8.6 WORKING OUT CONTRACTING CONDITIONS

The BON approach focuses on the theory of software contracting, which we consider the most important principle in existence today for constructing correct, reliable software. No other technique proposed so far has the potential to turn software development into the long-awaited discipline of engineering envisioned by Douglas McIlroy in his contribution to the now famous NATO conference in 1968 [McIlroy 1976], and meet the challenge for an, as yet unrealized, software components industry.

The theory, called design by contract [Meyer 1988a, Meyer 1992c], is actually an elegant synthesis of the essential concepts in three major fields of computing research: object-orientation [Nygaard 1970],[14] abstract data types [Guttag 1977, Gougen 1978], and the theory of program verification and systematic program construction [Floyd 1967, Hoare 1969, Dijkstra 1976].

Design by contract

Most pieces of software text in today's systems are only partially understood by their developers. The central task of each subprogram may be clear enough (and sometimes even documented), but as every experienced programmer knows it is the unusual cases, or even the ones that are never supposed to occur, that present the real problems. Since it is never exactly spelled out who is responsible for the exceptional cases—the supplier of a subprogram, or its clients—important prerequisites for various algorithms are often either checked in many places, or not checked at all.

The general feeling of distrust resulting from this practice has lead to a desperate style of blind checking known as *defensive programming,* which leads to even less understandable software with more errors because of the complexity introduced by the added redundant code. So the only solution is to create instead an atmosphere of mutual trust by specifying precisely who is responsible for what part of a complex system behavior. This is what design by contract is all about.

The idea is to treat each subprogram as a subcontractor who undertakes to deliver some service (specified by a postcondition), but only if certain prerequisites are fulfilled (the precondition). The key opening the gate to future trust and order—so that you can finally *know* that you are right when designing a program instead of just guessing [Mills 1975]—is not as one may think the postcondition (which specifies what the supplier will do), but instead the precondition (which specifies what the supplier will *not* do).

To take an example, suppose you are to define a subprogram to calculate the square root of a real number. If you expect this program to work under all conditions, you are in fact mixing two completely different tasks into one:

- Finding and returning the square root of a non-negative number.

---

[14] Simula, the first object-oriented language, not only introduced the remarkable concepts of inheritance and dynamic binding already in 1967, but was also the direct inspiration of almost all later work on abstract data types. It included the strong typing of Algol 60, but had generalized the single stack model into a multiple stack machine, which enabled encapsulation of autonomous objects.

- Returning something reasonable when the input turns out to be negative (assuming the output must be real).

For the first task we have a number of well-understood and efficient numerical methods dating all the way back to Newton to choose from as supplier. For the second task, we do not have a clue. Obviously the client has made a mistake, and there is no way we can know what is a reasonable response.

Therefore, the only approach that makes any sense is to lift the second problem off the shoulders of the supplier (who is not competent to handle it anyway) and instead let it be the responsibility of the client not to ask impossible questions. This may sometimes require explicit testing on the client side, but if it does, there is no better place to do it. Usually the context will lead the client to know without testing that the precondition is indeed fulfilled, something which is never true for the supplier.

Contracting as a mutual benefit

The software contracting model has much in common with standard practices in human society. For example, suppose you are in Stockholm and must deliver an important package to an address at the other end of the city.[15] Then you may either deliver the package yourself, or engage a mail carrier to do it for you. If you choose the latter alternative and employ the services of “Green Delivery” (Stockholm’s bicycle courier), the standard agreement between you and the courier looks like the one shown in figure 8.5. When two parties agree on something in detail, the resulting contract protects both sides:

- It protects the client by specifying *how much* must be done.
- It protects the supplier by specifying *how little* is acceptable.

The obligations of one party become the benefits of the other. As an aside

Party	Obligations	Benefits
Client	Provide package of maximum weight 35 kg, maximum dimensions parcel: $50 \times 50 \times 50$ cm, document: 60×80 cm. Pay 100 SEK.	Get package delivered to recipient within central city limits in 30 minutes or less without having to worry about bad weather or traffic jams.
Supplier	Deliver package to recipient in 30 minutes or less, regardless of traffic and weather conditions.	No need to deal with deliveries too big, too heavy, or not prepaid.

Figure 8.5 A contract

[15] The example is a slight modification of the one used in [Meyer 1992c].

(provided the contract covers everything) each obligation will also bring an additional benefit: if the condition says you must do X, then X is *all* you need to do. This may be called the No Hidden Clauses rule: sticking to the minimum requirements of the contract is always safe for each party.

Regardless of the No Hidden Clauses principle there are usually external laws and regulations whose purpose it is to prevent unfair contract clauses. For example, if your package happens to contain a famous oil painting by Anders Zorn the courier service is not permitted to drop it in the nearest garbage container simply because it violates the precondition by measuring 80 by 90 centimeters.

Such external regulations, which are part of the general context in which the contractors work, correspond to the class invariants of software contracts.

Laws of subcontracting

Polymorphism with dynamic binding is the main key to software flexibility. It has the power to remove most of the discrete case analysis so error prone and vulnerable to future changes—yet so abundant in traditional software. However, flexibility is meaningless unless the resulting software is correct, and polymorphism can be very dangerous in this respect.

Unless we are very careful when redefining an inherited operation, we may easily end up with a system where only some of the implementations that may be dynamically invoked will actually produce the expected result. What is there to prevent a redefined *area* function from returning, in some cases, the diameter instead? Without clear semantic rules, nothing but fuzzy naming conventions and the folklore of software engineering.

The problem is more subtle than it may appear at first sight, because even if every descendant class has a fully visible and correct specification of its behavior, chaos may still ensue. For example, if we need to compute the area of a list of geometric figures referred to by an entity of type *LIST* [*FIGURE*], all we can look at as client is the specification of the *area* operation as it appears in class *FIGURE*. During execution many different specialized versions of *area* may be called dynamically, but we cannot check their corresponding specifications when writing the list traversing code, if for no other reason than because some of the corresponding classes may not yet exist!

Therefore, we must have strict rules that guarantee that *any* future descendant class (whose operations may be invoked on our behalf whether we like it or not) must fulfill the promises that were given by its ancestors. This leads directly to the laws of subcontracting:

- A descendant class must fulfill the class invariant of the ancestor.

- A descendant class may never *weaken* the postcondition of a redefined feature (since this would mean delivering less than specified by the ancestor).
- A descendant class may never *strengthen* the precondition of a redefined feature (since this would mean imposing restrictions on the client not specified by the ancestor).

Note that nothing prevents a descendant class from strengthening postconditions (doing even better than promised) or weakening preconditions (imposing even fewer restrictions).

Note also that the above rules must be obeyed for every ancestor in the case of multiple inheritance, and will therefore prevent the combination of incompatible abstractions. This is extremely important for building the complicated inheritance lattices needed by, for example, the general data structure libraries of strongly typed language environments.

Where to put consistency checking

Design by contract offers an alternative to the blind checking of defensive programming by specifying a clear division of responsibility between client and supplier regarding the checking of various details of system consistency:

- The *client* is responsible for guaranteeing the precondition when calling.
- The *supplier* is responsible for guaranteeing the postcondition when returning.

Therefore, we have to choose where to put our tests for consistency. Either a condition is part of the precondition and must be ensured by the client, or it is removed from the precondition and must then be handled by the supplier.

Which alternative to choose must be decided case by case based on many factors, but the guiding star should always be the resulting simplicity and understandability of the system architecture. A rule of thumb is that if most clients need their own special treatment depending on some condition, it is better to put it in the precondition, while if the behavior alternatives are more or less standard for most clients, it may be simpler for the supplier to deal with them.

Classes as specification elements

As was argued in chapter 2, we should not strive to make the specification of an object-oriented system independent of its design, since this would defeat its purpose. Maintaining two entirely different descriptions (one for the system specification and one for its implementation) does not make sense, because

specifications of large systems become large no matter what language we choose. Therefore, separating the two worlds will only give us inconsistency problems and more difficult maintenance for rapidly evolving systems.

So the specification elements used must in the end be translatable into object-oriented expressions, involving feature calls on objects of the classes which are part of the system design. These classes are the only abstractions that can capture the complex behavior of the system through simple notation (provided the design is good) as using an independent system specification would necessitate starting from scratch. Instead, specification and implementation must share *the same abstraction base,* since the executable code should only be the innermost part of a layered design of abstractions.

Partial recursive specification will not tell us the whole truth about a system, but it will tell us nothing but the truth, and it has the flexibility and incrementality we seek. Any other approach breaks the seamlessness and is in our view doomed to fail as a road to mastering the industrial development of large and complex systems.[16] Complete specification of large industrial systems will probably never become feasible anyway, if only because of the constant changes involved.

Run-time monitoring of assertions

An important side effect of the recursive specification approach described is that assertions may be translated into procedural code and monitored during execution. Basic boolean expressions map directly to programming language primitives, while the first-order quantifications of BON assertions may be implemented as functions. We take the simple class *PERSON* in figure 8.6 as an example. Its invariant expresses that if you are a person, then each of your children has you for one of its parents. (@ is the BON graphical symbol for current object.)

The corresponding Eiffel code, which may be generated by a case tool, is shown in figure 8.7. The control structure of Eiffel, from−until−loop−end, should be self-explanatory and the *LINKED_LIST* class is a cursor structure, which may be traversed by invoking features *start* and successions of *forth* until the state *after* is reached (meaning the cursor is positioned just after the list). The standard feature *item* returns the list element at the cursor position and *Result* is a predefined local variable containing a function's return value. (Entities of type *BOOLEAN* are initialized to **false** by default, so *Result* can be used directly in the second function of figure 8.7.)

---

[16] We are talking about general systems development here; certain critical or highly specialized software may of course still at times benefit from other techniques.

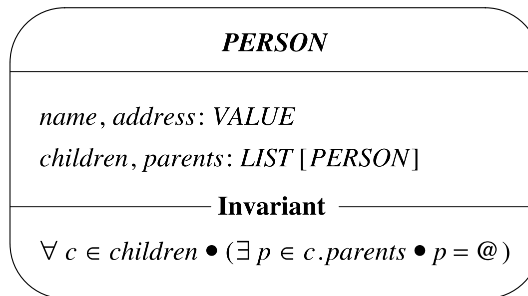

Figure 8.6 Consistency requirement: your children are really yours

```
class PERSON
feature
    name, address: VALUE
    children, parents: LINKED_LIST [PERSON]
    generated_assertion_1: BOOLEAN is
    do
        from
            children.start; Result := true
        until
            children.after
        loop
            Result := Result and generated_subassertion_1 (children.item)
            children.forth
        end
    end
    generated_subassertion_1 (c: PERSON): BOOLEAN is
    do
        from
            c.parents.start
        until
            c.parents.after
        loop
            Result := Result or (c.parent.item = Current)
            c.parents.forth
        end
    end
invariant
    generated_assertion_1
end
```

Figure 8.7 Class with generated assertion routines

When specification elements are implemented using procedural code, we must be extremely careful not to introduce any side effects, since this may change the semantics of the system when the assertions are monitored. Moreover, since *any* feature returning interesting information about the system state is a potential specification element, the rule of side-effect-free functions acquires an even greater importance.

Systematic exception handling

The contract theory also enables a very powerful exception handling mechanism to be applied during system execution. Since routines are not just small pieces of reusable software text, but precisely specified individual implementations, it is possible to introduce a notion of *failure.* Failure occurs when an execution of a routine is for some reason unable to fulfill its part of the contract.

An exception in a routine can be triggered in one of three ways: a supplier returns failure, an assertion is violated, or a signal is received from the surrounding hardware/operating system. (Note that assertion violation includes violation of the postcondition just before returning, as well as violation of a supplier precondition just before calling, since the latter is the *client's* responsibility.)

Exceptions may be processed by *handlers,* which will restore the class invariant for the current object and then either admit failure or else execute the whole operation again (after, for example, setting some flags). Admitting failure means triggering, in turn, a failure exception in the caller.

Since this book focuses on analysis and design and the details of exception handling are closely linked with the programming environment, we will not go further. The interested reader is referred to [Meyer 1992c, Meyer 1992a].

Finally, the violation of an assertion means that some implementation did not behave according to the specification. It is important to understand that this is a sign of a software (or possibly hardware) error. Things that may be expected to happen, no matter how seldom, must be part of the system specification and should therefore be handled in the main logic of the class features.