Dynamic Subjectivity: Implementation Issues and Computational Reflection

Jane Pryor and Natalio Bastán

UNICEN - Fac. de Ciencias Exactas
ISISTAN - Grupo de Objetos y Visualización
San Martín 57, (7000) Tandil, Bs. As. Argentina
TE: +54-293-40363 FAX: +54-293-40362
E-mail: {jpryor,nbastan}@exa.unicen.edu.ar
URL: http://www.exa.unicen.edu.ar/~isistan

Abstract

Subjectivity, and in particular dynamic subjectivity, is a desirable feature in programming languages, so as to allow the implementation of different views in order to enhance the reusability and integration of the key abstractions or components of these systems. This work presents a reflective meta-level architecture that supports dynamic subjectivity in an object-oriented system. This architecture has the advantage that the subjective behaviour is handled by a meta-level, such that the application that resides at the base level does not need to be modified.

Keywords: Object-oriented programming, Dynamic subjectivity, Computational reflection.

Dynamic Subjectivity:
Implementation Issues and Computational Reflection

1. Introduction

Complex software systems must be able to expand and change according to new requirements. This means that they should consist of components that are adaptable to these changes.

In the case of object-oriented systems, these are based on a set of abstractions modeled by a corresponding class in terms of abstract state and behaviour. When these systems expand into an integrated suite of applications, the different clients require context-specific views on such abstractions.

The language support for implementing such systems plays a central role as to the ease with which these large applications are modified and expanded. Experience has shown that a desirable feature of a programming language is to allow the implementation of different views in order to enhance the reusability and integration of the key abstractions or components of these systems.

The notion of Subject Oriented Programming was proposed as a vehicle for facilitating the development and evolution of suites of cooperating applications. In this case the term subject is used to mean a collection of state and behaviour specifications that reflect a particular perception of the world, such as is seen by a particular application or tool. The essential characteristic of subject-oriented programming is that different subjects can separately define and operate upon shared objects, without any subject needing to know the details associated with those objects by other subjects [Harrison93].

Dynamic Subjectivity (DS) is an alternative paradigm whereby the behaviour of an object is determined not by its class, but by a runtime context [Nolan97]. Contexts are descriptions used to dynamically bind identity, state, and behaviour to an object. The behaviour is subjective because the response to a given message will depend on the context in which it is sent. The identity of an object remains the same; however the state and behaviour may differ according to the context. This characteristic of a programming language would enable the development of much more flexible systems than conventional object-oriented programming. Viewing a system as a composition of subjective behaviours, these behaviours could be reused across different applications without any change in the structure of the objects. This facility, however, is not available in current object-oriented languages.

In this work we analyze different approaches to provide support for DS in the context of the Smalltalk environment. The analysis evaluates the flexibility and extendibility that these DS approaches provide. It also considers the ease with which DS can be incorporated to an existent programming environment. We show that a reflective approach based on a meta-level architecture brings the benefits of DS to an existing language. Computational reflection provides the advantage that the base application does not need to be modified, and the incorporation of subjectivity to a system at a meta-level, will provide a much more powerful, flexible and transparent environment. Under this approach, the paper presents a meta-architecture that meets the requirements mentioned and shows how this meta-architecture can be used to implement subjective applications.

This paper is structured as follows. Section 2 presents an overview of Dynamic Subjectivity. Section 3 analyzes different approaches to implement subjectivity. Section 4 describes an architecture for DS in Smalltalk 80 by means of computational reflection, and Section 5 presents some preliminary conclusions.

2. Dynamic Subjectivity

According to the accepted way of thinking about objects, an object encapsulates all its state and behaviour. Ideally, the designer of an object (or class) defines and implements the intrinsic properties and behaviour of an object, and all the other properties and behaviour required by clients can be derived from these by using the public properties .

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, in the case of a company that sells goods or services, the employees will have different responsibilities according to the work they do. They may act as managers, salesmen, administrative staff, etc. It is possible for the same employee to carry out more than one of these activities, as for example a salesman who also acts as a manager (Figure 1).

The different roles may need different features in the computer systems within the organization. The corresponding objects don't all have the same behaviour, but they may have some common behaviour. There are complications in applying inheritance, as an object may take on more than one role (or set of behaviours), or may take on new roles during its lifetime. This is the case of a manager who is also a salesman, or an administrative employee who also becomes a salesman.

The different clients of these objects may require different (subjective) views of the same employee. In some cases a sub-system will need to process all the views, in others it will be interested in only one view. A system that calculates salaries and commissions (payments) to employees will need to process all the views of the same employee, and a sales system will only be interested in the salesman view.

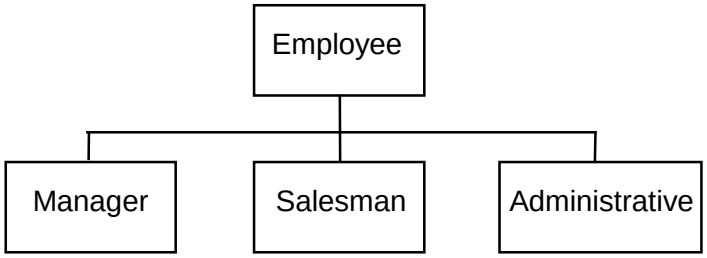

Figure1. Employee subtype relationship

A system of subjects is defined by [Harrison93] as meaning "a collection of state and behaviour specifications reflecting a particular gestalt, a perception of the world at large, as is seen by a particular application or tool". This proposal is based on a scheme of composition rules which instantiate subjects and route messages between them. These composition rules deal with matching classes and method names across dissimilar hierarchies and interface specifications.

There have been many different intents through workshops and discussion groups [Harrison94] [Harrison95] [Nolan96] to define the term subjectivity other than the many related and similar terms currently being used in this area of research (views, roles, perspectives, aspects).

For instance, [Kristensen95] defines "A role of an object is a set of properties which are important for an object to be able to behave in a certain way expected by a set of other objects. A role is a description and models an abstraction. A role instance has state and behaviour, and

models a perspective on a phenomenon. An object with its roles referred to as a single entity will be called a subject."

Aspects [Richardson91] have a similar purpose. They are sets of extensions or chunks of attributes and methods that an object may acquire or lose during its lifetime.

The overall goal of subject-oriented programming according to [Harrison93] is to facilitate the development and evolution of suites of cooperating applications. This remains a valid statement across all the various uses and implementations of subjectivity.

John Nolan in [Nolan97] presents Dynamic Subjectivity as a variant, where the behaviour of an object is determined not by its class but by the user's runtime context and viewpoint, the latter being determined by the current state of the object. The response to a message may then vary according to the context in which it is sent, and these variations in behaviour will therefore present different views of the object.

Thus, the context is constructed from a set of relationships among objects, with the context description similar to a set of rules. Methods are then defined with respect to a context, including in the declaration a series of conditions that must be met for this method to be used. When there are a number of appropriate implementations available, the one that describes the most accurate context is to be used.

The dynamic binding at runtime of behaviour based upon the current contextual information leads to greater possibilities in reuse and customizable systems. The technology required to build these languages is available: AI-like rule matching and object-message passing.

As an extension to Nolan's proposal, the context which will define the actual behaviour may not only depend on the current state, but also on the sender of the message, or even the actual application or sub-system being used (in the case of suites of applications).

There are several proposals of patterns for the design of roles, which in a more generalized fashion are adaptable to the concept of subjective views of a unique identifiable object. However, the current behaviour which corresponds to a role is determined by an explicitly defined context, with the limitation that the runtime context cannot be determined implicitly by the state or other factors.

The next section presents the most used design patterns for roles, with the proposal for the inclusion of rules sets which would permit the implicit runtime evaluation and determination of the context.

3. Some Approaches to Support Subjectivity

A variety of approaches for applying subjectivity have been explored and proposed in the last years. They vary in how and where the division between objective and subjective knowledge is made.

There have been various proposals for new or extended programming languages and tools that would support subjectivity. Most are at a prototype stage, such as the Watson Subject Compiler whose goal is to compose pre-compiled subjective code without recompilation, initially using C++ [Kaplan96]. Others include the programming language Cecil, a descendant of SELF, by Craig Chambers, which is a new object-oriented language which combines multi-methods with a classless object model, object-based encapsulation, and optional static type checking. Cecil uses "Predicate Classes" which define an object as being a member of a clas whenever it satisfies a predicate expression [Chambers95].

For the handling of subjectivity in object-oriented languages, different design patterns have been proposed. These patterns offer different advantages and disadvantages according to the characteristics of the application and the type of subjectivity involved.

The Role Object pattern [Bäumer97] [Fowler97] models context-specific views of an object as separate “role objects” which are dynamically attached to and removed from a “core object”. The core object models and implements a particular key abstraction, and each of the role objects model and implement a context-specific extension of the core object. This proposal makes the role object a decorator of the core object, as it provides a flexible alternative to subclassing for extending functionality [Gamma95].

The resulting composite object structure, consisting of the core and its role objects, is called a subject. A subject often plays several roles and the same role is likely to be played by different subjects. To work with a specific role of an object, the client must explicitly identify it.

The Role Object pattern has the following disadvantages:

- Clients are likely to get more complex due to a slight coding overhead. A client has to check whether the object plays the role in question. If it does, the client needs to query for the role; if it does not, the client is responsible for extending the core object in its use-context provided that the core object actually can play the role.

- Maintaining constraints between roles becomes difficult. Since a subject consists of several objects which are mutually dependent, maintaining constraints and preserving the overall subject consistency might become difficult.

- Constraints on roles cannot be enforced by the type system. It might be necessary to exclude certain roles from being attached to the same core object in combination. Also, some roles may depend on the existence of others. With the role object pattern, the type system cannot be relied on to enforce the constraints. Runtime checks will have to be used instead.

- Maintaining object identity becomes more complex. The core object and its role instances form a conceptual unit, which should have a conceptual identity of its own. While technical object identity can be handled directly in any given programming language (checking objects for technical identity is carried out by comparing object references), checking for conceptual identity requires additional operations.

Another pattern that allows for the incorporation of subjectivity is the Extension Objects pattern [Gamma97], which in a similar fashion to the Role Object pattern (and with the same disadvantages), represents a key abstraction that plays different roles for different clients.

In these design patterns the change of context (and role) must be specified explicitly. A possible alternative for the handling of Dynamic Subjectivity in an implicit fashion, whereby the runtime context determines the role, would be the incorporation of sets of rules in the core object.

This would permit the definition of the runtime context depending on the state of the object, such that the selection of the role object is handled implicitly and at runtime. Additionally, this means that the client object does not have to have prior knowledge of the available roles.

On the other hand, this proposal increases the complexity of the core object. It also limits the handling of new contexts or modification of existent rules, as recompilation would be necessary.

4. The support of Dynamic Subjectivity with Computational Reflection

This section presents the concept of computational reflection and a framework for meta-object support that provides a reflection mechanism for the Smalltalk 80 language. A reflective meta-level architecture that supports dynamic subjectivity is then presented and described.

4.1. Computational Reflection

Reflection is the capability of a computational system to reason about and act upon itself and adjust itself to changing conditions [Maes87]. The computational domain of a reflective system is the structure and the computations of the system itself.

A reflective system incorporates data representing static and dynamic aspects of itself; this activity is called reification. This self-representation makes it possible for the system to answer questions about and support actions on itself. In a reflective architecture, a computational system is viewed as incorporating an object part and a reflective part. The task of the object computation is to solve problems and return information about an external domain, while the task of the reflective is to solve problems and return information about the system itself.

The components that deal with the self-representation and the application reside at two different software levels: the meta-level and the base level, respectively. Components that deal with the functionality of the application are at the base level. Similarly, components that deal with the application's self-representation are at the meta-level.

- The *Base Level* contains program objects that solve a problem and return information about the application domain. According to Maes [Maes87] this is the external system domain.

- The *Meta-level* or *Reflective Level* is formed by objects that carry out computation about the computational system materialized by the objects at the base level. The computational domain, or internal system domain, deals with the information relative to the structures and mechanisms that carry out the program execution.

Both levels are related in such a way that changes at the base level are reflected at the meta-level, in a causal connection way [Maes86]. The meta-level has access to the information at the base level, but the base level does not have any knowledge about the meta-level.

4.2. Luthier MOPs: A Framework for the Support of Meta-Objects

The framework for the support of meta-objects, called Luthier MOPs (Meta Object Protocols), provides a flexible infra-structure for the association of meta-objects with classes, instances or specific methods. A meta-object can be associated with a method, a group of methods (of the same or different classes), an instance or a group of instances of different classes, with a class or a group of classes. A method, instance or class may be associated to various meta-objects [Campo97].

The most relevant aspect introduced by Luthier MOPs is the mechanism that associates and activates the meta-objects, called meta-object managers. A meta-object manager determines how meta-objects are associated with base-level objects and how those meta-objects are activated, hiding the mechanisms used to implement the reflective behaviour. When a reflected object receives a message, this message is captured and sent to the associated manager. The manager decides whether to transfer the control to a meta-object or execute the original method.

This framework for meta-object support provides a flexible and reusable reflection mechanism to the Smalltalk-80 system. The framework provides an infra-structure over which different policies of meta-object management can be built, based on managers and method interception models [Campo98].

4.3. A Reflective Meta-level Architecture for the Support of Subjectivity

In this work we propose a reflective meta-level architecture for the incorporation of dynamic subjectivity to an object-oriented system. The Luthier MOPs framework is used to support the reflective mechanism.

The proposed architecture consists of two levels: a base level and a meta level. The base level of this architecture consists of the (existent) object-oriented application. The meta-level manages the subjectivity of the system by means of rules that determine the runtime context and invoke the corresponding subjective behaviour.

Figure 2 presents a scheme of the architecture, which shows the association between the base and meta levels. The two-ended arrows indicate the interception of messages by the reflection mechanism, and the simple arrows show that the meta-object delegates the intercepted messages to the context-specific-view objects.

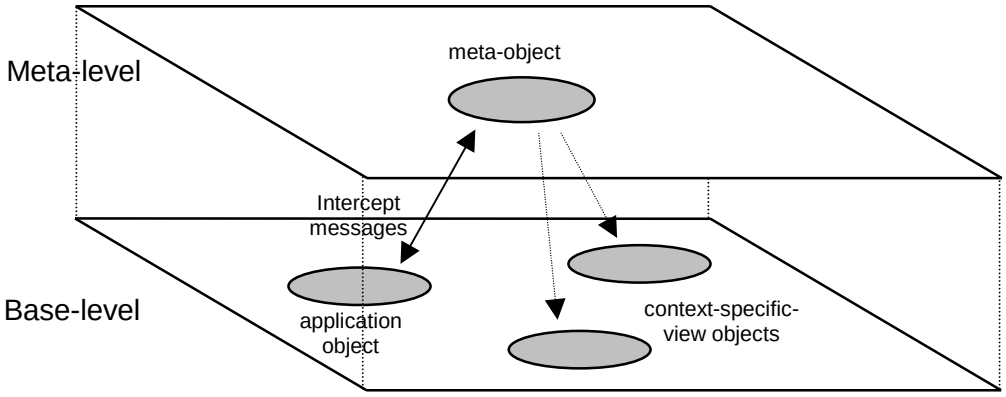

Figure 2. Reflective architecture behaviour

Each instance of a class of the application to which one wishes to add dynamic subjectivity has an associated meta-object. This meta-object maintains a set of rules that represent the different contexts pertaining to the associated object. Additionally, the meta-object contains the references to all the objects that correspond to the different subjective behaviours.

At execution time, when an object at the base level receives a message the reflection mechanism intercepts the message. The reflection mechanism then redirects the thread of control to the meta-object associated with that object. At the meta-level, the meta-object does its corresponding evaluation of the context, and sends the intercepted message to the context-specific-view object. When its execution finishes, the reflection mechanism returns the thread of control to the method that was intercepted.

Meta-objects can be added dynamically in the following ways:

ReflectionManager **reflectEachInstanceOf**: aBaseObjectClass **on**: aMetaObjectClass.

ensures that for each new instance of aBaseObjectClass a new instance of aMetaObjectClass will be created with the corresponding association between the new instances.

ReflectionManager **reflectObject**: aBaseObject **on**: aMetaObjectClass

this method creates a new instance of aMetaObjectClass for the specific object aBaseObject, establishing the corresponding association.

Similar methods are used to dynamically remove the meta-objects from the associated objects in the base level.

Figure 3 presents the example already described in Section 2, of a sales company whose employees can simultaneously carry out more than one role (salesman, manager, administrative staff).

In this example, Employee A is simultaneously a Salesman and a Manager, and Employee B is Administrative staff. Each of the employees will be associated with a meta-object at the meta-level. The meta-object associated to Employee A intercepts all the messages it receives, and evaluates the runtime context of the message. According to this evaluation, the message is delegated to the corresponding context-specific-view object, in this case either Salesman or Manager.

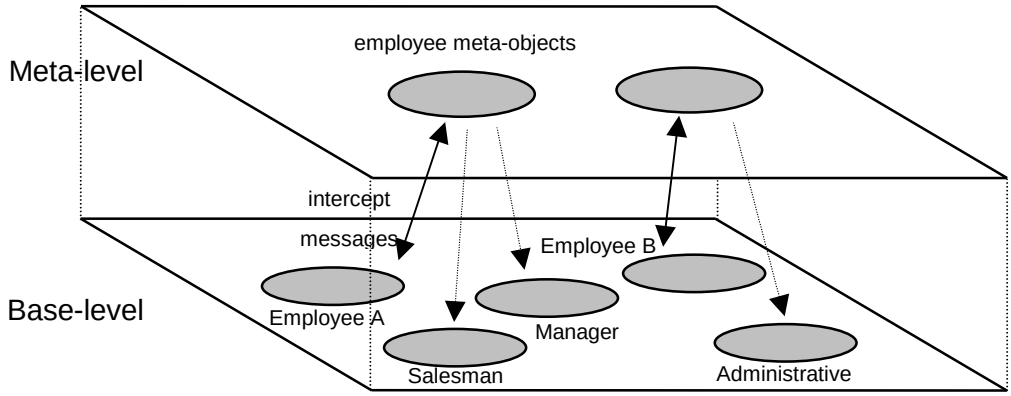

Figure 3. Reflective architecture behaviour in the example application

This reflective meta-level architecture that adds dynamic subjectivity to a system has the following advantages:

- It supports the desired characteristics that the incorporation of subjectivity offers.
- The subjective behaviour is managed at a meta-level in a transparent fashion, without the need for modifications to the base-level application.
- The handling of the subjectivity at a meta-level results in a flexible mechanism that permits the addition and modification of the rules that define the contexts.
- As the subjectivity is determined at a meta-level, the system may respond subjectively (at runtime) to a wide scope of factors. These factors include the message sender, the message arguments, the internal state of an object, and the general context such as the user, the sub-system and any other characteristic that can be evaluated by a set of rules.

- The combinatorial explosion of classes through multiple inheritance is avoided.
- It can handle constraints between the different subjective behaviours, by implementing them in the meta-objects.
- The client objects do not require any coding overhead in order to determine the subjective behaviour, as is the case in the Role Object pattern.

5. Conclusions

This paper presents an overview of subjectivity and different approaches that support it. These approaches are analyzed, and modifications that would enable subjectivity to be implemented dynamically are suggested.

A reflective meta-level architecture that permits the incorporation of dynamic subjectivity to a system is described and evaluated. This design handles dynamic subjectivity at a meta-level without the need to modify the object-oriented application which resides at the base level. Additionally, a wide scope of subjective factors can be handled dynamically with this architecture.

In conclusion, the proposed architecture has the advantages of adding dynamic subjectivity to classical object-oriented systems, in addition to a greater versatility and transparency due to the mechanism with which it is handled.

This architecture is being implemented with concrete applications in order to analyze and evaluate its effectiveness and efficiency.

References

[Bäumer97] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. “The Role Object Pattern”. In Proceedings of the 1997 Conference on Pattern Languages of Programs (PLoP ’97).

[Campo97] Marcelo Campo. “Compreensao Visual de Frameworks a través da Introspecao de Exemplos”. Ph.D. Thesis. UFRGS, Porto Alegre, Brasil., 1997. (in portuguese).

[Campo98] Marcelo Campo and Tom Price. “Luthier: : A Framework for Building Framework-Visualization Tools”. To appear in “Object-Oriented Application Frameworks”. Mohamed Fayad, Ralph Johnson (Eds.), John Wiley & Sons, USA, estimated November 1998.

[Chambers95] Craig Chambers. “The Cecil Language: Specification and Rationale, Version 2.1”. Technical report, 1995.

[Fowler97] Martin Fowler. “Dealing with Roles”. http://ourworld.compuserve.com/homepages/Martin_Fowler

[Gamma95] Gamma E., Helm R., Johnson R., Vlissides J. “Design Patterns. Elements of Reusable Object Oriented Software”. Addison Wesley. 1995.

[Gamma97] Erich Gamma. “Extension Object”. In “Pattern Languages of Program Design 3”. Robert C. Martin, Dirk Riehle, and Frank Buschmann (eds.). Addison-Wesley, 1998.

[Harrison93] William Harrison, and Harold Ossher. “Subject-Oriented Programming (A Critique of Pure Objects)”. In Proceedings of OOPSLA ’93.

[Harrison94] William Harrison, Harold Ossher, Randall B. Smith, and David Ungar. “Subjectivity in Object-Oriented Systems. Workshop Summary”. OOPSLA 1994.

[Harrison95] William Harrison, Harold Ossher, and Hafedh Mili. “Subjectivity in Object-Oriented Systems. Workshop Summary”. OOPSLA 1995.

[Kaplan96] Matthew Kaplan, Harold Ossher, William Harrison, and Vincent Kruskal. “Subject-Oriented Design and the Watson Subject Compiler”. In Proceedings of the 3rd Workshop on Subjectivity in Object-Oriented Systems. OOPSLA 1996.

[Kristensen95] Bent Bruun Kristensen. “Position Paper: Subjectivity and Roles”. In Proceedings of the 2nd Workshop on Subjectivity in Object-Oriented Systems. OOPSLA 1995.

[Maes86] Maes P. “Reflection in an Object Oriented Language”. AI MEMO n. 86-8. Artificial Intelligence Laboratory. Vrije Universiteit Brussel. Brussel, Belgique, 1986.

[Maes87] Maes, P. “Concepts and Experiments in Computational Reflection”. In Proceedings of OOPSLA ’87.

[Nolan96] John Nolan and Bruce Anderson. “Birds-of-a-Feather (BOF) session on subjectivity at Object Technology ‘96”. Object Technology 1996, organized by the British Computer Society. Oxford, England. 1996.

[Nolan97] Nolan, J. “Dynamic Subjectivity”. Object Expert. Vol 2 (3). March-April 1997.

[Richardson91] J. Richardson, and P. Schwartz. “Aspects: Extending Objects to Support Multiple, Independent Roles”. ACM-SIGMOD, International Conference on Management of Data. Denver, Colorado. 1991.