The method

Design by Contract™ is a method to reason about software that accompanies the programmer at any step of the software development. Even if it is called "design" by contract, it does not only target the design stage of an application. It is useful as a method of analysis and design, but it also helps during implementation because the software specification will have been clearly stated using "assertions" (boolean expressions). Design by Contract™ is also useful to debug and test the software against this specification.

The idea of Design by Contract™ is to make the goal of a particular piece of software explicit. Indeed, when developers start a new project and build a new application, it is to satisfy the need of a client, match a certain specification. The Design by Contract™ method suggests writing this specification down to serve as a "contract" between the clients (the users) and the suppliers (the programmers).

This idea of **contract** defined by some obligations and benefits is an analogy with the notion of contract in business: the supplier has some obligations to his clients and the clients also have some obligations to their supplier. What is an obligation for the supplier is a benefit for the client, and conversely.

Different kinds of contracts

There are three main categories of contracts: preconditions, postconditions, and class invariants:

- **Preconditions** are conditions under which a routine will execute properly; they have to be satisfied by the client when calling the routine. They are an obligation for the clients and a benefit for the supplier (which can rely on them). A precondition violation is the manifestation of a bug in the client (which fails to satisfy the precondition).

  Precondition clauses are introduced by the keyword **require** in an Eiffel routine:

```
routine_name (arg_1: TYPE_1; arg_2: TYPE_2): RETURN_TYPE is
        -- Comment
    require
        tag_1: boolean_expression_1
        tag_2: boolean_expression_2
    do
        ... Implementation here (set of instructions)
    end
```

Structure of an Eiffel routine with precondition

  Each precondition clause is of the form "tag: expression" where the tag can be any identifier and the expression is a boolean expression (the actual assertion). The tag is optional; but it is very useful for documentation and debugging purposes.

- **Postconditions** are properties that are satisfied at the end of the routine execution. They are benefits for the clients and obligations for the supplier. A postcondition violation is the manifestation of a bug in the supplier (which fails to satisfy what it guarantees to its clients).

Postcondition clauses are introduced by the keyword **ensure** in an Eiffel routine:

```
routine_name (arg_1: TYPE_1; arg_2: TYPE_2): RETURN_TYPE is
        -- Comment
    do
        ... Implementation here (set of instructions)
    ensure
        tag_1: boolean_expression_1
        tag_2: boolean_expression_2
    end
```

Structure of an Eiffel routine with postcondition

Of course, a routine may have both preconditions and postconditions; hence both a **require** and an **ensure** clause (like in the previous *BOOK* class).

- **Class invariants** capture global properties of the class. They are consistency constraints applicable to all instances of a class. They must be satisfied after the creation of a new instance and preserved by all the routines of the class. More precisely, it must be satisfied after the execution of any feature by any client. (This rule applies to **qualified calls** of the form $x.f$ only, namely client calls. Implementation calls — **unqualified calls** — and calls to non-exported features do not have to preserve the class invariant.)

  Class invariants are introduced by the keyword **invariant** in an Eiffel class

```
class
    CLASS_NAME
feature -- Comment
    ...
invariant
    tag_1: boolean_expression_1
    tag_2: boolean_expression_2
end
```

Structure of an Eiffel class with class invariant

There are three other kinds of assertions:

- **Check instructions**: Expressions ensuring that a certain property is satisfied at a specific point of a method's execution. They help document a piece of software and make it more readable for future implementers.

  In Eiffel, check instructions are introduced by the keyword **check** as follows:

```
routine_name (arg_1: TYPE_1; arg_2: TYPE_2): RETURN_TYPE is
        -- Comment
    do
        ... Implementation here (set of instructions)
        check
            tag_1: boolean_expression_1
            tag_2: boolean_expression_2
        end
        ... Implementation here (set of instructions)
    end
```

Structure of an Eiffel routine with check instruction

- **Loop invariants**: Conditions, which have to be satisfied at each loop iteration and when exiting the loop. They help guarantee that a loop is correct.
- **Loop variants**: Integer expressions ensuring that a loop is finite. It decreases by one at each loop iteration and has to remain positive.

This appendix will show the syntax of loop variants and invariants later when introducing the syntax of loops.

See "Syntax of loops", page 383.

Benefits

The benefits of Design by Contract™ are both technical and managerial. Among other benefits we find:

- *Software correctness*: Contracts help build software right in the first place (as opposed to the more common approach of trying to debug software into correctness). This first use of contracts is purely methodological: the Design by Contract™ method advises you to think about each routine's requirements and write them as part of your software text. This is only a method, some guidelines for software developers, but it is also perhaps the main benefit of contracts, because it helps you design and implement correct software right away.
- *Documentation*: Contracts serve as a basis for documentation: the documentation is automatically generated from the contracts, which means that it will always be up-to-date, correct and precise, exactly what the clients need to know about.
- *Debugging and testing*: Contracts make it much easier to detect "bugs" in a piece of software, since the program execution just stops at the mistaken points (faults will occur closer to the source of error). It becomes even more obvious with assertions tags (i.e. identifiers before the assertion text itself). Contracts are also of interest for testing because they can serve as a basis for black-box test case generation.
- *Management*: Contracts help understand the global purpose of a program without having to go into the code in depth, which is especially appreciable when you need to explain your work to less-technical persons. It provides a common vocabulary and facilitates communication. Besides, it provides a solid specification that facilitates reuse and component-based development, which is of interest for both managers and developers.

A.3 MORE ADVANCED EIFFEL MECHANISMS

Let's describe more advanced Eiffel mechanisms, typically the facilities on which the pattern library relies on.

See "Pattern Library", page 26.

Book library example

This section uses an example to introduce these mechanisms. Because we talked about books in the previous section, here is the example of a library where users can borrow and return books.

Here is a possible implementation of an Eiffel class *LIBRARY*:

Class representation of a book library

```
indexing
        description: "Library where users can borrow books"
class
        LIBRARY
inherit
        ANY
                redefine
                        default_create
                end
```