Research Topics in Software Composition [1]

Oscar Nierstrasz

University of Berne[2]

Abstract. Traditional software development approaches do not cope well with the evolving requirements of open systems. We argue that such systems are best viewed as flexible compositions of "software components" designed to work together as part of a *component framework* that formalizes a class of applications with a common software architecture. To enable such a view of software systems, we need appropriate support from programming language technology, software tools, and methods. We will briefly review the current state of object-oriented technology, insofar as it supports component-oriented development, and propose a research agenda of topics for further investigation.

1 Open Systems

Modern applications can be characterized as *open systems* in terms of topology, platform and evolution. Of these, the most difficult requirement to meet is the third, since it states, in effect, that all application requirements cannot be known in advance due to changing needs and technological demands. Traditional application development methods are inadequate to the extent that they assume closed and fixed requirements, and therefore result in inflexible systems that cannot be easily adapted to unexpected changes in requirements [10].

We can distinguish between Open Systems Requirements that are essentially *computational* (i.e., dealing with functionality, distribution, concurrency, etc.), and those that are essentially *compositional* (i.e., dealing with evolution, interoperability, etc.). A complete approach should take both kinds of requirements into account.

Current object-oriented technology and methods provide only limited kinds of support for both kinds of requirements. Object-oriented languages and models supporting concurrency and distribution are either experimental, or adopt approaches that are poorly integrated with an object model. Actual component models, such as CORBA, SOM, COM, are designed to address interoperability rather than software composition, and are not intended to support application evolution. Object-oriented Analysis and Design methods provide little or no explicit support for software reuse, evolution, or the development, use and adaptation of frameworks and component toolkits.

---

1. In *Proceedings, Langages et Modèles à Objets*, A. Napoli (Ed.), Nancy, Oct. 1995, pp. 193-204.

2. *Software Composition Group,* Institut für Informatik (IAM), Universität Bern, 3012 Switzerland.
E-mail: oscar@iam.unibe.ch *WWW:* http://iamwww.unibe.ch/~scg

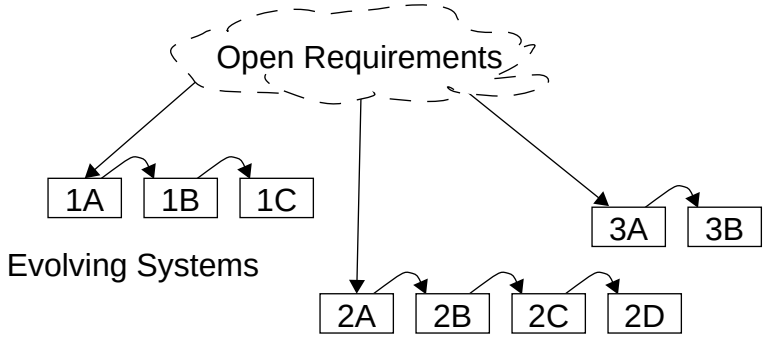

Figure 1 Open Systems as Families of Applications

In a sense, we need to view Open Systems as *families* of applications rather than as individual applications themselves. A truly open system can only be built by *anticipating* requirements evolution. An individual system, therefore, can be seen as an *instance* of a generic family of applications.

We shall continue by attempting to define software composition and explain its relevance to open systems development. Then, in sections 3, 4 and 5, we will consider briefly the open research issues in supporting compositional software development through languages, tools and methods. In our concluding remarks we summarize a research agenda.

2 Software Composition

Before attempting to define software composition, let us consider some approaches that seem to work well for composing applications in specific application domains:

4GLs:

- a specialized, high-level programming language characterizes a class of applications; typical 4GLs facilitate the rapid development of forms-based user-interfaces to database applications

Application Generators:

- a high-level language or interface describes application configurations
- source code is generated

Component Toolkits and Builders:

- a high-level language or graphical tool is used to combine and configure existing software components

Object-Oriented Frameworks:

- standard software components are specified as abstract/concrete object classes
- applications are programmed by specializing and extending framework classes

If we consider each of these approaches in turn, we can readily see a number of significant commonalities:

- the application domain is well-understood
- a generic software architecture captures families of applications
- parameterized software components are designed to be specialized or instantiated to meet specific requirements

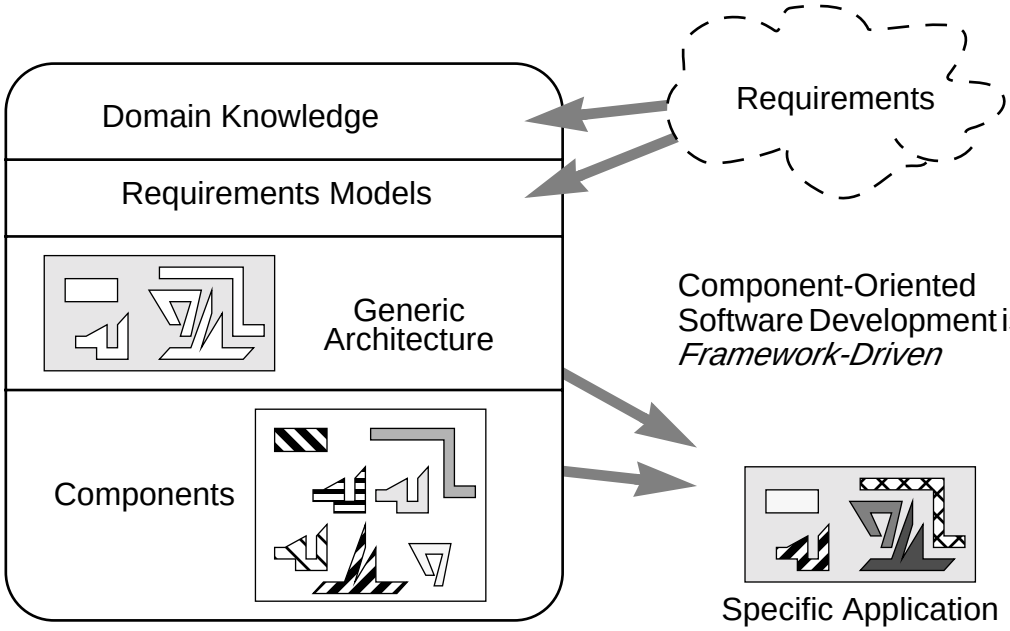

Figure 2 Component-Oriented Software Development

- the path from requirements collection to implementation is reduced to a recipe or formula (as much as possible)

Each of these points is true to a lesser or greater degree depending on how specialized or general the approach is. For example, software components are clearly visible in the latter three approaches, but are mostly hidden behind the language in a 4GL. On the contrary, the software development path is most streamlined with a 4GL, and less evident with a framework, since detailed knowledge of the implementation details of a framework is typically required before one can use it to build a specific application. The more general the approach, the more difficult it seems to apply.

Although each of these approaches works well in certain cases, they each have clear limitations:

- openness is hard to achieve: integration with external components, adaptation to new requirements are not necessarily straightforward
- approaches not easy to combine: the paradigms are largely disjoint; even combining two frameworks or class libraries can pose technical difficulties
- software architecture not always explicit: component toolkits are perhaps the best at making visible at a high level the relationships and interactions between software components; frameworks tend to hide these
- trade-off between convenience and generality: the most convenient techniques to use (e.g., 4GLs) tend to be the least general, and vice versa (frameworks)

2.1 Component Frameworks

The natural question to ask is: can we learn from and generalize these approaches so they can work well for arbitrary application domains? In the spirit of this question, we propose the following model for component-oriented software development (see figure 2): A *component*

framework is a collection of software artefacts that encapsulates domain knowledge, requirements models, a generic software architecture and a collection of software components addressing a particular application domain. Development of specific applications is *framework-driven*, in the sense that all phases of the software lifecycle, including requirements collection and specification are determined according to set patterns formalized within the framework. To a large extent, system design is already done, since the domain and system concepts are specified in the generic architecture. The remaining art is to map the specific requirements to the concepts and components provided by the framework. This is in sharp contrast to naive approaches that would apply either a traditional or an object-oriented method for analysis and design, and only during implementation attempt to find "reusable object classes" matching the design specification in a software repository. Experience shows that the most valuable kind of reuse occurs in the early stages of the software lifecycle.

Given this model of component-oriented development, we can define software composition as *the systematic construction of software applications from components that implement abstractions pertaining to a particular problem domain*. Composition is systematic in that it is supported by a framework, and in the sense that components are *designed* to be composed.

Clearly there are different levels of software composition. The most familiar to all programmers is at the level of the programming constructs of a particular programming language. The next level, of solution domain specific components is also familiar: pipes and filters in Unix are perhaps the best example of a successful composition paradigm. The third level, of application domain specific components, is the hardest to achieve, since it can only be achieved after first gathering and abstracting from the experience of a sufficient number of specific applications in a domain, and then developing an adequate model of components and composition for that domain. The best successes are in the domain of user interfaces (which is arguably just a solution domain). At any level, software composition requires that software components be systematically *designed* to be composed.

What, if any, are the differences between Objects and Components? First of all, objects encapsulate *services*, whereas components are *abstractions* that can be used to construct object-oriented systems. Objects have identity, state and behaviour, and are always run-time entities. Components, on the other hand, are generally static entities that are needed at system build-time. They need not exist at run-time. Components may be of finer or coarser granularity than objects: e.g., classes, templates, mix-ins, modules. Components should have an explicit composition interface, which is type-checkable (see figure 3). An object can be seen as a special kind of stateful component that is available at run-time.

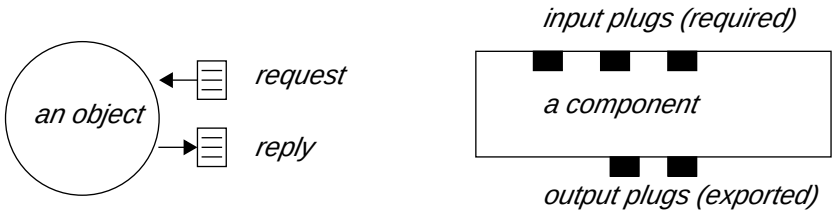

Figure 3 Objects and Components

Composition mechanisms mediate the interconnection between software components. Composition generally reduces to some combination of (i) generics/macro expansion; (ii) higher-order functional composition; or (iii) binding of communication channels. Composi-

tion mechanisms, therefore, are either static or dynamic. Static composition entails *interface compatibility* (e.g., type-checking), *binding* of parameters (e.g., binding of self and super in inheritance). Dynamic composition mechanisms may additionally entail buffering, protocol checking, translation between language/execution models, and service negotiation (this list is not intended to be complete).

Now let us be more precise about what we mean by a software architecture:

- A *software architecture* is a description of the way in which a specific system is composed from its components.
- A *generic software architecture* is a description of a class of software architectures in terms of *component interfaces*, *composition mechanisms*, and the *rules governing software composition.*

A component framework helps in the development of open systems by allowing a specific system to be viewed as a generic family of applications in the sense that its software architecture is derived from a generic one. The resulting system is open and flexible if its software architecture is *explicit* and *manipulable*. (This is clearly a necessary condition, since a system whose architecture is not explicit cannot easily be adapted to new requirements.)

Finally, we can define a *component framework* as a *generic software architecture* together with a corresponding *collection of generic software components*, that can be used to realize flexible applications whose architectures conform to the generic family.

4GLs, application generators and application builders follow a framework approach, but are closed to a single application domain. Object-oriented frameworks can be developed for arbitrary domains, but the generic software architectures they define are not explicit, and the rules for specializing and composing components can be difficult to determine from source code alone.

We shall now consider the following research issues related to software composition:

1. ***Languages:*** How, in general, can we specify software components and composition mechanisms? How can we specify generic architectures and component frameworks? How can we specify applications as compositions so that their architectures will be explicit?
2. ***Tools:*** How can we support framework-driven requirements specification? What kinds of *software information systems* are needed to manage the relevant software artefacts? What kinds of interactive tools can help in the construction and maintenance of applications composed from components?
3. ***Methods:*** To what extent can existing methods be adapted to framework-driven application development? What techniques and methods can help in the development and evolution of component frameworks (as opposed to individual applications)?

3 Composition Languages

Object-oriented languages and tools address open systems development by introducing objects (1) as an *organizing principle*, and (2) as a *paradigm for reuse.* Present-day Object-Oriented Languages emphasize *programming* (i.e., using language primitives to define new objects) over *composition* (i.e., building applications by composing higher-level abstractions), that is, they emphasize the first view to the detriment of the second. In general, it is not possible to build applications from an object-oriented class library or framework by simply

composing and linking objects instantiated from pre-existing classes. One must always program new classes that use, or are derived from those provided. In this sense, object-oriented technology fails to raise the level of abstraction from programming with core language concepts to composition with domain specific components.

We posit the need for a so-called *composition language*, which would support both the composition of applications from domain-specific components, as well as the definition of the component frameworks themselves. A composition language may or may not be seen as a programming language itself: in principle one could use it to define component frameworks at the level of basic programming language constructs, just as one may define such constructs on top of the λ calculus (or, for that matter, on top of pure Lisp). It is more interesting, however, to view such a language as enabling "glue" for composing pre-existing components, possible developed using different programming languages and platforms.

3.1 Interference of Object-oriented Features

Before considering the possible requirements of a composition language, let us consider the degree to which object-oriented programming languages support features needed for compositional software development. Wegner [21] has proposed a classification of object-based programming languages according to a set of "orthogonal" dimensions[1]:

- **Object-Based:** *encapsulation* (objects) [+ identity]
- **Object-Oriented:** + classes + *inheritance*
- **Strongly-typed:**+ data abstraction + *types*
- **Concurrent:**+ *concurrency* [+ distribution]
- **Persistent:**+ persistence + sets

An additional dimension not originally considered was *homogeneity:* in a homogeneous object-oriented language, *everything* (within reason) is an object. So Smalltalk is a homogeneous object-oriented language whereas C++ is not.

Orthogonality in Wegner's sense does not tell us anything about how easy it is to integrate orthogonal features within a single programming language. Numerous researchers have attempted to integrate such features [9] [11] only to discover that they interfere in unexpected ways [13]. In fact, just considering *objects*, *inheritance*, *types* and *concurrency,* we can see that each interferes with the others (figure 4): inheritance interferes with object encapsulation in that subclasses must typically be aware of implementation details that are hidden from ordinary clients — if the implementation of a class changes in arbitrary ways, encapsulation may be broken since subclasses may not necessarily continue to function correctly [20]. Concurrency interferes with object encapsulation in that objects that function correctly in a sequential environment may not when exposed to concurrent clients [17]. Combining inheritance with concurrency poses further problems in that it is difficult to define classes that make use of concurrency mechanisms and can be then inherited and extended in any meaningful way without exposing implementation details [5] [7]. Finally, the development of an adequate type model that addresses both objects and inheritance is still an open research problem [15], let alone one that addresses type compatibility for concurrent objects [16]. In gen-

1. Dimensions are considered to be orthogonal if features supporting them can be found independently in different programming languages. Concurrency is therefore considered orthogonal to inheritance, since some languages support concurrency features but not inheritance, and vice versa.

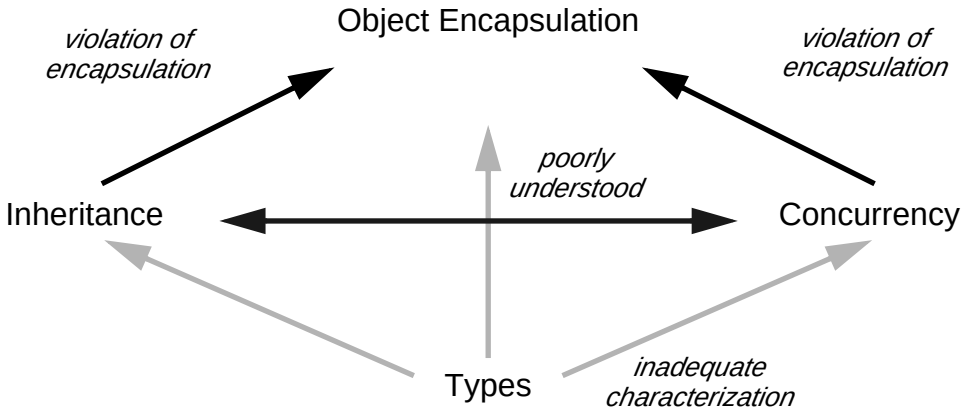

Figure 4 Interference of OO Features

eral, we can attribute these semantic interferences to weak support for software composition in object-oriented languages [13]: in each case, interference is due to inadequate characterization of the client/supplier contract. With inheritance in particular, this situation is aggravated by the two distinct interfaces a class must provide: one to clients of its instances and another to its subclasses.[1] A composition language could only hope to avoid the same pitfalls by drawing a clear distinction between the interfaces supported for each kind of client of a software component.

3.2 Requirements for Composition Languages

Composition languages would be used at two distinct levels: the framework level and the application level.

At the Framework Level, a composition language would support the specification of:

- generic software architectures
- standard component interfaces and protocols
- composition mechanisms (or “glue”)
- open, generic software components
- high-level syntax for compositions

A composition language in this sense serves as a tool for defining the kinds of software abstractions that are needed to build flexible applications in a particular domain. It serves as a kind of “meta-language” for defining the domain concepts.

At the Application Level, a composition language would support the specification of:

- elaboration and instantiation of generic components
- applications as compositions
- explicit, manipulable software architectures
- meta-level reasoning (access to framework level to support dynamic composition)

1. Aside from serving as component s for instantiating both (i) objects and (ii) subclasses, classes are often also asked to play the roles of (iii) types, (iv) sets of instances, and (v) meta-objects. Small wonder semantic difficulties arise in trying to integrate various language features!

This is the real domain of software composition: once the appropriate software abstractions are specified in a systematic way, flexible applications can be defined as instantiations, specializations and compositions of framework components. Ideally, applications are built at the level of the abstractions provided by the framework.

Let us consider what specific language features are needed to support these two levels:

1. ***Active objects:*** objects can be viewed as autonomous communicating agents [13]
2. ***Components:*** are first-class, higher-order abstractions [15]
3. ***Composition:*** should be general, not restricted to inheritance, structural composition, or other arbitrary paradigms
4. ***Types:*** must encompass both objects and components
5. ***Subtyping:*** means conformance to client/server contracts
6. ***High-level syntax:*** explicit and software architectures are supported by assigning high-level syntax to domain concepts
7. ***Interoperability:*** multi-language and multi-platform composition
8. ***Scalability:*** both small and large scale systems should be addressed; similarly rapid prototyping and production development (since component-oriented development spans all phases of the software lifecycle)

In order to address such diverse issues within a common language, a common semantic foundation is needed for reasoning about objects, components and composition, and an *object model* that relates objects and components is needed. The choice of object model will affect the ease with which composable behaviour and flexible component frameworks can be defined [13]. A kind of "object calculus" is sorely needed as a formal basis for specifying, comparing and evaluating possible object models, and as a possible core language for a composition language. As a bare minimum, we would need features to express:

1. ***Encapsulation:*** objects provide *services*, may change *state*, and have an *identity*
2. ***Active Objects:*** objects are *autonomous*, may be *internally concurrent*, and may cope with *multiple pending requests*
3. ***Composition:*** applications can be viewed as compositions of software components, and objects as compositions of object parts

An Object Calculus, it seems, would merge the operational features of a process calculus with the compositional features of the λ calculus. Recent progress in process calculi, in fact, have addressed many of the obstacles in developing an adequate object calculus [12][15], and an asynchronous variant [4] of the higher-order π calculus [19] appears in many ways to provide a suitable foundation: objects can be modelled as concurrent processes [18], and software abstractions (components) can be viewed as higher-order abstractions over the process space.

Because a composition language will cut across all dimensions of object-based languages, it is imperative to have a common semantic foundation, and a suitable mapping of language concepts to this foundation (i.e., an object model). Some issues to consider are:

1. ***Object Models:*** How can we model objects as processes, components as abstractions over the object space, and objects and object systems as compositions of components?
2. ***Composition:*** How can we express the basic forms of composition (generics, higher-order functions and communication)? How can we express more complex forms of composition (like inheritance) as software abstraction, and thus "unbundle them"?

3. ***Types:*** How can we develop an adequate type system with parametric polymorphism and subtyping for both objects and components? Can we "type the inheritance interface" [6] and thus achieve a more compositional view of inheritance? Can we model generic synchronization policies [8] as typed software abstractions, and reason about plug compatibility of protocols? [16]
4. ***Compositional reasoning:*** can we reason about the correctness of a composed system on the basis of the specifications of the components from which they are built?

4 Tools

Component-Oriented Software Development depends on systematic management of software information (i.e., domain knowledge, requirements models, component frameworks and applications). This leads us to the following problems:

- How to *represent* software information?
- How to *develop* software information?
- How to *drive* application development from software information?
- How to *evolve* and maintain software information?

Only the first of these problems is relatively well understood. Two classes of tools would appear to be essential to support component-oriented development: software information systems for representing and managing component frameworks, and visual composition tools to drive the interactive development and maintenance of software systems developed using component frameworks.

A *Software Information System* [2] represents and manages software information pertinent to component frameworks and derived applications. Some of the kinds of information to be managed include:

- Requirements models and dialogues
- Design guidelines
- Integration with development tools
- Component repository
- Evolution management for applications and frameworks

A Visual Composition Tool [3] supports the interactive construction of applications from plug-compatible software components by direct manipulation and graphical editing. A general approach to interactive software composition must be parameterized by component frameworks. Existing commercial tools are typically restricted to specific domains (UI, dataflow ...), and cannot be adapted to arbitrary domains. Experimental results [3] indicate that general-purpose visual composition is feasible by separating the tool from the component framework and the composition rules. Various technical and pragmatic difficulties nevertheless remain. Complex systems are hard to visualize, and require flexible filtering and representation techniques to support the needed user abstractions. A sufficiently flexible tool requires a framework and composition model *itself* to allow it to be easily adapted to different composition models and application domains [14].

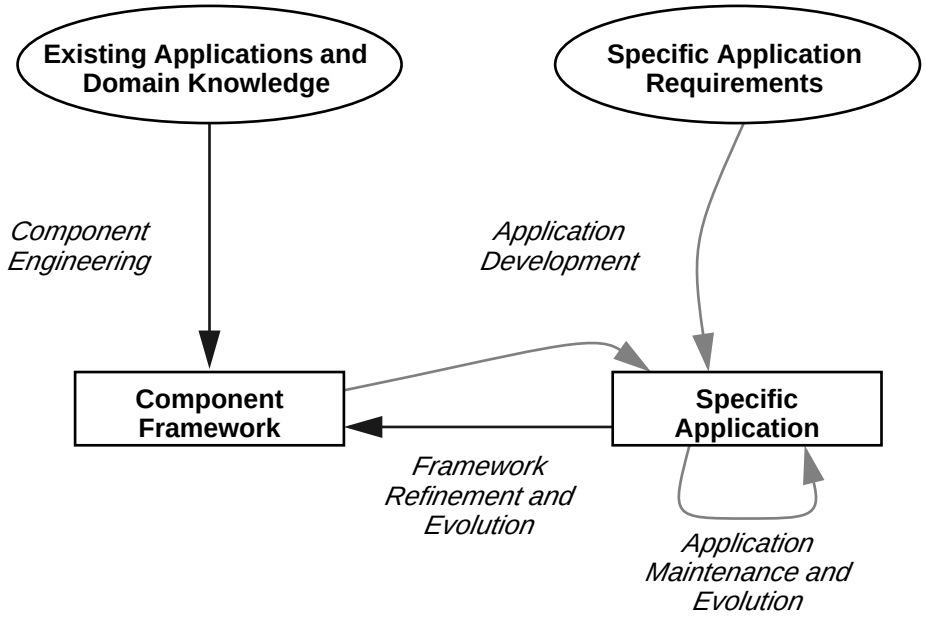

Figure 5 A Component-Oriented Software Lifecycle

5 Methods

In addition to the technological issues of component-oriented development, there are difficult methodological issues. First, how can we drive application development from component frameworks? Existing methods ignore reuse, or introduce it too late in the lifecycle. Traditional separation of analysis and design is incompatible with a framework-driven approach since framework reuse should be anticipated during requirements collection and analysis.

Second, where do the frameworks come from? Traditional methods do not address the development of generic systems from previously completed projects. Refactoring and framework evolution [1] are not yet well-understood or widely practised.

A Component-Oriented Software Lifecycle (figure 5) must take into account that *application development* (the construction of applications from component frameworks) is a separate activity from *component engineering* (the iterative development of the framework itself) [10]. Component engineering is capital investment whereas application development recovers the investment.

Since application development is ideally *driven* by component engineering, analysis and design are largely done already. The hard parts are: identifying the appropriate component framework to use, matching specific requirements to available components, building missing components and subsystems, and adapting components to unforeseen requirements. These aspects of object-oriented design and implementation fall outside the scope of today's object-oriented methods.

6 Concluding Remarks

Object-oriented languages typically provide only *limited* support for component definition and composition. This suggests a need for cleaner integration of (active) objects and compo-

nents within a so-called *composition language* for component-oriented development. Component reuse is always *systematic*, not accidental. This suggests a need for component frameworks and systematic software information management. Component engineering is difficult and iterative; present practice discourages design for reuse. This suggests a need for an evolutionary software life-cycle and corresponding methods.

A composition language would support an integrated object/component model based on a rigorous semantic foundation. Such a foundation — or *object calculus* — could well be based on a variant of the π calculus [4][12][18][19]. Challenges for such a language include the modeling of the foundational software abstractions (components, active objects, etc.), the development of an appropriate type system with type inference, support for interoperability with existing languages and component libraries, modeling of abstractions for concurrency and distribution, and modeling of reflective capabilities to support evolution of long-lived distributed systems.

The hardest problems are not in the technological domain of designing better programming languages, but in the domain of methodological support for component-oriented development. Software information systems and visual composition tools are two such ingredients in an environment that helps to drive application development from existing component frameworks. The most difficult problem is to develop those frameworks in the first place.

References

[1] Eduardo Casais, "Managing Class Evolution in Object-Oriented Systems," *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Ed.), Prentice Hall, 1995, pp. 201-244.

[2] Panos Constantopoulos and Martin Dörr, "Component Classification in the Software Information Base," *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Ed.), Prentice Hall, 1995, pp. 177-200.

[3] Vicki de Mey, "Visual Composition of Software Applications," *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Ed.), Prentice Hall, 1995, pp. 275-303.

[4] Kohei Honda and Mario Tokoro, "An Object Calculus for Asynchronous Communication," *Proceedings ECOOP '91*, ed. P. America, *Lecture Notes in Computer Science*, vol. 512, Springer-Verlag, Geneva, July 15–19, 1991, pp. 133–147.

[5] Dennis G. Kafura and Keung Hae Lee, "Inheritance in Actor Based Concurrent Object-Oriented Languages," *Proceedings ECOOP '89*, ed. S. Cook, Cambridge University Press, Nottingham, July 10–14, 1989, pp. 131–145.

[6] John Lamping, "Typing the Specialization Interface," *Proceedings OOPSLA 93, ACM SIGPLAN Notices*, vol. 28, no. 10, Oct. 1993, pp. 201–214.

[7] Satoshi Matsuoka and Akinori Yonezawa, "Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages," *Research Directions in Concurrent Object-Oriented Programming*, ed. G. Agha, P. Wegner and A. Yonezawa, MIT Press, Cambridge, Mass., 1993, pp. 107–150.

[8] Ciaran McHale, "Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance," Ph.D. Dissertation, Department of Computer Science, Trinity College, Dublin, 1994.

[9] Oscar Nierstrasz, "Active Objects in Hybrid," *Proceedings OOPSLA'87, ACM SIGPLAN Notices*, vol. 22, no. 12, Dec. 1987, pp. 243–253.

[10] Oscar Nierstrasz, Simon Gibbs and Dennis Tsichritzis, "Component-Oriented Software Development," *Communications of the ACM*, vol. 35, no. 9, Sept. 1992, pp. 160–165.

[11] Oscar Nierstrasz, "A Tour of Hybrid — A Language for Programming with Active Objects," *Advances in Object-Oriented Software Engineering*, ed. D. Mandrioli and B. Meyer, Prentice Hall, 1992, pp. 167–182.

[12] Oscar Nierstrasz, "Towards an Object Calculus," *Proceedings of the ECOOP '91 Workshop on Object-Based Concurrent Computing*, ed. M. Tokoro, O. Nierstrasz, P. Wegner, *Lecture Notes in Computer Science*, vol. 612, Springer-Verlag, pp. 1–20, 1992.

[13] Oscar Nierstrasz, "Composing Active Objects," *Research Directions in Concurrent Object-Oriented Programming*, ed. G. Agha, P. Wegner and A. Yonezawa, MIT Press, Cambridge, Mass., 1993, pp. 151–171.

[14] Oscar Nierstrasz and Theo Dirk Meijler, "Requirements for a Composition Language," *Object-Based Models and Languages for Concurrent Systems*, P. Ciancarini, O. Nierstrasz and A. Yonezawa (Ed.), LNCS 924, Springer-Verlag, 1995, pp. 147-161.

[15] Oscar Nierstrasz and Laurent Dami, "Component-Oriented Software Technology," in *Object-Oriented Software Composition*, ed. O. Nierstrasz and D. Tsichritzis, Prentice Hall, 1995, pp. 3-28.

[16] Oscar Nierstrasz, "Regular Types for Active Objects," in *Object-Oriented Software Composition*, ed. O. Nierstrasz and D. Tsichritzis, Prentice Hall, 1995, pp. 99-121.

[17] Michael Papathomas, "Concurrency in Object-Oriented Programming Languages," in *Object-Oriented Software Composition*, ed. O. Nierstrasz and D. Tsichritzis, Prentice Hall, 1995, pp. 31-68.

[18] Benjamin C. Pierce and David N. Turner, "Concurrent Objects in a Process Calculus," *Proceedings Theory and Practice of Parallel Programming (TPPP 94)*, Springer LNCS 907, Sendai, Japan, 1995, pp. 187-215.

[19] Davide Sangiorgi, "Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms," Ph.D. thesis, CST-99-93 (also: ECS-LFCS-93-266), Computer Science Dept., University of Edinburgh, May 1993.

[20] Alan Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, vol. 21, no. 11, Nov. 1986, pp. 38–45.

[21] Peter Wegner, "Dimensions of Object-Based Language Design," *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, vol. 22, no. 12, Dec. 1987, pp. 168-182.