reading characters. So clients should check that everything is fine before calling the routine. Although this check is almost always a guarantee of success, a rare combination of circumstances could cause a change of file status (because a user or some other system is manipulating the file) between the check for readable and the call to low_level_read_function . If we assume this latter function will fail if the file is not readable, we must catch the exception.

A variant would be

```
local
        attempts: INTEGER
do
        if attempts < Max_attempts then
                last_character := low_level_read_function (f)
        else
                failed := True
        end
rescue
        attempts := attempts + 1
        retry
end
```

which would try again up to *Max_attempts* times before giving up.

The above routine, in either variant, never fails: it always fulfills its contract, which states that it should either read a character or set failed to record its inability to do so.
In contrast, consider the new variant

```
local
        attempts: INTEGER
do
                last_character := low_level_read_function (f)
rescue
        attempts := attempts + 1
        if attempts < Max_attempts then
                retry
        end
end
```

with no more role for failed . In this case, after Max_attempts unsuccessful attempts, the routine will execute its ***rescue*** clause to the end, with no ***retry*** (the if having no ***else*** clause). This is how a routine **fails**. It will, as noted, pass on the exception to its caller.

Such a rescue clause should, before terminating, restore the invariant of the class so that the caller and possible subsequent ***retry*** attempts from higher up find the objects in a consistent state. As a result, the rule for an absent ***rescue*** clause — the case for the vast majority of routines in most systems — is that it is equivalent to

```
rescue
default_rescue§
```

where procedure *default_rescue* comes from *ANY* , where it is defined to do nothing; in a system built for robustness, classes subject to non-explicitly-*rescued* exceptions should redefine *default_rescue* (perhaps using a creation procedure, which is bound by the same formal requirement) so that it will always restore the invariant.

Behind Eiffel's exception handling scheme lies the principle — at first an apparent platitude, but violated by many existing mechanisms — that a routine should either **succeed** or **fail**. This is in turn a consequence of Design by Contract principles: succeeding means being able to fulfill the contract, possibly after one or more **retry;** failure is the other case, which must always trigger an exception in the caller. Otherwise it would be possible for a routine to miss its contract and yet return to its caller in a seemingly normal state. That is the worst possible way to handle an exception.

Concretely, exceptions may result from the following events:

• A routine failure (**rescue** clause executed to the end with no **retry**), as just seen.

• Assertion violation, if for a system that runs with assertion monitoring on.

• Attempt to call a feature on a void reference: x f (...), the fundamental computational mechanism, can only work if x is attached to an object, and will cause an exception otherwise.

• Developer exception, as seen next.

• Operating system signal:arithmetic overfolow; no memory available for a

requested creation or clone — even after garbage collection has rummaged everything to find some space. (But no C/C++-like “wrong pointer address”, which cannot occur thanks to the statically typed nature of Eiffel.)
.
It is sometimes useful, when handling exceptions in **rescue** clauses, to ascertain theexact nature of the exception that got the execution there. For this it is suffices to inherit from the Kernel Library class *EXCEPTIONS* , which provides queries such aş *exception* , giving the code for the last exception, and symbolic names for all such codes, such as *No_more_memory* . You can then process different exceptions differently by testing *exception* against various possibilities. The method strongly suggests, however, that exception handling code should remain simple; a complicated algorithm in a **rescue** clause is usually a sign that the mechanism is being misused.

Class *EXCEPTIONS* also provides various facilities for fine-tuning the exception facilities, such as a procedure *raise* that will explicitly trigger a “developer exception” with a code than can then be detected and processed.

Exception handling helps produce Eiffel software that is not just correct but robust, by planning for cases that should not normally arise, but might out of Murphy’s law, and ensuring they do not affect the software’s basic safety and simplicity.

Other applications of Design by Contract

The Design by Contract ideas pervade the Eiffel method. In addition to the applications just mentioned, they have two particularly important consequences:

• They make it possible to use Eiffel for analysis and design. At a high level ofabstraction, it is necessary to be precise too. With the exception of BON, object- oriented analysis and design methods tend to favor abstraction over precision. Thanks to assertions, it is possible to express precise properties of a system (“At what speed should the alarm start sounding?”) without making any commitment to implementation. The discussion of deferred classes (“Applications of deferred classes”) will show how to write a purely descriptive, non-software model in Eiffel, using contracts to describe the essential properties of a system without any computer or software aspect.

• Assertions also serve to control the power of inheritance-related mechanisms — redeclaration, polymorphism, dynamic binding — and channel them to correct uses by assigning the proper semantic limits.