

Subjectivity in Object-Oriented Systems

Workshop Summary

William Harrison,* Harold Ossher,* Randall B. Smith,† and David Ungar†

*IBM Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598

†Sun Microsystems Laboratories, Inc.
2550 Garcia Ave, Mountain View, CA 94043-1100

Abstract

Subjectivity in object-oriented systems is a new research area. At this, the first workshop in this area, there was much discussion of fundamental concepts and issues, as well as of perceived needs for subjectivity and models for realizing it. The discussion is summarized here, and a list of issues that were identified during the workshop is presented.

1. Introduction

According to the accepted way of thinking about objects, sometimes called the *classical model* [6], an object encapsulates all its state and behavior. Ideally, the designer of an object (or class) defines and implements the *intrinsic* properties and behavior of the object, and all other properties and behavior required by clients can be derived from these using the public operations.

This classical ideal is inadequate to deal with situations in which different, *subjective* views of shared objects are used in different parts of a system, by different users, or at different times. For example:

- *Construction of large and growing suites of applications manipulating shared objects.* New applications often introduce the need for new *extrinsic* properties and behavior that cannot be derived efficiently, or at all, from the intrinsic properties and behavior. Different applications might even need to classify the same objects differently.
- *Multiple views.* An array of data, for example might be displayed as a histogram or as a pie chart. These separate displays can be taken as different points of view on the same data. The display routines can be directly associated with the array object, and can even have access to encapsulated state. They should not be considered intrinsic properties of arrays, however, and it should be possible to write them as separate “applications.”
- *Versions.* Different state associated with a single object can be seen as a use of subjectivity. Users or applications may find it convenient to pass around or even store a reference to a single object, without worrying about updating the reference to refer to the latest version. One reference would suffice for all versions.
- *Debugging,* especially debugging code that the debugger uses. By giving the debugger a safe point of view on an application, it is possible for the debugger to be used on objects the debugger itself will use. For example, window system code might be viewed from a stable, base point of view, while a programmer adopts a temporary, trial perspective where s/he makes speculative changes to the window system. When an error arises, the programmer

can run the debugger using window system code from the known, stable point of view.

Allowing objects to present multiple views to different clients [3,4,7] is a first step towards subjectivity. The views must, however, either be implemented by the object designer as intrinsic, or be implemented as extrinsic in terms of already-available views. True subjectivity requires that the separate views contribute towards or even constitute the definition of the object. This approach has been taken in the past in some systems that are not object-oriented [1,2]. Support for subjectivity in a truly object-oriented setting has been addressed only recently [5,8]. It raises the promise of greater flexibility in writing, extending and combining object-oriented applications, and at the same time, many interesting technical and even philosophical issues.

This is a new area of research. The primary purpose of the workshop was, therefore, to facilitate exploratory discussion of the domain and identify some of the key concepts and issues. To this end, the workshop was structured as a morning of brief presentations to introduce ideas and to set the context, followed by an afternoon of discussion.

This report summarizes the morning presentations and the afternoon discussion groups, and concludes with a list of issues identified. Further details are contained in the thirty position papers submitted to the workshop.

2. Morning Presentations

In the morning, there were seven short presentations of material drawn from position papers to help set the context in terms of concepts, applications, and models and implementations.

2.1. Concepts

David Ungar and Randall Smith summarized the support that the language Us contains for subjectivity that allows an object to behave differently depending on the “perspective” employed by the client. A perspective is formed by a series of layers, nested so that there is greater sharing and greater stability nearer the core. Each layer contains a “piece” of each object in the perspective. Inheritance from inner layers exactly mirrors inheritance from parent objects. The perspectives are first-class computational elements. Two significant issues raised

by this approach are the problem of maintaining invariants when access to an object occurs through a perspective not constrained by the invariant and the problem of what it means to copy or initialize an object which is in many perspectives.

David Griswold presented an analysis of many of the abstract elements that make up a model of subjectivity. Since defining “something” which is relative to “some observer”, natural questions are: what is the “something's” identity, what is the observer, and how are they related? Once these questions are answered, the issues arise of where the objects' behavior and state are and whether the observer can be manipulated. Then one can ask whether the views are in the “something” or in the observer. If in the “something”, the elements of the view can be made available by visibility, by subtyping, or by adding to the base. These are called revealing, knowing, and burdening. If in the observer, one can ask how much is in the observer's “eye”: behavior only, added state and behavior, or all state and behavior. The first of these are called intercepting. Depending on the relationship between the observer and the “something”, the second is either called inheriting or viewing. When all state and behavior is in the observer, the third might be migrating if the same identity is used for the object in both observers or reparsing if even the identities are not preservable. In all cases where the state is in the observer the problem of specifying identity must be addressed.

2.2. Applications

Hayden Lindsey reviewed the software development problems that motivate interest in subjective approaches. As OO approaches are being used for larger projects, they are being applied in multi-project development of systems. This results both in overloading objects with the conceptual issues from a number of applications and also in overloading objects with the physical support for the many applications. One way of addressing these overloads is with refactoring, but refactoring of classes is hampered by lack of time and foresight. Furthermore, when extensions to object function for new projects require more state, the development responsibilities for the object become fragmented. Finally, there emerges a need for multiple versions of the same method from different development groups. These problems can be addressed to some extent in development environments; for example, in ENVY, behavior is spread across “applications” but adding state must be done in the “owning” applica-

tion. However, complex application development needs extension of the OO paradigm to provide for first-class treatment of “subjects” (modules that use their own particular view of shared objects), for the ability to create customized views of class state (especially to eliminate state), and for the ability to specify like-named methods within several applications

Dave Cleal addressed the need for support for subjectivity to enhance reusability. The needs were recognized in analyzing why the expected benefits of reusability did not emerge in the construction of a large securities management system. Many of the opportunities for reuse were frustrated by the need for different size/performance tradeoffs resulting from the use of common function in different settings. For example, the size and performance demands of trading tools which use “security” objects are high, and are satisfied by a class that has just a name and price, but analysis tools have complex time-projection capabilities and use a heavy-weight class for securities. Several approaches were investigated for providing different viewpoints on the security: wrappers, containers, ENVY extensions and extrinsic views. Each of these had several drawbacks. ENVY extensions, for example, can't multiply supply methods. Extrinsic views have problems with identity, so that eventually the core had all the support for a multiplicity of views. The frequently suggested approach of using subclassing and mix-ins causes a combinatorial explosion in the class hierarchy.

Michael Vanhilst described his experience building an astronomical viewer. The original design had been procedural and was suffering from many problems in making enhancements. He had tried to re-build it using OO technology without promising results. The classes became too big, and useful functionality had to be removed because the OO model made it too difficult to implement. An example is the many-to-many mapping of windows to images. He is now trying to use subjective approaches to solve some of these problems. One significant issue he has identified is that when aggregates of objects are used, the initialization order of visiting objects might have to depend on the subjects involved. He has found a number of features of subject-composition helpful: composing aggregates, augmenting the attributes, changing the behavior of methods by adding implementations,

and class composition with partial mappings.

2.3. Models and Implementations

Harold Ossher discussed the prototype he is building to support subjectivity and subject composition. This prototype emphasizes the use of “subjects” as a packaging construct and the ability to compose binary subjects obtained from multiple sources. In WASP (the WATson Subject-composition Prototype), a C++ program is compiled by a tool called the C++ Subjectifier, to produce a binary subject. Subjectifiers can also be written for other languages. Binary subjects are put together using a tool called a Compositor. The compositor is language-independent and is responsible for the interchange and composition of binary subjects produced from multiple source languages. Different compositors can be written to support a wide variety rules for how composition is performed. To aid both the human developer and the compositor in composing subjects, each binary subject has a human-readable “label” that describes how the binary code in the subject is used to implement various interfaces on objects. It has several elements, including interface and class definitions, constraints on which classes this subject requires to support which interfaces (even though some of that support may come from other subjects), and a mapping specifying which operation calls resolve to which (multiple) implementations in the binary code.

Craig Chambers described the ways in which Cecil contains support for subjectivity. Cecil is a pure OO language with multi-methods. The multi-methods are typically declared within scopes, but outside of the classes themselves. Other aspects may also be declared separately, e.g. attributes. The scoping and multi-method mechanisms are introduced to localize extensions to individual libraries for purposes of encapsulation and simplicity. The concept of “module” is added to the language as packaging principle to define scopes. Symbols can be explicitly imported from other modules using “private” declarations. An important issues is the visibility of a declaration. Visibility could be determined statically for type-checking or dynamically for method lookup. An important additional goal was to allow subclassing to extend the support for a class without impacting its clients.

3. Afternoon Discussion Groups

In the afternoon, the participants selected four areas for discussion and separated into groups for that purpose. The four topic areas were characterized as: Concepts and Terminology, Invariants, Models, and Models Meet Applications.

3.1. Concepts and Terminology

The discussion of Concepts and Terminology began by trying to clarify the differences in emphasis among two aspects of subjectivity addressed by the participants. One aspect emphasizes the fact that the subjective result of a message send is determined by the perspective adopted by a client. The client adopts a perspective, and the behavior depends only on its perspective. The other aspect emphasizes the fact that the subjective result depends not only on the perspective adopted by a client, but on some composition of all the other perspectives which are applied to the object at that time. [Although not developed during the discussion, we will use the terms “client subjectivity” and “participant subjectivity” for these two aspects of subjectivity.]

Some time was spent discussing the use of the term “subject” to stand for an orchestrated collection of perspectives on many classes of objects. Although it is natural to see the term as reflecting subject as a domain of interest (e.g. “the subject of physical modeling . . .”) its other common natural use as the object of interest (e.g. “Lee was the subject of several investigations . . .”) leads to confusion. It was explained that the term was originally chosen as a packaging term, not for single classes or objects, but as an alternative to “tool” or “application” which coordinate behavior on many objects. Subject connotes the perceiver of a subjective reality (in the sense used by philosophers like Burke and Hume) rather than the object in an objective reality (in the Platonic or Aristotelean sense).

A second issue discussed was that of completeness. In client subjectivity, a perspective is complete - all of the support needed for the behavior is present through the perspective. The layers of a perspective are incomplete, but do not actually exist without their underlying layers. In participant subjectivity, a perspective is complete with respect to definitions, but not implementations. In other words, the subject defines a complete view of the world,

but does not necessarily implement all of it; other subjects might supply essential and/or additional implementations.

A third issue discussed was whether subjectivity applies to individual instances or to whole classes. Client subjectivity tends emphasize support for subjectivity for individual objects while participant subjectivity emphasizes class-wide characterizations.

A fourth issue is whether to embed the constructs in a programming language (like Us or Cecil) or in a module-interconnection language (like the Subject Label language).

Discussion concluded with the question of a “litmus test” to determine if a system provided “real subjectivity.” Such a test might have the form “If I have an object and I send a message to it, is the message and object identity enough to know the behavior?” As a sample point, a system was proposed in which a single application derives two different behaviors by being linked (composed) with either of two libraries. Although some felt that the example wasn't even object-oriented, it could be viewed as a use of subjectivity because the behavior is not determined solely by the objects involved, but also the provider-subjective libraries that implement the function.

3.2. Invariants

When working with a subjective model, an invariant being maintained by one perspective may not be maintained by another. For example, consider two perspectives on a list in which one (#1) employs a count which always equals the number of objects in the list while the other (#2) simply keeps the list itself. There are two implementations of the method for adding to the list. In #1 it also updates the count while in #2, the method has no awareness of the count.

The invariant problem arises from the client aspect of subjectivity, because the behavior of the object varies by client. Participant subjectivity attempts to remedy this by allowing the mere presence of a subject to cause invocation of that subject's behavior. This provides the hooks needed for a subject to maintain its invariants irrespective of the client's subject. The invariant problem then arises only when the composition rules used exclude subjects that maintain invariants; this is the responsibility

of the person performing the composition, who is supposed to have a more global view of things.

Two problems emerge with respect to invariants: how to know there is a problem, and how to fix the problem. One approach to detecting a problem is to observe that any invariants involving instance variables present in only one of the perspectives/subjects is problem-free, as are invariants present in all of the perspectives/subjects. However, potential problems can be flagged when an invariant mentioning instance variables manipulated in several perspectives/subjects is not present in all perspectives/subjects that have those variables.

One way to repair the invariant failure is to create a (#3) subject which merely updates the count when element-adds are performed. The #3 subject is merged into the #2 subject using provider-subjective support, so that the required invariant is now true in the resulting composition.

3.3. Models

Four models (Us, Subject-Composition, Cecil, and Groups) were compared in three respects.

With respect to problems and issues in construction, Us raises issues in the difficulty of interpreting the meaning of cloning with respect to the object's pieces that are not in the perspective where the cloning is being performed. Subject-composition raises issues in initializing instance variables and in the meaning of deferred initialization. Cecil also raises constructor issues, preferring default initial values to C++ constructors (as does subject-composition). Finally, the group model performs initialization lazily because the group starts out "empty" by declaring its existence and then letting members join at leisure.

With respect to the "first-classness" of perspectives, Us and Groups treat the sender's perspective as an element manipulated by the applications, while Cecil and Subject-composition are intended to have the existence of other subjects be transparent to the clients. Perspective shifts can be orchestrated by re-composing the subjects, but, generally, this sort of computation would be outside the domain of most method implementations.

Finally, with respect to how a message send is expressed, Us employs a dispatch to one target with the dispatch

based on perspective and object. Subject-composition dispatches to many targets formed by method combination with multi-method dispatch that might (but often does not) select based on the (implicit) client subject. In Cecil, dispatch is to one method with multi-method dispatch based on scoping. In group models, messages sent to the group are multicast to all members of that group.

3.4. Models Meet Applications

Much controversy was sparked by the summary which posed the question: "Did any of the applications need Us's ability to compute with perspectives as first-class elements, or is SELF enough?" and the answer "None of the applications seemed to need more than one object with many perspectives, so is any of this really needed?"

Several people voiced the view that participant subjectivity answered their need for separate, decentralized development of applications that share objects. They reflected application domains including software representation and communications system design. Although the original question was posed from the point-of-view of the run-time execution, issues of support for packaging and reconstruction were seen to be very important as well.

4. Issues

The subfield of subjectivity in object-oriented systems is relatively new. A major goal of the workshop was to identify issues that require further examination, even if there was not time for solution or even discussion of all of them. The following is a list of such issues, presented here without discussion in the hope of sparking further interest, discussion and research.

Concepts

- What makes a system truly "subjective?"
- What is the difference between "subjectivity" and "multiple views?"
- Where are the objects' behavior, state and identity? In the objects or in the observer? Where are the views?
- Extensions of extensions? Perspectives of perspectives?
- "Client" versus "participant" subjectivity (see above)
- Instance versus class-based subjectivity
- Other kinds of subjectivity?

Use/Requirements

- Real-world, practical examples
- Does/how does subjectivity allow enhanced reuse?
- Different performance requirements in different situations can require radically different treatments of the same objects. It is not always viable merely to merge functionality.
- How can subjectivity be used to support the following development needs:
 - decentralized definition of state and behavior
 - packaging of just needed classes, methods and state
- How can subjectivity be used to support the following structures:
 - multiple aggregation hierarchies
 - windows with window-manager-dependent preferences

Structure

- Granularity of perspectives: object; layer, module or scope; entire program?
- Peer perspectives versus hierarchies of perspectives
- Is there always one right hierarchy for a system?
- Bundling of state and behavior

Methodology

- Notational devices for subjectivity
- Analysis and design issues

Semantics

- Static versus dynamic definition of perspectives and their contents
- Static versus dynamic composition
- Invariants and consistency
- Contracts and subjective viewpoints
- Privacy/encapsulation
- Are perspectives first-class?
 - If so, how should they be controlled?
 - If not, what are the limits?
- Creation, initialization and copying of objects
- Which perspective to use during evaluation
- When to change perspectives
- Static type checking of subjective code
- Method combination across perspectives, including details ordering and handling of return values.
- Managing name clashes across perspectives
- Flexible mapping of instances across multiple subjects: partial, many-many.

- Implementation issues

Relationship to Related Areas

- Relationship between subjectivity and reflection
- Can an object itself and the class that describes it's structure be considered just different perspectives of the object?
- Relationship between multi-methods and subjectivity
- What can be learned about subjectivity in OO from "group models" in distributed computing?

References

- [1] Gerard Boudier, Ferdinando Gallo, Regis Minot, and Ian Thomas. An Overview of PCTE and PCTE+. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium of Practical Software Development Environments*, pages 248–257, Boston, November 1988. ACM.
- [2] David Garlan. *Views for Tools in Integrated Environments*. PhD thesis, Carnegie-Mellon University, 1987.
- [3] I. P. Goldstein and D. G. Bobrow. An experimental description-based programming environment: Four reports. Technical Report CSL-81-3, Xerox Palo Alto Research Center, March 1981.
- [4] Brent Hailpern and Harold Ossher. Extending objects to support multiple interfaces and access control. *IEEE Transactions on Software Engineering* 16(11), pages 1247-1257, November 1990.
- [5] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '93)*, pages 411–428, Washington, D.C., September 1993. ACM.
- [6] Object Management Group. *Object Management Architecture Guide*, second edition, September 1992. OMG TC Document 92.11.1.
- [7] John J. Shilling and Peter F. Sweeney. Three steps to views: Extending the object-oriented paradigm. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications, (New Orleans)*, pages 353–361, October 1989. ACM.
- [8] Randall B. Smith and David Ungar. A simple and unifying approach to subjective objects. Self Group Technical Report, Sun Microsystems Laboratories, Inc., 1994.