Cite as: Bertrand Meyer, *Multirequirements*, in *Modelling and Quality in Requirements Engineering* (Martin Glinz Festscrhift), eds. Norbert Seyff and Anne Koziolek, MV Wissenschaft, 2013.

Multirequirements

Bertrand Meyer

ETH Zurich, NRU ITMO (Saint Petersburg) and Eiffel Software

Abstract: Requirements benefit from integrating several viewpoints, and from an object-oriented style.

Lack of structure, abstraction and hiding frequently make requirements hard to comprehend. We could get rid of these problems by treating requirements as objects. OO requirements would bring the benefit of OO principles, allow seamless application of OO methods through the development cycle, ensure a smooth transition from requirements to architecture and design, and support round-trip engineering.

Martin Glinz, from [1] (abridged)

1. The basic idea (self-referentially)

1.1 Requirement, requirements, project

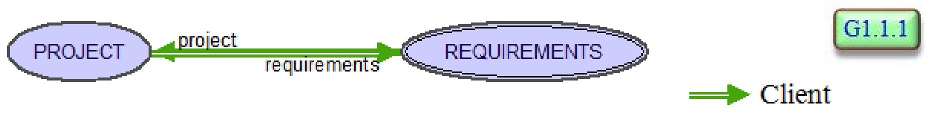
1.1.1 A software project /*PROJECT*/ exists to address some needs and must satisfy some constraints. The term "requirements" denotes the description of these needs and constraints. Any project, even one that most enthusiastically follows an agile, specify-as-you-go process, has requirements /*REQUIREMENTS*/; and conversely any "requirements" is relative to a project:

```
class PROJECT feature                                   -- E1.1.1
        requirements: REQUIREMENTS
invariant
        requirements.project = Current
end

class REQUIREMENTS feature
        project: PROJECT
invariant
        projects.requirements = Current
end
```

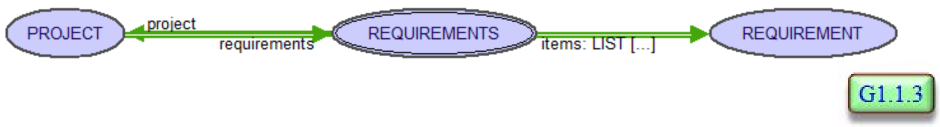
I1.1.1 (*Informative text*) It is convenient, the first time the requirements text introduces a term such as "project" describing an important abstraction that will be described by a class, to mention the class name in slashes, as in /*PROJECT*/. A tool should be available to collect all such occurrences automatically into an index.

1.1.2 We can enter the above information directly into our tools:



1.1.3 The term “requirements” is used as a collective (as in “the requirements document”), commanding a verb in the singular, but also as the plural of “requirement”. Indeed “a requirement” /*REQUIREMENT*/ is a specification of a particular system property; so “the requirements” as a whole contains a list of individual “requirements” (called that way, “individual requirements”, whenever confusion could arise):

```
class+  REQUIREMENTS feature+                                    -- E1.1.3
        items: LIST [REQUIREMENT]
end
```



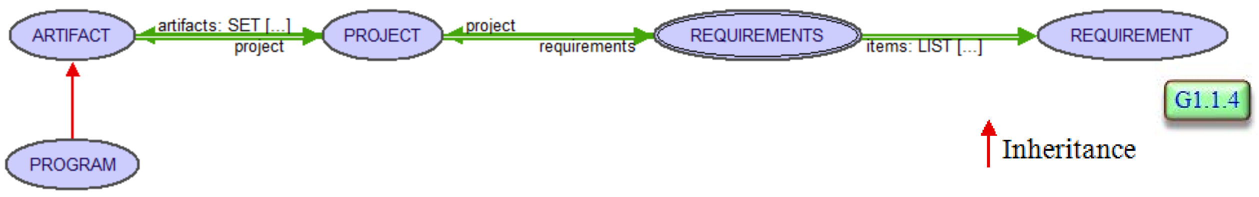
I1.1.3 (*Informative text*) As a general convention, we add a “+” to the keyword of a clause of a class, as here “**feature+**”, to indicate that the contents must be added to preceding ones for that class, here *REQUIREMENTS*. (For clarity, a class extract describing such an incremental extension starts with **class+** rather than just **class**.) The full corresponding clause for the class, for example here the full **feature** clause, is the concatenation of all such contents; supporting tools should reconstruct it on request. This convention is essential to support an incremental, iterative approach to constructing requirements specifications.

1.1.4 Similarly, a project produces a number of “artifacts” /*ARTIFACT*/. They can be of many different kinds, such as programs /*PROGRAM*/:

```
class ARTIFACT feature                                           -- E1.1.4
        project: PROJECT
end

class PROGRAM inherit ARTIFACT end

class+ REQUIREMENTS feature+
        artifacts: SET [ARTIFACTS]
end
```



1.1.5 The order of individual requirements in a “requirements” may be relevant, for example to produce a sequential compendium in the form of a requirements document; hence the use of a *LIST* (E1.1.3). For artifacts in a project, the order is irrelevant; hence the use of a *SET* (E1.1.4).

1.1.5 Artifacts can be composite: for example a program contains modules, which may themselves contain further components. In other words, an artifact can be part of another artifact. This aspect is fairly easy to describe using standard techniques, but for brevity the present discussion shall not detail it.

1.2 Traceability

1.2.1 Good requirements should satisfy a number of properties (see e.g. section 19.6 of [8]). We can model them through features and invariants of the classes *REQUIREMENTS* and *ARTIFACT*. One of the most important is traceability, which will serve as an example.

1.2.2 We may define traceability (although from the literature this does not seem to be the usual approach) as the combination of two properties: up-traceability /*up_traceable*/ and down-traceability /*down_traceable*/.

> I1.2.2.2 (*Informative text*) The /…/ convention for relating English terms to elements of the model is applicable to any construct, not just classes. Here *up_traceable* and *down_traceable* will be class features.

1.2.3 Traceability in either direction denotes a correspondence between artifacts and requirements. The correspondence is in the form of dependencies from individual artifacts to individual requirements (note the direction). The exact form of the dependency shall not be specified here; it suffices to characterize it abstractly through the notion that an artifact "follows from" /*follows*/ a requirement. The artifact and the requirement from which it follows shall belong to the same project /same_project/.

```
class+ ARTIFACT feature+                                        -- E1.2.3
        follows: detachable REQUIREMENT
invariant+
        same_project: follows /= Void implies follows.project = project
end
```

> I1.2.3-A (*Example*) Assume a requirement states that "an alarm shall be produced whenever the temperature rises to the specified threshold". If the implementation uses event-driven techniques, it may contain an instruction that subscribes an alarm routine to a temperature change event type. That instruction "follows from" the requirement.

> I1.2.3-B (*Informative text*) The "**detachable**" mention signals a property that may not always exist. If it does not, the value is **Void**. In the absence of a **detachable** qualification, the property is always defined.

1.2.4 Up-traceability is the property that every element of every artifact of a project follows from some element of the requirements:

```
class+ ARTIFACT feature+                                        -- E1.2.4
        up_traceable: BOOLEAN
invariant+
        up_traceability: up_traceable implies follows /= Void
end
```

I1.2.4 (*Informative text*) In requirements elements such as the present one describing a desired property *P* of certain artifacts, here down-traceability, it is possible to specify *P* directly as an invariant property; here the invariant would have stated *follows* /= **Void**. Then only systems that satisfy *P* are expressible. This specification style is often too restrictive, since one may need to deal with systems that may or may not satisfy *P*, and then ensure that they do. A more flexible strategy, illustrated here, is to use a boolean query *has_p* (here *up_traceable*) representing whether the system satisfies *P*, and use, as invariant clause , *has_p* **implies** *P*. This specification style is widely applicable.

1.2.5 Up-traceability can be defined not just for a single artifact but for an entire project:

```
class+ PROJECT feature+                                                    -- E1.2.5
        up_traceable: BOOLEAN
invariant+
        up_traceability: up_traceable = (across artifacts as art all art.item.up_traceable end)
end
```

I1.2.5 (*Informative text*) The **across** syntax provides a notation for predicate expressions: ∃ *x*: *E* | *p* (*x*) can be expressed as **across** *E* **as** *x* **some** *p* (*x.item*) **end**, and a universal quantification (∀) is similarly expressed through **all**.

1.2.6 Down-traceability is the property that for every requirement at least one artifact follows from it. It can be defined for an individual requirement (relative to a project), but here we define it only for a project as a whole:

```
class+ PROJECT feature+                                               -- E1.2.6
        down_traceable: BOOLEAN
invariant+
        down_traceability: down_traceable =
                                (across requirements.items as req all
                                        across artifacts as art some art.item.follows (req.item) end
                                end)
end
```

1.2.7 Both up- and down-traceability are fundamental properties for a system of any kind. It is equally hard to justify:

(1.2.7.A) Either the presence of an artifact that does not follow, however remotely, from a business need as expressed by some element of the requirements.

(1.2.7.B) Or a requirement element that does not, somehow, somewhere, have a consequence in some artifact of the project.

2. Perspective

2.1 The particularly perceptive reader may by now have realized that something a little strange is going on. Not so strange perhaps once you accept that the purpose of this article is to present a new method for requirements specification, "*multirequirements*", and that the previous section (section 1) is simply a self-referential example of applying the method, along with introducing a number of observations about requirements in general. The present section (the rest of section 2) explains the context and summarizes the method.

2.2 The choice of example in section 1 results from the combination of three observations:

(2.2.1) The purpose of requirements specification is to describe systems.
(2.2.2) It is often useful start the presentation of a new idea by example.
(2.2.3) A method is a system, even if a purely conceptual one, and hence provides an example.

2.3 Of course instead of choosing "requirements" as the example domain of discourse the discussion could have avoided the circularity — avoided going to a "meta" level — by choosing some application area; it could in particular have used a standard example from the specification literature such as boilers or libraries. It was felt preferable, however, to use the very domain of requirements, enabling us to talk about requirements right from the beginning. Boilers and libraries are interesting to factory and library people, but requirements people presumably prefer to think about requirements.

2.4 Although I have thought about the multirequirements method for a long time, practiced it at least in part, and even presented it in a couple of talks, it is neither completely documented nor even completely defined. The present article should therefore be considered as a draft of work in progress; in particular, some details of the method's requirements specification style, illustrated in section 1, may still change.

2.5 The present discussion, and the method so far, focus on functional requirements. Non-functional requirements may be a target too, but I have not examined this question.

2.6 More generally, the multirequirements method is at the present stage a proposal and has not undergone large-scale use, even less any form of empirical evaluation.

2.7 The multirequirements method relies on four theses:

(1.2.7.C) The requirements process should use interwoven layers of discourse (hence the name of the method).
(1.2.7.D) The requirements process should rely on object technology.
(1.2.7.E) The requirements process should enforce traceability as a key objective.
(1.2.7.F) The requirements process demands adequate tool support.

2.8 Section 3 presents a summary of the multirequirements method. Sections 4 to 7 present the four theses. Section 8 concludes.

3. A summary of the multirequirements method

Here is an overview of the multirequirements method. The method is defined by the following principles for developing requirements:

(3.1) Develop individual requirements incrementally on several layers, including the following three: formal, graphical, natural-language.

(3.2) Use these layers both in a complementary way (when one of them is more appropriate to the description of a system property) and redundantly (for example to combine the precision of formal descriptions with the convincing power of graphical descriptions).

(3.3) Model systems through object-oriented techniques: classes as the basic unit of decomposition, inheritance to capture abstraction variants, contracts to capture semantics.

(3.4) Use an object-oriented language (in the present discussion, Eiffel) to write the formal layer according to the principles of 3.3.

(3.5) Use the contract sublanguage of the programming language as the notation for the formal layer.

(3.6) As the goal is to describe models, not implementations, ignore the imperative parts of the programming language (such as assignment).

(3.7) Use an appropriate graphical notation (in the present discussion, BON [16]) for the graphical layer.

(3.8) Weave the layers to produce requirements descriptions, including a comprehensive requirements document if requested, but also any other appropriate views.

(3.9) Enforce and assess traceability between the layers and all products of the requirements process, and between requirements and other product artifacts, both down and up.

(3.10) Rely on appropriate tools to support the process, including incremental development.

4. Thesis A: multi-layer, interwoven requirements

4.1 Requirements techniques are of three main kinds distinguished by their mode of expression (reflecting fundamental conceptual choices): natural-language, such as English; graphical, such as UML or BON; formal (mathematics-based) such as Z.

4.2 These approaches are usually presented as alternatives, and much fighting continues between their proponents.

4.3 The fighting makes no sense. The approaches are complementary, because each has its unique strength: one cannot compete with mathematics for precision; with natural language for combining abstraction and detail; and with graphics for ability to convey a general structural idea quickly and convincingly. In each case, the other two approaches do not measure up.

4.4 The condition for using multiple descriptions is to guarantee that they remain **compatible**. This idea has been applied extensively, under the name of “Single-Product Principle” [7] (or “Single-Model” [13]), in the Eiffel context (and in the Java world to JavaDoc), leading to the rule that documentation should not be a separate product but a certain *view* of the software. More precisely, many views are possible — contract view, ancestor view, interface view... — describing different levels of abstraction. They can be extracted from the software by tools. This approach guarantees compatibility since everything is derived from a single base, the contractequipped program text, serving as the authoritative reference. For requirements the promise of compatibility is more difficult to enforce, but we can rely on a similar idea.

4.5 In the multirequirements approach, the requirements consist of items of three kinds: formal elements (see section 5); graphical diagrams, using BON; and natural-language paragraphs, numbered and structured. The example of section 1 has illustrated all three kinds.

4.6 The elements of these three kinds, or “layers”, are closely connected. This property was also illustrated in section 1: the natural-language description refers to formal elements, such as class names, with conventions such as the /.../ notation (for example, /*PROJECT*/) to mark the first occurrence of an abstraction that has a formal description. Similarly, the diagrams illustrate relations, client and inheritance, which usually have more precise specifications in the formal text and explanations in the natural-language text.

4.7 As these observations indicate, redundancies between the three layers are possible and in general beneficial, since expressing the same properties in different ways, each more directly evocative to different classes of project stakeholders — formal descriptions for advanced programmers and for the quality-assurance team, natural-language ones for non-software-technical customer representatives, diagrams for managers — increases the likelihood that misunderstandings and other potentially damaging requirements mistakes will be caught early on, one of the principal goals of a requirements process.

4.8 Redundancies are only acceptable if the descriptions are compatible. One way to ensure compatibility is to perform round trips between the three views. In particular, I described in an early article [6] how a detour through formal specification can yield a better natural-language specification, translated back from the formal description into English, but not necessarily the kind of English one would write without that detour: a more formal, more precise form of English. Michael Jackson cites newer examples (from Heimdahl et al.) in an article of the present volume [3]. As to the graphical representations, tools such as those of EiffelStudio can both produce them from the formal text, and, the other way around, allow users to enter them directly in graphical form and generate the formal text from them. For consistency, such tools should support full round-tripping.

4.9 There is no universally prescribed writing order between requirements elements or between the three layers (formal, graphical, natural-language). Each project may use its own guidelines. It is the task of supporting tools (section 7) to produce various compendiums, similar to the “views” of a program mentioned in section 4.4.

4.10 In particular, most projects need a “requirements document” that collects all the requirements in some defined order, for example the order dictated by the IEEE-830 requirements standard. Section 1 showed what such a document — rather, a typical extract — can look like, with the three layers interwoven. The order of such a document, however, is only one possible order.

4.11 The concept of weaving suggests an analogy between multirequirements and Knuth’s *Literate Programming* [5]. The basic idea is indeed the same. When I first read about literate programming I was seduced by the elegance of the approach, but found it inapplicable to modern, object-oriented programming which (as discussed in several publications including [7]) is fundamentally bottom-up as implied by the focus on reuse; literate programming seemed inextricably tied to the top-down, function-driven programming style of the nineteen-seventies. In that traditional view, a program implements a single "main" function; as a consequence the "literate" text is the sequential telling, cradle to grave, of a single story. What remains as compelling as at the time of Knuth's original presentation, however, is the idea of weaving several layers of discourse, in his case just two, the program and the documentation. Eiffel achieves this goal through different means, using the single-product principle discussed in section 4.4. With multirequirements we retain the weaving idea of literate programming, applied to three or more layers, and without the constraint of a single order or of a single story thread.

4.12 The three basic layers of description selected for multirequirements are the most immediately applicable, but more layers, such as Parnas's table-based requirements techniques [14] or Harel's statecharts [2], are natural candidates for addition.

5. Thesis B: object-oriented requirements

5.1 Object technology is essentially an architectural method [7], building the architecture of a system on the basis of a system's object types, described through the techniques of abstract data types, as opposed to modularization based on the system's functions. The goals are, among others, extendibility, reusability and reliability. These goals are important for requirements as well as for programs, and one may surmise that the same object-oriented solutions will be effective there too.

5.2 "Object-oriented" means based on a division into classes, connected through client links (which rely on interfaces only, applying information hiding) and inheritance links (which support classification of abstractions), and described through contract elements, in particular preconditions, postconditions and class invariants.

5.3 Object-oriented requirements are requirements built along the same architectural principles. The system is modeled as a set of objects defined by the corresponding classes, with client and inheritance links (no others) to specify the structure and contracts to specify the semantics.

5.4 One of the arguments for adopting an object-oriented policy from requirements on is to support the seamlessness of the object-oriented approach [7], eloquently described by Martin Glinz in the citation that opens the present article.

> I5.4 (*Informative text*) "Described", not advocated, as the paper from which the citation was taken also presents for fairness the opposite thesis — requirements are not objects. That counter-argument, however, is not pushed forcefully, so we may take a reasonable guess as to where the author's sympathies lie.

5.5 Seamless development is, in my experience, among the principal benefits of object technology. It should be noted, however, that not everyone shares this view; Michael Jackson, for example, concludes his contribution to the present volume [3] with the comment that "*in general, any aspiration to seamless development of a realistic cyber-physical system, in which the same structure is carried through all development phases from requirements to software architecture and design, must be regarded with deep suspicion*". More generally, his paper highlights the risks of using piecemeal specifications, and this criticism would presumably apply to the interwoven requirements of the multirequirements method, although I believe that proper tools would address the criticism. These issues are clearly fodder for future discussions.

5.6 Part of thesis B is that the contract sublanguage of a notation such as Eiffel covers the needs of the "formal" layer of requirements description (section 4), obviating the need for a special mathematical notation such as Z. If this sub-thesis is correct, the strategy provides considerable benefits: we avoid the need for mixing different mechanical notations, a programming language and a formal specification language. The example of section 1 provides, I hope, some arguments in support; note in particular the contribution of high-level constructs such as **across** expressions to provide first-order predicate calculus expressions at reasonable notational cost. Some sophisticated mathematical properties remain, however, heavy to express; we hope that new (and not yet implemented) mechanisms of Eiffel [11], providing notations equivalent to those of modern functional languages, plus type inference, will remove that issue. The full attainment of this goal remains, however, in the future.

5.7 The peculiarity of object-oriented requirements is that many teams, perhaps a majority, claim to be practicing it, but in my observations very few actually do at least in the sense described here. In particular, use cases [4], whatever their benefits, are not a form of object-oriented requirements: a use case is specific, whereas a class is general; it is concrete, where a class is abstract; and it is function-oriented, the reverse of object-oriented (section 4.11). The task of the requirements engineer is to go beyond the specific, the concrete and the functions to elicit the general, the abstract and the classes. Use cases can be useful [7, 10] for requirements elicitation (as a step in this abstraction process) and particularly as tests, but they do not constitute object-oriented requirements in the sense of the present discussion.

> I5.7 (*Informative text*) As the reader will have noted, the point of section 5.6 is not to offer a discussion of use cases, even less a criticism, neither of which is not in the scope of the present article, but to remove any ambiguity as to what "object-oriented requirements analysis" means in the context of the multirequirements method.

6. Thesis C: traceability

6.1 Requirements are only a tool towards building the program; they are not themselves the program. That is the reason why some agile and particularly "lean" approaches view them with suspicion, treating anything that is not deliverable as "waste". Such a reaction is exaggerated, but it is true that requirements not closely connected to the actual deliverables are worth little more than the keystrokes through which they were entered. Hence the central role, among quality criteria for requirements, of traceability as defined in section 1.2, including both up-traceability and down-traceability.

6.2 Traceability means in particular that tools should be able to establish and monitor dependencies between the requirements and other project artifacts such as the code.

6.3 A step (still tentative) in this direction is the EIS tool [9] of EiffelStudio, through which users can graphically record links between elements of the software text, typically classes or routines, and elements of the requirements, typically a paragraph in a Microsoft Word or PDF file. As a consequence, it is possible to trace the effect of a change on either side, to find out what elements on the other side may need updating.

6.4 Traceability is one of the reasons for which every element of the requirements (as well as every other artifact of a project) should be clearly and uniquely identified. The style illustrated in the present paper and particularly in section 1 relies on systematic conventions supporting this goal: every paragraph numbered; clear distinction between elements of the specification, informational comments and examples (standards documents systematically apply this convention, and requirements documents should follow it too); identification of formal elements through codes, such as E1.1.1, to associate each of them, in the EIS spirit, with the relevant natural-language paragraph; similar rule for graphical elements, with codes such as G1.1.1; frequent cross-references, supporting traceability analysis; for these cross references, use of precise values, such as "section 1.1", rather than contextual or order-dependent references such as "*as we have seen*" or "*in the previous section*" (that section might no longer be "*previous*" in the ordering produced by a different tool!).

6.5 These conventions are partial and subject to refinement, but highlight the special characteristics that must govern natural language when used for requirements, especially multirequirements.

7. Thesis D: tools

7.1 EIS is but one example of a tool supporting a multifaceted requirements method. The multirequirements method can only scale up if aided by a variety of tools for:

(7.1.1) Writing requirements in all layers (formal, natural-language, graphical).
(7.1.2) Recording dependencies, intra- and inter-layers.
(7.1.3) Interweaving the layers to produce a "requirements document" and other composites.

7.2 Rather than devising new stand-alone tools, it is preferable to integrate new functionalities in a comprehensive IDE, not only to leverage on existing mechanisms but more importantly to enforce traceability, beyond just requirements, through all kinds of software artifacts including requirements, designs and code, and all phases of the software lifecycle.

8. Conclusion

The multirequirements method recognizes the complexity of the requirements process and the need to rely on a compendium of complementary techniques. It renounces the traditional linear view of a single requirements document and the restriction to a single requirements notation (natural-language, graphical or formal). It takes advantage of object technology and particularly of Design by Contract techniques to express formal descriptions, using the notation of a modern programming language rather than a specialized mathematical language. It weaves together a variety of viewpoints and formalisms, textual and graphical, formal and informal, to produce requirements that can help projects and their diverse stakeholders. It relies on tools, integrated into an IDE, to support the requirements process. While no large-scale evidence of its benefits exists, I hope that the present discussion will have convinced the reader to try it out.

Acknowledgments

It is a pleasure to dedicate this article to Martin Glinz on the occasion of this Festschrift honoring his many contributions to the art, science and engineering of requirements.

The EIS system was built by Tao Feng, who contributed many ideas. The single-model work of Paige and Ostroff [13] has also been influential. Steele's "growing a language" [15] partly inspired the self-referentiality of the presentation.

The idea of applying software techniques to describe software processes goes back to Lee Osterweil [12] and was further developed by him and Lori Clarke. A difference with the assumptions of their work is that I believe object-oriented techniques, with Design by Contract,

provide the needed mechanisms without recourse to special process modeling languages; this idea, applied in section 1 of the present article to the modeling of requirements, has been the subject of a number of working discussions with Emmanuel Stapf.

Early versions of this article benefitted from perceptive comments by Michael Jackson and Ivar Jacobson.

Bibliography

[1] Martin Glinz: *Should Requirements Be Objects*? Tutorial Position Paper, *14th Annual International Symposium on Systems Engineering* (INCOSE), Toulouse, 2004.

[2] David Harel: *Statecharts: A visual formalism for complex systems*, in *Science of Computer Programming*, vol. 8, no. 3, pages 231-274, 1987.

[3] Michael Jackson: *Topsy-Turvy Requirements*, in *Martin Glinz Festschrift*, 2013 (this volume).

[4] Ivar Jacobson: *Object Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.

[5] Donald E. Knuth*: Literate Programming*, Cambridge University Press, 1992. (Extended from an article in *The Computer Journal*, vol. 27, no. 2, pages 97-111, 1984.)

[6] Bertrand Meyer: *On Formalism in Specifications*, in IEEE *Computer*, vol. 3, no. 1, January 1985, pages 6-25.

[7] Bertrand Meyer: *Object-Oriented Software Construction*, Prentice Hall, 1988 (first edition), 1997 (second edition).

[8] Bertrand Meyer: *Touch of Class: Learning to Program Well Using Objects and Contracts*, Springer-Verlag, 2009.

[9] Bertrand Meyer: *EIS: Putting into Practice the Single Model Principle* (blog entry, 5 July 2012), http://bit.ly/VlrRGZ. Includes references to EIS documentation.

[10] Bertrand Meyer: *A Fundamental Duality of Software Engineering* (blog entry, 14 October 2012), http://bit.ly/QEml00.

[11] Bertrand Meyer: *Eiffel as a functional language*, 2013, in preparation (draft available on request).

[12] Lee Osterweil: *Software processes are software too*, in ICSE '87, Proceedings of the 9th international conference on Software Engineering, IEEE/ACM, 1987, pages 2-13.

[13] Richard Paige and Jonathan Ostroff: *The Single Model Principle*, in *Fifth International Symposium on Requirements Engineering* (IEEE), 2001, pages 292-293.

[14] David Parnas: *Tabular Representation of Relations*, CRL Report 260, McMaster University, Telecommunications Research Institute of Ontario (TRIO), October 1992.

[15] Guy Steele: *Growing a Language*, in Journal of Higher-Order and Symbolic Computation, vol. 12, pages 221–236, 1999.

[16] Kim Waldén and Jean-Marc Nerson: *Seamless Object-Oriented Software Architecture*, Prentice Hall, 1995 (also available at http://www.bon-method.com/).