JOT
JOURNAL OF
OBJECT TECHNOLOGY

Object-Oriented Requirements: a Unified Framework for Specifications, Scenarios and Tests

Maria Naumcheva[*], **Sophie Ebersold**[*], **Alexandr Naumchev**[†], **Jean-Michel Bruel**[*], **Florian Galinier**[‡], **and Bertrand Meyer**[§]
[*] IRIT, University of Toulouse, France
[†]Unaffiliated
[‡]Spilen Corporation, France
[§]Constructor Institute, Schaffhausen, Switzerland

ABSTRACT A paradox of requirements specifications as dominantly practiced in the industry is that they often claim to be object-oriented (OO) but largely rely on procedural (non-OO) techniques. Use cases and user stories describe functional flows, not object types. To gain the benefits provided by object technology (such as extendibility, reusability, and reliability), requirements should instead take advantage of the same data abstraction concepts – classes, inheritance, information hiding – as OO design and OO programs.

Many people find use cases and user stories appealing because of the simplicity and practicality of the concepts. Can we reconcile requirements with object-oriented principles and get the best of both worlds?

This article proposes a unified framework. It shows that the concept of class is general enough to describe not only "object" in a narrow sense but also scenarios such as use cases and user stories and other important artifacts such as test cases and oracles.

Having a single framework opens the way to requirements that enjoy the benefits of both approaches: like use cases and user stories, they reflect the practical views of stakeholders; like object-oriented requirements, they lend themselves to evolution and reuse.

KEYWORDS Software requirements, use cases, scenarios, scenario-based testing, object-oriented requirements, specifications

1. Introduction

A good software system is an effective solution to a well-understood problem. As software engineering has progressed, it has become increasingly clear that achieving software quality involves achieving quality on both the solution side and the problem side: together with excellent design, implementation and project management techniques, a successful project requires an excellent description of the problem, known as the **requirements** of the system.

JOT reference format:
Maria Naumcheva, Sophie Ebersold, Alexandr Naumchev, Jean-Michel Bruel, Florian Galinier, and Bertrand Meyer. *Object-Oriented Requirements: a Unified Framework for Specifications, Scenarios and Tests*. Journal of Object Technology. Vol. 22, No. 1, 2023. http://dx.doi.org/10.5381/jot.2023.22.1.3

While a considerable body of knowledge exists about requirements engineering, the discipline as practiced in industry has not yet experienced the considerable progress that *object-oriented* (OO) concepts, methods, languages and tools have brought to solution-side tasks. The purpose of this article is to help advance the state of the art in requirements engineering through the application of OO ideas, and to show that this approach subsumes other widely applied techniques such as use cases and user stories. The research questions we tackle in this paper are (i) how to specify OO requirements? (ii) how to unify them with scenarios?

The *modeling power* of object technology has played a large part in its success for design and implementation, and can be even more useful for requirements. It comes in particular from the OO decision to define the architecture of systems on the basis of object types connected by well-defined relations ("client" and

"inheritance"), using structuring principles such as information hiding and Design by Contract. These notions are clear and simple, and help ensure that the structure of the software is closely related to the structure of the problem description. For example, a library-management system will have such classes as LIBRARY, BOOK, PATRON and CATALOG, directly reflecting concepts of the problem domain and their mutual relations. This property, known as *direct mapping*, helps achieve goals of reliability, extendibility and reusability. It is even more useful for requirements, which are all about modeling external systems and their environments.

In practice, however, requirements engineering does not widely apply OO techniques. The phrase "object-oriented requirements" itself is used (sometimes as "object-oriented *analysis*"), but often to describe techniques such as use cases and user stories, which are not object-oriented (Larman 2012).

The relationship between OO and non-OO techniques (the latter also known as "procedural") can be one of complementarity rather than confrontation. The core observation is that data abstraction, the idea at the heart of OO technology, has so much power that some of it remains untapped. One does not need to treat such techniques as scenarios, use cases, user stories and test scripts as independent of the OO framework or even antagonistic to it; they can instead find their natural place in it. The result is a unified approach to requirements engineering which has the potential to bring to this discipline the same remarkable advances that have proved so beneficial to solution-side aspects of software development. The first contribution of this paper is the evidence of scenarios limitations and how OO paradigm could address them. The second contribution is the description of OO requirements and a framework supporting OO requirements specification. The third contribution is the application of the approach to the case study of a Roborace, which can serve as a guide for practitioners who want to use the unified framework. These contributions allow requirements engineers to benefit from the advantages of conceptual consistency and unification of notations, and from the potential of scalability and extendibility of OO requirements.

Section 2 presents common approaches to requirements and their limitations. Section 3 describes fundamental OO techniques, that form the basis of the Unified Approach. Section 4 introduces the notion of object-oriented requirement. Section 5 describes in details a unified framework, based on OO requirements. Section 6 constitutes a proof of concept by applying the framework to a real-case application. Section 7 compares the approach with related works. Section 8 describes motivation for applying OO techniques to requirements, limitations of the approach and of the case study, and future work directions. Finally, section 9 summarizes the contributions and concludes the paper.

2. Scenarios and tests

Requirements in industry generally rely on different techniques, particularly use cases and user stories. In modern software development approaches, tests are viewed also as requirements artifacts.

2.1. Use cases

Use cases have become one of the major formalisms for expressing requirements thanks to Ivar Jacobson's work and his 1992 book (Jacobson et al. 1992). Their spread happened at about the same time as the spread of OO methods for programming and design. A use case describes a unit of requirements in the form of a possible scenario of user interaction with the system.

There are several use case notations; the examples in this article will rely on a notation due to Cockburn (Cockburn 2000). Table 1 presents an example, related to a library system. It specifies a scenario whereby a library user borrows a book from the library, with such steps as placing the book on hold, then checking it out, and returning it by the specified deadline.

The core part is the **main success scenario**, giving the sequence of steps of the use case. Additional possibilities of the notation, not used in the example, include:

- Steps consisting of several sub-steps, similar to calling routines in programming.
- Conditional steps, in which the use case follows either of two sub-scenarios depending on the outcome of a certain condition.

Such mechanisms suggest a strong analogy between a use case and an *algorithm* or a program. The two concepts also have differences: a use case (and other kinds of scenarios such as user stories, reviewed next) describes the interaction between a human actor and the system, whereas an algorithm or program is meant to be carried out by a computer.

The **level** entry characterizes the level of abstraction. A use case can describe a process at many levels, from the highest (a bird's eye view of an overall business process, meant to be complemented by further use cases for the details) down to the detailed descriptions of the system's actual operation.

A **precondition** is a limiting condition governing the applicability of the use case.

A **trigger** is an event that starts the use case.

Note the difference between the last two concepts: a precondition is a condition that must hold for the use case to be applicable, but does not by itself cause its execution; a trigger does. (The precondition is necessary, the trigger is sufficient.)

The **success guarantee** characterizes the state resulting from successful execution of the use case, for the "main success scenario". In the terminology of mathematical software verification, from which the term "precondition" is borrowed, it would be called a "postcondition".

An **extension** describes a departure from the "main success scenario". Extensions serve two separate purposes:

- As in the example, an extension can specify an alternative to the "main success scenario", to be applied when that scenario hits a condition that prevents it from proceeding normally.
- Extensions also support the reuse of elements common to several use cases, which can then be divided into a base use case, covering the common elements, and specific extensions.

Name	Borrow_a_book
Scope	System
Level	Business summary
Primary actor	Patron
Context of use	The patron wants to check out a book
Preconditions	The book is available
Trigger	The patron finds in a library catalog the book he wants to borrow and requests the system to place a hold on this book
Main success scenario	* The system changes the book status to on_hold * The patron checks out the book * The patron returns the book
Success guarantee	The patron has borrowed the book and returned it within the checkout duration.
Extensions	A. The book is not available *The system denies placing a hold on the book B. The hold expires due to exceeding the maximum hold duration. * The system changes the hold status to "expired" and the book becomes available C. The patron cancels the hold * The book status changes to available D. The patron does not return the book within the maximum check-out duration * The book status changes to overdue * The patron returns the book
Stakeholders and interests	Patron (borrows a book) Library personnel (enforces adherence to library policies)

Table 1 Use Case description for `Borrow_a_book`

The use case lists, towards the beginning, the **main actor** responsible for carrying out instances of the scenario. It concludes with a list of **stakeholders**: others who may be affected.

2.2. User stories

A use case is a complete path taken by an actor through the system. User stories (Jacobson et al. 2011) also express a typical interaction with the system, but at a much smaller level of granularity. They play an important role in requirements in **agile methods**, which promote incremental program construction: the basic agile development iteration involves a developer picking the next item from a list of functions to be implemented, implementing it, and moving on to the next one. A use case is generally too complex for such atomic units of development.

The standard format for a user story includes three elements:

- A role ("**As a...**"), corresponding to the main actor of a use case.
- A desired function ("**I want to...**"), part of the system's behavior.
- A business purpose ("**so that...**"), corresponding to one of the goals of a system.

An example user story in a library system is:

As a *patron,* **I want to** *check out a book* **so that** *I can read it at home*.

An alternative way of expressing it is in tabular form as presented in Table 2.

Role	Patron
Desired function	Check out a book
Business purpose	Read at home

Table 2 User Story Example

2.3. Use case 2.0 and use case slices

Use Case 2.0 (Schaaf 2012), a revision of the original use case methodology, combines ideas of use cases and user stories through the notion of use case slice. A use case slice is a selected part of a use case, which is also a unit of testing (usable in a test-driven design methodology). Breaking down a use case into slices makes it possible to consider it at different phases of development and to build the system incrementally.

As an example, a slice of the use case *Borrow_a_book* is *Overdue_checkout* (Table 3). It outlines its narratives as a set of flows: the basic flow of *Borrow_a_book* and one of its extensions (Alternative Flow D), "*The patron does not return the book within the maximum check out duration.*"

Name	Overdue_checkout
State	Scoped
Priority	Should
Flows	BF: "*Borrow a book*" + AF D: "*The patron does not return the book within the maximum check-out duration*"
Tests	The checkout duration exceeds the limit
Estimate	3/20

Table 3 Use case slice description of *Overdue_checkout*

The standard format of a use case slice includes:

- A **name**, used to track it through the development cycle.
- A **state**, with possible values *scoped*, *prepared*, *analyzed*, *implemented* or *verified*.
- A **priority**, expressed by the MoSCoW acronym: Must, Should, Could, Would.
- References: **flows** and **tests**.
- An **estimate** of the work needed to implement the slice.

2.4. Unit tests

An important development of software engineering in the past two decades, fostered in particular by agile methods, has been the recognition of test cases as essential artifacts of the software process. The various "xUnit" frameworks, where "x" represents a target programming language (for example, JUnit for Java), describe a test case in the form of a small program element listing a condition to be tested and an "oracle" specifying the correct expected result.

In a traditional, "waterfall" view of software engineering, requirements and test cases play entirely different roles and appear at opposite ends of the lifecycle: requirements at the beginning, tests at the end (in a final "verification and validation" step, often called V&V). The only relationship between them is that a test case, devised in a late stage of the project, will typically exercise a particular requirement, specified much earlier.

With modern views of software development, iterative and incremental, the concepts move closer to each other. In the extreme view (coming from the "Extreme Programming" approach to development) of *test-driven development* (Beck 2003), unit tests actually become the requirements, pushing aside more traditional forms of requirements. Even in less radical approaches, tests share many of the properties expressed as use cases or user stories.

The following listing is a Java test for the example library system (it corresponds to the requirement "*available books can be placed on hold by only one patron at any given time*" given in section 3.5 below).

```
public class HoldingAvailableBooksTest {
  private Book b; private Patron p1, p2; private
    LibraryBranch lb;
  private Library l;

  $@$BeforeEach
  public void setUp() {
    b = new Book("Crime and Punishment", "Fedor
    Dostoyevsky", "978-1703766172");
    p1 = new Patron("Ted"); p2 = new Patron("Fred
    ");
    lb = new LibraryBranch("Squirrel Hill");
    l = new Library("Carnegie Library of
    Pittsburgh");
    l.addBranch(lb); l.addPatron(p1); l.addPatron
    (p2);
  }

  $@$Test
  public void testHolding() {
    l.placeBookOnHold(b, p1, lb); l.
    placeBookOnHold(b, p2, lb);
    assertTrue(l.bookIsOnHold(b, p1, lb));
    assertFalse(l.bookIsOnHold(b, p2, lb);
} }
```

The `setUp()` method instantiates concrete input objects for the test. Then `testHolding()` executes the method under test and checks the correctness of the objects in the resulting state. The `@BeforeEach` annotation directs the unit testing framework to execute the `setUp()` method before running any tests, decorated with `@Test` annotations.

The role of tests extends beyond requirements (the focus of this article); note in particular the widely accepted role of the *regression test suite*, which collects all tests run on versions of a system since the project's inception. Another idea that has gained wide acceptance under the influence of agile methods is a milder form of Test-Driven Development: the rule that every addition to the code must also include a new test (even if we avoid misconstruing that test for a specification). Some agile approaches go so far as to accept only code and tests as legitimate artifacts (Poppendieck & Poppendieck 2003).

2.5. Benefits and limitations of scenarios and tests for requirements

The techniques reviewed above have gained front-row seats in modern software development. Use cases and user stories are important as checks on requirements, to verify that the requirements, expressed in any form, do cover the most important cases. They are not, however, a substitute for these requirements.

A good mathematical textbook presents, along with every important theorem, examples of its consequences. A good physics textbook presents, along with every important property,

examples of its applications. They help make the theorem or property more concretely understandable. No one, however, would accept the examples as a replacement for the theorem. Use cases are to a requirement what an example is to a theorem and a test to a specification.

A scenario or test describes only one path of system usage; the number of actual paths is, for any significant system, staggeringly large or theoretically infinite. Requirements should specify the system's behavior in all cases, not just the inevitably small subset of cases that have been foreseen.

The difficulty of software and particularly of software requirements comes from the myriad of variants that a system must be prepared to handle. One of the prime goals of requirements is to help ensure the system's correctness. In practice, the most serious bugs (violations of correctness) do not generally arise from the common example scenarios typically included in use cases, user stories and unit tests, since the testing phase will naturally focus on them; they arise out of less obvious cases, which such examples often leave out. Only by trying for a more abstract form of specification, covering all cases, can one expect to identify such problematic cases.

Even use cases, the most advanced form of scenario, suffer from this limitation. A use case describes only one path of interaction with the system, or a set of related cases if it takes advantage of conditional branching and extensions. In practice, use cases tend to focus on the most desirable cases, in particular the "main success scenario" and possibly some of its variants as illustrated earlier in section 2.1). While one can add extensions covering such situations, or write new use cases to address them, there are so many combinations for any non-trivial system that it is impossible to include all such extensions, or even a representative subset. These observations also apply to user stories and tests.

Scenario-based techniques suffer from another limitation: their exaggerated reliance on time ordering, which often leads to overspecification. A use case will often specify a sequential contract (step B must occur after step A) when in fact a logical constraint would suffice (step B requires a certain condition, which step A ensures). Turning logical constraints into sequential ones is often too restrictive. Section 5.1 discusses this point further. (Also note Glinz's critical analysis of use cases (Glinz 2000).)

The preceding analysis leaves two questions open:

- Is a more general form of requirement available, focusing on abstract properties rather than examples?
- Can we express use cases, user stories and tests in the same framework as those requirements?

For answers, we may turn to the object-oriented mode of specification.

3. Object-oriented fundamentals

To reach the level of abstraction that scenarios do not provide, we look into object-oriented techniques, which have been widely applied to programming and design.

3.1. OO principles and benefits

Some of the key object-oriented concepts are the following (Meyer 1988, 1997):

- *Type-based modular decomposition*: capture the properties of the key "things" manipulated by a system, better called **objects**, by abstracting them into types, or **classes**, and define the modular structure of a system as a set of interconnected classes.
- *Data abstraction*: describe each class not by implementation properties (what the corresponding objects "are") but by the applicable operations, or **features** (what the objects "have").
- *Contracts*: in such descriptions, include not only structural properties, such as the types of arguments handled by each operation, but also abstract semantic properties (preconditions, postconditions, class invariants).
- *Inheritance*: organize classes into taxonomies to take advantage of common traits. Techniques of polymorphism and dynamic binding, which follow from inheritance, enhance the architectural quality of OO systems.

The next subsections examine these concepts in more depth, with an eye on their application to requirements. The examples use the Eiffel notation, which includes built-in mechanisms of Design by Contract and was designed to cover requirements and design in addition to programming. Subsection 3.6 discusses the expected benefits.

3.2. Classes and their mutual relations

The central concept of object-oriented approaches to system modeling and structuring is the class, defined as a system unit specifying a type of object with the associated operations and their properties. A class has operations of three kinds, which we may illustrate through the example of a class describing the concept of book in a library system:

- Queries, providing information about the book: is_available, isbn, author.
- Commands, to update the corresponding objects: hold, checkout.
- Creators (or "constructors"), yielding objects of the type from other information, for example a book object defined by an author and title.

OO system descriptions are simple, relying on classes and only two relations between these classes: client and inheritance. For example, operations on a book (see the use case presented in section 2.1) involve, among other objects, a patron and a library branch. The class representing a book will be a **client** of the classes representing the other three concepts.

A class **inherits** from another if it represents a specialized or extended version of the other's concept. For example book and magazine both belong to the general category of library item, which can be represented by a class `LIBRARY_ITEM`. `BOOK` and `MAGAZINE` are "descendants" of `LIBRARY_ITEM` through the inheritance relation shown in Figure 1.

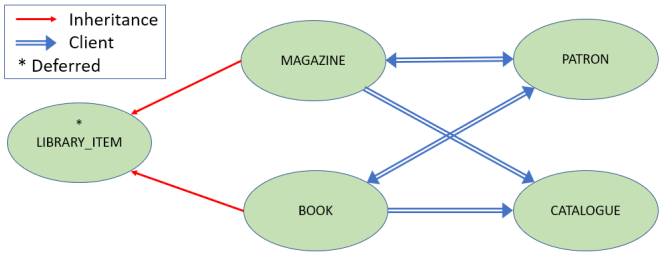

Figure 1 Examples of the two kinds of inter-class relations

3.3. Deferred (abstract) classes

In an OO program, most of the commands and queries of a class are "effective", meaning that the class specifies an algorithm for their implementation. A class whose features are all effective is itself called an **effective** class. In some cases, however, including classes describing requirements, it is useful to define features without providing an implementation. Features specified but not implemented are called **deferred**, and a non-effective class (one that has at least one deferred feature) is a deferred (abstract) class. In requirements for the library example, `LIBRARY_ITEM` will most likely be a deferred class (as marked by "*" in Figure 1) since it describes an abstract concept with several possible concrete realizations.

In requirements, declaring classes and features deferred makes it possible to specify that some functionality must exist while staying away from any consideration of implementation and hence from the risk of *overspecification*.

Even without an implementation, it will be possible in such cases to specify abstract properties of the desired behavior thanks to contracts, as discussed in section 3.4 below. It might seem that in requirements all classes and features should be deferred, but that would be too restrictive. As we will see in section 5, requirements classes can specify scenarios, such as use cases or tests; such classes and their features will, in general, be effective since they prescribe precise sequences of operations.

3.4. Contracts

Information hiding implies that modules of a system's description (in OO modeling, classes) can refer to others in terms of their abstract specification. To be useful, such specifications should not just be structural, giving the types of arguments and results of the operations in a class, but also semantic, describing the abstract properties of these operations and the class as a whole. Design by Contract techniques (Meyer 1992) provide this semantic specification for both operations (preconditions and postconditions) and classes (class invariants).

As an example, the class extract below expresses the requirement that *a public library allows patrons to place books on hold at its various library branches*:

```
class LIBRARY feature
  place_book_on_hold (b: BOOK; p: PATRON; lb:
    LIBRARY_BRANCH)
    require
        has_patron (p)
        has_branch (lb)
    do -- Future implementation
    ensure
        book_is_on_hold (b, p, lb)
    end
end
```

The **require** clause introduces a precondition, and the **ensure** clause a postcondition.

Contracts have many applications, including as documentation of the software and as a guide to both exception handling and the proper use of inheritance. When applied to code, they also provide a systematic approach to testing and debugging (in a development environment that provides the ability to evaluate contract elements at run time). The application of most direct relevance to requirements, as illustrated by the `LIBRARY` example sketch above, is to model the System or Environment by providing not only structural properties (object types and the applicable operations) but also through precise semantics (the properties of these types and operations).

Such semantic models lend themselves to automatic verification. Beyond run-time monitoring of contracts, which only applies to executable code, classes equipped with contracts can be verified using automatic tools such as JML (Leavens et al. 1999) and the AutoProof verifier for Eiffel (Tschannen et al. 2015).

3.5. Specification drivers

The object-oriented style of modeling relies on modular units (classes), each organized around one type of objects. Correspondingly, contract techniques apply within a class, making it possible for example to express the requirement that "*after a patron returns a book, it is considered available*" as a postcondition of an operation `return` in a class `BOOK`.

Some properties apply to several objects, possibly of different types. An example is "*available books can be placed on hold by only one patron at any given time*".

While the OO style would seem to break down in such cases, it actually handles it in a simple way through the introduction of "specification drivers" (Naumchev & Meyer 2016). The idea (generalizing techniques already present in the Visitor design pattern (Gamma et al. 1994)) is to express such cross-object properties through classes designed specifically for specification purposes.

In the last example, we may use the following specification-driver assertion, which describes a generic scenario of using the relevant features and specifies its effect through the postcondition:

```
holding_available_books (b: BOOK; p1, p2: PATRON; lb:
      LIBRARY_BRANCH; l: LIBRARY)
    require
        b.is_available; p1 ≠ p2
        l.has_patron (p1); l.has_patron (p2)
        l.has_branch (lb)
    do
        l.place_book_on_hold (b, p1, lb)
        l.place_book_on_hold (b, p2, lb)
    ensure
        l.book_is_on_hold (b, p1, lb)
        not l.book_is_on_hold (b, p2, lb)
    end
```

Specification drivers retain the OO specification style but make it more general by covering properties that may not be expressible within a single class of the original OO model.

3.6. OO benefits and their application to requirements

The case for applying OO principles and techniques summarized above rests on the expectation of a number of benefits, analyzed in detail in the literature (Meyer 1988, 1997). They include extendibility (ease of modifying software, by keeping the various elements of a system isolated from changes in others), reusability (ease of applying elements of a system to the development of a new system) and reliability (correctness of solutions, robustness in the presence of errors, and security). These goals are just as desirable for requirements as they are for design and programming. Remarkably, much of the rationale for using OO technology to achieve them does not depend on whether it covers implementation, design or (the case of interest for this discussion) requirements. For example:

- Extendibility (1). A core idea is that structuring the description of a system around the types of objects (classes) rather than around the functions leads to more stable architectures. A typical example found in the standard references is a payroll processing system. Initially, it is defined by a function: produce paychecks. But if you structure it based on that analysis (classical, non-OO "functional" decomposition) you will have to redo the architecture regularly as new functions come in: change salary scales, provide reporting mechanisms, allow interactive queries of salary information etc. In an OO approach the units of modularization are types of objects: `EMPLOYEE`, `SALARY_SCALE`, `PAY_RECORD`, `WORK_RECORD` etc. As new functions come (and go), these basic units remain stable. This flexibility accounts for a large part of the success of OO modeling. It is directly applicable to requirements. Note that use cases and other kinds of scenarios (see next section) break that structure by requiring the specifier to focus early — too early — on sequences of actions.
- Extendibility (2). Inheritance yields flexible architectures, based on taxonomy — the classification of object types into hierarchies, in line with scientific practices. It is directly applicable to requirements.
- Reliability (1). One of the OO reliability techniques is static typing, which catches many errors as type inconsistencies. It is directly applicable to requirements.
- Reliability (2). Contracts make it possible to express not just structural properties but semantic constraints. As illustrated in the examples of this paper, they are directly applicable to requirements.
- Reusability. Reuse is desirable for requirements as much as for other software development steps and artifacts, although in practice requirements reusability is still in its infancy. Some of the OO techniques favoring reuse, such as (again) the division into classes rather than functions, as well as inheritance, are directly applicable to requirements.

The next section describes in more detail the application to requirements of the OO techniques of the present section.

4. Object-oriented requirements

The scope of object-oriented concepts as described in the previous section is broad enough to encompass all tasks of software construction, from requirements to design, implementation of course, and even verification. Of direct interest in this article is the application to requirements.

4.1. OO requirements basics

Discussing a system as a set of object types (classes) characterized by the applicable operations yields a number of potential benefits, which have been widely recognized in the programming world but can apply to requirements as well. They include the following:

- Stability through the evolution of the software. Changes affect individual modules, not the architecture.
- Information hiding. With classes, we may declare, as part of the specification of a class, that some of the properties are for internal use only, within the class, and not accessible outside; limiting the effect of changes.
- Reuse. If we are trying to apply the results of one project to another, reusing individual operations will generally not work. Operations such as "place a hold on a book", "check out a book", "cancel a checkout" and others are closely connected. The notion of a book (as a class that includes all these operations) is a more realistic unit for reuse.
- Classification. Inheritance makes it possible to describe new classes as extensions or specializations of existing ones, without repeating common properties. Along with information hiding, inheritance is a key tool in harnessing system complexity.
- Modeling power ("direct mapping principle", as noted in section 1). OO concepts can yield system descriptions that are clear and intuitive, since the notions of object, class and inheritance are easy to grasp. Some classes have immediately tangible counterparts in the system environment. They are called "Environment classes", Concrete and Abstract, in the classification presented in a subsequent section (4.2).
- Abstraction. Deferred (abstract) classes and features make it possible to specify the presence of certain types of objects and of certain operations on them, without giving implementation details (but with abstract behavioral properties through contracts).

4.2. Modeling the system and its environment

Requirements in their general sense involve four aspects (called the "Four PEGS" in (Meyer 2022)): Project (a human effort to produce a system), Environment (the material or virtual reality in which the system will operate), Goals (objectives set by the organization), and System (the set of functional elements that will be provided). OO modeling is applicable to all four aspects.

The application to the System part is the traditional focus of OO ideas. When the project reaches the design and implementation stages, System classes will include both concrete classes describing implementation elements and more abstract design

classes, corresponding for example to design patterns such as Visitor or Observer.

At the requirements stage, it is not too early to use classes to identify the major components of the system and describe them through classes. Many of these classes will be deferred (abstract), but they may already include operational elements, including specification drivers and (as explained below) scenarios.

Of particular relevance are Environment classes; OO modeling is indeed appropriate to describe components of the environment. A typical example is the description of interfaces to and from other systems, through interface classes describing such elements as sensors, actuators, buttons and control panels. The corresponding objects are not part of the software under development, but they are directly monitored or controlled by the software.

Contract elements are particularly useful here, to describe delicate properties of the environment. Such properties include:

- Constraints: conditions imposed by the environment, such as a physical limit (in a cyber-physical system) or a legal obligation (for a business system).
- Effects: changes to the environment produced by the system (such as the triggering of an actuator, or a change in payroll processes).
- Assumptions (posited properties of the environment, making the system's construction easier).
- Invariants (properties that are both constraints and effects, as they can be assumed but must be preserved).

4.3. An example OO specification

The class text below is an example of how we can use an object-oriented formalism to describe requirements specifications, independently of any design or implementation consideration. It describes the notion of book in a library system. The following section will discuss some of its most important features, distinguishing it from other types of specifications such as use cases.

```
class BOOK feature
        -- is_available initialized to True
        -- is_on_hold, is_checked_out initialized to
        False
    is_available, is_on_hold, is_checked_out: BOOLEAN
    place_hold (patron: PATRON)
      -- Place a hold on a book. Set is_on_hold
        require
            is_available
        deferred
        ensure
            is_on_hold
            not is_available
        end
    checkout (patron: PATRON)
        -- Check out the book
        require
            is_on_hold
        deferred
        ensure
            is_checked_out
        end
    return
        -- Return the book to the library
        require
            is_checked_out
        deferred
        ensure
            is_available
        end
invariant
    is_on_hold implies not is_available
    is_checked_out implies not is_available
    is_checked_out implies not is_on_hold
    is_available implies not is_checked_out
end
```

5. Object-oriented requirements as the unifying framework

The object-oriented approach is a structuring discipline, which models systems — at all levels: requirements, design, implementation — as collections of classes equipped with contracts and related to each other through client and inheritance links. This framework is general enough to encompass all aspects of requirements and provides room for the various non-OO techniques (scenarios and tests) reviewed earlier. The following sections show how to use an OO model as an all-encompassing host for various applications.

5.1. Logical rather than sequential constraints

A distinctive feature of the above BOOK class sketch is its reliance on logical constraints (through contracts) in lieu of a strict specification (or overspecification) of sequencing constraints.

OO techniques avoid premature *time-ordering decisions*. While it is possible for an OO specification to express a time-ordered scenario such as a use case, object technology also supports a more general and abstract specification style, based on contracts.

The comparison of class BOOK with the use case specification of books in section 2.1 provides a good illustration. The use case version specifies the order in which operations will get executed; for example, in the "main success scenario":

- The system changes the book status to on_hold
- The patron checks out the book
- The patron returns the book

Enforcing such an ordering specification at the level of requirements is often a premature decision. In reality, the order of the steps is not cast in stone. Using a preset ordering is convenient to describe *desirable* scenarios, or more generally the *expected* ones. But what happens in life is not always what we hope for, or expect. What if the customer returns a damaged book? Should the book not remain unavailable until it is repaired?

To specify scenarios that depart from the standard ones, we saw that it is possible to use **extensions**. But this solution does not scale. Writing ever more use case extensions to cover all such situations leads to an explosion of special cases which soon becomes intractable. In practice, it is possible to write use cases to cover the most common scenarios, but they are only a small subset of the possible ones, in the same way that, in programming, tests can only cover a minute subset of possible inputs.

To get out of this predicament, we note that while constraints between the operations do exist, it is often more general and effective, instead of *timing* constraints, to rely on *logical* constraints.

The preceding example scenario provides a good illustration. As a specification, it is trying to express a few useful things; for example, the patron should place a hold on a book before checking it out. But it states them in the form of a strict sequence of operations which does not cover the wide range of possible scenarios.

In the same way, a user story such as "*As a patron, I want to check out a book so that I can read it at home*" describes the interaction between a human actor and the system. Describing a few such scenarios is useful as part of requirements elicitation, but to express the resulting requirements it is more effective to express the logical constraints.

Class BOOK (from section 4.3) specifies these logical constraints in the form of contracts. Notice the interplay between the preconditions and postconditions and the various boolean-valued queries they involve (`is_available`, `is_on_hold`, `is_checked_out`). It is possible to 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 make it possible to specify a much broader range of allowable orderings. Logical constraints are more general than sequential orderings.

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

```
        -- The following is the "Main scenario":
place_hold (patron: PATRON)
checkout (patron: PATRON)
return (patron: PATRON)
```

the postcondition of each step implies the precondition of the next one (the first has no precondition). Prescribing this order strictly is, however, overspecifying. For example, it may be possible to perform additional operations between `place_hold` and `check_out`.

The contract-based specification does cover the fundamental sequencing constraints; for example, the pre- and post-condition combinations imply that investigation must come before evaluation and that resolution must be preceded by either negotiation or imposition. But they avoid the non-essential constraints which were only an artifact of the sequential style of specification in the use case, not a real feature of the problem (Petre 2013).

5.2. Integrating scenarios into an OO model

The preceding discussion shows how a specification through classes and their contracts beats a scenario-style specification in precision and generality. Use cases, user stories and other scenarios do retain their attractive features mentioned in section 2, in particular their intuitive appeal to stakeholders, who can relate them easily to business processes, and their value as ways to validate the completeness of specifications. There is no need, however, to sacrifice scenarios – in particular use cases, retained as an example for this discussion – in an OO approach. A use case is simply a certain pattern of exercising the features (operations) of one or more classes; it can easily be expressed as a *routine* (method) of an appropriate class.

Here for example is the expression of the use case from section 2.1 as a routine that calls features of the class BOOK:

```
class BOOK feature
    borrow_and_return_book (p: PATRON, lb:
     LIBRARY_BRANCH)
        require
            book_is_available: is_available
        do
            place_hold (p)
            checkout (p)
            return (p)
        end
```

This use case is simply a routine, calling features of the appropriate class.

Where should such use case routines appear? Two possibilities are available:

- In the case of a use case characterizing a single data abstraction and applying to a single object, such as one book, it can be expressed as a routine of the corresponding class, in this case BOOK. Then it simply describes a specific behavior of the instances of the class, expressed in terms of the more fundamental operations of the class.
- A more general solution, and the one that fits the case of a use case involving several objects of possibly different types (hence, routines in different classes), is to group use cases into a separate class.

In both cases, a use case is an addition to one or more data abstractions from the rest of the requirements, intended to illustrate specific ways of using its features. The notion of "specification driver" (section 3.5) covers such specification elements exercising the features of one or more model classes. It was originally introduced for verification purposes (proofs and tests); we may view use-case class as an application of the concept to requirements specification.

In accordance with OO principles, a use-case class should include not just one use case but a group of logically related use cases, exercising features of other requirements classes. (The idea of describing a single use case as a class is not new; see for example (Cook et al. 2017). The use case classes described here are at a higher level of abstraction, covering a whole set of related behaviors, all pertaining to one or more data abstractions covered by other requirements classes.)

Here is an example of such a use-case class (a collection of specification drivers) exercising features of classes BOOK and LIBRARY:

```
class LIBRARY_BOOK_USAGE feature
    borrow_and_return_book
       do ... as given above ... end
    decommission_book
        do ... Specification of decommission use case
     ... end
    renew_book
        do ... end
    -- Other use cases
end
```

To avoid any confusion, note that the idea is not to describe *one* use case as a class or the corresponding object; that possibility is well-known, and not surprising since any well-defined abstraction can be modeled as an object. A use class such as LIBRARY_BOOK_USAGE gathers a *set* of important behaviors connected with an abstraction (or a group of abstractions) from the OO model of the environment and system. Such a set of related behaviors is in its own right a relevant abstraction in the OO model.

5.3. Relation of OO requirements to test cases

Test cases fit in the general OO framework just as use cases do. In modern "xUnit" approaches to software testing (section 2.4), the basic scheme is already there since, as noted in the earlier discussion, these frameworks require writing test cases as routines. The earlier example, HoldingAvailableBooksTest, takes the form of the class HOLDING_AVAILABLE_BOOKS_TEST depicted below.

```
class HOLDING_AVAILABLE_BOOKS_TEST feature
  test_holding
    local
        b: BOOK
        p1, p2: PATRON
        lb: LIBRARY_BRANCH
        l: LIBRARY
    do
      create b.make("Crime and Punishment", "Fyodor
     Dostoyevsky", "978-1703766172")
      create p1.make("Ted") create p2.make("Fred")
      create lb.make("Squirrel Hill")
      create l.make("Carnegie Library of Pittsburgh")
      holding_available_books (b, p1, p2, lb, l)
    end
    ...
end
```

As suggested by the final '...', the given routine test_holding does not need to be the only one in its class. Having one testing routine per testing class is, in fact, the typical xUnit style used in practice, but this practice misses the advantages of OO modularization. (It is a general rule of OO methodology that a class with just one routine is a "design smell", a sign of probably bad OO design.) In the context defined by the present discussion, the testing harness for a system should consist of a set of testing classes, each exercising some features of a system or environment class (or of a closely related group of such classes).

Like use-case classes, such test classes are specification-drivers classes. HOLDING_AVAILABLE_BOOKS_TEST, containing routines related to book borrowing, is a typical example.

5.4. Premature design?

An object-oriented requirements specification, as presented, uses an object-oriented notation, borrowed from an OO programming language. This notational resemblance may give the wrong impression of a specification involving premature choices of design and implementation. Such criticism is not justified; in fact, a proper OO requirements specification is more abstract, and less prone to overspecification, than a use-case or other scenario-based form of requirements.

The classes used in an OO specification are purely descriptive; they specify concepts (in particular, system and environment concepts) in an abstract way, using the OO style of specifying object types through the applicable operations and their abstract properties (contracts). In contrast, use cases are of a sequential, operational nature, presenting a risk of premature design, particularly as they make it tempting to write programs following the same ordering patterns, justified or not.

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. One of its consequences is reversibility: having everything expressed in the same notation makes it easier to update the requirements at any stage in the project, even deep into design, implementation or verification.

5.5. Applying the framework

While the systematic description of a comprehensive approach to requirements specification falls beyond the scope of this article, we may build on the BOOK example to obtain an outline of the general process of object-oriented requirements specification:

1. Express the fundamental abstractions in the form of requirements classes.
2. Express the fundamental constraints in the form of logical properties: invariants for these classes as well as preconditions and postconditions for their features (operations).
3. Express typical usage scenarios through use cases or user stories. (Unlike the previous two, this task does not make any attempt at exhaustiveness, since examples can only cover a minute fragment of all possibilities; instead, it concentrates on the scenarios of most interest to stakeholders, and those most likely to cause potential issues or bugs.)
4. As a consistency check, ascertain that the scenarios (item 3) preserve the logical properties (item 2). Update the logical properties if needed.

More generally, the combination of an object-oriented approach to structure the requirements (1), equipped with invariants (2) as well as other forms of contracts (preconditions, postconditions), with use cases to illustrate the requirements through examples of direct interest to stakeholders (3) and shown to preserve the invariants (4) provides a promising method for obtaining correct and practically useful requirements.

6. A case study – the Roborace software

This section illustrates the application of the suggested approach to a real-world case study: the Roborace (Roborace 2022). The code excerpts used in this section are available at (Naumcheva 2022). Although the complete requirements specification is

beyond the scope of the article, we illustrate the implementation of the key concepts of the framework.

6.1. Roborace: informal description

Figure 2 Devbot 2.0 – all-electric vehicle currently used in the Roborace competition (source: (Roborace 2022))

Roborace (Roborace 2022) is a global championship between autonomous cars. The hardware (the race cars) is the same for all participating teams; each gets access to an autonomous race car called Devbot 2.0, and develops software to drive it in races, in a completely autonomous way. Each season sees changes in the goals and rules, and the introduction of new conditions.

The race takes place on a circuit. The cars start at a designated spot on a starting grid and have to accomplish a given number of laps faster than the competing teams. The competitors race independently and their racing time is compared after all teams have finished the race.

No physical objects other than the competing race cars are present on a racetrack, but there can be virtual objects, of three kinds: static obstacles, loots and ghost cars. Cars get bonus time for collecting loots and penalty time for hitting obstacles or ghost cars. The total time is defined as the race time minus bonus time plus penalty time.

6.2. Roborace: use cases

A specification of the Roborace system would include many use cases, such as:

- Race without obstacles
- Avoid obstacles or stop
- Update speed limit
- Race with virtual obstacles
- Race with virtual race cars
- Move to pit
- Perform an emergency stop
- Perform a safe stop.

Let us pick the first of these, "*Race without obstacles*", for further analysis. This subsection shows a typical use-case model for it; in the next subsection we will see how to integrate it into a more general OO framework.

A specification of the use case in the style introduced in section 2.1 is presented in table 4.

6.3. Roborace: some requirements classes

We now temporarily set use cases aside and consider what an object-oriented requirements model for Roborace would look like. Its classes would cover both:

- Its environment, with such classes as RACE_TRACK, MAP and OBSTACLE.
- Components of the system, with such classes as RACE_CAR, PLANNING_MODULE, CONTROL_MODULE and PERCEPTION_MODULE.

Here is a sketch of two such classes, both belonging to the system.

```
class RACE_CAR feature
    control_module: CONTROL_MODULE
    perception_module: PERCEPTION_MODULE
    planning_module: PLANNING_MODULE
    localization_and_mapping_module:
     LOCALIZATION_AND_MAPPING_MODULE
end
```

```
class RACE_TRACK feature
    raceline: RACELINE
      -- Optimal raceline for the track
    map: MAP
      -- Coordinates of the bounding lines
end
```

During the development process, elements of the system's functionality are assigned as features of respective modules' classes and are enriched with contracts.

Below is an implementation of the requirement "At every position on a raceline the speed in the velocity profile shall not exceed the maximum racecar's speed":

```
class PLANNING_MODULE  feature
    car: RACE_CAR
    calculate_raceline (t: RACE_TRACK)
      --Calculate optimal raceline for a given
     racetrack
        do
        ensure
            across t.raceline.velocity_profile as rl
     all rl.item < car.max_speed end
        end
end
```

6.4. Roborace: integrating the use cases into the object-oriented model

We now see how to express the use cases (section 6.2) as part of the OO requirements (section 6.3).

The "race without obstacles" use case, previously expressed in tabular format (Table 4) becomes simply a routine race_no_obstacles in the requirements class ROBORACE_USE_CASES sketched above. It relies on conditional expressions to consider the use case alternative flows.

```
race_no_obstacles
    Note
        Callers: car_operator
    require
        not car.is_moving
        car.global_plan_is_calculated
        car.green_flag_is_up
        car.is_on_starting_grid
```

Table 4 A detailed description of the *"race without obstacles"* use case

Name	Race_no_obstacles
Scope	System
Level	Business summary
Primary actor	Race car Operator
Secondary actor	Roborace Operator
Context of use	Race car has to obey an instruction
Preconditions	* Race car is on the racetrack grid * Race car is not moving * The global plan (trajectory and velocity profile) minimizing the race time is calculated * The green flag is shown by the Roborace
Trigger	The system receives a request from the race car operator to start the race
Main success scenario	* The system calculates the local plan (path and velocity profile) during the race trying to follow the global plan as close as possible * The race car follows the local plan * After finishing the required number of laps the race car performs a safe stop
Success guarantee	The race car has completed the required number of laps and stopped.
Extensions	A. The red flag received during the race * The race car recalculates a global plan to perform an emergency stop * The race car performs an emergency stop B. The yellow flag is received during the race * The system sets the speed limit according to the received value * The race car finishes the race following the global trajectory and not exceeding the new speed limit C. The difference between the calculated (desired) location and real (according to the sensors) location is more than a given threshold * The race car recalculates a global plan to perform an emergency stop * The race car performs an emergency stop
Stakeholders and interests	Race car Operator (requests the car to start the race) Roborace Manager (sets the race goals and policies) Roborace Operator (shows the green, yellow, red flags)

```
    local local_plan: RACELINE
    do
        from --Sequence of system actions in use case main flow
          until car.race_is_finished or
                car.red_flag_is_up or
                car.location_error_is_detected
          loop
              if car.yellow_flag_is_up then
                update_speed
              end
          local_plan := car.planning_module.calculate_local_plan
          car.control_module.move (local_plan.speed,
                                   local_plan.orientation)
          end
          if car.red_flag_is_up or
             car.location_error_is_detected
          then emergency_stop
          else safe_stop end
    ensure
          not car.is_moving
          car.is_in_normal_mode implies car.race_is_finished
    end
```

The `race_no_obstacles` routine relies on implementing the routines `update_speed`, `safe_stop`, and `emergency_stop`: the respective features are called inside the use case. These features are implementation of the respective use cases, and such dependency corresponds to «include» and «extend» relationships between use cases.

The ROBORACE_USE_CASES class is thus a collection of routines corresponding to the system's use cases.

```
class ROBORACE_USE_CASES feature
    car: RACE_CAR

    safe_stop
        require
          ‘ car.is_in_normal_mode
        do
            car.control_module.safe_stop
        ensure
            not car_is_moving
        end

    emergency_stop
        require
            car.red_flag_is_up or car.location_error_is_detected
        do
            car.control_module.emergency_stop
        ensure
            not car.is_in_normal_mode
            not car.is_moving
        end


    update_speed
        require
            car.yellow_flag_is_up
        do
            car.update_max_speed (car.safe_speed)
        ensure
            car.current_max_speed = car.safe_speed
        end


    race_no_obstacles
        do
          --implementation is listed above
        end

    avoid_obstacle_or_stop
        do
        end

    race_with_virtual_obstacles
        do
        end

    race_with_virtual_race_cars
        do
        end

    move_to_pit
        do
        end

end
```

6.5. Roborace: relation between use cases and test cases

Use case stories define test cases for use cases (Jacobson et al. 2016). ROBORACE_USE_CASE_STORIES class inherits from ROBORACE_USE_CASES class. It includes a collection of routines corresponding to use case stories.

When a use case takes the form of a routine with contracts, extracting use case stories from such a routine becomes a semi-automated task. For example, the `emergency_stop` use case accepts two options in its precondition – (1) when the red flag is shown or (2) when a location error is detected. These options map to the following use case stories written in the unified approach / with OO requirements:

```
emergency_stop_red_flag_story
    require car.red_flag_is_up
    do emergency_stop end

emergency_stop_location_error_story
    require car.location_error_is_detected
    do emergency_stop end
```

These routines represent the two different paths through the `emergency_stop` use case, characterized by their preconditions. The connection with the parent use case is visible because the stories call the routine encoding the use case. The two routines must be exercised at least once with test input that meets their preconditions.

A similar analysis makes it possible to extract 5 use case stories from the `race_no_obstacles` use case:

- three for each possible loop exit condition.
- one corresponding to the true antecedent of the implication in the second postcondition assertion.
- one corresponding to the true consequent and false antecedent of the said implication.

The full collection of the extracted use case stories may be found in a publicly available repository (Naumcheva 2022).

6.6. Roborace: lessons from the example

The "main scenario" of the *"Race without obstacles"* use case (section 6.2) provides a good illustration of the difference between contract-based and scenario-based specification (section 5.1). As a specification, this scenario expresses, among other properties, that the system calculates a local plan and then follows it. It states this property in the form of a strict sequence of operations which, however, only covers some of the many possible scenarios.

It does list extensions, but only three of them, and does not reflect the many ways in which they can overlap. For example:

- It can happen that the green flag is shown some time after the yellow flag; but the extensions do not even list the green flag.
- In the same way, the red flag can be shown after a yellow flag.

An attempt to add extensions to cover all possibilities would have no end, as so many events may occur as to create a combinatorial explosion of possible sequencings.

One way out of this dead end would be to use temporal logic (Pnueli 1977), which provides a finite way to describe a possibly infinite but constrained set of sequences of events or operations. The Design-by-Contract-based technique discussed in the present work relies on a different idea: use logical rather than sequential constraints. For each operation, we specify both:

- The conditions it requires (precondition).
- The conditions it ensures (postcondition).

Sequential constraints become just a special case: we can express that `A` must come before `B` simply by defining a condition `C` as part of both the postcondition of `A` and the precondition of `B`. But the logic-based specification scheme covers many more possibilities than just this special case.

In the example just mentioned, we state (in class `ROBORACE`) the constraint on raising the yellow and red flags:

```
class ROBORACE feature
    raise_yellow_flag
        require
            green_flag.is_up
        do
        ensure
            yellow_flag.is_up
            not green_flag.is_up
            not red_flag.is_up
        end
    raise_red_flag
        require
            green_flag.is_up or yellow_flag.is_up
        do
        ensure
            red_flag.is_up
            not green_flag.is_up
            not yellow_flag.is_up
        end
end
```

Preconditions and postconditions apply to individual operations or events and cannot capture general environment constraints, such as the requirement that if the yellow flag is up cars should limit their speed to a dedicated "safe speed". For such requirements properties, we need contract elements of the third major kind, class invariants, as in the following extract from the specification of cars:

```
class RACE_CAR feature
    green_flag_is_up: BOOLEAN
    yellow_flag_is_up: BOOLEAN
    red_flag_is_up: BOOLEAN
    safe_stop_activated: BOOLEAN
    max_speed: REAL
    current_max_speed: REAL
        -- Current speed limit
    safe_speed: REAL
        -- Safe speed limit
invariant
    yellow_flag_is_up implies current_max_speed =
     safe_speed
    green_flag_is_up implies current_max_speed =
     max_speed
    red_flag_is_up implies safe_stop_activated
end
```

This example is typical of how invariants capture fundamental consistency constraints. Almost every problem domain has such constraints, defining what is and is not possible. Any good requirements should include them. They have no place, however, in a specification based solely on scenarios, which describe only examples of use, not the underlying invariant properties.

7. Related Work

7.1. UML and SysML

Use cases are an important modeling tool in UML (Cook et al. 2017). (Larman 2012) illustrates use-case-driven requirements specification in UML and (Overgaard & Palmkvist 2004) describes use-case patterns and blueprints for use-case modeling in UML.

UML makes it possible to treat use cases as objects, subject to specialization and decomposition. As noted in section 5.2, the use-case classes described in this paper cover a different concept: a group of related behaviors, pertaining to one or more data abstractions. UML use cases can have pre- and postconditions; in OO modeling as described in the present paper, pre- and postconditions (routine contracts) apply to individual operations. If a use case consists of a sequence of operations op_1, op_2, ... op_n, with each op_i characterized by pre_i and $post_i$, the pre- and postconditions of the use case are just pre_1 and $post_n$. It is possible in UML to associate contracts with individual operations through natural language or the OCL (Object Constraint Language) notation.

SysML (OMG 2019), an extended profile for UML, treats requirements as first class entities, establishing direct links between requirements and other software artifacts (such as tests). (Weilkiens 2011) illustrates requirements specification process with SysML and (Apvrille et al. 2020; Xie et al. 2022; Waseem & Sadiq 2018) provide applications of SysML to all phases of software development. SysML does not provide semantics for requirements although it is possible to associate contracts with individual operations through natural language or the OCL notation. SysML and UML are standardized notations, rather than methodologies.

7.2. Use-case modeling

The Restricted Use Case Modeling approach (Yue et al. 2016, 2013) relies on a use case template and a set of restriction rules to reduce the ambiguity of use case specification and facilitate transition to analysis models, such as UML class diagram and sequence diagram. The aToucan tool automates generation of UML class, sequence and activity diagrams (Yue et al. 2015). The approach does not advocate extracting abstract properties from use cases and domain knowledge, such as time-ordering constraints and environment constraints.

Like the use case classes presented in section 5.2, a Use Case Map (UCM) (Buhr & Casselman 1995; Amyot et al. 2005) depicts several scenarios simultaneously. UCMs represent use cases as causal sequences of responsibilities, possibly over a set of abstract components. In UCMs pre- and postconditions of use cases as well as conditions at selection points can be modeled with formal specification techniques such as ASM or LOTOS. UCMs specify properties of operations in relation to scenario sequences, rather than abstract properties of objects and operations.

Logical constraints, discussed in section 5.1 can also be formulated with state-based notations, such as Alloy (Jackson 2012), Event-B (Abrial 2010; Murali et al. 2016), Abstract State Machines (Börger 2010) and Statecharts (Whittle & Schumann 2000; Gomes & Costa 2003). Some approaches (Murali et al. 2016; Gomes & Costa 2003) apply formal modeling to specify use cases, but the formal specification is for an entire use case, not for its individual operations.

7.3. Use cases as requirements

There is disagreement among researchers about whether use cases are requirements. In the ICONIC process (Rosenberg & Stephens 2007) use cases are requirements and constitute the main input for software design. Larman claims in (Larman 2012) that use cases are only part of requirements, constituting functional requirements, yet not all requirements. Other requirements artifacts include supplementary specification, glossary, vision and business rules. In our perspective, although use cases are an important source of requirements, they have to be thoroughly analyzed together with environment constraints before proceeding to software design.

7.4. Contract-based approaches

To express properties of operations and their interactions, object-oriented requirements as discussed in this article rely on contracts rather than sequential ordering. Contracts also play a role in other requirements work, including approaches based on UML thanks to the Object Constraint Language, OCL. Larman's book on UML and patterns (Larman 2012) specifies "system operation contracts" in natural language or OCL to express how system operations change the state of domain model objects. Soltana et al. (Soltana et al. 2014, 2016) apply OCL contracts to express operational legal requirements as policy models.

The present study does not include empirical evidence, in particular about the value of using contracts for requirements. Such evidence does appear in an empirical study by Briand et al. (L. Briand et al. 2005; L. C. Briand et al. 2011), which identified a positive effect from producing OCL contracts in requirements analysis. The effect is moderate, but the authors point out that participants lacked training in UML and OCL.

Arnold et al. (Arnold et al. 2010) formalize use cases as grammars of responsibilities. Abstract Constraint Language contracts (pre- and postconditions and invariants) capture constraints that scenarios' or responsibilities' execution poses on the system's state. In this approach a dedicated binding tool maps elements of a requirements model to elements of candidate implementation. The approach preserves the procedural nature of specification, since it organizes specification around scenarios and operations, rather than types of objects of the application's domain.

The SIRCOD approach (Galinier 2021) provides a pipeline for converting natural language requirements to programming language contracts. The approach relies on the domain-specific language RSML for automating conversion from natural language to programming language. In the SOOR approach (Naumchev 2019; Naumchev & Meyer 2016), requirements are documented as software classes which makes them verifiable and reusable. Routines of those classes, called specification drivers, take objects to be specified as arguments and express the effect of operations on those objects with pre- and postconditions. The SIRCOD and SOOR approaches focus on translating existing requirements specifications to contracts expressed in a programming language, rather than extracting abstract requirements from scenarios.

8. Discussion and conclusion

8.1. Limitations

The work presented here is a conceptual contribution to the area of requirements methodology, and has not undergone any systematic empirical validation. The case study of section 6 serves as a proof of concept on a significant ongoing project (but only one).

The absence of large-scale empirical studies makes it impossible to state any guarantees that the approach will increase productivity or decrease defects. The case for it is based instead on arguments of a logical nature, in particular the observation that object-oriented technology with logical constraints is more general than scenario-based techniques and encompasses them as a special case. The next sections summarize these arguments.

The role of this article – in the tradition of work arguing for specific approaches in software methodology – is to provide the conceptual basis for studying OO versus scenario-based techniques. We do plan to conduct empirical studies to assess the actual consequences on actual projects. Studies conducted by others would be even more welcome.

8.2. Why OO requirements?

The idea of applying OO techniques to programming (their original focus since the appearance of the first object-oriented languages) is not controversial. Neither is their application to the next task up the abstraction level, software design. Moving up again one notch, to requirements, is not a new idea (Coleman

et al. 1994; Meyer 1988, 1997; Larman 2012); but OO requirements, unlike OO programming and OO design, are not widely practiced, for a large part because use cases and user stories have occupied the territory.

As the discussion in this article argues, these techniques (if used as the principal form of requirements) are a step backward since scenarios, as a requirements technique, lack abstractness, generality and precision. One of their principal limitations is that they prescribe specific orderings of operations, which is generally too constraining, as there are simply too many such orderings to understand and state. Instead of specifying that operations must appear in specified orders, a more flexible approach specifies that:

- Prior to executing, each operation may require some properties.
- After executing, each operation will ensure some properties.
- Specifying these before-and-after properties is an insightful way of specifying the operations, and determines, as a byproduct, which scenarios are legal and which are not.

A role remains for scenarios: to check that the more abstract logical specification does cover the specific cases (the most frequent usage patterns) that stakeholders have volunteered, as well as critical extreme cases that could lead to incorrect behavior. That role remains important, as long as we remember that:

- The specification is not given by the scenarios.
- It is given instead by the logical constraints (pre- and post-conditions, invariants) on operations (grouped, in an OO approach, into classes).
- The scenarios, while not providing the specification, provide *examples* (often, important ones) of the specification, useful to validate that specification.
- It is part of the tasks of the requirements process to check that the scenarios respect the contracts (each operation works properly if its precondition is satisfied, ensures its postcondition, and preserves the invariant).

Applying OO principles to requirements helps to produce more abstract and precise requirements due to the following mechanisms:

- The approach motivates asking precise questions of the non-technical stakeholders to clarify delicate points.
- Once a logical specification is produced, requirements engineer checks whether it corresponds to the stakeholders' informal view, thus avoiding incorrect implementation due to misunderstandings.
- Since time-ordering constraints are more abstract than behavioral sequences, formulating them with contracts covers a wider range of possible scenarios.

8.3. Seamlessness

Object-Oriented Analysis and Design (OOAD) approach relies on use cases along with object models as the techniques for requirements analysis (Pastor et al. 1997; Rumbaugh et al. 1991; Jacobson et al. 1992; Anda & Amyot 2020; von Olberg & Strey 2022). Unlike the Unified approach, in the OOAD approaches the process of software development is not seamless and involves several notations (natural language, UML, possibly formal languages). (Yue et al. 2015; Anda & Amyot 2020; Soavi et al. 2021; von Olberg & Strey 2022) propose methods for conversion between notations. However, the process of conversion is prone to errors and its outcome is never 100% correct As a result, the source code cannot be statically verified against the requirements.

As stakeholder requirements are formulated in natural language, a transition to programming language is inevitable at some point in the development process since software is written in a programming language. Seamless development makes the transitions as smooth as possible by encouraging the use of a single notation and a common set of concepts, embodied in the object-oriented paradigm. Transition to a programming language as the notation during requirements analysis makes it possible to resolve ambiguities in requirements early in the software development lifecycle.

8.4. Future work

One direction of future work is to apply the approach of unifying specifications, scenarios and tests to several case studies to provide more evidence for method validation. Another direction is to enrich the approach with tools that will improve its usability. Finally, we aim to perform a user study to evaluate the approach's usability and effectiveness.

9. Conclusion

Two of the central problems of software engineering, as relevant to requirements as for design and implementation, are size and change. Software systems can be large and complex; they must adapt to modification. Any approach to software construction must be judged by its ability to help address these issues.

On the design and implementation side, object-oriented techniques meet this criterion: thanks to techniques of class-based modularity, information hiding, genericity and inheritance, they have shown that they can support the development of large, evolving systems.

This article has presented the application of object-oriented ideas to requirements, where we may expect that they yield the same benefits. The key result is that we do not need to treat object-oriented requirements as a competitor to other popular requirements techniques such as use cases, use case slices and user stories. Object-oriented decomposition is at a higher level of generality than such procedural techniques and encompasses them. More specifically:

- The core idea of object-oriented decomposition is to use classes as the basic modular unit.
- The most intuitively appealing view of a class is that it describes a set of objects. For requirements, this concept of object is already rich in possibilities, since it makes it possible to describe, in a simple and natural way, concrete and abstract objects from a system's environment, such

as a racetrack or a book catalog (in the examples of this paper).
- Beyond this initial view of objects, however, the full notion of object is more general: the most important characteristic of a class is that it gathers a group of operations (commands and queries) applicable to a given abstraction. Beyond environment objects, such an abstraction can describe a system concept, such as elements of a design pattern.
- An object can also, in this spirit, describe a group of scenarios (use cases, user stories) relative to a set of abstractions, or a group of tests exercising these abstractions.

Applying these ideas results in a scheme that encompasses all the major requirements techniques in a general framework, with the advantages of conceptual consistency (as everything proceeds from a single overall idea, object technology) and unification of notations, and the potential of transferring the OO benefits of scalability and extendibility to the crucial discipline of requirements engineering.

Acknowledgments

We thank the JOT referees for many insightful comments on an earlier version of the article.

References

Abrial, J.-R. (2010). *Modeling in Event-B: system and software engineering*. Cambridge University Press.

Amyot, D., Roy, J.-F., & Weiss, M. (2005). Ucm-driven testing of web applications. In *International sdl forum* (pp. 247–264).

Anda, A., & Amyot, D. (2020). An optimization modeling method for adaptive systems based on goal and feature models. In *2020 ieee tenth international model-driven requirements engineering (modre)* (p. 11-20). doi: 10.1109/MoDRE51215.2020.00008

Apvrille, L., de Saqui-Sannes, P., & Vingerhoeds, R. (2020). An educational case study of using SysML and TTool for unmanned aerial vehicles design. *IEEE Journal on Miniaturization for Air and Space Systems*, *1*(2), 117-129. doi: 10.1109/JMASS.2020.3013325

Arnold, D., Corriveau, J.-P., & Shi, W. (2010). Modeling and validating requirements using executable cotnracts and scenarios. In *2010 eighth acis international conference on software engineering research, management and applications* (pp. 311–320).

Beck, K. (2003). *Test-driven development: by example*. Addison-Wesley Professional.

Börger, E. (2010). The abstract state machines method for high-level system design and analysis. In *Formal methods: State of the art and new directions* (pp. 79–116). Springer.

Briand, L., Labiche, Y., Di Penta, M., & Yan-Bondoc, H. (2005). An experimental investigation of formality in uml-based development. *IEEE Transactions on Software Engineering*, *31*(10), 833-849. doi: 10.1109/TSE.2005.105

Briand, L. C., Labiche, Y., & Madrazo-Rivera, R. (2011). An experimental evaluation of the impact of system sequence diagrams and system operation contracts on the quality of the domain model. *2011 International Symposium on Empirical Software Engineering and Measurement*, 157-166.

Buhr, R. J., & Casselman, R. S. (1995). *Use case maps for object-oriented systems*. Prentice-Hall, Inc.

Cockburn, A. (2000). *Writing effective use cases*. Addison-Wesley Longman Publishing Co., Inc.

Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., & Jeremaes, P. (1994). *Object-oriented development: The fusion method*. Prentice-Hall, Inc.

Cook, S., Bock, C., Rivett, P., Rutt, T., Seidewitz, E., Selic, B., & Tolbert, D. (2017, December). *Unified modeling language (UML) version 2.5.1* (Standard). Object Management Group (OMG). Retrieved from https://www.omg.org/spec/UML/2.5.1

Galinier, F. (2021). *Seamless development of complex systems: a multirequirements approach* (Unpublished doctoral dissertation). Université Paul Sabatier-Toulouse III.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. M. (1994). *Design patterns: Elements of reusable object-oriented software* (1st ed.). Boston MA: Addison-Wesley Professional. Retrieved from http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1

Glinz, M. (2000). Problems and deficiencies of UML as a requirements specification language. In *Tenth International Workshop on Software Specification and Design. IWSSD-10 2000* (p. 11-22). doi: 10.1109/IWSSD.2000.891122

Gomes, L., & Costa, A. (2003). From use cases to system implementation: statechart based co-design. In *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2003. MEMOCODE '03. Proceedings.* (p. 24-33). doi: 10.1109/MEMCOD.2003.1210083

Jackson, D. (2012). *Software abstractions: logic, language, and analysis*. MIT press.

Jacobson, I., Christerson, M., Jonsson, P., & Övergaard, G. (1992). *Object Oriented Software Engineering: A Use Case Driven Approach*. Boston MA: Addison-Wesley.

Jacobson, I., Spence, I., & Bittner, K. (2011). *Use-case 2.0 the guide to succeeding with use cases*. Alexandria, Virginia: Ivar Jacobson International SA. Retrieved from https://www.ivarjacobson.com/sites/default/files/field_iji_file/article/use-case_2_0_jan11.pdf

Jacobson, I., Spence, I., & Kerr, B. (2016, jan). Use-case 2.0: The hub of software development. *Queue*, *14*(1), 94–123. doi: 10.1145/2898442.2912151

Larman, C. (2012). *Applying UML and patterns: an introduction to object oriented analysis and design and interative development*. Pearson Education India.

Leavens, G. T., Baker, A. L., & Ruby, C. (1999). JML: A notation for detailed design. In *behavioral specifications of businesses and systems* (pp. 175–188). Springer.

Meyer, B. (1988). *Object-oriented software construction*. Prentice Hall.

Meyer, B. (1992). Applying "Design by contract". *Computer*, *25*(10), 40–51.

Meyer, B. (1997). *Object-oriented software construction, 2nd edition*. Prentice Hall.
Meyer, B. (2022). *Handbook of requirements and business analysis*. Springer.
Murali, R., Ireland, A., & Grov, G. (2016). UC-B: Use case modelling with event-b. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z* (pp. 297–302).
Naumchev, A. (2019). Seamless Object-Oriented Requirements. In *2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)* (pp. 0743–0748).
Naumchev, A., & Meyer, B. (2016). Complete contracts through specification drivers. In *10th int. symp. on theoretical aspects of software engineering (tase)* (pp. 160–167). IEEE Comp. Society.
Naumcheva, M. (2022). *Roborace repository.* https://github.com/mnaumcheva/Roborace_requirements_code. Github.
OMG. (2019). *OMG Systems Modeling Language (OMG SysML), Version 1.6* (Tech. Rep.). Object Management Group. Retrieved from https://www.omg.org/spec/SysML/1.6/
Overgaard, G., & Palmkvist, K. (2004). *Use cases: Patterns and blueprints*. Addison-Wesley Professional.
Pastor, O., Insfrán, E., Pelechano, V., Romero, J., & Merseguer, J. (1997). OO-Method: an OO software production environment combining conventional and formal methods. In *International conference on advanced information systems engineering* (pp. 145–158).
Petre, M. (2013). UML in practice. In *2013 35th international conference on software engineering (icse)* (p. 722-731). doi: 10.1109/ICSE.2013.6606618
Pnueli, A. (1977). The temporal logic of programs. In *18th annual symposium on foundations of computer science (SFCS 1977)* (p. 46-57). doi: 10.1109/SFCS.1977.32
Poppendieck, M., & Poppendieck, T. (2003). *Lean software development: an agile toolkit*. Addison-Wesley.
Roborace. (2022). https://roborace.com. Author.
Rosenberg, D., & Stephens, M. (2007). *Use case driven object modeling with uml: Theory and practice*. Springer.
Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. E., et al. (1991). *Object-oriented modeling and design* (Vol. 199). Prentice Hall.
Schaaf, R. (2012). *Use-case slice based product backlog - an example.* https://xebia.com/blog/use-case-slice-based-product-backlog-an-example/. Xebia blog.
Soavi, M., Zeni, N., Mylopoulos, J., & Mich, L. (2021). From legal contracts to formal specifications: A progress report. In *Joint proceedings of REFSQ 2021 workshops with the 27th international conference on requirements engineering: Foundation for software quality (REFSQ 2021), essen, germany, april 12, 2021* (Vol. 2857). CEUR-WS.org. Retrieved from http://ceur-ws.org/Vol-2857/nlp4re10.pdf
Soltana, G., Fourneret, E., Adedjouma, M., Sabetzadeh, M., & Briand, L. (2014). Using UML for modeling procedural legal rules: Approach and a study of luxembourg's tax law. In *International conference on model driven engineering languages and systems* (pp. 450–466).
Soltana, G., Sabetzadeh, M., & Briand, L. C. (2016). Model-based simulation of legal requirements: Experience from tax policy simulation. In *2016 ieee 24th international requirements engineering conference (re)* (pp. 303–312).
Tschannen, J., Furia, C. A., Nordio, M., & Polikarpova, N. (2015). Autoproof: Auto-active functional verification of object-oriented programs. In *International conference on tools and algorithms for the construction and analysis of systems* (pp. 566–580).
von Olberg, P., & Strey, L. (2022). Approach to generating functional test cases from bpmn process diagrams. In *2022 ieee 30th international requirements engineering conference workshops (rew)* (p. 1-5).
Waseem, M., & Sadiq, M. U. (2018). Application of model-based systems engineering in small satellite conceptual design-a sysml approach. *IEEE Aerospace and Electronic Systems Magazine*, *33*(4), 24–34.
Weilkiens, T. (2011). *Systems engineering with sysml/uml: modeling, analysis, design*. Elsevier.
Whittle, J., & Schumann, J. (2000). Generating statechart designs from scenarios. In *Proceedings of the 22nd international conference on software engineering* (pp. 314–323).
Xie, J., Tan, W., Yang, Z., Li, S., Xing, L., & Huang, Z. (2022). Sysml-based compositional verification and safety analysis for safety-critical cyber-physical systems. *Connection Science*, *34*(1), 911–941.
Yue, T., Briand, L. C., & Labiche, Y. (2013, mar). Facilitating the transition from use case models to analysis models: Approach and experiments. *ACM Trans. Softw. Eng. Methodol.*, *22*(1).
Yue, T., Briand, L. C., & Labiche, Y. (2015). aToucan: an automated framework to derive UML analysis models from use case models. *ACM Trans. on Soft. Eng. and Methodology (TOSEM)*, *24*(3), 1–52.
Yue, T., Zhang, H., Ali, S., & Liu, C. (2016). A practical use case modeling approach to specify crosscutting concerns. In *International conference on software reuse* (pp. 89–105).

About the authors

Maria Naumcheva is a PhD candidate at the University of Toulouse/IRIT. Her research interests include Requirements Engineering and Software Engineering Education. You can contact the author at maria.naumcheva@irit.fr.

Sophie Ebersold is a full professor at the University of Toulouse. She is head of the SM@RT team of the IRIT CNRS laboratory. Her research areas include the modeling of complex systems, methods/model/language integration, with a focus on Requirements and Systems Engineering. You can contact the author at sophie.ebersold@irit.fr.

Alexandr Naumchev holds a Ph.D. degree in Computer Science from the University of Toulouse and a Master of Science degree

in Software Engineering from Carnegie Mellon University. His doctoral thesis explores the idea of using an object-oriented language with contracts for requirements specification, verification, and reuse. Currently an independent researcher, Alexandr is focusing on static analysis of multithreaded programs and relational database queries. You can contact the author at anaumchev@outlook.com.

Jean-Michel Bruel is head of the Computer Science department of the Technical Institute of Blagnac and a member of the Strategic Research Committee of the IRIT CNRS laboratory since 2021. His research areas include the development of software-intensive Cyber-Physical Systems, and methods/-model/language integration, with a focus on Requirements and Model-Based Systems Engineering. You can contact the author at bruel@irit.fr.

Florian Galinier is co-founder and CEO of SPILEn, a company specialized in software and requirements engineering best practices. His research focuses on software engineering and model-driven engineering, with an emphasis on requirements engineering and domain-specific modeling languages. He holds a PhD in Computer Science from the University of Toulouse (2021). You can contact the author at fgalinier@spilen.fr.

Bertrand Meyer is Professor of Software Engineering at Constructor Institute in Schaffhausen, Switzerland, and Chief Architect of Eiffel Software in Santa Barbara. He is interested in various aspects of software engineering including software verification and object technology. You can contact the author at bm@sit.org.