Principles for Modeling Language Design

Richard F. Paige [a,1], Jonathan S. Ostroff [a,1], Phillip J. Brooke [b]

[a] *Department of Computer Science, York University, 4700 Keele St., Toronto, Ontario M3J 1P3, Canada.* {paige,jonathan}@cs.yorku.ca

[b] *CESG United Kingdom*
pjb@mithlond.demon.co.uk

Keywords: Modeling languages; Design principles; UML; Unification.

Abstract

Modeling languages, like programming languages, need to be *designed* if they are to be practical, usable, accepted, and of lasting value. We present principles for the design of modeling languages. To arrive at these principles, we consider the intended use of modeling languages. We conjecture that the principles are applicable to the development of new modeling languages, and for improving the design of existing modeling languages that have evolved, perhaps through a process of unification. The principles are illustrated and explained by several examples, drawing on object-oriented and mathematical modeling languages.

1 Introduction

The key difficulty in producing quality software is specifying and designing the conceptual construct that underlies the software [2]. This conceptual construct is usually complex. Complexity is an essential difficulty that cannot be dealt with by using more powerful programming languages or tools, or by using modeling languages that abstract it away. Complexity must be dealt with by the developer, who must choose and apply the most appropriate languages and tools, based on their knowledge and experience.

Suitable *modeling languages* are needed to describe the conceptual construct underlying software. These languages, which are often graphical, can be used to produce a satisfactory description of the conceptual constructs, frequently prior to writing

[1] Supported by the National Sciences and Engineering Research Council of Canada.

any code. Prior construction of a model for a derived software system is as essential as having a blueprint for a building or a schematic for a circuit before building them.

Recently, the Unified Modeling Language (UML) [29] has been proposed as a standard modeling language, particularly but not exclusively for modeling object-oriented (OO) systems. The UML is a unification of a collection of software development approaches. Unification of these approaches was carried out to help provide stability to the OO technology marketplace, while giving a standard notation for developers to apply. As such, the designers of UML focused on: *standardizing* the syntax and informal semantics of the modeling language; and, easing the transition from existing technologies to the UML, in part by ensuring a degree of syntactic and conceptual compatibility.

Unification is not the only way that modeling languages have been developed. They have also been designed from scratch, typically with only minor concern for compatibility with existing languages. In designing anew, language designers embody experience and previous work in the syntax, semantics, and tools that support the new language. As the language becomes used in practice, the designers are often compelled to make extensions or additions of new features to meet previously unknown requirements, and so the languages evolve. Examples of such modeling languages include the Business Object Notation (BON) [36], SOMA [6], and some of the predecessors of UML, as well as real-time approaches such as HOOD [28] and MASCOT [31]. Also in this category are formal notations, e.g., CSP [12], CCS [21], Z [32], and B [1], which should be viewed as modeling languages: they give a mathematical model of a system that can be reasoned about (via theorem provers or model checkers).

In both unification and new development, modeling languages are constructed to meet certain goals. In the case of unification, ease of transition from earlier notations, as well as standardization, are important goals. For other languages, there may be goals such as *ease of drawing*, *simplicity*, or *ease of reasoning*. These goals drive the development of the syntax, semantics, documentation, and tool support for modeling languages.

An implication of this discussion is that, analogous to programming languages, modeling languages have to be designed. A part of the design process for modeling languages requires the formulation of a plan, consisting of a set of *goals* for the language. By carefully delimiting goals for a modeling language, it can be determined whether or not a language meets the needs of its intended users, and also can lead to better documentation for the language, especially in terms of the language's intended domain of practical use. A valid question to now ask is: what goals may be of interest in the design of a modeling language? To help answer this question, we can turn to the realm of programming language design.

Criteria and principles for programming language design have been discussed and presented over the last thirty years [11,33,34,38]. These principles have provided programming language designers with guidance on whether to include features in their languages, as well as criteria for the qualitative comparison of languages. We suggest that some of these principles are directly applicable to the design of modeling languages, as we shall discuss. Where programming languages and modeling languages primarily differ is in their intended domain of use – programming languages describe executable systems, while modeling languages need not.

In this paper, we discuss the *goals* that may be of interest in designing modeling languages, as well as *principles* that can help designers meet those goals. These principles in turn can suggest ways to qualitatively compare modeling languages.

How can we determine whether we have captured the most important and essential principles for the design of modeling languages? Ideally, these design principles will be used to improve the quality of modeling languages, in the sense that the modeling languages produced by following the principles can be used to create higher-quality software than others. Thus, the ideal approach to validating the principles would be to design a collection of metrics that quantify the *quality* of the software and documents produced using a modeling language. Based on these metrics, we could carry out a number of experiments in software development and measurement, using a modeling language. In the experiments, we would determine whether the modeling languages actually abet the development of quality software, and thus whether the design principles are useful in producing higher-quality modeling languages.

Such an experimental process requires a well-founded, widely accepted, and quantifiable definition of software quality, which currently does not exist. In the absence of such a standard definition, we must view the principles suggested in this paper as a starting point for the design of these experiments.

2 The Use of Modeling Languages

A key question to begin our presentation is, 'For what do we intend to use a modeling language?' A modeling language, as stated in [29], is a language used to 'specify, visualize, construct, and document a software system'. Fowler [5] extends this definition to use modeling languages to describe concepts and constructs in the *problem domain*. We use these two definitions as the starting point for our discussion.

As [5,29] suggest, it may be useful in some cases for a modeling language to be graphical. This requirement is aimed at abetting the development of large software systems, which possess a myriad of inter-related components. Visual modeling lan-

guages provide the means to effectively convey these components and their interactions, and allow developers to focus their attention on parts of a system at any time.

However, primarily textual modeling languages, such as CSP or B, achieve these intended uses more concisely without the problems of secondary notation [26]. It is this secondary notation (e.g., layout and typographic cues) that sometimes appeals to users, and causes confusion in interpreting graphical notations.

There is an increasing amount of work on the subject of visualisation in a software development context [7,8,25]. It is largely inconclusive; it is not clear what an appropriate notation (textual or graphical) is in a given context. There are no obvious rules to apply: thus we conclude that both graphical, textual, and hybrid notations are useful depending on specific circumstances.

Modeling languages are used by designers. They are most often applied *before* program code is constructed (though, as we shall see when we discuss reversibility, this is not always the case). Modeling languages are tools to support the designers, and as such should provide assistance in those most critical and complicated tasks. What are the critical tasks of users of modeling languages? We suggest that there are four critical tasks for designers:

(1) architectural description,
(2) behavioural description,
(3) system documentation, and
(4) forward and backwards generation.

2.1 Architectural description

Modeling languages are used to describe a system in terms of abstractions (which may be classes, processes, use-cases, programs, and so on) and their relationships (which include inheritance/subtyping, data flow, sequencing, et cetera). Thus, a useful modeling language will provide designers with assistance in expressing necessary abstractions and their relationships at appropriate levels of detail and in an appropriate form (i.e., visually or textually). A document that consists of such abstractions and their relationships is often called an *architectural description.*

An example of using a modeling language for architectural description is shown in Figure 1. The modeling language UML is used to describe the architecture of an office management system. The model is made up of classes (drawn as rectangles), associations (drawn as lines between classes), methods and attributes, and annotations, such as multiplicity constraints.

Modeling languages are used at several different stages in the development life-

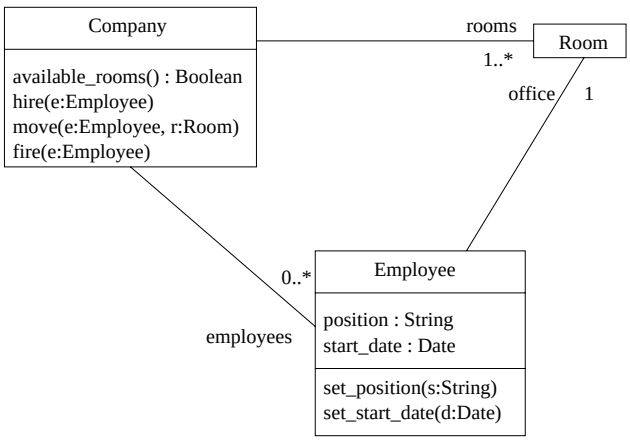

Fig. 1. Using UML for architectural description

cycle. Most commonly, they are used during *requirements analysis* (for conceptual modeling) and *design*. In requirements analysis, the languages will be used to describe abstractions associated with the problem domain [5]. In design, the languages will be used to describe abstractions of the solution domain (though, as Jackson points out [13], some abstractions will belong to both the problem and solution domain). Thus, the modeling language should be usable for describing abstractions throughout the development lifecycle.

Modeling languages should be applicable to the development of large systems, which possess many abstractions and inter-relationships. The language should be able to describe systems at various levels of detail, from the highest level where the system is itself a single abstraction, to a level of abstraction where requisite system behavior is acquired by interactions of a large collection of implemented software components. Formal or informal refinement relationships, and notions of traceability between levels of abstraction can usefully be defined to improve the overall quality of the software product being developed.

Language designers should not sacrifice applicability to small systems so as to make the language more applicable to large systems, otherwise users will have difficulty learning the language. But the language must still scale up, otherwise it cannot be useful for more than toy problems. Thus, the language must support concise architectural description.

2.2 Behavioural description

An architectural description is generally insufficient to completely express the details of a system. An improved description will include details of what each abstraction in the model represents, what each does, and when interactions between model components occur. This is the purpose of the behavioural description.

There are many styles of notation that can be used for behavioural description, e.g., process algebras (CSP), state-based descriptions (written in, e.g., Z, B, and the Object Constraint Language), natural language (e.g., used in process specifications associated with data flow diagrams), or variants of finite state machines (say, Statecharts, also as used in UML). Each style of notation can be used in different ways: finite state machines are useful for animation, or for model checking, while variants of state-based modeling are useful for automated code generation and for semi-automatic proof and reasoning. The most appropriate notation to use for behavioural description will depend on the project context.

As an example, consider the architectural description in Figure 1. The description says nothing about the behaviour of methods associated with the classes. To describe the behaviour of the method `fire` of class $Company$, a modeler could use the Object Constraint Language to give the method a pre- and postcondition, as follows.

```
Company :: fire(e : Employee)
pre : employees → includes(e)
post : not employees → includes(e) and
       employees → forAll(f | f.office ≠ (e@)pre.office)
```

The precondition (indicated by the `pre` annotation) requires that the employee `e` (the employee to be fired) is included in the collection of all employees. The postcondition ensures that the argument `e` is no longer in the collection, and that after firing the employee's former office is now unoccupied.

The preceding model of the office management system, written as a UML class diagram and OCL constraints, separates architectural and behavioural descriptions. These descriptions can also be combined in the same model. With the BON object-oriented modeling language, state-based descriptions of behaviour – in the form of pre- and postconditions – are included within the architectural description. Figure 2 shows a BON description of the same office management system that integrates both architectural and behavioural description. The architectural description is graphical: classes are represented as rounded rectangles or ellipses (the latter being used when class details are not described), and the association relationships between classes are represented as directed arrows (e.g., between $COMPANY$ and $EMPLOYEE$). The behavioural description of method $fire$ of class $COMPANY$ – again, written as a pre- and postcondition – is integrated with the architectural description.

A similar approach is taken with the B method, where abstract machines combine architectural and behavioural description. UML, on the other hand, allows developers to describe architecture using class diagrams, and behaviour with separate OCL

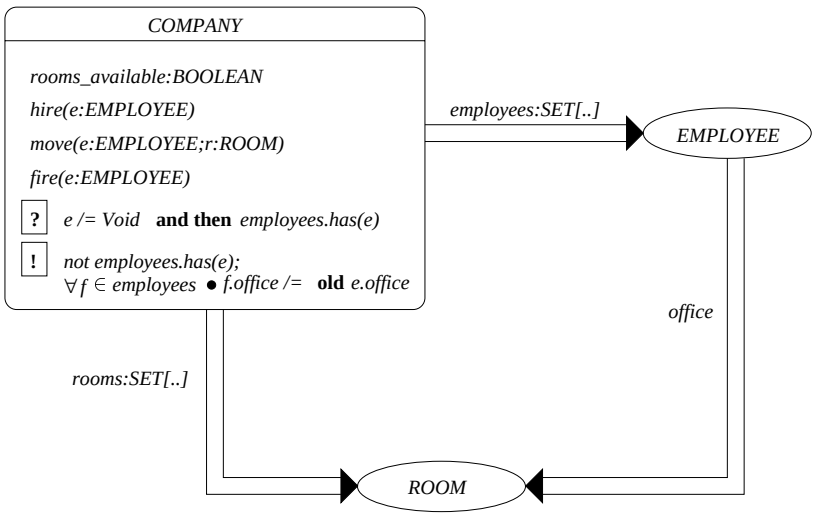

Fig. 2. BON diagram integrating architectural and behavioural description

constraints (as the previous example demonstrated) or separate Statechart models. UML also permits integrated description of architecture and behaviour; examples in [24,29] demonstrate this.

2.3 System documentation

System documentation is provided to explain how a system works, and what needs to be done in order to adapt or maintain it, to meet changing requirements or to correct mistakes or omissions. A model, produced with a modeling language, is itself a form of system documentation. It can present a high-level architectural view of a system, details for abstractions — e.g., method signatures for a class, inputs and outputs for processes, pre- and postconditions — and (from a design perspective) an effective summary of the implementation of the system. A quality modeling language will thus encourage designers to write clear, self-documenting systems. The language will also support the designer in maintaining the documentation as the system evolves, such as when programs change under system maintenance.

System documentation must of course be supplemented by other forms of documentation, including justifications of the design and modeling decisions that were made during production, as well as instructions on its use.

2.4 Forwards and backwards generation

Very often, modeling languages are used in the front-end of the development life-cycle, when developers are trying to understand the problem they have been given, and when they are planning which abstractions to use in the solution.

Models of software systems are eventually implemented, typically in a high-level programming language, but ultimately in object code. Thus, a modeling language should be based upon abstractions that can be transformed as easily as possible into program code. Clearly, behavioural descriptions that are mathematical, e.g., pre- and postconditions or process algebras, may require further refinement and elucidation to map into code. However, for architectural abstractions, such as classes, modules, and types, it should be possible to directly map them into corresponding program code elements. Without this, we make it harder to trace program errors back to model elements, thus making maintenance all the more difficult. Thus, a modeling language should include at least the architectural description elements that will be used in the eventual implementation language. This corresponds with the notion of *seamlessness* described in [20].

This requirement should not supersede earlier ones. Clearly, the simplest modeling language to map into a programming language is the programming language itself. However, the effectiveness of a programming language for describing architecture – especially for problem analysis – is questionable, in part because such languages often require the modeler to provide detail which may be irrelevant for the purposes of understanding the problem. This should be contrasted with the Eiffel language [20], which has been suggested as appropriate for use throughout the software development lifecycle, in part because of the language's support for behavioural description *and* implementation. However, visual descriptions of software architecture are still important and useful when applying the Eiffel language: for large systems, a diagrammatic representation can help in understanding the Eiffel code.

Easy mapping of architectural abstractions to programs is only one requirement for a modeling language. In previous subsections, we described the need for making it easy to document a system using a modeling language, and to maintain the models as programs change. Very often, models and programs are allowed to get out of synch because maintainers have been changing the code but not changing the models. This immediately destroys the value of the model for the purposes of documentation; designers and programmers can no longer rely on it. It is therefore useful to be able to produce models from programs. A typical modeling loop, then, is to produce a model, generate code from it, and then whenever the code is changed by programmers, generate a new model (which may require further improvements or augmentation by developers). This is often called *round-trip engineering*.

A good modeling language should support forward and backwards generation, and therefore make it as easy as possible for developers to have round-trip engineering.

Our first observation regarding these goals is that they are dependent. By designing a language to meet one of these goals, we may affect another. For example, documentation can be a part of an architectural design; the BON language, the SOMA language [6], and other modeling languages that focus on behaviour and responsibilities work along these lines. BON, for example, includes documentation about the behavior of functions and procedures within the class model itself, as Figure 2 demonstrated. Further, as already discussed, backwards generation of models from code is an effective means of producing documentation and architectural descriptions.

There are two implications of this. One is that it may be difficult to satisfy each goal in isolation. Language designers must therefore decide which goals are essential for the intended use of the modeling language. A second implication is that it will be non-trivial, because of this dependence, to satisfy all of these goals in one modeling language. From this, we can draw two conclusions. First, it may be very useful to *combine* or *integrate* separate modeling languages [22,23] for specific purposes, when it is discovered that one language does not satisfy all of the goals of its users. And, developers should be prepared to *select* a modeling language based on their reading of its appropriateness for a specific project.

An evident condition for achieving any of the design goals is the simplicity of the modeling language. Without simplicity, the language designer cannot evaluate the consequences of language design decisions, nor can they easily determine if they have met their goals. Without simplicity, designers will be overwhelmed by a collection of different abstractions and relationships. Without simplicity, forwards and backwards generation may only be possible on a subset of the language, if it is possible at all.

This correlates directly with the discussion of Hoare [11] and Wirth [38] on the design of programming languages. Hoare, writing about goals in the design of a programming language, says:

> *A necessary condition for the achievement of any of these objectives is the utmost simplicity in the design of the language.*

The main beneficiary of a simple modeling language is the user. A simple language is more likely to be fully understood by its users than a more complicated one. A user who fully understands the modeling language he or she is using will be more capable of handling more complex tasks. Such users will have come across the deficiencies in the language, and will know how to avoid them or mitigate their effects. Moreover, they may know how to use the simple modeling language in unexpected ways, and will likely know all the consequences of using language features in combination.

Many of the principles of programming language design that are presented by Hoare and Wirth also appear to be applicable directly to modeling languages. This is not surprising because both programming languages and modeling languages can be used for describing software systems, albeit at different levels of abstraction. Thus, many programming language principles should carry over to the modeling language domain. However, some principles of programming language design – related to compilation, execution efficiency, debugging, i.e., those concerning the specific form of software models expressible in a programming language – will not. We should expect that there will be principles for designing modeling languages that are not applicable to the programming language domain as well.

We conject that the main principle for the design of a modeling language should be simplicity. Without it, we will likely have significant difficulty in achieving the goals mentioned in previous subsections. It also aids the modeler in learning and remembering all the features of the language, and in selecting the best feature for their purpose. Modelers can expend most of their effort on using the language, rather than on learning it.

With complex modeling languages, a significant amount of effort may have to be expended in learning the language before applying it. Such language complexity also unfavourably impacts on the tool implementer as well, who is responsible for ensuring that the tool is compliant with the modeling language specification. Certainly, one can always use a subset of a complex modeling language, especially at first when newcomers are familiarizing themselves with the language, but if it is feasible to usefully apply a subset of a modeling language, one can rightly ask if it is *necessary* and *useful* to include and use the remaining parts. Further, how would newcomers choose a useful subset?

3 Designing a Modeling Language

As previously suggested, there are two broad ways in which modeling languages have been built. They have been built from scratch, without explicit consideration of compatibility with earlier approaches and techniques (excepting, of course, for that knowledge which is encompassed in the experience of the language designers and which is therefore implicit in any design decisions that are made). Thereafter, there may have been further additions to the language, as it evolves through use. They have been produced by unification from existing approaches. There are advantages and disadvantages to each approach, which we now discuss.

3.1 Design from scratch

A modeling language designed from scratch has little or no history. It is not required to remain compatible with languages that have been used before. It does not have to include features just because other languages have them. It can be constructed to meet specific design goals; tradeoffs in order to make it easy to transition from previous languages to the new language may not have to be made. Moreover, the language can be designed to include exactly the features that are needed to satisfy its design goals; all other features can be discarded. As a result, modeling languages built in this way can usually be constructed to satisfy the goal of simplicity.

The difficulty with designing languages from scratch is that it is a complex problem. It is difficult to decide what features to include and what to omit. It is also hard because we need precise requirements before we can start building the language. We must be able to precisely express the goals of the language as well as the satisfaction criteria: for example, how will we know if our language is simple? Because of this complexity, we should not expect to be able to produce a method for designing model languages from scratch. Instead, we should look for *principles* and *criteria* that can guide us in making design decisions.

A language designed from scratch (or produced by unification; see the next section) is not a static entity. It will change — through extension or removal of features – as it used and is found wanting. All modeling languages evolve as the tasks to which they apply change; their origins are what will differ.

3.2 Unification

A modeling language that is unified from existing languages has two key goals to its design: it should standardize an existing set of features, so that users need know how to use only the standard version; and, it should be backwards compatible with its ancestors. Backwards compatibility need not require that valid models in the ancestral languages are valid in the new language; rather, it may simply mean that moving from the old to new language — where modelers must transition a syntactic and semantic gap between languages — is made as easy to accomplish as is reasonably possible.

Unification is one way to merge modeling languages that are independently evolving towards each other, or to merge divergent languages that have a common core of concepts, notations and features. The intent with unification is to help provide stability to the users of the modeling languages, while providing a standard notation for developers to apply. Unification is more likely to lead to a complex language with many features, where concepts are included in a language because one group of participants in the development process has used them and found them useful

in the past. Political necessity may also mean sacrificing the simplicity of the language design. However, unification is a simpler process to understand than designing a language from scratch: the process starts from existing concepts and syntaxes. It is easier to debate and discuss existing languages and their features than it is to argue about abstract concepts such as design principles.

Unification may ignore language design principles, in part because of the nature of its design process. However, we shall conject shortly that language design principles can be applied to a unified modeling language, after the unification process has been completed. That is, unification is only one step towards producing a useful and well-designed modeling language.

4 Principles for Modeling Language Design

The previous sections argued that the preemptive principle in the design of a modeling language should be simplicity. But implicit in this argument is that there are other principles that also should be considered. In this section we discuss key principles and the reason for their consideration.

4.1 Simplicity

We have already discussed simplicity. We have argued in detail that it should be the leading principle in designing a modeling language. Hoare describes that some programming language designers have replaced the goal of simplicity with the goal of *modularity*[2] [11], by which it is possible for programmers to apply the language by understanding only a subset of it. He argues that, in the domain of programming languages, this does not make sense because it is possible for programmers to accidentally invoke unknown features, and because it is more difficult to satisfy this goal than that of simplicity. We suggest that a similar argument also holds for the design of modeling languages: it is easier to aim for simplicity of modeling language than it is to aim for a language that satisfies the goal of modularity, in part because it is difficult to provide pragmatic advice on how to achieve modularity.

If a modeling language is simple, then it will be small and memorable, and it can be learned in its entirety by its users. The principle of modularity is useful, but is of questionable value in designing a modeling language.

Some examples of modeling languages that may be considered simple, small and memorable include BON, CSP, and predicative programming [10].

[2] Not to be confused with the very useful concept of modularity in the architectural description of a system.

4.2 Uniqueness

Uniqueness is sometimes referred to as *orthogonality*. It is discussed in [11] as well as in [18]. The principle is easy to express. A language that satisfies the principle of uniqueness provides one good way to express every concept of interest, and it avoids providing more than one.

The rationale for this principle is obvious. By avoiding duplication of features, the language is kept smaller and more explainable. There is no need to try to explain to users when to use one feature over another – potentially very similar, if not identical – feature.

Uniqueness does not imply minimization of features. A feature should be included in a modeling language if it is necessary for modeling a required concept and if there is no way of modeling it using current features. The intent with uniqueness is to have languages defined by a small number of *powerful* features that may be useful in more than one context. By keeping the number of features small, it is easier to understand the consequences of using the features together. The formal modeling language CSP, based on process algebras, satisfies the principle of uniqueness: each modeling construct in the language is not duplicated by any other, and makes it possible to model all systems of interest. UML, by comparison, does not satisfy the principle – it possesses both sequence and collaboration diagrams, which are semantically equivalent [24].

4.3 Consistency

The principle of consistency is described in [18]. It is simple to explain: consistency means that there is a purpose to the design of the language. All of the small number of powerful features that are included (or are to be added) to the language must further this purpose. Any feature that does not support the purpose must be discarded. An excellent example of consistency in modeling language design is BON: the purpose of BON is to support seamless and reversible development [36]. Any modeling concept that does not allow seamless or reversible development has been removed from the language. With UML, it is more difficult to determine whether the language is consistent or not, because there are no precise design goals beyond standardization of modeling concepts.

The B formal method was designed to make it easy to produce executable programs from formal models. At the same time, it was desired to provide automated assistance for this process. Z, in contrast, was designed to support formal modeling in a set-theoretic language. It is more difficult to produce programs from Z models than B models, in part because Z was not originally designed for such a task; it does not possess an executable subset to its specification language. Tools and methods that

support development of programs from Z models are, typically, more difficult to use than corresponding B techniques, in part because of this design characteristic

Consistency of language should not be confused with consistency of the *models* produced using the language; the latter is more of a reliability issue (see Section 4.8). Some modeling languages, e.g., UML, allow designers to describe a system in several independently constructed models – e.g., a class diagram, deployment diagram, use-case diagram, sequence diagram, et cetera – and at implementation time, these models must be checked for consistency, i.e., that something said in one model is not contradicted by something said in another model. Because the collection of models that can be produced using UML is large, and because each model itself may be complex (containing many different abstractions and relationships), checking the consistency of a UML specification is non-trivial, and it is questionable whether it can be automated. A contrasting approach is offered by BON: therein, a single model is constructed for each class [24], and checking the consistency of this model is straightforward and can be assisted by automated tools. Similarly, with the formal modeling languages Z and B, a single model of a software system is constructed, and tool-assisted reasoning about the model can be performed.

4.4 Seamlessness

The seamlessness principle contributes to being able to generate code from models, and also is a significant contribution towards producing maintainable software. Seamlessness allows the mapping of abstractions in the problem space to implementations in the solution space without changing notation, thus avoiding the impedence mismatches that often arise throughout the development process. Seamless software development in an object-oriented setting occurs by adding new classes, or by enriching already existing class from earlier phases with additional features [20]. In all stages of the software lifecycle, developers work with the same kind of abstraction, e.g., classes, processes, etc. At the end of development, a tool (typically a compiler) will have to render some executable code from the design.

Modeling languages for object-oriented development are well-suited to satisfy this principle. BON, for example, has as one of its primary design goals the support of seamlessness. BON supports seamless development with the Eiffel language by obeying the *single model principle* [24]: all information associated with a class in a system is contained in one place: the single model for the class. Different views of the model may automatically be generated. This model contains, amongst other pieces of information, an implementation of the class. Thus, seamlessness is guaranteed. UML, by comparison, is not seamless. In UML, object behavior can be specified using finite state machines, and thus further translation is necessary to express such details in an object-oriented programming language. A contrasting

language and method that supports seamlessness is B, where abstract machines are used throughout development until code is generated automatically by a specialised tool.

4.5 Reversibility

The principle of reversibility contributes to the production of maintainable software, and to producing better documentation for software systems. The principle of reversibility requires that changes made during one stage of the development lifecycle can be automatically reflected back to earlier stages. So, a modification made to an implementation class written, e.g., in Java, can be reflected in diagrammatic models, written in, e.g., UML. This captures the notion of feeding back to the design level pragmatic constraints from the implementation.

Reversibility, combined with seamlessness, allows programs and models to be kept in synch, and thus helps create and maintain system documentation. Changes made to models can be reflected in code; changes made to code can automatically be reflected in changed models. This is exactly what is required to in the maintenance process, as well as to abet future maintenance. It is unlikely that, without reversibility, models will be kept up to date with the code. The implication here is that without reversibility, it is the code that will be maintained, and the documentation provided by the model will thereafter be of minimal use.

A requirement for reversibility is that it be supported by tools. Typically, the principle is supported by CASE tools for a particular modeling language that also generates code for particular programming languages. Common combinations include UML and C++ or Java (e.g., Rational Rose [27]), OMT and C++, and BON and Eiffel (e.g., EiffelCase [20]). The primary focus of these tools and languages is to support production of architectural descriptions from programs; a mapping from programs to more abstract behavioural descriptions, e.g., state machines or predicates, may not in general be feasible.

In general, with formal modeling languages, it is difficult to support reversibility: this would require automatic production of formal specifications from programs. This is complex for a number of reasons: a program may implement many different formal specifications, or, the data structures used in a program may not correspond to data structures available in a specification language, for example.

4.6 Scalability

The principle of scalability states that a modeling language should ideally be useful for both small and large systems, i.e. not just ‘toy’ systems. It should be useful for

modeling systems with a few components and inter-relations, and systems with thousands of components and inter-relations.

To satisfy this principle, modeling languages must have certain characteristics. For one, they must provide a concise mechanism for describing the fundamental abstractions for their problem domain. In an OO setting, for example, this means that there must be a concise notation for describing classes and objects; typically, rectangles or ellipses are used here. The modeling language must also provide the means to hide details of abstractions. Again, in an OO setting, this means that, for example, class interface details (names of attributes, method signatures, invariants, et cetera) should be revealed at the discretion of the modeler. Finally, the language must also provide a *grouping mechanism* that allows the modeler to collect abstractions, name them, and hide their details. Such a mechanism is present in UML (where it is called a *package*) and BON (where it is called a *cluster*). Such a grouping mechanism can best be supported by a tool, akin to a hypertext browser.

Scalability has long been an issue with formal modeling languages; it has been claimed, in the past that formal methods do not 'scale up' to large problems. This is in part claimed because of a lack of structuring mechanisms in formal modeling languages, as well as limited tool support and an inability to hide details associated with abstractions. Work on the B method, which supports modular specification and powerful tool support, has aimed at addressing this issue. Recent work on method integration [22] and industrial-strength tool support [3] has also dealt with this problem.

4.7 Supportability

A modeling language is meant to be used, by humans, for writing or drawing models, as noted in Leveson's criteria [15]. Very often, this will be done on a whiteboard, or with pencil and paper; thus, models should be easy to produce by hand. But sometimes humans want to have software to help them produce models. The software should be expected to provide help in producing correct models (i.e., checking that the syntax of the diagrams is correct), in generating programs from models, and in producing models from code (i.e., reverse engineering).

For building large software systems using a modeling language, tool support is essential, not only for drawing and managing the models, but for maintaining them as development proceeds. Thus, the principle of supportability states that a modeling language should be designed to be implementable and supportable by software tools. This places restrictions on the notation syntax (i.e., it should also be easy to draw and display on a computer screen, it should be concise) and the semantics (i.e., it should be defined so that it can be automatically or semi-automatically translated into code, and it maybe should even support reverse engineering, although extract-

ing an appropriate abstraction will often be difficult and will require considerable human guidance).

With formal modeling languages, B has certainly been designed with tool support in mind. Work with the Z modeling language has aimed at providing tools, e.g., CADiZ [35], but it has been found to be more difficult to provide effective, industrial-strength tool support for Z, in part because of the language's design, as mentioned earlier.

4.8 Reliability

The goal of software development is to produce *quality* software. There have been many definitions of software quality proposed, but a common factor throughout many definitions is that quality software is *reliable*: it meets its specifications; and it reacts appropriately whenever it is given unexpected or erroneous input – that is, it is *robust*.

Methods for producing software must emphasize quality. Thus, modeling languages must support the production of reliable programs. They should provide support to ensure that the programs meet their specifications, e.g., via formal analysis or traceability combined with testing. They should provide support for developing software that reacts appropriately when given erroneous input, e.g., via design-by-contract mechanisms as supported in BON or UML, or by use of exception handling. And they should provide support for ensuring that the models being produced are consistent – i.e., are devoid of contradictions (as discussed in Section 4.3).

UML, with the addition of its Object Constraint Language (OCL) [37], attempts to support the production of reliable software via design by contract, although this has been criticised [24]. Part of the argument against the approach to design by contract offered by UML and OCL is that contracts and classes can be kept separate, potentially leading to inconsistencies between descriptions. BON, B, Z, and all other formal methods offer similar approaches to abet the production of reliable software (though the former is object-oriented, while the latter is state-based), supporting reliability by assertion-based techniques, run-time checking, and proof. The work of the Precise UML group, e.g., see [4], has focused on improving the UML for developing correct software, in part by giving parts of the modeling language a formal semantics.

4.9 Space economy

The final principle we mention is the simplest. The principle of space economy states that models should take up as little space on the printed page as possible. The

reason for this is obvious: smaller models have less to understand, and there is less work for modelers and tools to perform in order to maintain the models. Certainly, this principle can be taken too far, so space economy should not sacrifice simplicity nor the understandability of the language as well.

4.10 Summary of Principles

The principles we have suggested can be summarised thus:

Simplicity	No unnecessary complexity is included in the language.
Uniqueness	There are no redundant or overlapping features.
Consistency	Language features cooperate to meet language design goals.
Seamlessness	The same abstractions can be used throughout development.
Reversibility	Implementation changes can be propagated into the model.
Scalability	Large and small systems can be modeled.
Supportability	The language is usable by humans, and supportable by tools.
Reliability	The language encourages the production of reliable software.
Space economy	Concise models are produced.

5 Applying the Principles to UML

The Unified Modeling Language is rapidly becoming a standard modeling language, particularly for object-oriented systems. It is not without its limitations nor beyond criticism [30,9,19,24]. UML has been produced by a unification process, taking the best modeling concepts from OMT, Booch, and Objectory, producing a single syntax and meta-model. It is currently undergoing standardization by the Object Modeling Group.

As we have discussed throughout Section 4, UML, as it currently stands, does not satisfy many of the design principles that we have discussed. It is certainly not simple (the latest revision of its notation guide is over 160 pages), and for principles like seamlessness, consistency, and uniqueness, it falls short [24]. This is not surprising, as the language has been developed from several others. However, the fact that the UML does not satisfy the design principles may hinder its utility and effectiveness. That said, the principles can be applied to the UML in a process of *a posteriori* rationalization. The UML can be analyzed and evaluated to determine

how it can be adapted to satisfy the principles it fails to satisfy at the moment.

We make the following preliminary suggestions.

- *Seamlessness.* To support seamless development with UML requires the removal of *translation* in the development process. That is, the same concepts should be used in analysis, design, and implementation.

  Seamlessness is in part dependent on the programming language that is to be used for development. For example, if Java is the implementation language, then seamless development is not possible if multiple inheritance or deferred classes (where class features may have specifications, but not implementations) are used in design. BON, by comparison, works seamlessly with Eiffel (it was designed to), and requires no translation.

  A good step towards seamless development with UML would be to use finite state machines, which break seamlessness, only to provide a different view of an object's behaviour. This view should be automatically generated, e.g., from OCL constraints, akin to what is done with Graham's SOMATIK system [6].

  Another approach would be to produce specific dialects of the UML for specific programming languages, e.g., UML-Java, UML-C++, et cetera. These would likely be subsets of the full UML, perhaps with modifications to tune the syntax for behavioural description to fit most closely with the programming language syntax. This is possible in UML 1.3, which has an extension mechanism called *profiles*.
- *Uniqueness.* There is significant overlap among modeling techniques in UML; there are several ways to express concepts of interest. For example, sequence and collaboration diagrams are semantically equivalent [29]. There are also several ways to express constraints (e.g., as notes and separately using OCL constraints), and the concepts of abstract class and interface semantically coincide. Overlap such as this should be eliminated, or it should be made clear why it is necessary to have such overlap, as well as how to determine when to use particular versions of overlapping concepts. Alternatively, for a duplicated concept, one version could be chosen for standard use, and all other versions could be automatically generated different views of the concept.
- *Consistency.* This is the most difficult design principle to establish with a unified language like UML. Consistency aims at directing the design of a modeling language. The UML has already been designed, so what consistency can imply for the UML is that the UML may have to be redesigned – or restructured – to satisfy it. We suggest that a goal in the redesign of UML could be *rationalization*, that is, obtaining a simpler version of the modeling language where there is one way of modeling a particular concept. This would eliminate redundancy and duplication, and would make it easier for tool developers to ensure that they are complying with the language standard. Work on developing a 'core' UML is proceeding, and this may abet the satisfaction of this principle. However, we point out that developing the core UML is now primarily a political problem, and not primarily a problem in language design. An approach to design where a core

language is produced, and then extensions to the core developed, would make it easier to satisfy the principle of consistency.

- *Simplicity.* The UML is not simple. Its fundamentals can be learned in a few hours, but to understand all its concepts, their interrelations, and the underlying meta-model, requires a great deal of time and effort.

  Simplification of the UML can be accomplished, in significant part, by attempting to satisfy the previous three principles. By striving for consistency, uniqueness, and seamlessness in rationalizing the UML, a simpler language of necessity will be produced.
- *Reliability.* As it stands, the UML and its constraint language do not possess mature, rigorous semantics. Consider that individuals and groups (such as the Precise UML group) are attempting to strengthen the UML [4]. Without a rigorous semantics, we cannot consider the UML to be appropriate for the development of reliable software. Also, as suggested in [24], support for behavioural description via the OCL in terms of software contracts is weak, at best. Support for behavioural description is fundamental for producing quality software.

  Improvements to the UML with regards to satisfying this goal are suggested in [14,24]. It is a challenging problem to improve the UML along these lines.

6 Conclusions

Modeling languages, which are used in the software development process, must be designed. In this sense, modeling languages should be considered no different than programming languages. Techniques, criteria, and principles for the design of modeling languages should be produced and given the accord that they are for the design of programming languages. These principles should ultimately be validated by experiment, showing that modeling languages that are well-designed help in the production of better-quality software.

A great deal of effort has been spent on studying and producing principles and criteria for the design of programming languages. Programming languages, and their related tools, are just one component of the software development process. Modeling languages and their tools are another, and they should be designed and developed with the same care as programming languages. Many of the principles that have been developed over the years for the design of programming languages, like simplicity, consistency, and uniqueness, are equally applicable to modeling languages. Because of the nature of modeling languages — typically being used earlier in the software lifecycle than programming languages, and typically being diagram-based — there are also design principles that are not obviously applicable to programming languages, like seamlessness and reversibility.

Modeling languages *should* be designed to satisfy the necessary principles from the start. For some problem domains, or for some situations, some principles may

need to be discarded or deemphasized; thus, what is an essential principle to follow in the design of one language may be unessential in another. This is exactly the case as for programming languages. It may be cheaper to re-design and rationalize modeling languages than programming languages, after their development. In part, this is possible because, unlike programming languages, modeling languages do not have to be (but may be) executable. They are first and foremost a language for the whiteboard, the piece of paper, and the CASE tool.

The danger with *a posteriori* changes to a modeling language is that the modifications may invalidate existing tools. This is a not insignificant concern, especially considering the cost of modern CASE tools. Careful tool design, and use of meta-CASE techniques [17], can help reduce the cost of changes in modeling language. But if these changes are those of rationalization, and involve removal of duplicated or redundant concepts, then the likelihood of invalidating existing tools is reduced. However, it does suggest that it is best to design the language to accomplish the necessary tasks from the start.

References

[1] Abrial, J.-R. *The B-Book* (Cambridge University Press, 1996).

[2] Brooks, F. *The Mythical Man Month* (Addison-Wesley, 1995).

[3] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. *Proc. WIFT'95*, IEEE Press, 1995.

[4] Evans, A., France, R., Lano, K., and Rumpe, B. The UML as a Formal Modeling Notation. *Computer Standards and Interfaces,* 19(7), 1998.

[5] Fowler, M. *Analysis Patterns* (Addison-Wesley, 1996).

[6] Graham, I. *Requirements Engineering and Rapid Development* (Addison-Wesley, 1998).

[7] Green, T.R., and Petre, M. When visual programs are harder to read than textual programs. In van der Veer, G., Tauber, M., Bagnarola, S., and Antavolits, M., editors, *Human-Computer Interaction: Tasks and Organisation. Proceedings of ECCE6 (6th European Conference on Cognitive Ergonomics)* (CUD, 1992).

[8] Green, T.R., and Navarro, R. Programming plans, imagery and visual programming. In *Proceedings of INTERACT '95* (1995).

[9] Hamie, A., Civello, F., Howse, J., Kent, S., and Mitchell, R. Reflections on the Object Constraint Language. In *Proc. UML'98* (Springer-Verlag, 1998).

[10] Hehner, E.C.R. *A Practical Theory of Programming* (Springer-Verlag, 1993).

[11] Hoare, C.A.R. Hints on Programming Language Design. In *Proc. ACM Principles of Programming Languages 1973* (ACM Press, 1973).

[12] Hoare, C.A.R. *Communicating Sequential Processes*. (Prentice-Hall International UK, 1985).

[13] Jackson, M. *Software Requirements and Specifications* (Addison-Wesley, 1995).

[14] Kent, S., and Howse, J. Mixing visual and textual constraint Languages. *Proc. UML'99*, LNCS 1723 (Springer-Verlag, 1999).

[15] Leveson, N., Heimdahl, M.P., Hildreth, H., and Reese, J.D. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[16] Leveson, N., Heimdahl, M.P., and Reese, J.D. Designing specification languages for process-control systems. In *Proc. FOSE'99*, ACM Press, September 1999.

[17] MetaPHOR Project Group, MetaPHOR: Metamodeling, Principles, Hypertext, Objects and Repositories. Technical Report TR-7 (University of Jyvaskyla, 1994).

[18] Meyer, B. *Eiffel – the Language* (Prentice-Hall, 1992).

[19] Meyer, B. UML: The Positive Spin. *American Programmer*, March 1997.

[20] Meyer, B. *Object-Oriented Software Construction*, Second Edition (Prentice-Hall, 1997).

[21] Milner, R. *Communication and Concurrency* (Prentice Hall, 1989).

[22] Paige, R. A meta-method for formal method integration. *Proc. Formal Methods Europe 1997*, LNCS 1313, Springer-Verlag, 1997.

[23] Paige, R. When are methods complementary? *Info. Soft. Tech.* 41(3), February 1999.

[24] Paige, R., and Ostroff, J. A comparison of the Business Object Notation and the Unified Modeling Language. *Proc. Second International Conference on the Unified Modeling Language (UML'99)*, LNCS 1723 (Springer-Verlag, 1999).

[25] Petre, M., Blackwell, A., and Green, T. Cognitive questions in software visualisation. In Stasko, J., Domingue, J., Price, B., and Brown, M., editors, *Software Visualisation: Programming as a Multi-Media Experience*. (MIT Press, 1997.)

[26] Petre, M. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, June 1995.

[27] Quatrani, T. *Visual Modeling with Rational Rose 2000 and UML* (Addison-Wesley, 1999).

[28] Robinson, P.J. *Hierarchical Object-Oriented Design* (Prentice-Hall, 1992).

[29] Rumbaugh, J., Jacobson, I., and Booch, G. *The Unified Modeling Language Reference Manual* (Addison-Wesley, 1999).

[30] Simons, A., and Graham, I. 37 Things that Don't Work in Object-Oriented Modeling with UML. In *Proc. ECOOP'98 Workshop on Precise Behavioral Semantics* (TU-Munich Report 19813, 1998).

[31] Simpson, H.R., and Jackson, K. Process Synchronization in MASCOT. *Computer Journal* 22(4), 1979.

[32] Spivey, J.M. *Z Reference Manual* (Prentice-Hall, 1989).

[33] Steele, G. Growing a Language. Invited talk at *OOPSLA'98*.

[34] Stroustrup, B. *The Design and Evolution of C++* (Addison-Wesley, 1994).

[35] Toyn, I. and McDermid, J. CADiZ: An Architecture for Z Tools and its Implementation. *Software – Practice and Experience*, 25(3):305-330, March 1995.

[36] Walden, K., and Nerson, J.-M., *Seamless Object-Oriented Software Architecture* (Prentice-Hall, 1995).

[37] Warmer, K., and Kleppe, A. *The Object Constraint Language* (Addison-Wesley, 1999).

[38] Wirth, N. On the design of programming languages. In *IFIP World Congress 1974* (North-Holland, 1974).