Cataloging Object Oriented Software Reuse

Doug Lea

SUNY Oswego, Oswego NY 13126 *and*
NY CASE Center, Syracuse NY 13244-4100
Tel: (315) 341-2688
Email: dl@g.oswego.edu
Fax: (315) 443-4745

Abstract

Object-Oriented design is often said to be a positive factor in reuse. This position paper attempts to clarify some of the contributing factors and a few problem areas.

Keywords: Reuse, Object-Oriented, Design

Workshop Goals: Learning; networking;

Working Groups: Reuse and OO methods, Reuse and formal methods, Domain analysis/engineering, Design guidelines for reuse - OO

1 Introduction

Among the many reasons people adopt Object-Oriented (OO) methods is the widely held reputation for enhancing software reuse. However, many different, even conflicting accounts have been offered in explanation of this reputation. The remainder of this position paper attempts to categorize and clarify some of the ways in which OO developers obtain reuse.

2 Composition

OO methods support a version of the most basic and essential form of reuse, the combination of possibly many individual components to serve some aggregate purpose. *Object composition* is different in both concept and execution than procedure, function, or module composition. However it may be used to identical or similar effect. In compositional OO design, a developer defines a class whose instances serve as "hosts". Hosts maintain the identities of other "helper" objects and send them messages in the course of achieving required functionality.

Many variants exist. For example, one common form of composition is often termed "delegation". Here, any of some subset of messages are simply forwarded (delegated) to the object referred to by a certain connection link, but the exact object on the other side of the link may change (be rebound) over time.

Compositional designs may also be categorized with respect to their encapsulation and modularity properties. Pure *closed* composition is a form of *black box* reuse. It is among the safest and most conservative possible design strategies, due to:

- *Outward Closure.* Helpers never send messages except simple replies to objects outside some known scope. Replies never reveal the identities of inner objects.
- *Inward Closure.* The host maintains the sole link to each helper. The host never reveals the identities of helpers. No other objects may access the helpers or otherwise exploit their existence.

ADT-style OO designs rely on closed composition. The success of basic libraries containing many such black-box components attests to the utility of closed composition in addressing in-the-small reuse needs.

2.1 Architecture

The above closure restrictions may be lifted either by choice or by necessity, resulting in a continuum of *open* composition strategies involving collaboration networks, resource sharing, and larger scale opportunities for reuse. And larger scale opportunities for disaster: aliasing, interference, and lack of independence and local verifiability. These must be dealt with via combinations of design rules (e.g., access control policies), organizing constructs (e.g. "groups", "roles"), and worst-case implementation techniques (e.g., locking, atomic transactions). The presence of such dangers, along with the sheer difficulty of their diagnosis, prevention, and control can be a limiting factor in successful reuse of individual classes. However, semi-isolated networks in which these factors are known to be under control are often reusable as a group.

Moreover, by abstracting over the functionality of a system or subsystem of components, and only paying attention to their static and dynamic interrelations, one may use/reuse the resulting design architectures. Architectural reuse ranges from the everyday use of "standard forms" of static connections (e.g., simple linked lists), to the incorporation of design patterns that represent solutions to common structuring problems (e.g., model-view-controller designs), to the system-wide adoption of large-scale architectural frameworks (e.g., software bus, Linda tuple space).

Architectural reuse may or may not involve any actual code reuse. Some architectural elements may indeed be completely isolated from application-specific functionality. The isolation of elements into nodes and use of parameterized code to construct lists of such nodes represents a familiar example. Similarly, "protocol objects" such as multicast controllers may be reused in the construction of group broadcast designs without regard for the contents of multicast messages.

However, there may be at least an equal number of examples in which no code reuse is possible. For example, OO components assisting in the construction of many general graph structures seem little used, perhaps because of typical application-specificity of node and edge semantics. Instead, the general ideas are manually adapted to the situations at hand. It is not clear (and not worth debating) where to draw the line between this form of reuse and the application of "general design knowledge".

3 Inheritance

3.1 Classes as Specifications

Class constructs provide developers the opportunity to declaratively specify the behaviors of objects. While there is an enormous range in support for declarative specification, nearly all methods, languages, database tools, etc., possess some means of indicating what objects do without saying how they do it, or at least without allowing other software to depend on how they do it. Constructs may include simple method interfaces, method signatures, invariants, preconditions, postconditions, and other specification devices. Unlike similar module-based constructs, classes allow the reuse of the same specification for an arbitrary number of software entities (objects).

3.2 Specification Inheritance

The simplest way to reuse a specification that does not quite apply to a new problem is to extend it. Specification inheritance (also known as property inheritance) supports *extension by addition* of specifications. Subclasses may add new properties (e.g., methods, attributes) and/or add further constraints on existing ones. However, additions may not invalidate any property listed in any superclass. Beyond its technical uses, the emphasis on specification inheritance in OO development can lead to increased human understandability and conceptual reuse that accompanies any effort to hierarchically categorize entities in a given domain.

3.3 Polymorphism

Specification is an analysis and design concept. However, OO languages and tools integrate at least some declarative and operational aspects of types and inheritance, thus linking the expression of properties and code in software. The resulting type systems enable exploitation of subclass polymorphism: A software client `c` may require that a particular server object `s` possess the properties listed in some class `A`. However, `s` may well be an instance of a subclass `SubA` with additional properties. Specification inheritance rules, taken seriously, ensure that none of the extra properties in `SubA` invalidate `c`'s assumptions and requirements about `s`.

The primary effect on reuse is support for *interoperability*. Since client code is decoupled from inessential features, new subclasses of `A` may be developed and plugged while at the same time reusing existing client code. In practice, this remains more of an ideal than an everyday occurrence. Client code is too often written in a way that needlessly depends on inessential features of a specialized class rather than a less committal, more general superclass. (Design rules and languages distinguishing interfaces from implementations can alleviate this but tend to leave architectural specifications hidden as "implementation" matters.) Also, the introduction of a new subclass may suggest better ways of factoring and expressing superclasses and clients.

OO *frameworks* (e.g., for GUIs) typically combine these forms of polymorphic client code reuse with architectural reuse. Frameworks declare one or more inheritance hierarchies in which most classes are listed mainly in terms of their abstract interfaces. However, the classes may also contain code-based commitments to interconnection rules and protocols that hold independently of how other functionality is implemented. Framework users must add subclasses that implement (and perhaps add to) the desired functionality in an application-specific manner while still obeying the connection patterns and protocols listed in superclasses.

Additional reuse opportunities stem from constructs allowing classes to be parameterized with respect to other types. The techniques are similar to those used in modular design techniques for languages such as Ada. However, inheritance and parameterization (genericity) are not purely orthogonal concepts. Interactions among them lead to new design strategies, about which there is as yet little to conclude with respect to reuse.

3.4 Defeasible Inheritance

Few languages enforce pure property inheritance rules. They instead support a form of defeasible inheritance in which some superclass properties may be *overridden* in subclasses. In most languages, this overriding is limited to redefining method implementations in subclasses.

Overriding can enhance reusability. Overriding rules can make it easier to obtain code decoupling. In a system with overriding, clients cannot depend on any features of listed method code, since it may change in subclasses. (Since in most languages, only method code, not data representation commitments may be overridden, representational coupling remains a potential problem.) It also provides additional opportunities for internal reuse. A superclass may include *default* method implementations that are reused by most subclasses without forcing all of them to do so.

However, this form of reuse-driven inheritance can have harmful effects on other aspects of reuse and software quality. Most languages do not contain powerful enough declarative constructs to require subclass designers to preserve all essential guarantees of their superclasses. Even in languages that do possess these features, programmers tend not to use them enough. It is often so much easier to write code that implements functionality than declarations describing its effects that people don't even attempt the latter. Thus, a subclass implementation of a method may alter semantics in a way that does not preserve interoperability properties that clients depend on. As a limiting but very common case, a new class may list another as a superclass solely to reuse a few bits of its implementation without in any way preserving superclass declarative properties. (Some languages do provide constructs that distinguish at least this extreme case.)

Sometimes this form of implementation inheritance is used by developers just because it is easy to express in OO languages. Reuse goals could be better preserved via compositional techniques such as using another object as a delegated helper rather than listing its class as a superclass. Worse, the fact that many programmers see these two options as nearly equivalent can lead to other conceptual design errors. Worse still, languages, user manuals, and textbooks themselves obscure these issues by attempting to relate subclassing concepts to the use of runtime storage layout schemes that can be viewed as representationally "embedding" superclass objects inside subclass objects. Design decisions are sometimes based on these considerations. Sometimes this is due to an unfortunate but correct concern for representational compatibility with existing reused software (e.g., between C and C++), but more often results from simple confusion about efficiency consequences.

Defeasible inheritance is also used by programmers in order to evade language-based encapsulation rules. In most languages, a client/host is not allowed to even reference internal implementation matters listed in a class, but a subclass is. Thus, inheritance is used to simplify white-box reuse of incidental implementation features. While such code mangling is not an optimal form of reuse, it is often preferable to no reuse at all, so does have its place. It is unfortunate that OO languages force such tradeoffs against non-local reusability goals.

To the extent to which specification reuse is more important than code reuse, defeasible inheritance appears to be a net loss. There are certainly better language constructs available that still support decoupling, default code, and opportunistic white-box reuse, and even the convenience of expressing specifications, interfaces, and code in the same framework. In fact, even without adding new constructs, it is very possible to ban all use of defeasible inheritance in languages including C++ and Smalltalk, and still obtain these forms of reuse. However, the constructions are awkward and unnatural enough that programmers do not employ them in practical development.

4 Biography

Doug Lea teaches computer science at the State University of New York at Oswego, teaches software engineering courses at Syracuse University, codirects the Software Engineering Lab at the New York State Center for Advanced Technology in Computer Applications and Software Engineering, and works on projects involving object-oriented design, specification, distribution, compilation, reusability, and libraries.