Ensuring Semantic Integrity of Reusable Objects (PANEL)

Webb Stacy, *CenterLine Software, Inc.* (moderator)
Richard Helm, *IBM Thomas J. Watson Research Center*
Gail E. Kaiser, *Columbia University*
Bertrand Meyer, *Interactive Software Engineering*

The object oriented paradigm provides new opportunities for development via reuse. However, those opportunities are accompanied by new challenges. In particular, consumers of reusable components want to ensure that they have maintained the semantic integrity of the reused components. Several special approaches have been proposed to describe and enforce semantic integrity constraints in reusable components, for example class invariants [1], contracts [2], and inspection gauges {3]; others have discussed more conventional means such as simple assertions and unit and module testing {4,5].

This panel will address issues surrounding quality control of reusable components. Specifically, panelists will address these issues:

What are the challenges to semantic integrity maintenance presented by the object oriented paradigm?

How can semantics be verified in the presence of dynamic binding of methods to messages, especially if a method supplied by the reuser is bound to a message invoked from a reusable component?

Does encapsulation conflict with white-box testing? Is white-box testing appropriate for the consumer of reusable components?

Does the granularity of the component make a difference? For instance, is less semantic verification required when an entire framework is reused than when a single class or method is reused?

How can the consequences of semantic violations be expressed? Should reusers be allowed to violate them if the consequences aren't important to their application?

What are the economics of semantic integrity maintenance of object oriented components in the face of reuse? How can the costs be kept low enough that reuse is still attractive?

To what extent can the creator of the reusable components ensure their quality in all reuses? To what extent must this be the responsibility of the reuser?

What means are available for expressing semantic integrity constraints? What constraints can't be expressed?

To what extent should (and can) the enforcement of semantic integrity be provided by the language itself?...the development environment?...unit and module testing?...integration testing?

The panelists represent a range of viewpoints. Bertrand Meyer has pioneered investigations of assertions that describe class invariants and method pre- and post-conditions, and proposes correct software construction as a succession of documented supplier-consumer contract decisions [6]. Richard Helm also sees a role for contracts in software construction, but views them more as constraints on interclass behavioral compositions; he finds composition, especially in conjunction with contract specification mechanisms, a safer and more mature software construction technique than subclassing. Gail Kaiser addresses common misconceptions about adequate testing under inheritance and composition, finding that, in general, more testing is required than might be apparent at first glance.

These views are not mutually exclusive. Each is an important perspective on the picture of semantic integrity under reuse. Together, they serve to bring the picture into sharper focus.

Gail Kaiser

It is commonly assumed that properly constructed reusable classes can be tested in isolation and/or within the context of the original system, and then reused without retesting in a wide variety of systems. Although this belief is intuitively appealing, it turns out to be false for certain widely accepted testing criteria, such as statement coverage and branch coverage, that fulfill the axioms of "adequate testing" developed in the testing community [7,8].

Contrary to one's intuition, reusable classes -- particularly code inherited from superclasses -- should be retested in most cases of reuse [4]. New errors can be introduced by interactions with the new context. Further, old bugs that were never detected during previous testing or use of the class may come to light only during reuse.

Much of the confusion is due to the fact that reusers often treat adequate testing as "proving" that the code is correct, when in fact it does no such thing; adequate testing only provides a level of confidence that the code has been sufficiently exercised. While this confidence may be well-placed within the original context, it should no longer apply when a class is reused.

Richard Helm

Semantic Integrity of Reusable Objects: Re-use and Abuse of Software Components

Software components only operate under certain assumptions: parameter values, calling conventions etc. Reusers of software must not violate these assumptions if the semantic integrity of components is to be maintained. Object-oriented technology raises new issues concerning semantic integrity of reusable components (both objects and classes). An object's mutable state means its operating assumptions can change over time. "White-box" reuse via class inheritance, and "black-box" reuse (see [9]) via object composition each present different assumptions to the reuser. How can we ensure these assumptions will not be violated when reusing a component?

Unfortunately, the current situation is that the burden is largely placed on the reuser to do the right thing and not abuse the component. This is partly as it should be. The reuser has some responsibility in the matter. However, we often find that the burden is too large. There is high intellectual overhead for the user in order to truly understand and be sure that they are violating any assumptions.

For example, when subclassing, reusers may be required to

> implement methods deferred to subclass by parent,
>
> ensure overridden methods do not violate parent classes' assumptions about that method,
>
> respect protocol supported by parent classes, and

initiate notifying actions when subclass state changes.

In reusing and composing objects, they may be required to

create objects with correct parameters, compose it correctly and

only with compatible objects, respect allowable sequence of

operations that may be called on the object, provide appropriate

responses to calls initiated by the object.

To lighten the user's burden, what choices exist? Although programming language features provide prescriptive support, for example C++'s pure virtual functions and to a lesser extent Eiffel's assertions (which only detect violation of integrity, rather than ensure integrity), currently the reuser mostly has to rely on descriptions of operating assumptions. On the whole, we remain pessimistic for prescriptive and automated support for semantic integrity -- descriptive approaches will continue to dominate. Consequently, the problem of ensuring semantic integrity comes down to one of careful design that minimizes the opportunities for semantic violations to occur, coupled with adequate descriptions of these designs.

Given this to be the case, what should authors of components be designing, describing, and to a lesser extent prescribing to reusers? A clue lies in the structure of the more mature object-oriented frameworks such as InterViews, Unidraw, or ET++. A key quality of these systems is their reliance on object-composition (black-box reuse) as a mechanism for reuse and obtaining new functionality. This contrasts with less mature systems in which new functionality is obtained by creating new subclasses (white-box reuse), and contrasts even further with pre-mature system where new functionality is obtained by the addition of a new class (no reuse).

This experience suggests that the key to reusable software is found in flexible and abstract composable objects supported by a rich set of composition mechanisms. Helper objects that are responsible for checking composing objects correctly are also important. Systems based on object composition also reflect our desire to minimize the chances for violations. Because reuse via composition uses high level domain objects and abstractions, and composes these via standard interfaces, there are fewer chances for violations of integrity. In contrast, reuse through designing sub-classes via inheritance uses relatively low-level programming language abstractions, exposes class internals, and is prone to programming errors.

How then can we describe and prescribe object-compositions. As we described in [2], one technique is to use contracts. Contracts describe cooperating objects in terms of their individual obligations, patterns of communication between objects, inter object invariants, pre-conditions or pre-nuptuals required to enter the contract, and how the contract is to be instantiated with "live" objects. Contracts also included the notions of conformance declarations which describe how class and any subclasses do their part to fulfill the obligations required via the contract. However, contracts provide only a high-level description of an object composition. One approach to move away from the purely descriptive is to push contracts specifications closer to the programming language. This approach has been explored recently in [10] to specify reusable components, and by some of our recent experiments that assume the existence of contract constructs which provide means to explicitly instantiate contracts. Other means to describe object compositions include protocol and interface definition languages that include the notion of typing and subtyping. These can be checked at component composition time. Classes can also be instrumented to check that the protocol is obeyed, although this can have serious performance problems.

Finally, we remark that inheritance cannot be totally ignored as it provides a way to quickly populate the

system with components. But, as we mentioned previously, inheritance requires programming in low level abstractions and exposes the danger of introducing code that violates integrity. Coding is thus to be avoided. Fortunately, there exist class design techniques that reduce the amount of coding. For example, template methods combined with simple pure virtual functions allow rapid and simple customization of families of related classes.

Bertrand Meyer

Achieving semantic integrity through "Design by Contract"

One of the limitations of usual approaches to reuse, including many of the techniques available in object-oriented environments, is that they force potential "reusers" (software developers wishing to take advantage of existing library components) to choose between two equally unsatisfactory types of component documentation:

1 - The component's source code.

2 - Some documentation written separately from the component itself.

Solution 1 is inappropriate for large-scale reuse because it overwhelms reusers with low-level information, and fails to protect them against the effects of eventual internal changes. Solution 2 assumes supplementary effort on the part of the components' developers and, worse yet, cannot guarantee that the documentation will remain up-to-date when the components evolve (which almost inevitably occurs).

A better solution is to generate documentation from the class text itself. This approach can only work if the class text includes information that is both *semantic* as with technique 1 above, and *high-level*, as with technique 2.

To be high-level, the information must only describe interface properties, excluding any implementation properties. To be semantic, the information must not limit itself to the signatures (number and types of operation arguments): it must express the usage properties of each operation.

Eiffel's assertions fulfil this role. The semantic information for a routine is the combination of a precondition (input condition, imposed on clients, i.e., callers) and a postcondition (output condition, imposed the supplier, i.e., the routine itself). In addition, classes may be equipped with invariants.

The underlying theory is "Design by Contract" [1, 6, 11,12]. In fact I believe this is the main theoretical basis of the object-oriented method as a whole. Using Design by Contract, developers can ensure the reliability of software systems not through numerous and often redundant checks, but by specifying the precise conditions that govern communication between the various components of these systems. The contracts proper are expressed by preconditions and postconditions; the invariants are additional semantic constraints, providing crucial information to understand the semantics of classes. Invariants are accumulated in the inheritance process, giving its full semantic value to the view of inheritance as being (among other things) the "is-a" relation.

These ideas pervade the whole realm of object-oriented ideas. Two of their applications, as implemented in Eiffel, are particularly important:

- The Design by Contract principle yields a theory of inheritance, and a rule as to what kind of routine redeclaration is permissible. In the context of polymorphism and dynamic binding, redeclaration is subcontracting; for a subcontractor to be "honest", it must keep the precondition or weaken it, and it must keep the postcondition or strengthen it. These important rules are directly enforced by the language.

- Another consequence is a disciplined exception mechanism, based on the idea that an exception (abnormal case) is the result of some party's inability to fulfil its obligation in a contract.

Then only two responses are possible: resumption (try again, usually with a new strategy) and failure (carrying the exception over to the client).The Eiffel exception mechanism is the direct application of this analysis.

Assertions also serve as a powerful debugging tool, especially in connection with the use of libraries of reusable components if (as with the Eiffel libraries) the components are heavily equipped with preconditions, postconditions and invariants. They provide an excellent documentation mechanism: the "short form" of Eiffel classes, used as the key documentation format and generated automatically by tools of the environment, provides the semantic yet high-level form announced at the beginning of this note. Even more importantly, assertions serve as a constant methodological guide for the production of correct and robust object-oriented software, making "Design by Contract" a powerful analysis, design and implementation principle.

References

[1] Meyer, B., *Object Oriented Software Construction.* New York: Prentice-Hall, 1988.

[2] Helm R., I. Holland, & D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. *OOPSLA* 1990.

[3] Cox, B. Planning the software industrial revolution. *IEEE Software, 7*(6): 25-33, November, 1990.

[4] Perry, D. & G. Kaiser. Adequate testing and object-oriented programming, *Journal of Object-Oriented Programming, 2*(5): 13-19, Jan/Feb 1990.

[5] Smith, M.D., & D.J. Robson. A framework for testing object-oriented programs. *Journal of Object-Oriented Programming, 5*(3), June, 1992.

[6] Meyer, B. Design by contract. In D. Mandrioli & B. Meyer (Eds.) *Advances in Object-Oriented Software Engineering.* New York: Prentice-Hall, 1992.

[7] Weyuker. E.J. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering SE-12*(12):1128-1138 (December 1986).

[8] Weyuker, E.J. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM 31*(6):668-675 (June 1988).

[9] Johnson, R.E., & B.Foote. Designing reusable classes. *Journal of Object-Oriented Programming, 1*(2). June/July. 1988.

[10] Holland, I. Specifying reusable components using contracts. *ECOOP* 1992.

[11] Meyer, B. *Eiffel: The Language.* New York: Prentice-Hall, 1991.

[12] Meyer, B. Applying design by contract. *IEEE Computer*, October, 1992 (to appear).