Component-Oriented Software Development *

Oscar Nierstrasz, Simon Gibbs and Dennis Tsichritzis

Université de Genève†

Introduction

Object-oriented programming techniques promote a new approach to software engineering in which reliable, open applications can be largely constructed, rather than programmed, by reusing "frameworks" [3] of plug-compatible software components. Although the dream of a components-based software industry is very old [9], only now does it appear that we are close to realizing the dream. The reason for this is twofold:

- Modern applications are increasingly *open* in terms of topology, platform and evolution, and so the need for a component-oriented approach to development is even more acute than in the past;
- Objects provide an organizational paradigm for *decomposing* large applications into cooperating objects as well as a reuse paradigm for *composing* applications from pre-packaged software components.

Despite the contributions of object-oriented technology, there are several open research problems that must be resolved to reach the goal of effective component-oriented development. First, object-oriented mechanisms for composition and reuse must be cleanly integrated with other features, such as concurrency, persistence and distribution. Second, effective reuse of software presupposes the existence of tools to support the organisation and retrieval of components according to application requirements and the interactive construction of running applications from components. Third, the design of reusable frameworks is an iterative, evolutionary process, so it is necessary to manage software and software information in such a way that designs and implementations can evolve gracefully.

Finally, present object-oriented methodologies do not explicitly address the design of reusable frameworks. Not only the languages and tools, but the economics, methods and culture of software development must ultimately adapt to a new evolutionary software life-cycle if we are to realize the benefits of large-scale software reuse [2][17].

We shall outline a series of ongoing research projects at the University of Geneva that address component-oriented software development at the levels of languages, tools and frameworks, in particular, (1) the integration of object-oriented language features that support software composition with features concerned with other issues, like concurrency, (2) application development tools to support composition and reuse, and (3) the development of reusable application frameworks, specifically in the domain of multimedia applications.

Language Feature Integration

Object-oriented programming languages (OOPLs) are characterized by features that support (1) the definition of *objects* encapsulating state and behaviour as a set of services, (2) *classes* for instantiating new objects, and (3) *inheritance* as a mechanism for defining new (sub)classes [18]. These features provide the means to model applications as collections of cooperating objects (i.e., for *decomposition)*, and the potential to derive new specifications of objects from existing ones (i.e., for *composition*). Unfortunately these object-oriented features are *not* orthogonal to other language design issues. Semantic interference may occur between features if they are designed independently (or even if they are not!) [16]. We shall briefly summarize the key issues and point out some promising directions for feature integration.

Hybrid is a concurrent object-oriented language, developed at the University of Geneva, that is an attempt to cleanly integrate the following features [10]:

1. *Objects, Classes and Inheritance:* "all" run-time entities are instances of classes; subclasses may inherit instance (state) variables and methods (operations) from multiple super-classes;
2. *Strong-Typing and Genericity:* methods are statically typed-checked — it is not possible to invoke an unsupported operation on an object; classes may have type parameters;
3. *Concurrency:* all objects are active entities with their own thread of control; objects may issue concurrent requests and may delay their response to a pending request;
4. *Persistence:* objects may outlive the process that created them and may, in general, live indefinitely.

Although a working compiler for Hybrid has been successfully implemented [8], a number of difficulties were encountered:

- *Encapsulation and Inheritance:* as with other OOPLs, classes have two kinds of clients (objects and subclasses), but the services provided to each are not distinguished; reusability would be enhanced by making explicit the different client/server contracts [14];

*.In *Communications of the ACM*, Vol. 35, No. 9, special issue on Analysis and Modelling in Software Development, Sept. 1992, pp. 160-165.

†.*Authors' address:* Centre Universitaire d'Informatique, 24 rue Général Dufour, CH-1211 Genève 4, Switzerland. *E-mail:* {oscar,simon,dt}@cui.unige.ch. *Tel:* +41 (22) 705.7664. *Fax:* +41 (22) 320.2927.

- *Homogeneity:* although Hybrid was intended to be a fully object-oriented language, certain entities, such as classes and *delay queues* (used for delaying requests for selected services), are not first-class objects; a more homogeneous approach would be both simpler and more general;
- *Types:* in Hybrid the notions of class and type are separated to avoid confusion between subtyping (refinement of specification) and inheritance (incremental modification of implementation); nevertheless, operational interfaces used as types tell us little about the client/server contract when concurrency and synchronization are at issue;
- *Concurrency and Inheritance:* it is difficult for a subclass to extend the synchronization policy of its superclasses in a meaningful way [7].

To a large extent, these difficulties can be traced to the lack of any generally accepted computational model for active objects suitable for defining the semantics of language constructs, and to the lack of good prototyping tools for experimenting with language design alternatives. As a consequence, we are now pursuing a radically different approach to the integration of OOPL features in which objects are viewed as *patterns of communicating agents* [11].

At the lowest semantic level, we only have *agents*, which are entities that communicate by synchronously exchanging messages. At any point in time, an agent may make a number of *communication offers* to input or output certain specific messages, or it may silently change state by means of an internal communication. Communication events take place when there are matching input and output offers between concurrent agents.

The behaviour of agents may be described precisely by a simple language, or "object calculus," called OC [13]. It is possible to define higher-level abstractions such as remote procedure calls, various synchronization mechanisms, control structures, classes, and even inheritance, as patterns, that is, as functions over the space of agents. Since OC is an executable language, it can serve simultaneously as a semantic target for formally specifying language features and as a tool for rapidly prototyping them. We are presently working on the definition of a *pattern language for active objects* based on OC, in which the basic patterns representing language features can be augmented by libraries of patterns representing reusable software components.

Application Development Tools

The use of an object-oriented programming language for developing applications is not sufficient to guarantee an improvement in programmer productivity. The development of languages to better support component-oriented software development addresses only part of the problem. It is necessary to *invest* in development of reusable component sets and to streamline the application development activity as much as possible by providing an environment that encourages application construction through reuse over programming from first principles.

To provide the appropriate context, we can distinguish between two quite different activities:

1. ***Application Engineering*** is the activity of abstracting the domain knowledge for selected application domains, developing reusable software components to address these domains, and encapsulating this knowledge into *generic application frames* (GAFs);
2. ***Application Development*** is the activity of instantiating a specific application from a GAF to meet some particular requirements.

Capital investment occurs during application engineering. This is an expert activity that requires detailed understanding of both the application domain and of the mechanisms available for packaging and composing software components. The return on investment should occur during application development. Applications will be easier to develop, will be more robust and reliable, and will be more flexible and easy to adapt to evolving requirements. The degree to which application development is streamlined depends on (1) the quality of application engineering preceding it, and (2) application development tools to support reuse.

Concerning the first point, we will only say that application engineering is *necessarily* iterative and evolutionary. One cannot be sure that software is truly reusable until it has been successfully reused. We must better understand how application engineering works in some selected application domains before we can hope to offer any insight into how to accomplish it in general. It is for this reason, we suspect, that object-oriented analysis and design methodologies have been slow to address the issue of reuse.

Concerning the second point, it is critical to take into account that reuse occurs by design, not by accident. As a consequence, it is essential that the environment support and exploit the GAF structure. In effect, GAFs function as "specialized methodologies" for object-oriented development in selected application domains. They drive the application development process by means of pre-packaged requirements models, application designs and software components.

ITHACA

ITHACA is a Technology Integration Project in the Office & Business section of the European Community's Esprit II Programme*. The goal of ITHACA is to produce a complete object-oriented application development environment that can be easily adapted to various application domains [1]. The environment consists of:

- *An Object-Oriented Programming Environment:* including an object-oriented language with support for persistence and transactions, an object-oriented database, and programming and debugging tools;

*.The partners are Siemens/Nixdorf (Berlin), Bull (Paris), Datamont (Milan), TAO—Tècnics en Automatitzaciò d'Oficines (Barcelona), FORTH—the Foundation of Research and Technology, Hellas (Iraklion) and CUI—the Centre Universitaire d'Informatique of the University of Geneva.

- *A Software Information Base (SIB):* for storing all information concerning software reuse and the development process (i.e., GAFs), and including a graphical browsing and querying interface;
- *A Requirements Collection and Specification Tool (RECAST):* for guiding the application developer in the instantiation of a GAF;
- *A Visual Scripting Tool (Vista):* for interactively constructing applications from pre-packaged, plug-compatible software components;
- *Application Workbenches:* for selected application domains, including Public Administration, Office Systems and Financial Systems.

The application workbenches constitute the software components and tools specific to a particular application domain. Knowledge concerning the development of these applications is stored in the SIB (i.e., as GAFs). The GAFs then drive application development by constraining the development process to the context of the particular application domain. The requirements model is provided and all generic design choices are fixed in advance. Only the specific application requirements and design choices that depend on them should, in principle, be specified during application development. The degree to which this is true will depend on how general or specific the GAF is. For example, a GAF for defining workflow procedures that coordinate the processing of electronic office documents may be completely general, or it may be specially tailored to the needs of a specific organisation and leave open only very few design alternatives.

Vista

Visual scripting is the interactive construction of applications from pre-packaged, plug-compatible software components by direct manipulation and graphical editing. *Vista* is the visual scripting tool developed within ITHACA [12].

Scriptable software components are characterized by the following properties:

1. *Scripting Interface:* every component has a set of *output ports* where it makes available services that it provides, and a set of *input ports* where it receives connections to services it requires;
2. *Visual Presentation:* direct manipulation is supported by providing every component and each of its ports with some editable visualization;
3. *Run-time Behaviour:* a component, when all its input ports are connected, provides some set of services to its clients.

A *script** is a set of software components with compatible input and output ports connected. The types of ports defined for a set of components and the rules that determine plug-compatibility constitute what is called a *scripting model*. This very general scheme can accommodate a variety of programming and composition models. For example, in a dataflow-based scripting model, connections cause values to be propagated from output ports to input ports [12]. Dataflow components compute some values as a function of their inputs, and make them available at their output ports. There is no distinction between composing a dataflow script and executing it since information can flow as soon as connections are made.

In an object-oriented scripting model, on the other hand, connections represent service availability. A connection between an input port and an output port binds the client to the service provider. When an object-based script is used, its components may (or may not) make use of available services according to their needs.

An example of an object-based scripting model is one that was developed within ITHACA for generating specifications in an Activity Definition Language (ADL) for coordinating office procedures. Each component (representing office objects, activities, etc.) is equipped with methods for generating various code fragments. An office procedure is then defined as a visual script by connecting existing code generation components. A complete script can then be packaged as an office procedure, and requested to generate its own ADL specification.

Clearly, scriptable components are not limited to generating code — in an equally reasonable scenario, the scripting of an office procedure would cause run-time entities to be connected, and changes to scripts would immediately entail changes in the behaviour of the associated office activities.

By decoupling scripting models from the underlying programming language, a variety of different software composition paradigms can be supported. Furthermore, the ability to encapsulate a *script as a component* (SAC) makes it possible to develop higher-level abstractions through scripting rather than by programming. To define a SAC, one must provide script with (1) a scripting interface (by specifying which ports of the script are to become ports of the SAC), and (2) a visual presentation (which can be composed of existing presentation components). The behaviour of a SAC is simply the behaviour of the script it encapsulates.

Multimedia Frameworks

Languages and tools only provide *technology* for achieving component-oriented software development. Application frameworks are needed to provide a discipline for component development and reuse. Since new ideas, methods and tools fare best in new application areas where inertia is not a factor, the realm of multimedia applications provides a good test-bed for component-oriented software development. We are presently developing a multimedia framework that is intended to simplify the programming of multimedia applications.

An early version of the framework has been described elsewhere [5] — here we shall summarize the main concepts. The framework introduces classes of *media values* and *media objects*. Media values are temporal sequences. They may be either synchronous streams (in which elements occur at regular intervals) or asynchronous "bursts" (in which elements occur irregularly). Media objects produce, consume and transform media values. Media values are grouped into multimedia values by

*.The term "script" is intended to suggest a parallel between an application and a theatrical performance, in which the desired behaviour is obtained by means of a script that specifies how actors are to interact.

specifying temporal transformations (which determine when particular values start and stop) — we call this *temporal composition*. Media objects, in turn, are grouped into multimedia objects by specifying the flow of values from one object to another — we call this *flow composition*.

Flow composition is perhaps closer to component-oriented software development than is temporal composition. Whereas temporal composition is a form of data structuring, flow composition actually produces applications. Flow composition is intended for applications in which many media streams undergo simultaneous processing. The scripting model is based on a *hi-fi component metaphor:* media objects have input and output ports that may be connected and through which media values flow*. These objects correspond to pieces of hardware (e.g., a video digitizer), or software processes (e.g., a process that generates audio events). Both forms of media object are examples of *active objects*, objects that can spontaneously perform actions, even in the absence of method invocation. This results in a high degree of concurrency, and, consequently, special techniques are needed for synchronization [5]. During construction of an application using flow composition, the scripting model determines which port connections are valid; for instance, an input port can be connected to an output port if the two handle compatible media values. Another constraint is that some ports may require single connections while others may allow multiple connections.

As a specific example of flow composition, we will describe a prototype we are developing called the *virtual museum* [15]. The system is based on a 3D model of a building. Users can navigate through the model with a joystick-like device. At selected locations, various "exhibits" are displayed. At the moment exhibits include raster images, video frames, simple animated geometric objects and 3D surface scan data (similar to raster images, but with depth information).

The script for the virtual museum is shown in Figure 1. This

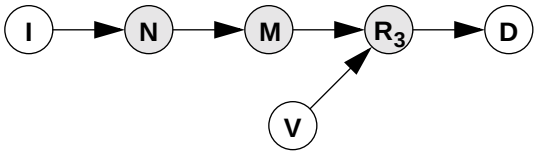

Figure 1 Basic Virtual Museum Script.

script specifies a multimedia object that is composed of six individual media objects. These objects produce a data stream from the input device (I), translate this stream into requests for movement within a 3D space (N), interpret movement requests in the context of a geometric model of the museum (M), render geometric models and overlay video frames (R_3), provide video frames (V), and display the output of the rendering process (D). In Figure 1, software-based media objects are shown with shading, hardware-based without. A list of the port descriptions of these objects is provided in Table 1.

Table 1 Virtual Museum Components.

component	input port(s)	output port(s)
Input Device Object (I)		raw device data
Navigation Object (N)	raw device data	render requests
Museum Object (M)	render requests	render requests
Video Source Object (V)		video frames
Render Object (R_3)	render requests video frames	raster frames
Display Object(D)	raster frames	

One benefit of flow-based composition is that new functionality can be added, or removed, by simple modifications to the script. As an example of reduced functionality, note that the Museum Object (M) can be bypassed by connecting the Navigation Object (N) directly to the Render Object (R_3). The resulting application allows the user to interactively move through an "empty" 3D space. This simpler configuration is useful for testing components. An example of an extended configuration is shown in Figure 2 where an Animator Object (A)

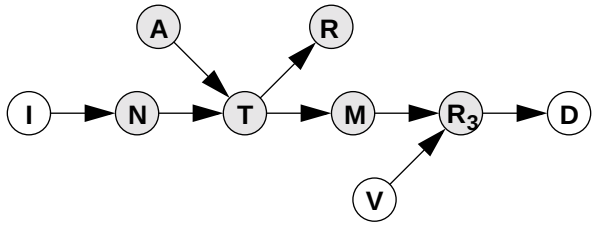

Figure 2 Extended Virtual Museum Script.

transforms or displaces parts of the museum (e.g., rotates an exhibit), a Tee Object (T) duplicates its input to produce two output streams, and a Recorder Object (R) produces a time-stamped log of requests. By again slightly changing the script one could playback the log, possibly in reverse, possibly at different speeds.

These brief examples illustrate the potential of component-oriented software development. We believe that our experience, and that of others, indicates that this form of development is suitable when components are derived from a framework designed for a particular application domain.

Conclusions

There is today still too much emphasis on object-oriented *programming* rather than on application *composition* from prepackaged components. There are a number of reasons for this, one of which is that present-day object-oriented languages do not fully support a *component-oriented* software development approach. A second reason is that application development

*.This view of multimedia applications resembles the "digital production studio" model proposed for Intel's DVI system [6].

tools tend to emphasize programming and debugging rather than composition and reuse.

There are two other difficult problems that are not technological but methodological and cultural:

1. *The Framework Design Problem:* How do we abstract from acquired domain knowledge in order to engineer plug-compatible components for composing new applications?
2. *The Large-Scale Reuse Problem:* How can we obtain a satisfactory return on our capital investment in reusable software components?

The first problem is an order of magnitude more difficult than that of developing a single, specific application. One must acquire domain knowledge, factor out functionality, anticipate future requirements, develop reusable software components, package the results for generations of future application developers, *evaluate* the ease with which new applications can be composed, and iterate. It is necessarily an evolutionary process. Reusable object classes are like poems — it is easy to talk about them, but it is hard to write a good one!

The second problem has to do with the way software is packaged and marketed, and the way in which software communities share and exchange information [4]. Software developers have always recognized the need for standards, but standards for interoperability of object-oriented applications are only now being explored. Present-day project management practices discourage reuse by leaving little room for capital investment in reusable software. Existing software is reused only if it is part of the basic environment, if it is free, or if it constitutes a complete subsystem (such as a database). New approaches to software licensing and exchange of software information are needed if developers of reusable software are to see a return on their investment.

References

[1] M. Ader, O. Nierstrasz, S. McMahon, G. Müller and A-K. Prö-frock, “The ITHACA Technology: A Landscape for Object-Oriented Application Development,” Proceedings, Esprit 1990 Conference, Kluwer Academic Publishers, Dordrecht, NL, 1990, pp. 31-51.

[2] B.J. Cox, “Planning the Software Industrial Revolution,” IEEE Software, vol. 7, no. 6, Nov. 1990, pp. 25-33.

[3] L.P. Deutsch, “Design Reuse and Frameworks in the Smalltalk-80 System,” in *Software Reusability, Vol. II*, (eds. T.J. Biggerstaff and A.J. Perlis) ACM Press, 1989, pp. 57-71.

[4] S. Gibbs, D. Tsichritzis, E. Casais, O. Nierstrasz and X. Pintado, “Class Management for Software Communities,” Communications of the ACM, vol. 33, no. 9, Sept. 1990, pp. 90-103.

[5] S. Gibbs, “Composite Multimedia and Active Objects,” Proceedings OOPSLA’91, ACM SIGPLAN Notices, vol. 26, no. 11, Nov. 1991, pp. 97-112.

[6] J.L. Green, “The Evolution of DVI System Software,” Communications of the ACM, vol. 35, no. 1, Jan 1992, pp. 52-67.

[7] D.G. Kafura and K.H. Lee, “Inheritance in Actor Based Concurrent Object-Oriented Languages,” in *Proceedings ECOOP ’89*, Cambridge University Press, Nottingham, July 10-14, 1989, pp. 131-145.

[8] D. Konstantas, O. Nierstrasz and M. Papathomas, “An Implementation of Hybrid, a Concurrent Object-Oriented Language,” in *Active Object Environments*, ed. D.C. Tsichritzis, Centre Universitaire d’Informatique, University of Geneva, June 1988, pp. 61-105.

[9] M.D. McIlroy, “Mass Produced Software Components,” in *Software Engineering*, ed. P. Naur and B. Randell, NATO Science Committee, Jan 1969, pp. 138-150.

[10] O. Nierstrasz, “Active Objects in Hybrid,” Proceedings OOPSLA ’87, ACM SIGPLAN Notices, vol. 22, no. 12, Dec 1987, pp. 243-253.

[11] O. Nierstrasz and M. Papathomas, “Viewing Objects as Patterns of Communicating Agents,” Proceedings OOPSLA/ ECOOP ’90, ACM SIGPLAN Notices, vol. 25, no. 10, Oct 1990, pp. 38-43.

[12] O. Nierstrasz, D. Tsichritzis, V. de Mey and M. Stadelmann, “Objects + Scripts = Applications,” Proceedings, Esprit 1991 Conference, Kluwer Academic Publishers, Dordrecht, NL, 1991, 534-552.

[13] O. Nierstrasz, “Towards an Object Calculus,” in *Proceedings of the ECOOP ’91 Workshop on Object-Based Concurrent Computing*, ed. M. Tokoro, O. Nierstrasz, P. Wegner, A. Yonezawa, LNCS, Springer-Verlag, Geneva, Switzerland, July 15-16, 1991, to appear.

[14] R.K. Raj and H.M. Levy, “A Compositional Model for Software Reuse,” in *Proceedings ECOOP ’89*, Cambridge University Press, Nottingham, July 10-14, 1989, pp. 3-24.

[15] D. Tsichritzis and S. Gibbs, “Virtual Museums and Virtual Realities” Proc. Intl. Conf. on Hypermedia & Interactivity in Museums, Archives and Museum Informatics Technical Report, no. 14, Pittsburgh, October 14-16, 1991, 17-25.

[16] D. Tsichritzis, O. Nierstrasz and S. Gibbs, “Beyond Objects: Objects,” IJICIS, vol. 1, no. 1, 1992, to appear.

[17] P. Wegner, “Capital-Intensive Software Technology,” IEEE Software, vol. 1, no. 3, July 1984.

[18] P. Wegner, “Dimensions of Object-Based Language Design,” Proceedings OOPSLA ’87, ACM SIGPLAN Notices, vol. 22, no. 12, Dec 1987, pp. 168-182.