Below is section 8.7 dpmk af_nrcp6mdrfc F_lb`mmi* *M`hcar+Mpgclrcb Pcosgpckclrq*,

8.7 LOGICAL CONSTRAINTS VERSUS PREMATURE ORDERING

As the stack example illustrated, object-oriented specifications stay away from premature time-order decisions by focusing on object types (classes) and their operations (queries and commands), without making an early commitment to the order of executing these operations.

8.7.1 A contract-based specification

Here again is the essence of "main success scenario" in the insurance example (the original with more details was on page 130).

1. A reporting party who is aware of the event registers a loss to the insurance company.
2. A clerk receives and assigns claim to a claims agent.
3. The assigned claims adjuster:
   - 3.1 Conducts an investigation.
   - 3.2 Evaluates damages.
   - 3.3 Sets reserves.
   - 3.4 Negotiates the claim.
   - 3.5 Resolves the claim and closes it.

(Note about the example's Environment: as noted on page 130, a *reserve* in the insurance business is a monetary amount that an insurer, when receiving a claim, sets aside as a financial provision estimating the financial liability that may eventually result from the claim. Reserves are important as an accounting precaution for difficult cases that may cause a prolonged analysis, or even litigation, and incur a high cost which will only be known at the end of the process.)

As a specification, this scenario is trying to express a few useful things; for example, you must set reserves before starting to negotiate the claim. But it expresses them in the form of a strict sequence of operations, a *temporal* constraint which does not cover the wide range of legitimate scenarios. As in the stack example, describing a few such scenarios is useful as part of requirements elicitation, but to specify the resulting requirements it is more effective to state the *logical* constraints. Here is a sketch of how the class *INSURANCE_CLAIM* could specify them in the form of contracts.

```
class INSURANCE_CLAIM feature
    -- Boolean queries (all with default value False):
  is_investigated, is_evaluated, is_reserved, is_agreed, is_imposed, is_resolved: BOOLEAN
  investigate
          -- Conduct investigation on validity of claim. Set is_investigated.
    deferred
    ensure
      is_investigated
    end
  evaluate
          -- Assess monetary amount of damages.
    require
      is_investigated
    deferred
    ensure
      is_evaluated
      -- Note: is_investigated still holds (see the invariant at the end of the class text).
    end
  set_reserve
          -- Assess monetary amount of damages. Set is_reserved.
    require
      is_investigated
      -- Note: we do not require is_evaluated.
    deferred
    ensure
      is_reserved
    end
```

```
    negotiate
            -- Assess monetary amount of damages. Set is_agreed only if negotiation
            -- leads to an agreement with the claim originator.
        require
            is_reserved
            is_evaluated
        deferred
        ensure
            is_reserved
            -- See the invariant for is_evaluated and is_investigated.
        end
    impose (amount: INTEGER)
            -- Determine amount of claim if negotiation fails. Set is_imposed.
        require
            not is_agreed
            is_reserved
        deferred
        ensure
            is_imposed
        end
    resolve
            -- Finalize handling of claim. Set is_resolved.
        require
            is_agreed or is_imposed
        deferred
        require
            is_resolved
        end
invariant       -- "⇒" is logical implication.
        is_evaluated ⇒ is_investigated
        is_reserved ⇒ is_evaluated
        is_resolved ⇒ is_agreed or is_imposed
        is_agreed ⇒ is_evaluated
        is_imposed ⇒ is_evaluated
        is_imposed ⇒ not is_agreed -- Hence, by laws of logic, is_agreed ⇒ not is_imposed
end
```

Notice the interplay between the preconditions, postconditions and class invariant, and the various boolean-valued queries they involve (*is_investigated*, *is_evaluated*, *is_reserved*…). You can specify a strict order of operations o_1, o_2…, as in a use case, by having a sequence of assertions p_i such that operation o_i has the contract clauses **require** p_i and **ensure** p_{i+1}; but assertions also enable you to specify a much broader range of allowable orderings as all acceptable.

The class specification as given is only a first cut and leaves many aspects untouched. It will be important in practice, for example, to include a query *payment* describing the amount to be paid for the claim; then *impose* has the postcondition *payment* = *amount*, and *negotiate* sets a certain amount for *payment*. Such aspects are not hard to complete but have been left out above to keep the example short. (Remember that the Companion to this Handbook has many practical specification examples.) The following subsections describe the key points to be learned from the example as it stands.

Even in this simplified form, the specification includes a few concepts that the original use case left unspecified, in particular the notion of imposing a payment (through the command *impose*) if negotiation fails. Using a logical style typically uncovers such important questions and provides a framework for answering them, helping to achieve one of the principal goals of requirements engineering ("Requirements Questions Principle", page 22).

8.7.2 Logical constraints are more general than sequential orderings

The specific sequence of actions described in the use case ("main success scenario") is compatible with the logical constraints: you can check that in the sequence

```
                -- The following is the "main success scenario", already reproduced on page 147.
investigate
evaluate
set_reserve
negotiate
resolve
```

the postcondition of each step implies the precondition of the next one (the first has no precondition). In other words, the temporal specification satisfies the logical one. But you can also see that prescribing this order is a case of overspecification (as defined in 4.7.3, page 59): other orderings also satisfy the logical specification. It may be possible for example — subject to confirmation by Subject-Matter Experts — to change the order of *evaluate* and *set_reserve*, or to perform these two operations in parallel.

The specification does cover the fundamental sequencing constraints; for example, the pre- and postcondition combinations imply that investigation must come before evaluation and resolution must be preceded by either negotiation or imposition. But they avoid the non-essential constraints which, in the use case, were only an artifact of the sequential style of specification, not a true feature of the problem.

The logical style is also more conducive to conducting a fruitful dialogue with domain experts and stakeholders:

- With a focus on use cases, the typical question from a requirements engineer (business analyst) is "*do you do A before doing B*?" Often the answer will be contorted, as in "*usually yes, but only if C, oh and sometimes we might start with B if D holds, or we might work on A and B in parallel*…", leading to vagueness and to more complicated requirements specifications.
- With logic-based specifications, the two fundamental question types are: "*what conditions do you need before doing B*?" and "*does doing A ensure condition C?*". They force stakeholders to assess their own practices and specify precisely the relations between operations of interest.

8.7.3 What use for scenarios?

Use cases and more generally scenarios, while more restrictive than logical specifications, remain important as complements to specifications. They serve as both input and output to more abstract requirements specifications (such as OO specifications with contracts):

- As **input** to requirements: initially at least, stakeholders and Subject-Matter Experts often find it intuitive to describe typical system interactions, and their own activities, in the form of scenarios. Collecting such scenarios is an invaluable requirements elicitation technique. The requirements engineer must remember that any such scenario is just one example walk through the system, and must *abstract* from these examples to derive general logical rules.
- As **output** from requirements: from an OO specification with its contracts, the requirements engineers can produce valid use cases. "Valid" means that the operation at every step satisfies the applicable precondition, as a consequence of the previous steps' postconditions and of the class invariant. The requirements engineers can then submit these use cases to the SMEs and through them to stakeholders to confirm that they make sense, update the logical conditions if they do not (to rule out bad use cases), and check the results they are expected to produce.

8.7.4 Where do scenarios fit?

While many teams will prefer to write scenarios (for the purposes just described) in natural language, it is possible to go one step further and, in an object-oriented approach to requirements, gather scenarios in classes.

We will see later in this chapter (section 8.8) that an OO requirements specification includes classes of several kinds. One of the categories (see 8.8.1) is expressly intended for scenarios. A *scenario class* will include a number of routines, each typically representing a use case or user story. Such routines are expressed in the notation of an object-oriented programming language, as convenient and expressive here, to describe processes of interaction with a system, as it is for its traditional use of describing programs.

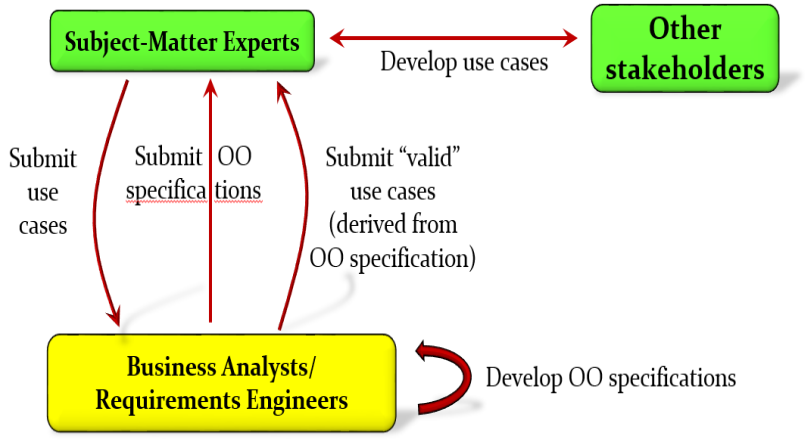
A scenario routine will use these mechanisms to express a use case or user story in terms of the features of other (non-scenario) classes of the OO specification. Scenario classes, taking advantage of OO structuring mechanisms, particularly inheritance, can provide groupings at different levels, as introduced in 7.3: epics and use case slices.

Scenario classes are examples of **specification drivers**. Classes in the other categories describe types from the environment, the project or the system (its design or its implementation). A specification-driver class, for its part, specifies patterns of *exercising* the mechanisms provided by such classes. Besides scenario classes, another example of a specification driver is a *test* driver, which triggers features of implementation classes and checks the outcomes through test oracles. Specification drivers generalize this notion to any class whose purpose is to exercise features of a given set of classes. They serve various purposes, in particular for software verification; scenario classes are a particularly useful application of the idea.

8.7.5 Different roles for different techniques

The preceding discussions suggests that a complementarity exists not only between scenarios (such as use cases) and OO specifications but also between the *roles* involved. Let us now review this human aspect.

Scenarios, while not appropriate as requirements specifications, are a productive tool in the dialog with Subject-Matter Experts and other non-IT-specialist stakeholders since they reflect the intuition of how one will work with the system. Using them alongside OO specifications (both as inputs to them and outputs from them) leads to a productive distribution of tasks represented by the following figure.



Working with both use cases and OO specifications

This scheme is subject to variation; in particular, depending on matters specific to each project and discussed elsewhere in this Handbook:

- All stakeholders might qualify as Subject-Matter Experts; then the top-right box goes away.
- Even if "other stakeholders" exists as a distinct category, there is no universal rule forcing BAs (business analysts) to go through designated SMEs only to obtain information from stakeholders. Each project sets its own policy on this matter.

The diagram, covering the general case, shows the use of scenarios and OO specifications by various categories of requirements builders. One possible complaint against logic-based specifications is that they are not to everyone's taste. In reality, anyone can learn to use them (the underlying concepts are very simple), but the focus on logical reasoning is one of the reasons for having a separate category of requirements engineers or business analysts. These requirements experts should fully master the corresponding techniques. The distribution of tasks then becomes clear, as already outlined above:

- SMEs and other stakeholders may prefer, at least initially, to reason in terms of scenarios.
- It is the task of BAs to abstract these specific cases into general specifications, using the techniques of object-oriented requirements and contracts.
- A constant back-and-forth takes place; in particular, BAs show the consequences of their OO modeling to the other two groups, to check that their understanding is correct.

Many specialists of non-IT areas will (as experience shows) naturally take to the OO style and the associated logical mechanisms. But even if they do not, the BAs should.

It is in fact paradoxical to encounter resistance to such methods based on the argument that "*it is too hard for our people*". The basic justification for having requirements experts, be they called business analysts or requirements engineers, is precisely that they master techniques unique to the engineering of requirements. If requirements were all about listing individual scenarios — "*the user can do this and then that, the system will respond by this and then that*" — there would hardly be a need for a separate profession. Good requirements analysis is about abstracting from special cases to general specifications. The best path is to abstract from scenarios to contracted classes.

8.7.6 Towards formal methods and abstract data types

The intrusion of logic into the descriptions — see for example the invariant of class *INSURANCE_CLAIM* on page 149 — is a first step towards making use of mathematical techniques in requirements. Simple mathematics, based on elementary logic, but mathematics all the same.

The general application of mathematical techniques to reasoning about software is known as *formal methods* and goes beyond this first step. Chapter 9 is devoted to formal methods and followed by a discussion in chapter 10 of a particular formal notion, *abstract data type*, the mathematical counterpart to the software notion at the core of object technology: class.

8.7.7 "But it's design!"

Although at this stage you should have no doubt about the suitability of object-oriented principles for requirements, it is still useful (if only for discussions you may have with colleagues steeped in more traditional approaches) to dispel a common misunderstanding: that such specifications as illustrated in the preceding example, based on classes and contracts (plus inheritance to organize the classes into hierarchies), are premature designs or even — heaven forbid! — premature implementations. If so, they would be subject to the reproach of "overspecification" (4.7.3, page 59).

Of course they are not. In the present discussion, classes are modeling tools at the requirements level. OO modeling means that you describe elements (from the Project, Environment, Goals or System) through types of objects and the applicable operations. These classes are purely descriptive; they need not contain any inkling of future design decisions.

The "*but it's premature design*!" reaction is ironic when used to advocate use cases, which present a clear risk of premature design since they specify ordered sequences of operations, leading to the temptation of writing programs along those same ordering patterns.

8.7.8 Towards seamlessness

While free from design and implementation considerations, the classes written for requirements purposes are still classes and can be expressed in an object-oriented language that also supports design and implementation classes. The benefit here is to avoid harmful changes of concepts and notations when going through successive steps of the software lifecycle.

This approach is known as seamless development and discussed in more detail in connection with the place of requirements in the software lifecycle (see "Seamless development", 12.6, page 218). One of its beneficial consequences is **reversibility**: having everything expressed in a single notation makes it easier to update the requirements at any stage in the project, even deep into design, implementation or verification, in line with the Requirements Evolution Principle ("The evolution of requirements", 2.1.4, page 23).

Without having seen yet the full discussion of seamlessness, we may take advantage of the preceding discussion to analyze the diverse nature of classes that appear in requirements and at other stages of software development. This analysis leads us, in the next section, to an important classification of the various kinds of class.