# THE PECULIARITIES OF SOFTWARE COMPOSITION MODELS

K. Chandra Sekharaiah
Distributed Object Systems Group
Department of Computer Science & Engineering
JNTU College of Engineering
Hyderabad 500 072
shakes123@sify.com

D. Janaki Ram
Distributed Object Systems Lab
Department of Computer Science & Engineering
Indian Institute of Technology
Madras India 600 036
djram@lotus.iitm.ernet.in

Mohd. Abdul Muqsit Khan
Department of Computer Science
Moulana Azad National Urdu University
Hyderabad India 500 032
abdul_muqsit_khan@yahoo.co.in

## ABSTRACT

*Concerns are at the core of software engineering and composition. Concerns apply in terms of objects, methods, subjects, aspects, roles. This paper explores such various concerns from a comparison perspective. It concludes that none of the software composition techniques explored so far give adequate treatment in addressing the object schizophrenia problem for a complete solution. Our work is related to HRI and robot motion controls based on software composition.*

## Keywords

Roles, Subjects, Aspects, Object Schizophrenia Problem (OSP), Human Computer Interaction (HCI)

## 1. INTRODUCTION

Object-orientation provides latest development methodology in software engineering. In this methodology, the decomposition of a system is based on the concept of an object. Inheritance and object composition are two basic, standard composition models of object-oriented languages [1, 2]. Delegation enriches object composition with inheritance properties.

Delegation is considered more powerful than class-inheritance because the latter can be simulated using the former but not vice versa. Object state is shared among instances in prototype systems whereas it is not so in class-based systems. On a contrary note, Stein demonstrated with a model for class-inheritance that it also captures delegation [3]. Delegation and class-inheritance are considered equally powerful. When inheritance mechanism is emulated using message passing, a user is not allowed to designate another object to reply in place of the original object. This is known as the "self problem". In [4], inheritance is said to breach encapsulation whereas in [5], it is argued that delegation breaches encapsulation [6]. Delegation-based languages such as Self allow more flexible composition. But, they mainly focus on how to share common functions at the instance level and have little intention of letting two or more objects combine together to form a larger structure. Basically, delegation is unidirectional and two objects related by delegation relation preserve their relatively independent nature.

Contracts [7] is a construct for the explicit specification of behavioral compositions. A contract defines a set of communicating participants and their contractual obligations. This notion of *participants* corresponds to roles but participants are actually objects and thus the separation of objects and roles is somewhat blurred. A contract specifies a composition at the class level and by instantiating a contract, a behavioral composition is created. In that sense, compositions of the contracts model remain in the conventional object-oriented framework.

In the realm of role modeling, Gottlob et al. [8] do not use the word composition. In their model, a role type hierarchy paves the way for the creation of role instance hierarchies. Inheritance is at object level. This is seen as composing a role instance with an object instance that already exists. The role hierarchy specifies the route of delegation at the instance level. The delegation hierarchy could also be interpreted by composition, i.e. a role at the node of the tree can be regarded as a composite packing of nodes along the path from the role node to the root.

Recently, different software composition techniques such as roles, subjects and aspects have aroused much interest among the object community. These techniques provide different kinds of modules that encapsulate different concerns of a system. Separation of concerns can provide many software engineering benefits such as reduced complexity, improved reusability, and simpler evolution.

Separation of concerns provides modular design. In object orientation, modularization is by class (or object). Hence, changing a data representation in the system is well localized. Objects of different types can provide a natural separation of concerns. However, objects are 'not like islands' and always exist in relation with other objects. Conventional object-oriented programming languages support a one-dimensional space wherein all concerns are "class" concerns. However,

a second dimension is necessitated for providing separation of concerns in modules that qualify objects. Roles, subjects, and aspects provide three distinct approaches for modeling separation of concerns.

Separation of concerns is also aimed at eliminating the invasive changes to many classes when a new feature is added to a system. Generally, adding a new feature to a system leads to scattering of feature code across multiple classes. Also, the feature code is tangled with other code among these classes. The term feature refers to either methods or attributes. Different modularizations are needed for different purposes in the form of class, feature, aspect (e.g. distribution or persistence), role, variant or other criterion. Subjects and aspects provide modules that cut across classes which are the dominant composition-decomposition mechanism in object-oriented languages.

Stakeholders are entities related with the software development and prevail at all stages of the development process. Stakeholders determine the kind of separation of concerns to be employed in the system development. Stakeholders at various phases of the software development may hold idiosyncratic viewpoints. Concerns are related to different stakeholders. Stakeholders work with their concerns depending on their role in the system development life cycle. Clients for the system, its users, developers, operators, vendors and so on form the stakeholders of the system. A concern space is the union of concerns of all the stakeholders. Generally, a view is a partial model of a system. Viewpoints are fixed items. For example, the functional viewpoint and the data viewpoint are two kinds of viewpoints in structured design methods. There are no definite, formal definitions for concerns and viewpoints in the literature. View is an instance just as viewpoint is a type. A view of a system typically addresses one or more concerns. Viewpoints are stakeholder-dependent. They describe the requirements from the perspective of a stakeholder. The concerns are related to the current context of the system with which the stakeholders work.

It is rather not possible to ensure separation of concerns in true sense of the term. Concerns are, often, not altogether independent or "orthogonal". In practice, a support for interacting concerns is necessary even as useful separation is achieved. Concerns exist on the sender side and the receiver side. The forces of state, sender and context on the sender's side given in [9] are but different concerns in system development. Some of the concerns are synchronization constraints, addressing the history information, evolution of behavior, coordinated behavior, security and reliability measures, real-time behavior, distribution aspects, multiple views concerns, persistence, concurrency, performance etc.

In [10], the concept of composition of different concerns must be also applied during the software development process. Certain design concerns such as access control, synchronization and object interactions cannot be expressed in current OO languages as separate software modules.

Objects are a means of providing separation of concerns. Different software composition models have been built based on object orientation. They capture separation of concerns in different ways with various modeling requirements, composition semantics and complexity. Subject-oriented composition, aspect-oriented composition and role-oriented composition have come up as extensions to the basic object-oriented composition mechanism. [11] compares and contrasts such concerns as roles, subjects and aspects. These composition models are explained below. The composition models are prone to object schizophrenia problem. They are closely related to role modeling and yet maintain different features. A solution approach to OSP is significant to reduce the modeling complexity with separation of concerns.

Schizophrenia is increasingly becoming an investigative topic amongst cognitive science researchers including linguists [12], computer scientists, neuroscientists, software composition and modeling community [13, 14, 15]. The term has etymological significance as explained below.

Schizophrenia = *Schiz* or *Schism* as the root word + *phreno* as the root word

The root word *schiz* or *schism* denotes the splitness or dividedness property in things. It means that the thing of relevance exists as fragmented, split or divided. *'phren'* is another root word in the term. It means 'related to the brain'. Taken together, the term refers to the split issues related to the structural and functional semantics of the memory and processing systems in human beings.

The rest of the paper is organized as follows. The next two sections explain how separation of concerns is facilitated or eliminated in terms of composing and decomposing them in the concerns space. In section 4, the fundamentals of object schizophrenia problem are explained. In section 5, various facets of object-oriented composition are detailed. Sections 6, 7, 8, and 9 highlight the failure of the composition techniques involving subjects, aspects, glues and roles respectively to provide a complete solution to the object schizophrenia problem. Section 10 discusses approaches to human-computer interaction. Section 11 gives the directions for future research in the arena of human-robotic interface. Section 12 concludes the paper.

## 2. OBJECT SCHIZOPHRENIA PROBLEM: WHAT IT MEANS

The notion of schizophrenia has been explored in software systems such as behavior systems which include object-oriented systems [13, 14, 15] and linguistic systems [12]. In this paper, we explore to what extent various software composition techniques address solu-

tion to the object schizophrenia problem.

Object schizophrenia problem is defined as "a condition of an object which is under object schizophrenia and also characterized by such symptoms as broken delegation (BD), broken assumptions (BA), doppelgangers (DG), wrong method interpretation (WMI) due to message forwarding mechanism, security problem (SP) due to message forwarding mechanism etc". Thus, OS is a necessary but not sufficient condition for OSP. OSP is an offshoot of OS, but not the vice versa. In [16], three symptoms of OS are provided as broken delegation, broken assumptions and doppelgangers. In this paper, they are categorized as symptoms of OSP to avoid the confusion between OS and OSP. OSP symptoms are explained in the context of role modeling. Message forwarding mechanisms such as *delegation* and *consultation* are main causes of OSP. A role model has to provide non-intrusive evolution (problems can be addressed at without modifying the existing code) of objects playing roles. If OSP symptoms prevail, this requirement can not be met. Hence, a role model has to be free from the OSP. In terms of the identity semantics, split identity is sure to result in OSP. Broken identity may or may not result in OSP.

***OSP = OS + at least one symptom such as wrong method semantics, insecure message forwarding, broken contracts (for e.g., in a role hierarchy), violation of the identity principle, etc.***

## 3. DECOMPOSING CONCERNS: PROVIDING SEPARATION OF CONCERNS

In the context of software decomposition, the notion of *concern* is defined as a domain used as a decomposition criterion for a system or another domain with that concern [17]. Concerns can be used during analysis, design, implementation, and refactoring. Presently, in software construction, there are many types of decomposition such as functional decomposition, separation of interface and implementation, separation of implementation and policy, separation of responsibilities, and decomposition based on various data structures.

A composite object can be decomposed. The set of part objects is a result of decomposition of a whole concept. Nevertheless, decomposition is not supported if it is available only implicitly as the result of aggregation. No language supports decomposition. However, it is possible to design language constructs that support this process [18]. Object identity combines the distinct notions of the facility of object reference and the facility of object comparison. Object reference permits object correlation and access to object's internal states. Object comparison facilitates to decide if two variables actually point to the same object. The two OID concerns are separated in such patterns as the Decorator pattern [19]. The Decorator pattern can be employed to dynamically attach additional responsibilities to objects. By separating the two notions of object reference and object comparison that are usually subsumed in the concept of object identity, we can greatly increase the expressiveness of a corresponding model. Since the Decorator pattern can be seen as a kind of role model, and role models are known to be closely related to the concepts of *aspects* and *subjects*, separation of the OID concerns may further improve the possibilities of separation of concerns. With the usage of the Decorator pattern, one cannot rely on object identity. In [19], "...from an object identity point of view, a decorated component is not identical to the component itself."

## 4. COMPOSING CONCERNS: ELIMINATING SEPARATION OF CONCERNS

Issues such as reuse concerns and identity concerns are important in composition. Multi-dimensional separation of concerns is a new sub-branch of software engineering. Its goals are to enable: one, encapsulation of all kinds of concerns in a software system such that dynamic (re)selection of concerns is permitted; two, overlapping and interacting concerns; three, on-demand remodularization, and to encapsulate concerns that are identified anew in the software life cycle.

Separation of concerns is a notion that is at the core of software engineering since Parnas' seminal work in [20]. It refers to the ability to identify, encapsulate, and manipulate such parts of software that are relevant to a particular concern (concept, goal, purpose etc.). Concerns are the principal motivation for organizing and decomposing software into manageable and comprehensible parts. Different developers who work at different phases of the software life cycle have different kinds of concerns. For instance, class is an important concern in object-oriented programming. It encapsulates data concerns and behavior concerns together. Concerns such as printing, persistence, and display capabilities are known as feature concerns even as there are other concerns such as aspects, roles, variants, and configurations. Proper separation of concerns reduces software complexity, improves comprehensibility, promotes traceability, and facilitates reuse, non-invasive adaptation, customization, and evolution, and simplifies component integration.

## 5. OBJECT-ORIENTED COMPOSITION

Basically, system development involves stakeholders at various stages of the development process. The development life cycle itself consists of such phases as analysis, design, implementation, coding and testing. At language level, a particular language may not be able to to meet some of the specifications. This is because a language is set toward a particular goal set. Language development is the process of modifying it to satisfy the trends in programming. In [21], the four programming languages Oberon-2, Modula-3, Sather and Self are compared as to how support for inheritance, dynamic dispatch, code reuse, and information hiding is provided in very different ways and with different levels of efficiency and simplicity. In [18], a comparison of object-oriented languages such as Smalltalk , Beta [22], C++, Eiffel [23], CLOS [24],

Self [25], Objective C [26] Ada, SIMULA [27] and Object Pascal [28] was made. The framework could be used to measure the extent to which a programming language supports conceptual understanding. As recalled in [18], in [29], the language definition includes: "A language *supports* a programming style if it provides facilities that makes it convenient (reasonable, easy, safe, and efficient) to use that style. A language does not support a technique if it takes exceptional effort or skill to write such programs; in that case, the language merely *enables* programmers to use the technique."

In class-based languages, inheritance is supported by subclassing, whereas in prototype-based languages, it is supported by delegation. Often, inheritance is used to represent composition [30]. In spite of a tendency to equate reusability with inheritance, in [31], composition of one object from other objects is another important form of reusability. In [31], roles are suggested to enhance the reusability of object-oriented models. It was shown that objects can be less reusable if they are aware of their containment in *aggregates* (sets. lists, etc.). Roles allow us to satisfy behavioral protocols without a necessity to resort to modifying the type-hierarchy.

An object may consist of part objects, the components, that are physically part of a whole object. An object may also consist of components that are references to other objects. An object may also consist of such components as patterns. Furthermore, an object may contain components that are references to patterns. Usually, such references are closures that denote patterns and the lexical scopes. In the Scandinavian school on object orientation in [30], variance in support for composition in such languages as C++, Eiffel, Beta, CLOS and Smalltalk is detailed. The comparison perspective is in terms of part objects, object references, part patterns and pattern references.

In [18, 32], *decomposition* is the inverse process of aggregation. In [32], *aggregation* is the second main abstraction principle. The term *composition* is used as a synonym for aggregation. Aggregation refers to the principle of considering collections of things as single higher-level things called aggregates. Things are described in terms of parts and wholes and the principle represents the *has-a* relationship among things. In [33], things are the abstractions that are first-class citizens in a model whereas relationships tie these things together. The aforesaid abstraction principle in [32] is considered a special kind of association. Nevertheless, in [32], *grouping, association, partitioning* and *cover aggregation* synonymously represent the fourth abstraction principle.

In [34], composite concepts can be modeled from multiple perspectives. Role classes and role objects provide multiple perspectives on the aggregation of a whole with same atomic parts. Subjects [35] do not involve multiple aggregation hierarchies. The UML notation [33] considers aggregation and composition differently. Simple aggregation is entirely conceptual and does not change the meaning of navigating across the association between the whole and its parts. It does not link the lifetimes of the whole and its parts. Composition is a variation of simple aggregation with strong ownership and coincident lifetime as part of the whole. Composite aggregation is contrasted with simple aggregation. In a composite aggregation, an object may be a part of only one composite at a time. A part may be shared by several wholes. A *wall* may be part of multiple room objects. Further, the whole is responsible for the disposition of its parts. In other words, the composite must manage the creation and destruction of its parts. For example, in a windowing system, a created *Frame* must be attached to an enclosing *Window*. Destroying a *Window* object must in turn destroy its *Frame* parts. Composition is a special kind of association. A concept may be decomposed into multiple sets of concepts [36], i.e. several decompositions help to understand the same concept better. For instance, elephant may be decomposed into trunk, leg, body, head and tail; a second kind of decomposition of the elephant would be hair, meat, bone etc.

## 6. SUBJECT-ORIENTED COMPOSITION

Separation of concerns in a software system may be supported by roles i.e., by applying various subjects in the application domain. Subject-oriented programming [35, 37] is a program-composition technology that supports building object-oriented systems as compositions of subjects. A subject is a collection of classes or class fragments. The abstraction hierarchy of a subject may model its domain in its own, subjective way. A subject may be a complete application in itself, or it may be a complete fragment that must be composed with other subjects to produce a complete application. Subject composition combines class hierarchies to produce new subjects that incorporate functionality from existing subjects. The programming paradigm supports building object-oriented systems as compositions of subjects, extending systems by composing them with new subjects, and integrating systems by composing them with one another.

Like subject-oriented programming, subject-oriented design supports decomposition of object-oriented software into modules, called *subjects*, that cut across classes, and integration of subjects to form complete designs. In subject-oriented design, an object-oriented design model is divided into design subjects which encapsulate some concerns in an object oriented design.

A subject-oriented program is described by subjects. Subjects are object-oriented models which are composed according to a set of rules. Subjects provide partial views of a system. A subject is a context of its domain of application. In [16], subject composition is claimed to lead to no object schizophrenia since only one object identity is used. In [9], roles were used for subject composition. An object-role hierarchy is a subject.

Object has a "self". But, subject has no self. It is a good idea to maintain a design object that can contain

many selfs. SOP cannot model RPCM objects. Modeling an object as multiple subjects is different from modeling an RPCM object. An RPCM object is based on a single role hierarchy whereas modeling an object as multiple subjects means that multiple classification hierarchies might be prevalent in the design.

Object schizophrenia and object schizophrenia problem are addressed in [16]. However, the two notions are not well contrasted and not well elaborated. In [16], subject composition avoids broken delegation because only a single object identity is used. On similar grounds, subject composition avoids broken assumptions symptom of object schizophrenia. Many of the problems that arise on account of the prevalence of OS could be avoided if a master identity could be established and it could be ensured that no OID other than that of the master is ever used (except within the master object). Seemingly simple, this solution is, nevertheless, not workable in practice. For instance, to avoid the broken delegation symptom, the pseudo-variable "self" or "this" must actually refer to the master, even in the slave objects. Noticeably, most of the programming languages do not facilitate the enforcement of such a protocol.

## 7. ASPECT-ORIENTED COMPOSITION

Separation of concerns in a software system may be supported by aspectualization, i.e., by the expression of parts of that system in terms of separate aspects. Aspects crosscut. They include different use cases, collaborations, distribution, persistence, error detection/handling, logging, tracing, caching, requirements, features, qualities, perspectives, processes, implementation structures, synchronization etc. Like object-orientation and other separation of concerns technologies, aspect-orientation has implications throughout the software life cycle.

Aspects allow different aspects of a system to be described separately, and in different languages where this is appropriate. An aspect-oriented program comprises a number of *aspects* which are *woven* together to produce the relevant system. Aspect weaver composes different aspects to form a composition.

Aspect-Oriented Programming (AOP) [38] is a programming paradigm that is used where a concern that *cross-cuts* a group of objects is modularized as an aspect. Cross-cutting between *components* and *aspects* is a key feature of AOP. In Figure 1, $n$ components and $m$ aspects are depicted by vertical bars and horizontal bars respectively to represent cross-cutting. Filled circles represent *join points* where aspects interact with components. The same aspect may interact with all components as is the case with Am or only some of them as is the case with A1, and vice versa (for e.g. C2 and C1). This cross-cutting is the main feature of AOP. Aspects would not be called aspects if they do not cross-cut components. A compiler, called *weaver*, weaves aspects and objects together into a system. Concerns such as error-checking strategies, synchronization policies, resource-sharing, distribution concerns and performance optimizations are examples of aspects. AspectJ, AOP language, is an aspect-oriented extension to Java. A program in AspectJ is composed of aspect definitions and ordinary Java class definitions. An *aspect* is an AspectJ specific language extension to Java. AspectJ weaver weaves together Aspects and classes. Main language notions in AspectJ are *introduces* and *advises*. *Introduces* adds a new method in which cross-cutting code is described to a class that already exists. *Advises* modifies a method that already exists. *Advises* can append cross-cutting code to a specified method. In the approach in [39, 40], *introduces* weaving only adds a method interface and the body of the method is added through *advises* weaving. *Before* is used in order to append code before a given method. *After* is used in order to append code after a given method.

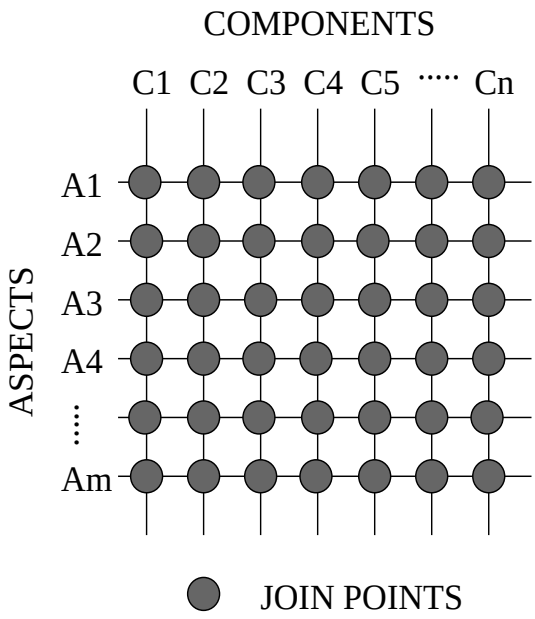


**Figure 1: Components and Aspects Crosscut in AOP**

Aspects are cross-cutting concerns over a set of objects. Join points should be easily located in systems. Join points are directly related to weaving. Weaving does not necessarily rely on weavers and is always a process achieved once or several times as it is presented in [38]. Role models synthesis and subject composition are examples of such processes. The rules and principles of weaving form the important feature of weaving. For example, subject composition rules [41], specification of roles as domains for activities [42], or even sharing specified by delegation links for split objects in [43] facilitate weaving. The notion of identity is an important notion implicitly implied in weaving in the sense that roles are attached to an object with a unique identity. An object might appear in several subject activations and yet has a unique identity used as a basis for subject composition whereas pieces [43] belong to an object with a unique identity. Absence of this feature leads to inherent object schizophrenia problems in the composition models.

In [39, 40], role model designs and implementations with AspectJ were proposed. In the model, an object has core/intrinsic attributes/methods. A role adds extrinsic attributes/methods and facilitates perspectives that can be used by other objects. The approach recommended is: (1.) introduce the interface for the role-specific behavior to the role class; (2.) advise the implementation of the role-specific behavior to instances of the core class; (3.) add role relationships and role contexts in the aspect instance. In AOP, an executable program can be constructed only by objects even if there are no aspects that add cross-cutting properties to objects.

The work in [40] presents aspect-oriented role model designs and implementations to provide a promising approach to reduce object schizophrenia, interface bloat, support of dynamic role assignment at an instance level and for flexible integration of object hierarchies. However, the claims are weak in that it is explicitly stated that the findings are preliminary. Basically, there is no clear notion of object schizophrenia and OSP in the literature. Our work brought out analytical results of contrast OS vis-a-vis OSP and to provide a sound basis for the two notions in the realm of role modeling.

## 8. ROLE-ORIENTED SOFTWARE COMPOSITION

Separation of concerns in a software system may be supported by roles i.e., by applying the role mechanism to objects either dynamically or statically. An object can play several roles. This provides systematic separation of concerns by describing different phenomena in different role models [44]. The OOram method [44] provides the notion of role model synthesis that supports the construction of complex models from simpler ones in a safe and controlled manner. Here, a role is an idealized description of an object in the context of a pattern of collaborating objects. The OOram method uses a policy of *divide and conquer* to focus on the object aspects that are relevant for the role model's area of concern. Role-based modeling supports the separation of functional aspects by means of contracts. The different roles a subsystem can play within a design can be described by distinct contracts. This is made possible by separating roles from each other. Roles are a well-known mechanism to provide multiple views, role-based approach to access control, object migration [45, 46], and composition-decomposition for object evolution [14]. Important properties for roles include visibility, dependency, identity, multiplicity, dynamicity, and abstractivity [14, 47, 48, 49, 13]. The properties were also explained in [8]. This set of properties are part of the role paradigm [14, 49]. Role paradigm conformance is a requirement for advanced role models. A role model which conforms to the role paradigm is called *role paradigm conformance model* (RPCM). The role models in [8, 48] are RPCMs.

In [14, 13], seminal work on OS and OSP is presented. Various kinds of OSPs were presented. Role Modeling Problem (RMP) is the problem of developing an RPCM that is also OSP-free. None of the existing role models have strong claims for a solution to RMP. Compositions in advanced role models are delegation-based. Objects with roles have to satisfy certain characteristics called role paradigm conformance characteristics. In the subject-oriented paradigm, only OID is necessarily shared. In advanced role models, composition with roles satisfies the identity property by following the OID integrity principle [13]. In advanced role models, object and its roles are supported for singular, isolated applications. Hierarchical organization of object behavior is essential. Object-role hierarchies are used. In role modeling, an object is not defined as composition of independently defined part objects. The role objects are not independent; they are existence-dependent on the player object. On the other hand, SOP supports decentralized development of objects. The class definition is decentralized. SOP supports multifarious behaviors spread across multifarious interrelated, interoperating, integrated applications. Classification hierarchies are application-specific. Hierarchical organization of object behavior is not a fundamental issue. An object is defined as composition of independently defined part objects, which can be dynamically added or removed.

Most of the role-oriented composition models use is-role-of inheritance [14] kind of reuse support in contrast with the traditional specialization and aggregation models which use white-box reuse and black-box reuse support respectively. Nevertheless, some of the early, rudimentary role models [50, 51] do not provide this inheritance model for objects with roles. Roles describe the responsibilities of objects in a collaboration, but how a role is actually modelled or specified is often left open. Whereas some design techniques model roles using interfaces [44], some others use as part of a behavioral contract between participants [7]. In contrast to the design idea of separating concerns as distinct collaborations, at run-time, things look much more complex. Several design collaborations may be interleaved and instances of the same class often play different roles in one call sequence. This makes it difficult to disentangle concerns and to reconstruct collaborations as they were conceived at the design stage.

## 9. GLUE-ORIENTED COMPOSITION

There have been many an attempt in the literature to provide clean separation between the concepts of interface and implementation. Some object-oriented programming languages do provide such separation. Languages Cecil [52] and its ancestor BeCecil [53] provide a clean separation between these two notions as well as between the two notions of subtyping and subclassing. Emerald [54] also distinguishes between types (interfaces) and classes (implementations), but does not support implementation inheritance. Theta [55] and Lagoona [56] are other languages that provide such a separation. Galileo [57] provides partial separation between interface and implementation. It is generally known that implementation inheritance implies interface inheritance [32, 58] whereas interface

inheritance does not include implementation inheritance [33]. Java is a language which supports either interface inheritance or implementation inheritance.

Glue model was proposed for object reuse by customization in object-oriented systems [59, 60, 61] providing interfaces and plug-and-play support. It supports multiple interfaces in a single class definition facilitating the multiple views approach specified in [62] to extend the OO paradigm. The class can consist of a public interface which captures the time-invariant behavior of the object and a set of Typeholes. A Typehole abstracts the variant or context-dependent behavior. It supports an interface for the object. It consists of a set of method declarations in the base class. The implementation itself for the Typeholes is provided in the *Glue* classes. A class with a Typehole can be instantiated even though its context-dependent behavior is not defined. An instance of a glue class provides a context-dependent behavior. During runtime, instances of the classes with the variant and the invariant behaviors can be composed.

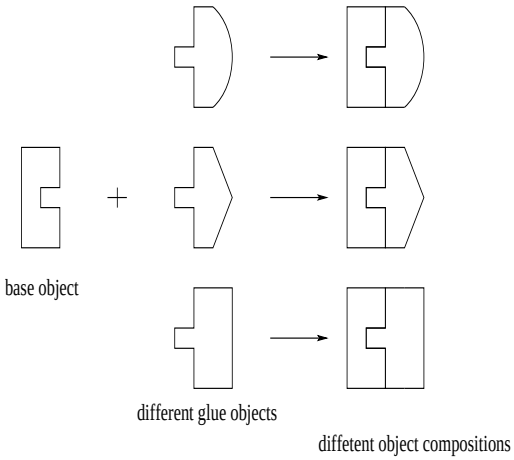


**Figure 2: Object Reuse by Customization: The Glue Model Approach for Multifarious Compositions**

Corresponding to a Typehole, there may be multiple glue classes. Specification of glue relationship does not directly result in the composition of the instances of the base class and the glue class. Two operators viz. *plug* and *unplug* are used to compose the base object and the glue object and to decompose the objects respectively. Facilitating the multifarious glue objects to be composed with one and the same base object by these operations results in object reuse by customization as shown in Figure 2. The base object and the glue object can be dynamically composed, decomposed and recomposed using these two operators. There are four compositional patterns supported by the model, viz. *in, out, part-of* and *using*. The base object and the glue object can be dynamically composed, decomposed and recomposed using these two operators. The model has been implemented with the proposed language constructs by extending the Java language. For this purpose, a parser has been written in Perl language. The parser takes glue code as input and generates pure Java code as output. It is designed to permit separate compilation of the classes with the basic and the variant behaviors and to permit late binding of their instances.

One of the kinds of OSPs is broken delegation. A broken delegation problem is said to arise if a *this* call is made in a Typehole method of a part object and that call is delegated to the glue object rather than the base object. In the case of Glue-oriented composition, a reference to the base object is always made available in the glue object. Any call on *this* is handled on the base object. A new pseudo-variable *glue* is introduced, and any call on *glue* is handled by the glue object. Nevertheless, glue model does not address a a complete solution to all the object schizophrenia problems. It addresses a solution approach to broken delegation only.

## 10. HUMAN-COMPUTER INTERACTION (HCI)

HCI is a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them [63]. The focus is on interaction and specifically between one or more humans and one or more computational machines e.g. GUI (an interactive graphics program support on a workstation). Instead of workstations, computers may be in the form of embedded computational machines, such as parts of spacecraft cockpits or microwave ovens. HCI studies both the mechanism side and the human side. A computer mouse, a touch screen, a program running on machine that includes a trashcan, icons of disk drives and folders, pulldown menus are different forms of HCI features.

The emerging HCI systems have such characteristics as ubiquitous communication, high functionality systems, mass availability of computer graphics, and interactive animation, mixed media (systems will handle images, voice, sounds, video, text, formatted data), high-bandwidth interaction, large and thin and light weight and low-power consumption displays (enabled by paper-like pen-based computer systems), embedded computation, group interfaces (e.g., for meetings, for engineering projects, for authoring joint documents), user tailorability in such application areas as characterization of application areas (e.g., individual vs. group, paced vs. unpaced), document-oriented interfaces (text-editing, document formatting, illustrators, spreadsheets, hypertext), communications-oriented interfaces: Electronic mail, computer conferencing, telephone and voice messaging systems, design environments (programming environments, CAD/CAM), online tutorial systems and help systems, multimedia information kiosks, continuous control systems (process control systems, virtual reality systems, simulators, cockpits, video games), embedded systems (copier controls, elevator controls, consumer electronics and home appliance controllers (e.g., TVs, VCRs, microwave ovens, etc.).

HCI has a common theme, to some extent, with ergonomics, cognitive systems and cybernetics. GUIs form a part of HCI. Human robotic interface (HRI) is emerging as a subtheme under HCI. Often, HCI involves transducers between humans and machines and because humans are sensitive to response times, viable human interfaces are more technology-sensitive than many parts of computer science. Out of the various aspects of HCI, we are currently involved in research about the human-robot interface from a software composition perspective because such a perspective will facilitate evaluation of various permutations and combinations of robot motion controls in the execution of a task.

## 11. FUTURE DIRECTIONS FOR RESEARCH: HUMAN-ROBOT INTERFACE (HRI)

HRI systems use a combination graphical and non-graphical language and dialogue styles and interaction metaphors (tool metaphors, agent metaphor) and content metaphors (desktop metaphor, paper document metaphor) and other media such as film, theater and graphic design.

An intelligent robot can determine its own behavior and conduct through functions of both transducer-based sensing and recognition. Human-robot communication is facilitated by discrete word recognition, teach and play-back and high-level programming languages [64]. Robot control languages such as WAVE, AL, AML, AUTOPASS, HELP, JARS and MAPLE [64] use a variety of features for software composition. We are exploring the merits and demerits of the various software composition approaches discussed in this paper for joint/wrist control of robots and the relevance of OSP in cybernetics. The various motion controls of robots could extensively use delegation feature in trajectory planning and it is in this regard that the work is explored.

## 12. CONCLUSIONS

New perspectives are brought out about understanding object-oriented software composition in terms of delegation, class inheritance, contracts, roles, subjects and aspects. The concerns space in software development is manipulated in terms of composition and decomposition. It is important to look into whether a software composition model is prone to object schizophrenia problem so that a solution for the same for that software composition model can be addressed. The survey shows that different composition models such as subjects and aspects give only partial treatment and partial solution approaches to the notions of object schizophrenia and object schizophrenia problem. Even as aspect-oriented implementations do exist for roles, there are only preliminary findings and weak claims as regards elimination of the object schizophrenia problem. The glue model shows a solution approach to only broken delegation. However, it does not address other problems that tend to arise due to object schizophrenia.

## 13. REFERENCES


[1] Klaus Ostermann and Mira Mezini, "Object-oriented composition untangled," *OOPSLA Proceedings in ACM Sigplan Notices*, pp. 283–299, 2001.

[2] Klaus Ostermann, "Dynamically composable collaborations with delegation layers," in *in Proceedings of 16th European Conference on Object-Oriented Programming*, (University of Málaga, Spain), June 2002.

[3] Lynn Andrea Stein, "Delegation is inheritance," *OOPSLA Proceedings in ACM Sigplan Notices*, pp. 138–146, Oct. 1987.

[4] Lieberman, H., "Using prototypical objects to implement shared behaviour in object-oriented systems," in *Proceedings of OOPSLA*, Oct. 1986.

[5] Patrick Steyaert and Wolfgang De Meuter, "A marriage of class- and object-based inheritance without unwanted children," in *ECOOP'95*, vol. 952, (Aarhus, Denmark), pp. 127–144, Springer-Verlag, Aug. 1995.

[6] Daniel Bardou, "Delegation as a sharing relation: Characterization and interpretation," in *Position paper at the workshop on Prototype-based Object-oriented Programming, ECOOP'96*, (Linz, Austria), 1996.

[7] Richard Helm, I.M. Holland, and D. Gangopadhyay, "Contracts: Specifying behavioral compositions in object-oriented systems," in *Proceedings of ECOOP/OOPSLA'90*, vol. 1, (Ottawa), pp. 169–180, June 1990.

[8] George Gottlob, Michael Schrefl, and Brigitte Röck, "Extending object-oriented systems with roles," *ACM Transactions on Information Systems*, vol. 14, pp. 268–296, July 1996.

[9] Bent Bruun Kristensen, "Subject composition by roles," in *Proceedings of the Fourth International Conf. on Object Oriented Information Systems(OOIS)*, (Brisbane, Australia), 1997.

[10] Mehmet Aksit, "Composition and separation of concerns in the object-oriented model," *ACM Computing Surveys*, vol. 28A, no. 4, 1996.

[11] D. Bardou, "Roles, subjects and aspects: How do they relate?," July 1998. Position paper at the Aspect Oriented Programming Workshop, ECOOP'98, Brussels, Belgium. Extended abstract published in ECOOP'98 Workshop Reader, Serge Demeyer and Jan Bosch, editors, Lecture Notes in Computer Science (LNCS), vol. 1543, Springer, 418–419, December 1998.

[12] Chandra Sekharaiah, K. and D. Janaki Ram, "The Dynamics of Language Understanding," in *Language Engineering Conference*, (Hyderabad, India), pp. 197–200, IEEE CS Press, Dec. 2002.

[13] Chandra Sekharaiah, K. and D. Janaki Ram, "Object schizophrenia problem in object role database system design," in *Object Oriented Information Systems (OOIS'02)*, (Montpellier, France), pp. 494–506, Springer Verlag, Sept. 2002.

[14] Chandra Sekharaiah, K. and D. Janaki Ram, "Object schizophrenia problem in modeling is-role-of inheritance," in *Inheritance Workshop, 16th European Conference on Object-Oriented Programming*, (University of Málaga, Spain), pp. 88–94, June 2002.

[15] K.Chandra Sekharaiah, "Modeling Objects with Roles," in *Ph.D. Thesis, Dept. of C.S.E. IIT Madras, India*, Jan. 2003.

[16] IBM Research: Subject-oriented programming group, "Subject-oriented programming and design patterns," (http://www.research.ibm.com/sop/).

[17] Krzysztof Czarnecki, Ulrich W. Eisenecker, and Patrick Steyaert, "Beyond objects: Generative programming," in *Position paper, ECOOP'97 Workshop on Aspect-Oriented Programming*.

[18] Bent Bruun Kristensen and Kasper Osterbye, "A conceptual perspective on the comparison of object-oriented programming languages," *ACM Sigplan Notices*, vol. 31, pp. 42–54, Feb. 1998.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns*. Addison-Wesley, 1995.

[20] D. Parnas, "On the criteria to be used in decomposing systems in modules," *Communications of the ACM*, vol. 15, pp. 1053–1058, Dec. 1972.

[21] Robert Henderson and Benzamin Zorn, "A comparison of object-oriented programming in four modern languages," *Software-Practice and Experience*, vol. 24, pp. 1077–1095, Nov. 1994.

[22] O.L. Madsen, B. Moller-Pedersen, and K. Nygaard, *Object-oriented programming in the Beta programming language*. Addison-Wesley, 1993.

[23] B. Meyer, *Eiffel: the language*. Prentice Hall, 1992.

[24] S.E. Keene, *Object-oriented programming in Common LISP*. Addison Wesley, 1989.

[25] Ungar, D. and R. B. Smith, "SELF: The power of simplicity," in *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages,and Applications*, (Orlando, Florida), pp. 227–241, Oct. 1987.

[26] B. J. Cox and A. J. Novobilski, *Object-oriented programming, an evolutionary approach 2/E*. Addison Wesley, 1991.

[27] O. J. Dahl, B. Myhrhaug, and K. Nygaard, *SIMULA 67 common base language*. Norwegian Computing Center edition February, 1984.

[28] *Macintosh programmer's workbench Pascal 3.0 reference*. Apple Computer, 1989.

[29] Stroustrup, B., "What is object-oriented programming," *IEEE Software*, pp. 10–20, May 1988.

[30] Ole Lehrmann Madsen, "Open issues in object-oriented programming- A scandinavian perspective," *Software-Practice and Experience*, vol. 25, pp. 3–43, Dec. 1995.

[31] Michael F. Kilian, "A note on type composition and reusability," *ACM OOPS Messenger*, vol. 2, pp. 24–32, July 1991.

[32] Antero Taivalsaari, "On the notion of inheritance," *ACM Computing Surveys*, vol. 28, pp. 438–479, Sept. 1996.

[33] Grady Booch, James Rumbaugh, and Ivar Jacobson, *The unified modeling language user guide*. Addison Wesley, 2000.

[34] Lars Kirkegaard Baekdal and Bent Bruun Kristensen, "Aggregation from multiple perspectives by roles," in *Proceedings of the TOOLS Pacific '99*, (Melbourne, Australia), 1999.

[35] William Harrison and Harold Ossher, "Subject-oriented programming (A critique of pure objects)," in *Proceedings of OOPSLA*, pp. 411–428, 1993.

[36] Bent Bruun Kristensen and Kasper Osterbye, "Conceptual modeling and programming languages," vol. 29, no. 9, 1994.

[37] Harold Ossher, William Harrison, Frank Budinsky, and Ian Simmonds, "Subject-oriented programming : Supporting decentralized development of objects," in *Proceedings of the 7th IBM Conference on Object Oriented Technology*, (Santa Clara, CA), pp. 570–577, July 1994.

[38] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming* (M. Akşit and S. Matsuoka, eds.), vol. LNCS 1241, pp. 220–242, Berlin, Heidelberg, and New York: Springer-Verlag, June 1997.

[39] Elizabeth Kendall, A., "Role model designs and implementations with aspect-oriented programming," in *Proceedings of OOPSLA*, pp. 353–369, 1999.

[40] Elizabeth Kendall, A., "Aspect-oriented programming for role models," in *Position paper, Proceedings of the ECOOP'99 Workshop on Aspect-Oriented Programming*, pp. 49–55, 1999.

[41] Harold Ossher, Matthew Kaplan, William Harrison, Alexander Katz, and Vincent Kruskal, "Subject-oriented composition rules," in *Proceedings of OOPSLA*, pp. 235–250, 1995.

[42] Bent Bruun Kristensen and Daniel C. M. May, "Activities: Abstractions for collective behavior," in *Proceedings of the 10th European Conference on Object Oriented Programming (ECOOP'96)*, vol. LNCS 1098, (Linz, Austria), pp. 472–501, 1996.

[43] Daniel Bardou and Christophe Dony, "Split objects: A disciplined use of delegation within objects," in *Proceedings of OOPSLA, ACM Sigplan Notices*, pp. 122–137, 1996.

[44] Trygve Reenskaug, *Working with objects: The OORAM software engineering method*. Manning PublicationsGreenwich, CT, 1995.

[45] Papazoglou, P. and B.J. Kramer, "A database model for object dynamics," *The VLDB Journal*, vol. 6, pp. 73–96, 1997.

[46] Roel Wieringa, Wiebren DeJonge, and P. Sprint, "Roles and dynamic subclasses: A modal logic approach," in *Proceedings of the European Conference on Object Oriented Programming*, pp. 32–59, 1994.

[47] Bent Bruun Kristensen, "Object-oriented modeling with roles," in *Proceedings of the Second International Conf. on Object Oriented Information Systems(OOIS)*, (Dublin, Ireland), pp. 57–71, Springer Verlag, 1995.

[48] Chandra Sekharaiah, K., D. Janaki Ram, and A.V.S.K. Kumar, "Typehole model for objects with roles in object-oriented systems," in *Fourteenth Europeon Conference on Object Oriented Programming (ECOOP 2000), LNCS 1964*, (Sophia Antipolis, France), pp. 301–302, Springer Verlag, June 2000.

[49] Chandra Sekharaiah, K., Arun Kumar, and D. Janaki Ram, "A security model for object role database systems," in *International Conference on Information and Knowledge Engineering (IKE'02)*, (Las Vegas, Nevada, USA), pp. 381–385, June 2002.

[50] Joel Richardson and Peter Schwarz, "Aspects: Extending objects to support multiple, independent roles," in *Proceedings of the ACM SIGMOD Int. Con. on Management of Data*, vol. 20, pp. 298–307, May 1991.

[51] Barbara Pernici, "Objects with roles," in *IEEE/ACM Conference on Office Information Systems ACM SIGOIS*, vol. 1, (Cambridge, Massachutes), pp. 205–215, Apr. 1990.

[52] Craig Chambers and Gary T. Leavens, "Typechecking and modules for multimethods," *ACM Transactions on Programming Languages and Systems*, vol. 17, pp. 805–843, Nov. 1995.

[53] Craig Chambers and Gary T. Leavens, "BeCecil, A core object-oriented language with block structure and multi-methods: Semantics and typing," in *Proceedings of The Fourth International Workshop on Foundations of Object-Oriented Languages*, (Paris), Jan. 1997.

[54] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchison, and Eric Jul, "Emerald: A general-purpose programming language," *Software Practice and Experience*, vol. 21, pp. 91–118, Jan. 1991.

[55] Day M.,, R.Gruber, Barbara Liskov, and A.C. Myers, "Subtypes vs where classes: Constraining parametric polymorphism," *SIGPLAN Notices*, vol. 30, pp. 156–168, Oct 1995.

[56] M. Franz, "The programming language lagoona - a fresh look at object-orientation," *Software - Concepts and Tools*, vol. 18, pp. 14–26, 1997.

[57] Antonio Albano, Luca Cardelli, and Renzo Orsini, "Galileo: A strongly-typed, interactive conceptual language," *ACM Transactions on Database Systems*, vol. 10, pp. 230–260, June 1985.

[58] Steve Vinoski, "CORBA: Integrating diverse applications within distributed heterogenous environments," in *IEEE Communications*, vol. 14, Feb. 1997.

[59] Anjaneyulu, P. and D. Janaki Ram, "Seamless integration of mobility into distributed systems using glue components," in *Proceedings of Ninth Annual IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM'98)*, (Delaware, USA), pp. 65–75, Oct. 1998.

[60] Janaki Ram, D. and O. Ramakrishna, "The glue model for reuse by customization in object-oriented systems," in *Tech. Report no IITM-CSE-DOS-98-02 (Communicated after first revision to Software Practice and Experience)*, (IIT Madras, Chennai, India), 1998.

[61] Janaki Ram, D. and Chitra Babu, "A framework for dynamic client-driven customization," in *Proceedings of Object Oriented Information Systems (OOIS)*, pp. 245–258, 2001.

[62] John Shilling, J. and F. Peter Sweeney, "Three steps to views: Extending the object-oriented paradigm," in *Proceedings of OOPSLA*, vol. 24, pp. 352–361, Oct. 1989.

[63] Hewett, Baecker, Card, Carey, Gasen, Mantei, Perlman, Strong, and Verplank, "ACM SIGCHI curricula for human-computer interaction, url: http://sigchi.org/cdg/cdg2.html/2_1."

[64] K.S.Fu, R.C.Gonzalez, and C.S.G.Lee, *Robotics: Control, Sensing, Vision, and Intelligence*. MC-GrawHill International, 1987.