(Article published in IEEE Computer, September 1996)

Reversibility in software engineering

Kim Waldén, ***Enea Data***

What can we expect from analysis and design methods? The basic need is simple: obtaining a conceptual model of what software we want to build, before we build it. For small systems, informal descriptions may suffice, and the architecture gradually emerge as the hopefully well-structured source code takes shape. But for large, many-developer, long-evolving systems, we need to invest more in high-level specifications. Such specifications can be used in two different ways, both equally important:

A. As a high-level model to help us arrive at good solutions to complex problems by gradual refinement.

B. As an abstraction of an existing implementation to help us understand and control the software during its whole lifecycle.

The problem is that specifications using natural language and ad hoc notation tend to become ambiguous, inconsistent, and expensive to maintain.

Software development methods try to improve on this by offering a set of concepts and notations with more precise semantics. But the big challenge is consistency, which raises two questions relating to A and B above. First, how do we check *implementation consistency*, meaning that the final code really implements what was specified? Second, how do we check *specification consistency*, meaning that the high-level model is still a correct description of the evolving software?

To be able to check implementation consistency, we need to use high-level modeling concepts that are compatible with those of the programming language, so that all parts of the model can be traced to the source code. However, forward compatibility is not enough. We must also be able to see the high-level effects of implementation changes. The key to maintaining specification consistency is ***reversibility***, which is the ability to map the source code structure back into the high-level models.

Reversibility is a strong requirement, since it forces the high-level modeling concepts to be not only compatible, but in fact similar to those used at the implementation level. If they are not similar, the mapping will not work in both directions and the evolving system will gradually diverge from the specifications. To avoid this, the tacit requirement in textbooks on analysis and design is that all software changes must start at the highest level models and work their way down to the source code. However, this is not feasible in practice for two reasons. First, since non-trivial manual translation is required it is just too expensive to constantly iterate over the different models. Second, the top level analysis and design models are only the tip of an iceberg. Most of the detailed decisions collectively forming the behavior of a system needs to be specified using a programming language.

The key to maintaining specification consistency is *reversibility*, which is the ability to map the source code structure back into the high-level models.

Therefore, at some point during development, a conceptual model of the system in terms of implementation abstractions will emerge. This model, usually residing in the heads of the developers and passed on informally to new project members, is needed to understand and control the evolving system. It reflects directly what is running on the machine(s) and can never be replaced by analysis and design models expressed in a different conceptual system (like entity-relationship diagrams or state-transition tables).

This brings us back to the initial question, what can we expect from analysis and design methods? Well, unless the modeling concepts chosen are in close correspondence with those supported by the implementation language, reversibility is lost. This means that as the software changes, it will be impossible (at reasonable cost) to continuously update the high-level models and ensure specification consistency. Thus the most we can hope to get from non-reversible methods is initial help to arrive at a first version of a system. Since enhancements and

maintenance typically account for 80% or more of the total software cost, the models of such methods cannot be expected to play more than a minor role in software engineering.

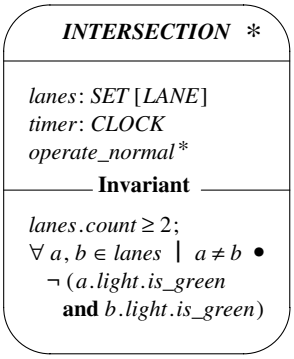
If analysis and design models are to serve as support throughout the software lifecycle, we need a different approach. We must use structuring concepts that can be preserved as part of the implementation, and again be extracted from the evolving source code to update the high-level models. So what does this mean? Are we supposed to use low-level programming constructs to describe our analysis model and overall system architecture? Certainly not. The idea is to use abstract object-oriented specification at all levels.

All concepts—from those mirroring the problem domain to those added later in the computational model—are viewed as classes whose operations define the behavior of the corresponding objects. If we stick to the class as the basic structuring mechanism, and only use inheritance and client relationships to describe how the classes relate to each other, the desired reversibility comes for free. This is because these concepts are directly supported by the major industrial O-O languages, so there will be no major impedance mismatches. If, on the other hand, we include concepts that cannot be mapped unambiguously to and from the eventual source code, reversibility is lost.

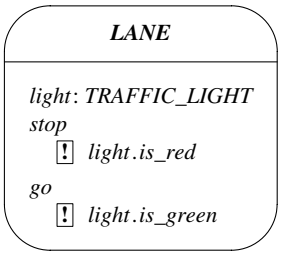
But what about the power of a notation based only on class abstractions connected by client and inheritance relations? Will it be expressive enough, or do we have to pay a high price for reversibility through a less general and less precise high-level notation? Can we do without the associations, multiplicities, and state diagrams which are included in nearly all published notations for OOA/D? The answer is yes. Since the approaches are different, you lose some and you gain some, but two crucial means of specification: strong typing and class contracts, can more than compensate for the loss of free associations. (For an introduction to software contracts see this department in the March 1996 issue of *Computer*.)

Strong typing makes it possible to specify the signatures (types of arguments and return value, if any) of each operation in a class. Contracts make it possible to also (partially) specify the semantics of each operation in a purely declarative form, independently of any implementation. For example, using the BON[1] notation, an intersection in a traffic control system might be defined by the class interface below.

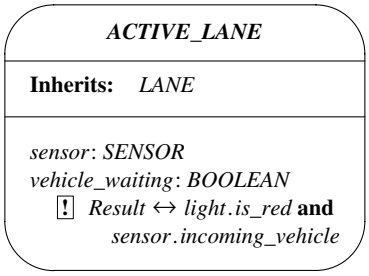
The intersection controls a set of lanes whose lights can be set to red or green. (Each lane may represent several physical lanes whose lights are all connected, showing the same color.) It has access to a timer which can be used by the *operate_normal* command to iterate through the stop and go cycles. The asterisk shows that the command is deferred, meaning that it has no implementation at this level. Proper behavior must be supplied by descendant classes.



The first part of the invariant section states that the number of lanes must be at least 2 (not much of an intersection otherwise). The second part says that for each pair of lanes in the set, at most one can have the green light on (important for avoiding accidents). The lanes in the set may look like:



The traffic light of each lane is accessible through the query *light*, and the commands *stop* and *go* have postconditions specifying the semantics of each command in terms of the current state of the light directly after executing the command. To avoid unnecessary waiting when traffic is low, some lanes may have built-in sensors:



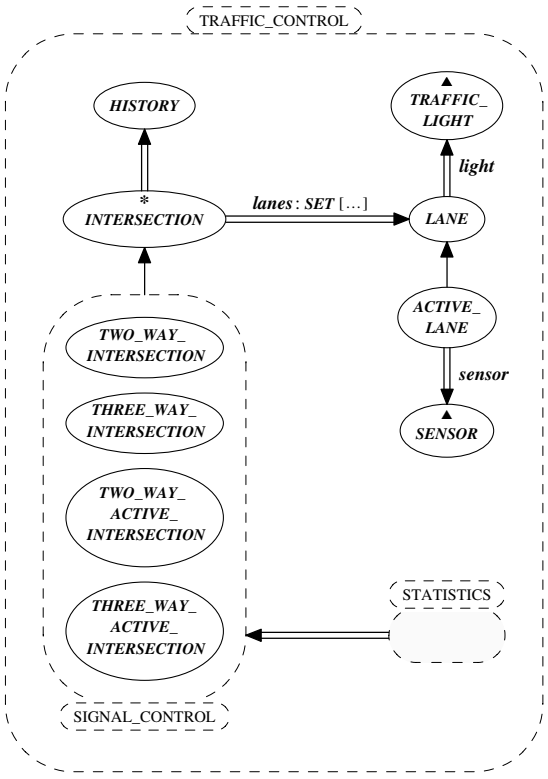
This type of lane has a query *vehicle_waiting*, which will return true if and only if the light is red and a vehicle is approaching the intersection.

The specification language is simple, essentially boolean expressions augmented by first-order predicate logic, so we can reason about sets of objects. Its main strength does not lie in the notation, but in the fact that queries to other objects may be part of the expressions. This opens the possibility of introducing new specification primitives as queries to previously defined abstractions (like the expressions *lanes.count*,

light.is_red, and *sensor.incoming_vehicle* in our example). The advantages are twofold: first, we may check the semantics of the queries by looking at the contracts in their respective classes. Second, the names of the queries and their signature types will reflect meaningful concepts in the system. This leads to a specification language whose primitives will automatically adapt themselves to the application at hand.

This leads to a specification language whose primitives will automatically adapt themselves to the application at hand.

Because of their simplicity, contracts can only cover part of the semantics of a piece of software, but unlike fully formal methods they are easy to read and to apply routinely in everyday software development. The gain in software quality can be quite dramatic. Strong typing, which is crucial for specification of contracts, also has the side-effect of giving us most client relations for free. Thus, in the overall architecture depicted below we can infer the static dependencies directly from the operation signatures in our example.



We have added a cluster of classes implementing the behavior of four different types of intersection, a history class, and a statistics cluster to collect and present information about traffic intensity, queue lengths, waiting times *etc*.

All class interfaces have been compressed into class headers containing just the names and some ornamentation showing that *INTERSECTION* is a deferred class, and that *TRAFFIC_LIGHT* and *SENSOR* interfaces external software or hardware. Client relations are shown as double arrows and inheritance relations as single arrows, both applicable to classes and/or clusters. Some of the client relations have been labeled with the names of the features causing the dependency. The *STATISTICS* cluster has been compressed (all classes hidden).

Strong typing, which is crucial for specification of contracts, also has the side-effect of giving us most client relations for free.

The ability to group classes recursively into named clusters (the dashed rounded boxes) combined with a general facility for showing or hiding details, addresses another important issue, namely *scalability*. Being able to zoom up and down layered models without losing track of how the currently visible components fit into the overall structure is essential when trying to understand large systems. We will not discuss scalability further here, since it would require a column of its own.

It is time to conclude, so once again: what can we expect from analysis and design methods? In my experience, most developers who use methods systematically—a small minority, to begin with—do so to get help in coming up with a good initial system design. Whether or not the resulting models turn out useful in practice, they usually lose their importance after the first software release (if not before) and the source code starts living its own life.

As long as incompatible modeling primitives, such as free associations and general state machines, are included as essential ingredients in a method, reversibility is lost. Without reversibility, it is impossible to ensure specification consistency in the long run, and analysis and design methods can only play a marginal role in software engineering. I think their potential is higher. Object-oriented abstraction, if used in its pure form, offers a unique possibility to attain reversibility. Let's go for it.

References

1. K. Waldén and J.-M. Nerson, *Seamless Object-Oriented Software Architecture*, Prentice Hall, Englewood Cliffs, N. J., 1995

Kim Waldén *is senior consultant at Enea Data in Stockholm, Sweden. He has been engaged in introducing O-O techniques in the Swedish software industry since 1987. His email address is kim@enea.se.*