The Single Model Principle

Jonathan S. Ostroff and Richard Paige

Technical Report CS-2001-06

December 19, 2001

Department of Computer Science

4700 Keele Street North York, Ontario M3J 1P3 Canada

The Single Model Principle

Jonathan Ostroff and Richard Paige [*]
Department of Computer Science, York University,
4700 Keele St., Toronto, ON M3J 1P3, Canada.

December 19, 2001

Abstract

Modelling languages such as UML are increasingly used to describe software systems at different levels of abstraction. There are two very different ways of using such languages. One approach is based on the manifestation of a *single model*, with construction of different views from this model, and with automatic or semi-automatic consistency checking among these views. This follows what we term the *single model principle.* The second approach (of which the use of unrestricted UML is an example) is based on the independent construction of multiple models of the same system, but with no guarantee of the consistency of the various models. We propose that to best support seamless, reversible software development of reliable software, it is preferable to follow the single model principle for a specific subset of development tasks.

We describe the single model principle and its supporting infrastructure. We show how the BON/Eiffel description language, which supports both high-level abstract specifications as well as code implementations, can be enhanced to satisfy the essential tenets of the single model principle, both for static and dynamic descriptions. We describe how a UML profile (including the use of Java) might be expanded to provide weak support for the principle. We also consider situations and tasks when following the principle is insufficient, particularly when capturing early (goal-oriented) requirements.

Keywords: modelling languages; UML; BON; view consistency; object-orientation

[*]Email: {jonathan, paige}@cs.yorku.ca. Supported by a grant from NSERC.

Contents

List of Figures

List of Tables

1 Multiple and Single Models

UML is a major step towards the development of a standard language for the visual specification and design of object-oriented systems. It supports the modelling of systems (and not just software) using object-oriented concepts, and addresses the issues of scale inherent in complex, mission-critical systems. The modelling language is usable by both humans and machines. Such a standard is useful and is used extensively in this article.

However, UML is also open to significant criticism. The latest version of the language specification is about 500 pages long, with the notation guide alone taking 175 pages. UML is complex, and allows many independent views of a system to be constructed. The UML 1.3 Standard [14] states:

> "Every complex system is best approached through a small set of *nearly independent*[1] views of a model; no single view is sufficient. In terms of the views of a model, the UML defines the following graphical diagrams:
>
> - use-case diagram
> - class diagram
> - behaviour diagrams [...]
> - implementation diagrams [...]
>
> These diagrams provide multiple perspectives of the system under analysis or development. The underlying model integrates these perspectives so that a *self-consistent* system can be analyzed and built. These diagrams, along with supporting documentation, are the primary artifacts that a modeler sees, although the UML and support tools will provide for a number of derivative views."

The phrase "self-consistent" in the above quote is problematic. UML allows for multiple *views* of the system. This alone is not problematic; indeed, experienced practitioners know that no single view of the system will suffice. What is problematic is the inference that these views will be consistent with each other. In fact no such guarantee exists with UML, and very little guidance has been provided for how one would achieve such consistency.

Since there is no guarantee of consistency, what we really have are *multiple models* of the system. We call this the *multiple model principle*, which is to be distinguished from the single model principle advocated in the sequel.

As a simple-minded example illustrating problems with the multiple model approach, consider a UML statechart with a guard p on a transition describing the change in the state of an object. Elsewhere in the collection of documents describing the system, the same change might be described using an Object Constraint Language (OCL) precondition $\neg p$. There is no way of detecting inconsistencies of this kind, because there is no formal link between statecharts and OCL present in the semantics of UML.

By using multiple UML models, developers can work independently on separate parts of a system, and can apply the most appropriate diagrams or notations for describing each part. Some potential inconsistencies, e.g., that classes generalize themselves, or that packages have unique names, will be caught by the UML metamodel. But, when it comes time to construct

[1]Italics are not in the original

executable code from the models, the models must be integrated into a single description that satisfies all the constraints and descriptions contained in the individual constructs. Since there is nothing in the UML, in its metamodel, or in its supporting processes to prevent many of these inconsistencies from arising, executable code that implements these models consistently cannot be easily generated. This of course raises the substantial concern that the actual software artefact delivered does not satisfy its UML specification.

By contrast, the single model principle (defined precisely in the sequel) also supports multiple views of the system. However, it aims to ensure consistency of views either by construction or by rigorous analysis. No current method, with the expressive power of UML, can claim to guarantee consistency. In this article, we hope to show how, and to what extent, the goal might be achieved.

2 Software Description Languages

The Unified Modeling Language (UML) is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system. The UML, is not intended to be a visual programming language, in the sense of having all the necessary visual and semantic support to replace programming languages, although it does have a tight mapping to a family of OO languages such as C++ and Java.

The single model principle addresses the whole scope of software development, including implementation. In order to explore the single model principle in full, we will need to choose a programming language (e.g. Java) to complement the UML. We can then contrast the support that UML/Java and BON/Eiffel provide with respect to the single model principle.

2.1 BON/Eiffel

Eiffel is an object-oriented programming language and method [10]; it provides constructs typical of the object-oriented paradigm, including classes, objects, inheritance and client-supplier relationships, generic types, polymorphism and dynamic binding, and automatic memory management.

However, Eiffel is not just a programming language — the notation also includes the notion of a *contract* to specify the duties of clients and suppliers. A valid Eiffel program may consist only of specifications – that is, it may possess no program code whatsoever – or it may be a combination of specification and code or code only. Eiffel is thus a true wide-spectrum language in the style of [6, 12]. An Eiffel program may therefore also be called a "model" or a "description" of a system, as it is more than just code.

The assertion language for Eiffel used in this paper includes first-order predicate logic. Eiffel as a compilable language has a more restricted assertion language, though the agent mechanism [11] in recent versions of Eiffel makes it possible to achieve nearly the same level of expression. The Eiffel Refinement Calculus [17] shows how class implementations can be refined from contractual specifications. The Eiffel language is thus applicable to modelling concepts and constructs from late requirements engineering (when customer goals are well-defined, and alternatives in goals have been considered and selected) through to implementation.

The BON modelling language [22] complements Eiffel with a set of concepts and corresponding graphical notations to support object-oriented modeling centered around the three principles of seamlessness, reversibility and contracting. Although BON is a language independent method[2], its basic concepts are close enough to Eiffel that its graphical notation may be viewed as a graphical dialect of Eiffel for the purpose of this article. In this article, we will use the terms "Eiffel" and "BON" interchangeably.

The analysis and design of object-oriented systems in BON will result in static and dynamic descriptions of the system under development, as shown in Fig. 1. A static class diagram tells us how the program modules (i.e. the classes) are organized. A dynamic diagrams, by contrast, documents how the system will behave over time. In an object-oriented system, this means describing how objects interact at execution time (i.e. how they pass messages to each other). Each object will behave as prescribed by its blueprint, i.e. its corresponding class. BON's dynamic diagrams are similar to UML collaboration or interaction sequence diagrams.

A static description includes the various classes (e.g *CITIZEN* and *HOUSE* in Fig. 1), as well as the formal description of the class interface, as shown for *CITIZEN* in the figure. A dynamic description specifies system execution scenarios which specifies the events or messages that are passed in the order that they occur.

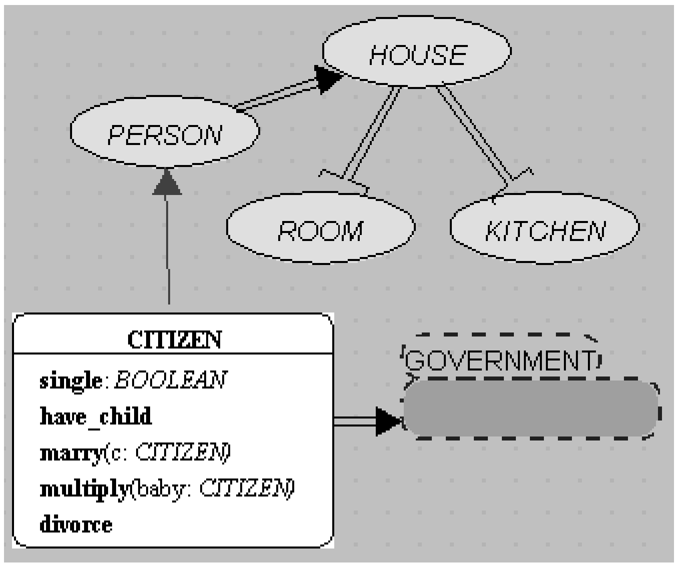
The *CITIZEN* class has three attributes (*spouse*, *parents* and *children*), a boolean-valued function *single*, and a state-changing procedures such as *marry* and *divorce*. A **require** clauses describes a precondition, and an **ensure** clause the postcondition. Postconditions can refer to the value of an expression when the feature was called by prefixing the expression with the keyword **old**. Classes may also have *invariants*, which are predicates that must be maintained by all visible routines. Visibility of features is expressed by annotating **feature** clauses with lists of client classes permitted to access the features.

Eiffel supports inheritance relationships (e.g., between *PERSON* and *CITIZEN* above), association relationships (e.g., between *CITIZEN* and itself, via attribute *spouse*, and between *PERSON* and *HOUSE*) and aggregation relationships (e.g. between *HOUSE* and *KITCHEN*). A *cluster* shows a set of classes, and possibly other clusters. The cluster *GOVERNMENT* in the figure is compressed, i.e. the details are hidden from view. Visual descriptions of relationships using BON can be found in [22].

Eiffel compilers (such as Eiffelstudio) can automatically reverse engineer class diagrams and the class interfaces from the implementation code. This "reversibility" property is important for keeping implementations consistent with their specifications. Also, given a class interface as shown in the class diagram, it is a simple matter to forward engineer the class interface, to which the programmer can easily add implementation code. Very little research has been done on integrating dynamic diagrams into the seamless and reversible engineering process. This integration will therefore be developed further in Section 4.

[2]It has been used successfully over the years at Enea in Sweden in industrial projects with such diverse languages as C++, SmallTalk and Object Pascal

```
class CITIZEN inherit PERSON
feature {ANY}
    spouse : CITIZEN
    children, parents : SET[CITIZEN]
    single : BOOLEAN
        ensure Result = (spouse = Void)
feature {BIG_GOVERNMENT}
    marry ···
    have_child ···
    divorce is
        require ¬ single
        ensure single ∧ (old spouse).single
invariant
    single ∨ spouse.spouse = Current;
    parents.count ≤ 2;
    ∀ c ∈ children • ∃ p ∈ c.parents • p = Current
end
```



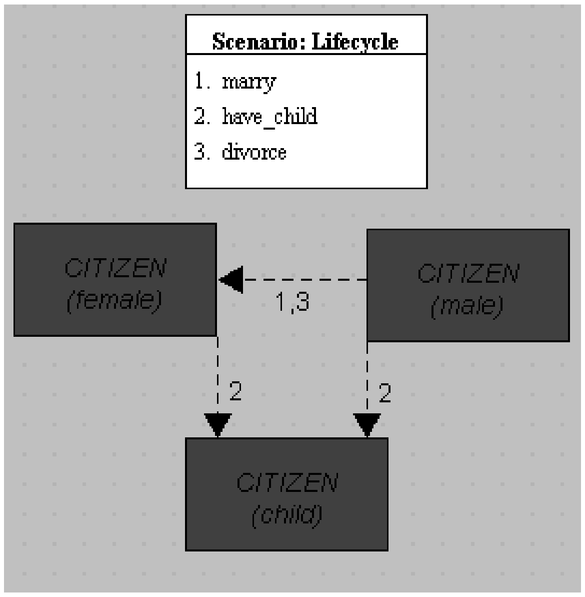


Figure 1: BON/Eiffel – Top: interface. Left: Static Class Diagram. Right: Dynamic Diagram

3 Definition of the Single Model Principle

We have informally explained the single model principle, so far, in terms of consistency of views of a system. In this section we list further factors and criteria that will allow us to provide a more precise definition of the single model principle.

We must first decide on a minimum set of essential views. Without knowing what these different views are, we cannot describe any relationships, such as consistency, between them.

A software system (or subsystem) is usually conceived of as consisting of an assembly of modules (or classes) in some relationship with each other. There is therefore the module level and the system level (for simplicity, in the sequel, the system level will also include subsystems).

The notion of a *package* is used in UML for the system level descriptions. This raises a frequently debated issue — the need to have super-modules above the level of a class [10, p209]. The package is one such notion of a super-module, and has its own rules for hiding and exporting information as the rules for packages are different than the rules for classes. To re-use a class in a package, the whole package must be imported. By contrast, Eiffel has the notion of selective export (e.g. in Fig. 1, the only class that can invoke marriage and divorce is *BIG_GOVERNMENT*), and thus does not need the additional layer of a super-module. The notion of a *cluster* is used in BON to group classes and/or sub-clusters selected by the designer according to some criteria to form a conceptual unit or subsystem. The same set of classes can, in other views, be clustered differently. The cluster therefore does not have any semantic effect, and is not part of the class syntax.

3.1 Views at the module level

At the module level we need at least two views:

1. the *implementation* view, i.e. the actual executable code that describes how the the module performs its intended function, and

2. the higher-level *specification* that describes what the module function is.

To describe the specification and implementation views in accordance with the single model principle, we need a *seamless and reversible wide-spectrum* modelling language. The language must be wide-spectrum if it is to encompass both specifications and implementations. It must be seamless if the implementation is to be developed smoothly from the specification. It must be reversible if the specification is to be obtained automatically from the implemented source code.

Most programming languages are not wide-spectrum, as they do not support the full power of contracts (e.g. C++, Ada and Java), unless they are enhanced with add-ons. For example, Ada supports a two-tiered notion of a module (called a "package") involving a "specification" and a "body". But the name "specification" is too strong for a construct that supplies only typing information (the signature in ADT terms) but not the behaviour (the ADT axioms). The modest word "interface" is more applicable [10, p1081].

As mentioned earlier, Java must be enhanced with external add-ons such as *iContract* or JML [8] (and OCL at the UML level) to achieve the intended effect. The combined

UML/Java notation is wide-spectrum, but not seamless. This is because there is a certain amount of impedance mismatch between OCL and the Java assertion languages (such as *iContract*) as they are different logical languages, and hence some conversion would be required. Furthermore, a UML class can be constrained not only via OCL, but also via constraints in class diagrams and statecharts. Checking such a collection of varying constraints and converting them to a Java assertion language is not seamless. For the same reason, it would be difficult to reverse engineer the OCL contracts from the Java code and assertions.

In BON/Eiffel, specifications are predicate logic contracts (pre/post conditions and class invariants). Seamlessness and reversibility are there by construction. The high-level specification language is a superset of Eiffel's own expression syntax, and is checked at compile time for type correctness. Once a query is defined, it becomes part of the assertion language, thus enriching the constructive expressivity of assertions. There are no special mathematical operators as one would find in languages such as Z [21]. Since the specification is part of the code, reversibility is automatically present, and the contract view can be extracted automatically.

A rudimentary check of consistency between the specification and implementation views can be made because: (a) the specifications are typechecked against the implementation by the compiler, and (b) the specifications (contracts) can be checked against the implementation for violations at run-time by turning assertion checking on. More refined consistency methods are available such as the Eiffel refinement calculus [17] which is in the spirit of the wide-spectrum methods of Hehner and Morgan [6, 12].

Thus, one approach to making multiple consistent views of a single product, is to make the concepts and notations needed to write code abstract enough so that they can serve just as well as tools for *modelling*. As Meyer's *self-documentation principle* states [10, p55]:

> "...software becomes a single product that supports multiple views. One view, suitable for compilation and execution, is the full source code. Another is the abstract interface documentation of each module, enabling software developers to write client modules without having to learn the module's own internals... Other views are possible."

The fully implemented code containing well-designed preconditions, postconditions, invariants and the careful choice of names for both classes and features, so that various views can be extracted from this single product at various levels of abstraction, is therefore, by construction, the simplest kind of consistent single model at the module level.

The fact that all the details of a class *CITIZEN* are contained in a single file (e.g. *citizen.e*) gives the module *physical integrity*. In C++, it is possible to avoid physical integrity by defining the interfaces of classes in files separate from the class definitions (i.e., through use of `.h` and `.cc` files). This places the burden either on the programmer or the compiler to keep track of the different parts and to ensure that they are consistent.

3.2 Views at the system level

At the system level, there are also at least two views that must be supported. These views allow the designer to document the system architecture, i.e. the major components of the system and the structural and dynamic relationships between them.

The two views in the UML idiom are (a) class diagrams (i.e. the static structure of the system), and (b) behavioural diagrams (e.g. statecharts, activity diagrams, and sequence or collaboration diagrams) that describe the system dynamics. David Harel writes:

> As to the UML itself, one must remember that right now UML is a little *too* massive. We understand well only parts of it; the definition of other parts has yet to be carried out in sufficient depth to make crystal clear their relationships with the constructive core of UML (the class diagrams and statecharts) ... my personal feeling is that in the wake of the initial excitement about a standard for modeling software the UML will have to be made smaller and tighter. Otherwise, it will become too cumbersome and multifaceted to be really useful. I think it will gradually shrink, leaving only three or four types of diagrams that are really needed and are useful. The rest will probably become obsolete and will eventually disappear [3, page xx].

What we have here is essentially a statement of the need for *conceptual integrity*. Use a small number of powerful descriptions that work together to help describe the software product; provide one good way to describe every construct of interest. This will prevent the documentation from becoming unmaneagable, keep it relatively easy to read, and will allow one to easily teach and communicate the ideas to designers and programmers.[3]

The Java language possesses a variety of syntactic constructs (classes, interfaces[4], and primitives) all of which describe the same semantic concept. As shown by Eiffel and C++, there is no need for separate syntactic constructs. UML uses classes, interfaces, datatypes, nodes, and components as classifiers, and further constraints (e.g., via stereotypes, in OCL, via multiplicity, etc.) can be associated with classifiers to form new ones. Similarly sequence and collaboration express the same details, but in different ways. A critical question to ask about the design of a language is: why have many different ways of expressing the same concepts? Conceptual integrity would dictate the choice of one of the equivalent views, and find techniques to automatically check consistency of equivalent views (as in the SOMATIK tool [4]) where these additional views are felt to be necessary.

By contrast, Eiffel/BON has the kind of conceptual integrity described above, that is to say a small number of powerful orthogonal descriptions that work together to describe the software product. At the module level, contracts can be used to express the various constraints that in UML are performed by a variety of constructs (OCL, statecharts, multiplicities and other classifiers). At the system level, there are only two kinds of diagrams: static class diagrams and dynamic diagrams that are closely linked with each other. The class diagrams show the the static structure of arbitrarily large systems (using recursive clustering and compression), and can be forward or reverse engineered from code (Section 2.1). Dynamic diagrams show system behaviour over time. Dynamic diagrams describe interactions of only those objects that are instances of classes that appear in the static class diagram. Messages sent betweeen two objects can then be related to calls to features described in the

[3]For example, in language design, Meyer [9, p498] explains why multiple variants of the loop construct (e.g. test at the beginning, test at the end, provide a "for" loop for automatic transition to the next element etc.) are not needed when a single construct will be easier to remember, program and get right. Loops are hard enough as it is.

[4]Interfaces are included in Java to eliminate the apparent complications that arise with multiple inheritance.

class interface. Some current research is aimed at using this close link between static and dynamic diagrams to check consistency [?].

3.3 Definition

Based on the preceding discussion, we can define the *single model principle* as requiring the use of a seamless and reversible wide-spectrum language for sofware description, possessing conceptual integrity at both the module and system levels, while maintaining view consistency at different levels of abstraction.

Criterion	**UML/Java**	**BON/Eiffel**
Seamless and reversible wide-spectum descriptions	No (e.g. impedance mismatch between OCL and *iContract*, or between statecharts and classes)	Yes (by construction)
Conceptual integrity	No (e.g. constraints can be expressed on dependency arrows, in notes, via OCL and in statecharts; collaboration and seqence diagrams are similar semantically)	Yes (by construction)
View consistency	No (in general, no algorithms or methods available to check the constructive part – classes and statecharts — against the other views, e.g. OCL)	Qualified Yes (static views have good consistency checking mehods; basic checks can be made for the consistency of dynamic and static diagrams, but better methods needed)

Table 1: Single model principle — comparison

Table 1 summarizes the discussion so far by comparing UML/Java and BON/Eiffel with respect to the single model principle.

The *typical* use of UML/Java does not obey the single model principle. Developers use UML to produce multiple independent models, while Java is used to constructively describe a single executable model of the system under description, but there are no algorithms or methods to guaranteee the required consistency. However, as will be shown in the sequel, there is nothing to prevent us from taking a core subset of UML and using that as a platform to satisfy the single model principle.

4 Deliverable dependencies

A simple way to further illustrate the single model principle is by considering development deliverables and their dependencies. We present the deliverables using UML's package and dependency notation, in part because it is a standard. The diagrams aim to depict *typical use* of UML/Java and BON/Eiffel. In these diagrams, dashed arrows represent *dependencies.* A dependency relationship exists between two elements where a change to the supplier element (the target) may potentially affect the client element (the source). The dependencies are annotated with stereotypes. Three stereotypes are used.

- ⟪ **refine** ⟫**:** where the client element is a refinement (i.e., a more detailed description) of the supplier element. There is no guarantee that the refinement is consistent with the supplier. There is also no guarantee that the refinement can be done automatically, and developer intervention will usually be needed to establish the relationship.
- ⟪ **consistent_refine** ⟫**:** This is refinement in which the client is guaranteed to be consistent with the supplier in the sense of [6, 17], wherein any scenario satsifying the client also satisfies the supplier, with the client adding additional constructive information. Thus, for example, source code is considered a refinement of a class diagram. This relationship, in general, is only partially automatible, and will require developer intervention to establish. Seamlessness will of course aid the designer to do the refinement more easily.
- ⟪ **derive** ⟫**:** where the client element can be derived automatically and consistently from the supplier element.

The **derive** relationship is obviously the most desirable, as a tool can do this automatically for the developer. An example of this is the automatic generation of BON class diagrams and specifications from Eiffel code (using the self-documentation principle) as depicted in Fig. 2.

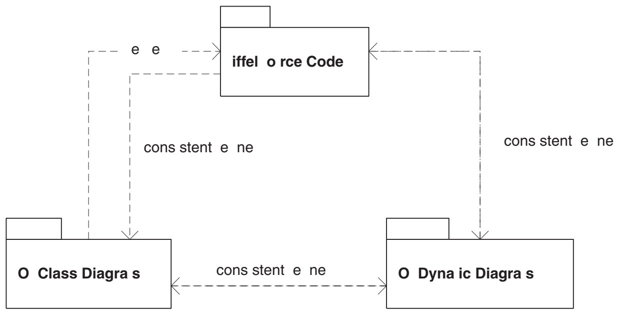


Figure 2: Eiffel deliverable dependencies

The minimum required for the single model principle is the **consistent_refine** relationship, which ensures that the two views are consistent with each other. Thus, for example, Eiffel source code can be refined from the contracts inherent in BON class diagrams, with a guarantee that the code implements the contracts [17].

So, too, the consistency of dynamic diagrams can be checked both against source code and the class diagrams by treating messages as feature calls [**?**]. For example, to check a dynamic diagram against a class diagram, we must first ensure that both diagrams are syntactically and semantically (obey typing and scope rules) correct. Then we check that each object in the dynamic diagram is an instance of a class in the corresponding class diagram. Finally, we must check that that the message being fired is allowed given the pre/postconditions of the corresponding feature calls in the class. These checks are stated and proved using the PVS theorem prover, but other methods are possible.

Thus all BON/Eiffel deliverables – i.e., class diagrams, source code, dynamic diagrams – are related and are dependent. They are either related through automatic construction (e.g., automatic generation of class diagrams from source code, or dynamic diagrams from source), or by algorithmic consistency checking and developer modification. Hence they meet, at the very least the consistency requirement, and in some cases the stronger **derive** criterion.

By contrast, the situation with UML/Java (depicted in Fig. 3) is a very different situation.

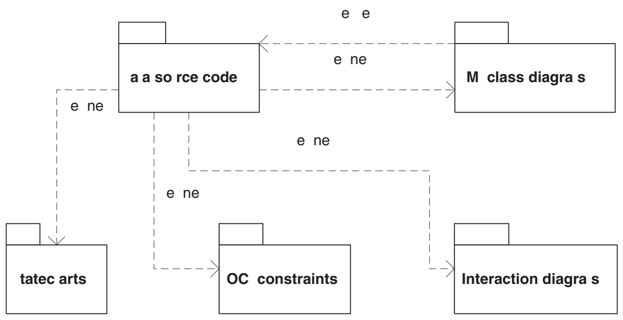


Figure 3: Deliverable dependencies using UML/Java

With UML/Java, deliverables (e.g., class diagrams, state machines, OCL constraints, collaboration diagrams, etc.) can and usually are constructed independently and thus can introduce overlap and inconsistencies. For example, consider a UML model consisting of a class diagram (where methods have pre- and postconditions written using OCL) and a state transition diagram. For consistency, preconditions must correspond to guarded transitions in the state machine, but nothing in the modelling language enforces this, nor does the language provide necessary theory, methods, or tools to check or enforce consistency. It is thus left to the modeller to ensure or enforce consistency in their descriptions.

The single model approach, which is taken by Eiffel, is more fundamental than the multiple model approach, exemplified by UML/Java. Consider the problem of checking the consistency of independently constructed multiple models. To do this, the models must be combined in a common framework and reasoning must be carried out. But this is the process of constructing a single model — containing all descriptions regarding a software system of interest. Now, the construction problem and the consistency checking problem is more difficult than had development started with a single model in the first place. To construct a single model, we must combine information and constraints from several possibly very large, complex separate models. In contrast, with the single model approach, a large model is constructed piece by piece, by adding new descriptions (e.g., method signatures, code for methods, contracts) over the course of the entire development, and different views are produced from the model automatically or semi-automatically.

5 The single model principle and UML

Suppose that it is desired to follow the single model principle while using UML and Java. Is it possible to do so? In [19], we provide a profile (i.e. a subset of UML and Java) that can be used to this purpose. This profile is targetted at supporting Extreme Programming principles and guidelines [1] while using UML; it is not intended directly to support the single model principle. However, with some further, minor, restrictions, the profile can be used to support the single model principle as well. An overview of the deliverables of the proposed method and their relationships is shown in Fig. 4.

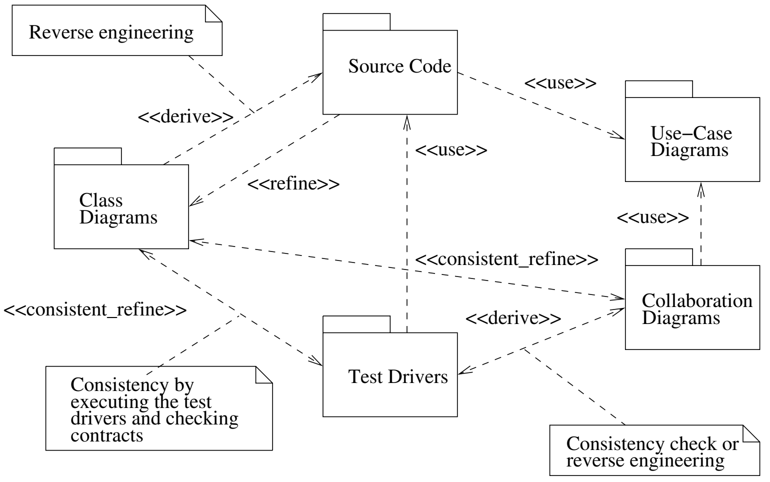


Figure 4: Proposed deliverables following the single model principle

The key deliverables with this profile are class diagrams (restricted to support only a subset of the typical constructs allowed in UML), collaboration diagrams, test drivers, and Java source code. Use-case diagrams may also be used, but they are considered as rough sketches or informal documentation, and their consistency with other development products is not guaranteed. Two types of dependencies exist between deliverables: automatic construction dependencies (where a tool generates one deliverable from another and guarantees semantic consistency); and algorithmic consistency checking dependencies (where an algorithm is used to detect all inconsistencies between two deliverables, and a report is thereafter generated for the developers).

Contracts are used in both class diagrams and in Java source code (via something like *iContract*). The profile emphasises testing, due to the inclusion of test drivers as a deliverable. Test drivers are small Java programs that execute the system on a suite of test data, and inform the testers as to the results of testing.

Collaboration diagrams are viewed as an abstraction of test drivers; thus, they can either be reverse engineered automatically from the test drivers (for a description of a suitable algorithm for this, see [**?**]) or their consistency can be checked against the test driver. A tool is required to implement the reverse engineering and the algorithmic consistency checking, as discussed further in [**?**].

Collaboration diagrams are used only in the development of test drivers, not in capturing behaviour of methods — thus, there is no direct relationship between collaboration diagrams and source code or class diagrams. However transitivity between colloaboration diagrams and test drivers, and test drivers and code can be used to obtain the required consistency check.

Such a profile, with some minor restrictions, can be used in the spirit of the single model principle. If the test driver runs succesfully, then a basic kind of consistency test is passed. Conceptual integrity arises due to restrictions placed on the use of UML and Java constructs (e.g., to satisfy the construct uniqueness by using only classes for modelling, and avoiding multiple inheritance), and through supporting tools, which provide a repository for all information related to abstractions that appear in the model. Consistency of views is established via algorithms and reverse engineering facilities, as well as the successful execution of the test drivers. Wide-spectrum applicability is obtained through use of contracts (specifications) in both UML diagrams and Java source code.

6 The single model principle and metamodelling

Both BON and UML are modelling languages, possessing metamodels that specify the well-formedness constraints that all models must obey. By examining the metamodels for each language, we gain further insight as to why BON supports the single model principle and why UML does not without the restrictions of the previous section.

Fig. 5 depicts a fragment of the BON metamodel using the UML notation as a metamodelling notation. The diagram shows that in BON, a model consists of a set of abstractions. An abstraction may be a class, a cluster, an object, or an object cluster. These abstractions may have relationships with other abstractions. Each metaclass in Fig. 5 has constraints written as clauses in class invariants (the invariants are fully described in [18]).

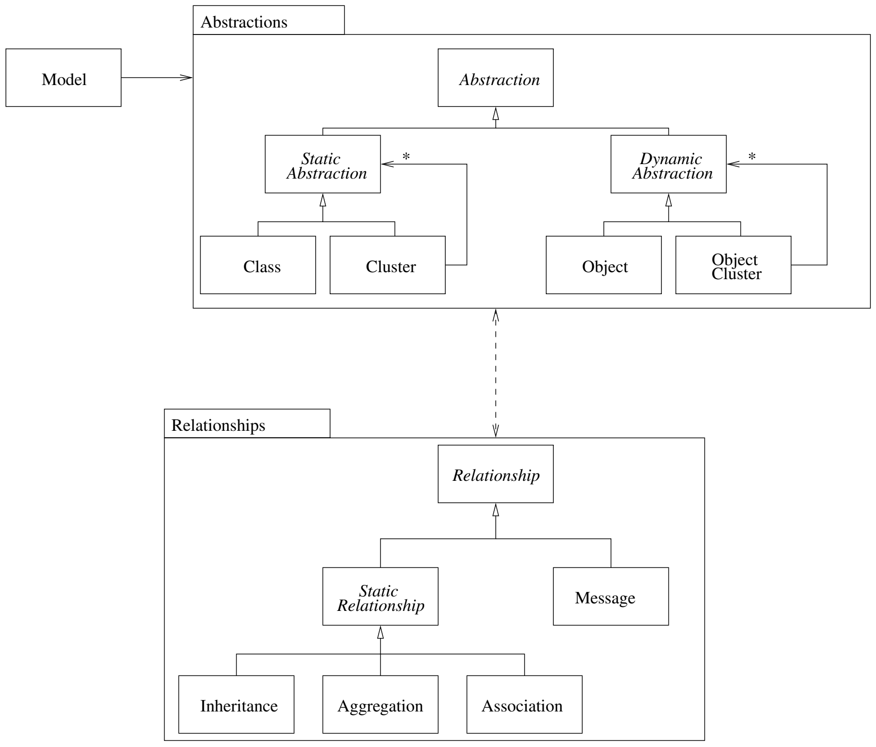


Figure 5: A fragment of the Eiffel metamodel

The diagram clarifies why Eiffel supports the single model principle. All views are related static or dynamic abstractions which can be constrained by well-formedness constraints (defined in **Relationships**). For example, these could be constraints on messages to ensure that each message corresponds to a feature provided by a class or to ensure that each dynamic object is an instance of a corresponding class in the static class diagram [19].

Consider now Fig. 6, presenting a fragment of the UML metamodel, extracted from the complete metamodel in [15]. The fundamental concept in a UML model is a **Model element**, which is subclassed by concepts such as **Classifier**s and **Interface**s.

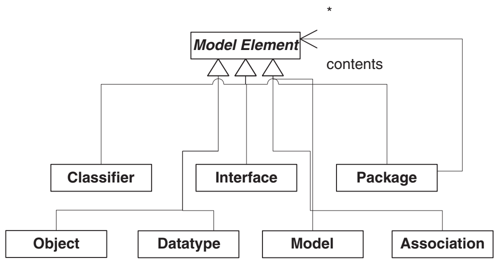


Figure 6: A fragment of the UML metamodel

Notice that a **Model** is also a subclass of **Model Element**; that is, according to the metamodel, a valid model may be constructed from several different models, where each is an instance of the metaclass **Model**, each potentially containing different abstractions and relationships. These models may therefore be independently constructed – indeed, each view is of itself a model – and nothing in the metamodel or in the semantics of UML guarantees consistency of these views. This is why UML is a multiple model approach. Dependencies between separate views must be dealt with by the developer. There is no theory provided with the UML that can help to test or verify that a set of models is consistent or inconsistent.

7 Where the principle is insufficient

We have advocated the single model principle for building reliable software seamlessly and reversibly. We now examine situations where the single model principle appears to be, or actually is, insufficient for rigorous development of reliable software. In particular, we want to focus on dynamic modelling, early requirements engineering, and legitimate inconsistencies.

7.1 Legitimate inconsistency

Traditionally, inconsistency in software desciptions is a matter to be avoided. After all, inconsistencies may lead to misunderstandings and errors that will result in faulty software. By contrast, Nuseibeh et al [13] write that inconsistency in software descriptions must be tolerated, and support the idea of "making inconsistency respectable". This position is not necessarily a contradiction to the major thesis of this article.

In the early part of the requirements phase, and even at later stages, the designer may want to use informal descriptions and rough sketches, whose consistency cannot be formally verified. Checking the inconsistency of a large description may be computationally expensive, even in those cases where procedures exist for such detection. Inconsistency may also indicate areas of legitimate uncertainty, or where the designers shared understanding has broken down. In these cases, inconsistency may be a necessary part of the design process. As stated in [13], the problem is not in the inconsistency per se, but with inconsistency that remains undetected.

Requirements engineering typically occurs in two distinct phases. *Early* requirements engineering is focussed on understanding, capturing, and analysing specific customer goals (which have a clear-cut criterion for satisfaction) or soft-goals (which need not have a precise specification of satisfaction). *Late* requirements engineering occurs when goals are well-defined and real-world entities can be modelled. Goals are critical in dealing with non-functional requirements.

A modelling language like BON/Eiffel, based on the single model principle, is insufficient for modelling goals, in part because of of the need for seamlessness. BON/Eiffel provides no built-in techniques for modelling goals: they would have to be treated informally (e.g., as comments, rough sketches, or informal documentation), and would not be directly expressible in executable code, thus defeating seamlessness. In order to treat goals formally, a richer modelling language, e.g., KAOS [2] could be used. However, this would defeat seamlessness and would introduce impedance mismatches, particularly in mapping designs to programs, unless the resultant object oriented programming language supports goal-based constructs as well. The impedance mismatch can be minimized by providing rigorous (automatic or semi-automatic) translations from a language for goal-based modelling to a language such as Eiffel, which obeys the single model principle. A translation from KAOS to Z has been defined, and mappings from Z to BON (and thereafter to Eiffel) appear in [16]. This approach is also followed by Graham [4], in his task-based modelling techniques that support traceability. Thus, the single model principle seems incompatible with goal-based requirements description, but translation methods could be used to ameliorate the incompatibility in practice.

Our conclusion is that we must allow modelling views, perhaps even inconsistent ones, outside of the profile supporting the seamless and reversible single model principle. A more comprehensive framework, involving the single model principle, would thus include both rough sketches [7] and consistent descriptions:

- **Rough sketches**, i.e. modelling views that are not necessarily consistent, such as goals, use cases etc. that are needed primarily in the early stages of the project. They provide informal, though useful, documentation that is ascribed no precise semantics.

- **Consistent descriptions**, i.e. a variety of views satisfying the single model principle that allows for seamless reversible development of code from specifications starting at late requirements and onwards.

Obviously, the single model principle dictates that we maximize the power of consistent descriptions and only use rough sketches where consistent descriptions run out of expressive power.

7.2 Dynamic modelling

In the BON/Eiffel version of the single model principle only one behavioral construct — the dynamic diagram – was supported. So too, the UML/Java profile described earlier used only colloboration diagrams. The unrestricted UML supports many behavioural descriptions such as sequence diagrams, statecharts and activity diagrams that all have their uses.

What is the status of these additional descriptions? As mentioned earlier, we can treat them as rough sketches. There are also possibilities, in future work, for automatically generating them from a single dynamic diagram.

For example, sequence diagrams can be treated as a generated view, constructed automatically from Eiffel implementations of methods of a class (that is, the messages depicted in a dynamic model simply show the sequence of function or procedure calls within an executing system). From this perspective, dynamic models cannot be constructed independently from static models (e.g., as is permitted with UML). This perspective on the use of sequence diagrams differs from that of Harel [5], wherein they are posited as a modelling technique for capturing requirements. It also differs from the use of sequence diagrams in the Rational Unified Process, wherein they are produced from use-case diagrams.

Statecharts are used in UML for describing the behaviour of objects; Harel posits them as a mechanism for modelling design [5]. Eiffel does not support state transition diagrams, but they can be generated automatically from class interfaces using an approach similar to that of the SOMA toolset [4].

8 Discussion and Conclusions

Independently generated multiple models of a system cause more problems than they solve in developing software. It is claimed that multiple models are useful because they allow developers to work independently on different parts of a software system, and thereafter their individual work can be integrated. We have already remarked on problems with this approach, particularly with consistency: checking that one independently created model does not contradict a second independently created model is a very complicated problem, even for small systems. The problem is avoided by obeying the single model principle.

We have been careful to phrase our arguments in terms of developing *reliable* software. Consistency of descriptions is an essential facet of developing reliable software, and thus we suggest that obeying the principle in this specific domain is necessary. However, the single model approach appears incompatible with early requirements engineering and for modelling inconsistency: systematic and automatable translations between appropriate languages may be necessary to help in these phases.

The multiple model approach offered by languages such as UML is not a good way to build reliable software. It is not a good mechanism for ensuring consistency, nor to help trace errors in programs back to errors in models. A single model approach, wherein different views of a system can be automatically or partly automatically generated from a single model of the system, should be preferred for developing high-quality software systems.

We have used BON/Eiffel to illustrate the single model principle, but our arguments are not limited to this modelling language: they apply to any language which provides a unique way of describing abstractions of a system of interest, and which relies on automatic generation of views. We have illustrated how UML and Java might be used to satisfy the single model principle.

References

[1] Beck, K. *Extreme Programming Explained*. Addison-Wesley Longman, 1999.

[2] Dardenne, A., A.v. Lamsweerde, and S. Fickas. Goal-Directed Requirements Acquisition. *Science of Computer Programming*, 20(p3-50, 1993.

[3] Douglass, B.P. *Real-time UML : developing efficient objects for embedded systems (2nd edition)*. The Addison-Wesley object technology series, Addison-Wesley, Reading, MA, 1999.

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

[5] Harel, D. From Play-In Scenarios to Code: an Achievable Dream. *IEEE Computer*, 34(1): p53-60, 2001.

[6] Hehner, E.C.R. *A Practical Theory of Programming*. Springer-Verlag, New York, 1993.

[7] Jackson, M. *Software Requirements and Specifications*. Addison-Wesley, 1995.

[8] Leavens, G.T., K.R.M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion*, ACM, 2000. ftp://ftp.cs.iastate.edu/pub/techreports/TR00-15/.

[9] Meyer, B. *Eiffel the Language*. Object Oriented Series, Prentice Hall, 1992.

[10] Meyer, B. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[11] Meyer, B. Agents, iterators, and introspection in Eiffel. ISE Inc. 2000.

[12] Morgan, C. *Programming from Specifications*. International Series in Computer Science, Prentice Hall, 1994.

[13] Nuseibeh, B., S. Easterbrook, and A. Russo. Making inconsistency respectable in software development. *Journal of systems and software*, 58(p171-180, 2001.

[14] OMG. OMG Unified Modeling Language Specification: Version 1.3. 1999. www.omg.org

[15] OMG. OMG Unified Modeling Language Specification: Version 1.4. Object Management Group. 2001.

[16] Paige, R. and J.S. Ostroff. From Z to Bon/Eiffel. In *13th IEEE international Conference on Automated Software Engineering (ASE)*, Hawaii, IEEE Computer Society, 1998. www.cs.yorku.ca/techreports/1998/CS-98-05.html

[17] Paige, R. and J.S. Ostroff. ERC — an Object-oriented Refinement Calculus for Eiffel. Department of Computer Science, York University. Technical Report CS-2001-05, 2001. http://www.cs.yorku.ca/techreports/2001/CS-2001-05.html

[18] Paige, R.F. and J.S. Ostroff. Metamodelling and Conformance Checking with PVS. In *Proc. Fundamental Aspects of Software Engineering 2001*, Springer-Verlag, LNCS 2029, pp2-16, 2001.

[19] Paige, R.F. and J.S. Ostroff. A Proposal for a Lightweight Rigorous UML-Based Development Method for Reliable Systems. In *Proc. Workshop on Practical UML-Based Rigorous Development Methods 2001 (co-located with UML 2001)*, Lecture Notes in Informatics (GI Series), German Society, p192-207, 2001.

[20] Paige, R.F., J.S. Ostroff, and P.J. Brooke. Checking the Consistency of Collaboration and Class Diagrams with PVS. 2001.

[21] Spivey, J.M. *The Z Notation: A Reference Manual (2nd edition)*. Prentice-Hall, Englewood Cliffs, N.J., 1992.

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