Beyond Objects: Components [1]

Theo Dirk Meijler and Oscar Nierstrasz
Software Composition Group, 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.[3]

1 Introduction

In large scale networks, such as the internet, many different kinds of resources are available. These resources include not only information systems and their contents, but also information processing programs, expert system shells, and other kinds of computational resources. In order to synthesize information from various sources and avoid having to duplicate information processing resources, it is necessary to make information systems and computational resources cooperate. Cooperation can take various forms: in *decentralized* cooperation, resources are agents that cooperate actively (and possibly interactively) to arrive at a common result, and are thus *visible* to each other; in *centralized* cooperation an integrating agent manages underlying resources that are *not* visible to each other, issues requests to the resources, and is responsible for synthesizing the results.

In order to realize different forms of cooperation, technological support is needed: First of all, an infrastructure is needed to allow heterogeneous resources to communicate either with each other or with an integrating agent. Since there are now several industrial standards available that provide this kind of support (i.e., CORBA, OLE, OpenDoc, etc.), we will assume in this chapter that the necessary infrastructure is in place. Second, reliable solutions for coordination and synthesis are needed. This encompasses such aspects as "brokering" (i.e., deciding which requests should go where), coordinating concurrent or simultaneous access to shared resources, establishing a valid execution order for servicing requests, maintaining consistency of per-

1. In *Cooperative Information Systems: Current Trends and Directions*, M.P. Papazoglou, G. Schlageter (Ed.), Academic Press, Nov. 1997, pp 49-78.

2. *Authors' address:* Institut für Informatik (IAM), Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland. *Tel:* +41 (31) 631.4618. *Fax:* +41 (31) 631.3965. *E-mail:* {meijler, oscar}@iam.unibe.ch. *WWW:* http://iamwww.unibe.ch/~scg.

3. Some of the material presented here was previously published in "Research Topics in Software Composition" in *Proceedings, Langages et Modèles à Objets*, A. Napoli (Ed.), Nancy, Oct. 1995, pp. 193-204.

sistent state, gathering and integrating results from various resources, and so on. In general we will distinguish *non-functional* behavioural aspects of a system and the functional aspects. In a Cooperative Information System the latter correspond to the resources that are integrated in a cooperation.

Cooperative information systems are essentially *open* systems: systems that are open in terms of topology, platform and evolution[55]. A key characteristic of open, network applications is that requirements continuously change. This implies that coordination and synthesis must not only be reliable, but they must be robust, flexible and configurable. In other words, it is necessary to identify and implement software abstractions that encapsulate efficient and reliable solutions to standard coordination and synthesis problems. These abstractions, or "components", can then be used across many different applications, and can be reconfigured when application requirements change.

Thus "software reuse" is the key to building these systems: not only are the cooperating resources themselves software components that are used across multiple applications, but also the components that realize non-functional behaviour will be used in various configurations to address a variety of requirements.

In this chapter we will summarize the state of the art in software reuse, evaluate the extent to which available approaches support (or fail to support) the construction of flexible, open information systems. We shall especially focus on the possibilities to configure and specialize non-functional behaviour independently from functional behaviour as needed to realize open cooperative information systems. We shall identify a series of open research problems to be resolved.

We start by noting that object-oriented languages and techniques presently offer the most relevant and promising support for our problem. Objects encapsulate data and operations by providing an interface that only responds to messages. They can therefore hide the fact that they might encapsulate existing programs, act as proxies for remote resources, or even coordinate multiple, concurrent requests. In short, objects provide a uniform way to hide distribution and heterogeneity. If we assume that resources will be encapsulated as distributed objects, the question then becomes how to realize coordination and synthesis abstractions that can be applied in a reusable way to these distributed objects.

It is useful to distinguish between "white box" reuse — in which the implementation of reused components is exposed to some degree — and "black box" reuse, in which components can only be reused according a specially-provided reuse interface, or *contract*. We will take the position in this chapter that the most desirable form of reuse is "black-box" or compositional reuse, since this frees the application developer from having to study implementation details of components to be reused. Since the reuse contract is explicitly specified, it is possible to check the contract, and to actively support it in a development environment. Furthermore, links and dependencies between black box components must be explicitly specified, thus making it easier to adapt a composition to new requirements. With white box reuse, these links are often hidden and implicit in the extension code, and therefore harder to understand and change.

In section 2 we will give an overview of black box components, illustrate what problems they address through their support for *variability* and *adaptability*, and provide a scenario for component-oriented application development.

In the next section we shall evaluate how well current object-oriented technology supports this form of black-box reuse, and at the same time indicate what the consequences are for realizing components for non-functional behaviour. We see for example in section 3.1 that many problems arise when trying to integrate such non-functional aspects in object-oriented programming languages. Furthermore, in section 3.2 we shall see that subclassing is really a white-box reuse mechanism, which makes it quite difficult to reuse by inheritance classes that implement coordination abstractions.

Section 3 gives us an overview of necessary object-oriented technology, but also shows us the current limitations of that technology. In section 4 we give an overview of future directions. We focus on the requirements and possible realization of a composition environment. One important aspect of such an environment will be the distinction between two separate roles with separate concerns: (i) *application developers* develop specific applications by composing both domain-specific functional components and generic, coordination components, in a black box fashion; (ii) *component developers* build black box components by identifying useful software abstractions and factoring out both domain-specific and generic components. The implementation of the components themselves may incorporate white box reuse, but this should not be visible to application developers. Furthermore, a clear distinction will be made between *extensional* object composition and *intentional* class level composition, components for non-functional behaviour mostly being part of the latter. In such a composition environment a set of rules, together called the *composition model*, determines what compositions are legal. A composition environment will allow for visual composition, and support the developer to do so in compliance with the composition model.

We conclude by noting that present-day software development methods do not yet support component-oriented development in two important senses: first, component reuse is often considered far too late in the software lifecycle, after detailed design is complete, whereas systematic reuse of component requires that software architectures also be reused. Second, none of the well-known methods gives any hint how to develop reusable software components. Methods to support component development are still an open research topic.

2 Software Composition for Open Systems

If we examine successful approaches to developing open, adaptable systems — such as 4GLs, application generators, component toolkits and builders, and object-oriented frameworks — we find that there are striking similarities. In each case, (i) the application domain is well-understood, (ii) a generic software architecture captures families of applications, (iii) parameterized software components are designed to be specialized or instantiated to meet specific requirements, and (iv) the path from requirements collection to implementation is reduced (at least to some degree) to a recipe or formula.

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 often 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.

In each case, *variability* — how much variation can be achieved — is attained by providing components on top of which variations can be introduced. *Adaptability* — how easy it is to adapt existing applications — is achieved by providing a generic application architecture that can be adapted to different needs. Ease of use is achieved by providing "black-box" interfaces to components that on the one hand constrain the ways in which components can be used and on the other hand limit the need to understand implementation details of components.

2.1 Components and Black-Box Reuse

Software reuse addresses two seemingly contrasting sets of requirements: (i) streamlining the development process, and (ii) ensuring robustness and run-time efficiency of products. In the introduction we asserted that "black box" reuse is preferable to "white box" reuse. We will now try to make this distinction precise by explaining what we mean by the term "component."

We may see a program as a structure: a structure of statements, or of procedures, methods, classes etc. In the most basic form of software development the developer has to create these structures from scratch. We can abstract away from the elements of a structure in order to scale up to various levels of reuse. A programmer who is provided with certain pieces of structure that can be adapted and combined (e.g., a sequence of statements or a group of cooperating classes) can already achieve a certain degree of reuse. We call this "open" or *"white-box"* software reuse, since the structures that are reused are not encapsulated.

Adapting white-box structures can be very difficult, however: one has to understand what each element in the structure means and how the elements work together in order to reuse the structure. The complexity of adaptation of course depends on the complexity of structure to be adapted. Moreover, putting several complex structures together to form a bigger system (e.g., merging together groups of statements or different class hierarchies) is also difficult. This is where software components help us.

A *component* is an abstraction of a software structure that may be used to build bigger systems, while hiding the implementation details of the smaller structure. Putting together components is simple, since each component has a limited set of "plugs" with fixed rules specifying how it may be linked with other components. Instead of having to adapt the structure of a piece of software to modify its functionality, a user plugs the desired behaviour into the parameters of the component.

There are therefore two important aspects to components: (1) *encapsulation* of software structures as abstract components, and (2) *composition* of components by binding their parameters to specific values, or other components. A simple example is a function or a procedure parameterized by its run-time arguments. An object-oriented example is a generic or "template" (In C++ [13]) container class, which can be parameterized by the type of the contained elements. Encapsulation is the means to achieve variability, since the possible variation is expressed in the parameters (or "plugs") of the component. Adaptability is achieved during composition, since a software structure composed from components can be more easily reconfigured than an unencapsulated structure.

We can exchange the open variability of white-box structures for the fixed variability of possible connections to the plugs of the component. This restriction of variability is possible due

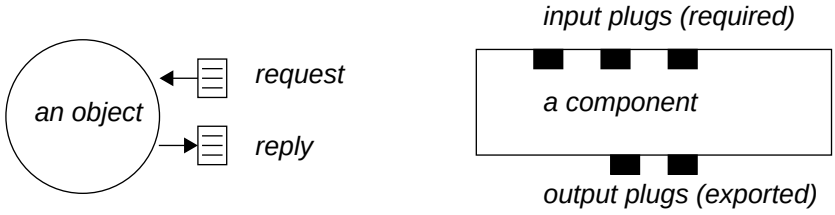

Figure 1 *Objects and Components.*

to the fixed intended purpose of the component; it also includes the possibility to check the correctness of combinations of parameters. We call this "closed" or *"black-box"* software reuse.

When we build systems by putting together pieces of software, the need for compositionality is even clearer. It is difficult to integrate open pieces of software structure, as anyone who has "cut-and-pasted" software code can testify. Creating new classes through inheritance can pose a similar problem, since object-oriented languages do not support the specification of an explicit, typed "inheritance interface" for programmers who develop subclasses [23]. Putting together components is much simpler, due to a well defined composition interface that defines how components may be "plugged" together.

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 do not necessarily 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 1). An object can be seen as a special kind of stateful component that is available at run-time.

It is certainly possible to *implement* many kinds of components as objects, which is the source of a great deal of confusion. By encapsulating components as objects, one achieves a great deal of flexibility, since components can then be configured and substituted at run-time. This notion is fundamental to all interactive component-based development environments (such as user interface builders). On the other hand, this does *not* mean that either every object is usable as a component, or that components must be implemented as objects! Functions, modules, templates and even whole applications can be seen as components. Conversely, objects that are not designed to be connected to other objects are not "pluggable", and hence cannot be seen as components.

2.2 Why Do We Need Components?

Let us now consider what specific problems are addressed by components:

Fast time-to-market. Applications that can be built from reusable components can be developed more quickly and thus brought to market and sold more cheaply than custom-made applications.

Reliability. Components that are reused across many applications are bound to be more reliable than new, hand-coded components. Applications built according to tested frameworks are bound to be more reliable than newly designed and implemented applications.

Division of labour. Components with well-defined interfaces are natural units for distribution to software teams. The development of applications from software components and the development of reusable components themselves are tasks requiring different kinds of skills and experience.

Variability. Families of applications can be developed using a common software base only if the software base can accommodate sufficient variability. Software components support variability through parameterization. Parameters represent functionality that must be provided by the client of the component, or (as is often the case with object-oriented components) default functionality that may be overridden.

Adaptability. A flexible application is one that can be easily adapted to changing requirements. Software components support adaptability if an application can be viewed abstractly as a configuration of components linked together. If the components have been well-designed, many changes in requirements can be addressed at this abstract level by reconfiguring the application's components. In well-understood application domains, many possible changes in requirements can be anticipated and incorporated into the design of the components and the ways in which they may be composed.

Note that adaptability may be viewed as a form of reusability, since it entails the reuse of an existing application to create a changed version. However, it is a special form since it does not focus on newly building (larger) systems from (smaller) existing software components. Variability is a pre-requisite to adaptability but increasing variability in the components may damage adaptability since adaptation becomes correspondingly increasingly complex.

Distribution and concurrency. In order to use hardware resources optimally, systems are becoming more distributed and consequently concurrent. Since distributed systems are notoriously difficult to implement correctly, application developers need software abstractions that can simplify the task. Components offer on the one hand natural units for distribution, and on the other hand may encapsulate protocols and concurrency abstractions, thus hiding the complexity of distributed programming from application developers.

Heterogeneity. Open systems are inherently heterogeneous. Components of a distributed system will be developed using different platforms and programming languages. Components help by hiding differences in implementation platform behind interfaces that are (in principle) independent of programming languages, as in component models such as COM and CORBA [22][47].

A piece of software can be called a *component* if it has been *designed* to be composed with other components. In general this is done to address a particular class of applications. If that is the case[1] we say a *component framework* has been developed for that application area. This notion still needs to be better understood. The key principles, however, are:

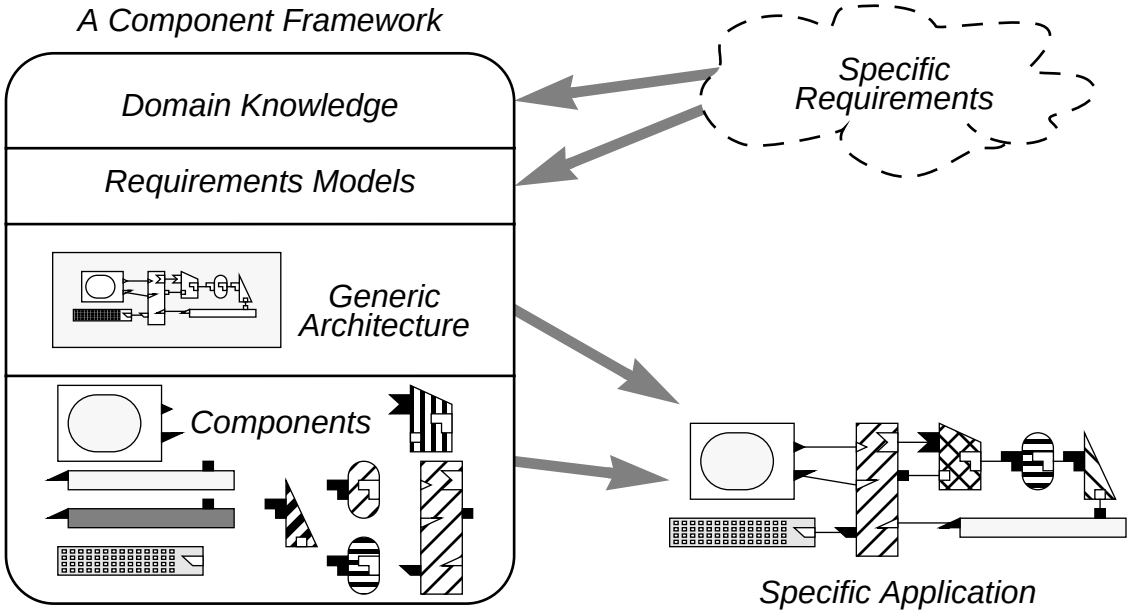

Figure 2 *Component-Oriented Software Development*

- A component framework does not just consist of a library of components, but must also define a generic *architecture* for a class of applications.
- Flexibility in a framework is achieved through *variability* in the components and *adaptability* in the architecture.
- Flexibility in an application is promoted by making the specific architecture *explicit* and *manipulable*.

2.3 A Scenario for Compositional Development

Since black-box components are not isolated entities, but only become useful in the context of a framework, or at least of an environment[37], this form of reuse cannot be achieved by just starting to develop components. So, how should software development be organized in order to achieve compositional, black-box reuse? In an attempt to answer this question, we propose the following scenario for component-oriented software development (see figure 2):

- A *component framework* is a collection of software artefacts that encapsulates (i) domain knowledge, (ii) requirements models, (iii) a generic software architecture and (iv) 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.

1. Note in the Componentware [37], [56] approach a whole component environment may be said to serve as a “framework”.

- 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 [14][15].

Such a scenario would therefore correspond to a new, framework-driven method of software development that is much more strongly directed towards software reuse than existing object-oriented development methods.

The scenario assumes that all parts of the component framework are formally specified, and managed by an application development environment. The environment guides the requirements collection and specification activities, and helps to guide the specialization and configuration of the application from available components.

Given this scenario 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.

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 (*cf.* [50]).
- A *generic software architecture* is a description of a class of software architectures in terms of *component interfaces*, *connectors*, and the *rules governing software composition*.

Connectors [50] mediate the interconnection between software components. Connectors may either be *static* or *dynamic*. Static composition entails *interface compatibility* (e.g., type-checking), and *binding* of parameters (e.g., binding of self and super in inheritance). Dynamic connectors additionally entail any kind of run-time behaviour, such as buffering, protocol checking, translation between language/execution models, and service negotiation. Composition often reduces to some combination of (i) generics/macro expansion; (ii) higher-order functional composition; or (iii) binding of names to resources (e.g., object identifiers, communication channels).

Composition rules formalize the kinds of components that may be composed using the available connectors, and may be expressed in a type system, in the semantics of the programming language used, or as part of a tool or environment. For example, the fact that certain kinds of applications may be composed with "Unix pipes" depends partly on the definition of the components (they must be designed as "sources", "filters" or "sinks"), on the semantics of the shell programming language, and on the run-time environment (i.e., the buffering of input and output by the operating system). All composition rules together, in whatever form, make up the *composition model*.

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 ar-

chitecture 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.)

3 Object-Oriented Software Composition

From the previous sections we can now distil some requirements for the construction of open information systems in general, and Cooperative Information Systems in particular. In the terminology of section 2, we see that we need to develop component frameworks. In the context of Cooperative Information Systems, the components of a such a framework would include. first of all, the individual systems that cooperate, and, second of all, the components that realize the coordination and synthesis. This is necessary to keep the cooperation as flexible as possible, allowing us, for example, to have both central and decentralized cooperations. In order to avoid having undesirable and undocumented dependencies between components, it is important that cooperative systems be built from components in a "black box" fashion.

We assert that the fundamental problem to be addressed by a framework for Cooperative Information Systems and for open systems in general, is to provide black box components that encapsulate both functional and non-functional aspects of behaviour (i.e., systems and their coordination), that can easily be combined. This separation of concerns is both critical — for ensuring that systems remain flexible and reconfigurable — and non-trivial — since functional and non-functional aspects are typically intertwined in programs.

In our search for technology that supports this, we turn to object-oriented programming languages and methods, where combining functional and non-functional features in a reusable form has been studied extensively in the past. In this section we consider both approaches to reuse and approaches to separating functional and non-functional aspects of behaviour.

3.1 Interference of Object-oriented Features

Wegner [58] has proposed a classification of object-based programming languages according to a set of "orthogonal" dimensions:

- **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.

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. 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 [39],[5] only to discover that they interfere in unexpected

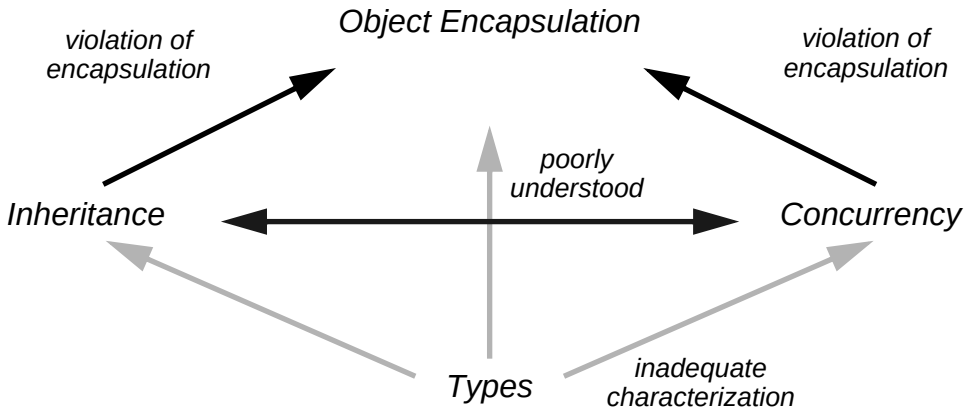

Figure 3 *Interference of OO Features*

ways [41] (see [46] for an overview). In fact, most of the problems arise because inheritance is basically a white box reuse mechanism. Inheritance conflicts with encapsulation since subclasses are dependent upon implementation details of superclasses in a way that is not described by an explicit interface (figure 3).

Concurrency and persistence are examples of non-functional issues. Combining inheritance with concurrency poses 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 [19] [27]. That one would like to configure these non-functional aspects independently from the functional issues can also seen from the fact that objects that function correctly in a sequential environment may fail when exposed to concurrent clients [46]. Similarly, one would like to switch persistence “on” and “off” independently of the way functionality is inherited in an inheritance hierarchy. Introducing persistence through the inheritance hierarchy reduces flexibility.

The development of an adequate type model that addresses both objects and inheritance is still an open research problem [42], let alone one that addresses type compatibility for concurrent objects [43].

3.2 Inheritance

Even in approaches where functional and non-functional aspects are separated, both may still be developed independently through inheritance. Although inheritance is an important mechanism for sharing interfaces and implementation in object-oriented design, its principle weakness is that it essentially supports “open”, or white-box reuse: the superclass is generally viewed as an open structure of methods and instance variables that the builder of the subclass adapts and extends. This is really a problem since the subclass is not even a copied version of the superclass: The subclass remains dependent on the superclass, and the “openness” thus remains. The superclass cannot be viewed as a component with respect to the subclass, since the implementation of the superclass cannot be changed independently.

There have been several developments that address this problem. Both Eiffel [36] and C++ [13] provide mechanisms for controlling which features are visible to subclasses, and/or for controlling visibility of sets of features to specific client classes. Beta [25] provides a special

construct (called “inner”) that can be used to control exactly where subclasses may extend inherited behaviour. Lamping [23] has proposed a discipline for explicitly typing the inheritance interface, which goes a long way towards turning inheritance into a black-box form of reuse.

Bracha [4] has proposed a finer granularity approach to class composition based on composition of “mix-ins.” Mix-ins are comparable to abstract classes, in the sense that they define incomplete sets of methods and instance variables, but then can be combined using a variety of operators, not just by inheritance.

As mentioned above, once functional and non-functional aspects have been combined in a single class, adapting the functional aspects in a subclass requires that the non-functional aspects be adapted as well, so the two become intertwined and can no longer be independently configured. Still, as demonstrated by the ACE toolkit[1], it is possible to (i) provide support for non-functional aspects — such as synchronization — in a set of dedicated classes, (ii) provide “pure” functional aspects in a separate set of classes, and then (iii) finally build a subclass that combines the two. No further subclassing should be done on basis of this latter class.

3.3 Object Composition

A more dynamic approach to separating non-functional and functional aspects is to use object composition. In ACE, for example, both requests and the mechanism for handling requests can be “reified” (made explicit) as objects that are separate from the object that implements the functional part [24]. Having a separate object manage the non-functional aspect of request handling (such as queuing, copying to other server objects etc.) separates this concern, and makes it easier to flexibly substitute different policies.

Object composition provides for more flexible and disciplined reuse than inheritance, since (i) object compositions may be changed at run-time, and (ii) objects are composed according to explicit interfaces. In fact, many of the basic design patterns [14] of object-oriented development introduce flexibility through object composition. For example, the effective behaviour of an object can be changed at run-time if an object delegates some of its responsibilities to other supporting objects, and these supporting objects can by dynamically substituted. Object composition addresses such diverse problems as adaptation of object interfaces, augmentation (or “decoration”) of an object’s services, dynamically changing the effective behaviour of an object, provide transparent interfaces to remote objects, and so on [14].

Object-composition involves instantiating objects, parameterizing those objects and linking them together. Objects can be parameterized by object-specific methods. Object-composition should take place in the context of a component framework that provides sets of objects that can be linked together. Links have meaning in the sense of a certain corresponding run-time cooperation.

Presently the only way to specify rules for the composition of objects is by means of the type system: objects may be composed if the dynamic type of an object conforms to the static type of the variable used to store the link. There are two shortcomings to this approach.

1. ACE is an object-oriented network programming toolkit for developing communication software; it is well-known for its design and its flexibility.

First of all, in most (typed) object-oriented languages (such as Eiffel) there is "equality" between classes and types. A class is considered to be a subtype of another only if it inherits from the latter (and also satisfies substitutability constraints). This is especially a problem if we want to be able to acquire or replace an implementation later, possibly over the net or from some independent vendor. The Java language [16] shows how the separation between interface and class can allow for this kind of "pluggability". Note that the use of "untyped" linking, as in Smalltalk or in Objective-C is not really a solution, since such mechanisms provide no support for creating correct compositions.

Second, in certain cooperations, instances from one class will not merely play the role of servers for instances of the other class. A more detailed cooperation protocol is involved. Plugging in another class, which has the same interface, but does not use that protocol leads again to an incorrect composition. Thus checking on interface only is not enough [43]. So far no object-oriented language supports checking of cooperation protocols.

Although object composition can help to make applications more flexible, it does not necessarily help make application architecture more explicit. The way in which a system is composed of objects is typically hidden in the implementation of the objects themselves. Hiding implementation details is, of course, what objects are good at, but this does not help the system architect who wants to explicitly view a system as a composition of objects. In this sense, object composition is not well supported by existing object-oriented languages.

An environment for object composition (see section 4) would not only represent compositions of objects explicitly, but would help to manage what kinds of links can be established between components. In a visual tool, type checking may not only be "corrective", that is denying incorrect links, but also "supportive" that is, suggest correct links.

Commercial tools exist that support visual object composition, but these tools are always specialized for a particular composition domain, such as user interface constructing. General commercial tools for visual composition that are adaptable to different component frameworks have not yet been introduced, though some experimental systems have been developed [34].

3.4 Class-level Black-Box Composition: Genericity

Genericity is a form of parameterization where a component, for example a class or a procedure, has a parameter which is a type (or a class) rather than a value. Genericity can be supported to varying degrees by a programming language. In C++, for example, generic classes, called "templates", are little more than glorified macros, since their parameters must be bound before any type-checking is performed. STL (The Standard Template Library) is a well-known example of the use of templates in C++ [38]. In Ada 95 and Eiffel, on the other hand, generic classes are well-integrated into the language, and can be independently type-checked and compiled. Being a special form of parameterization, genericity can be viewed as separating the variability of a software component from that component.

McHale [29] has shown that genericity can be used in concurrent object-oriented programming languages to separate the programming of the synchronization control from the "normal" programming of the methods. McHale provides many examples of "generic synchronization policies" that can be independently specified and later bound to arbitrary classes. An example is a "readers writers" policy, a synchronization policy that allows either several readers to si-

multaneously access the state of an object, or a single writer at a time. Such a policy is a generic abstraction that can be applied by (i) linking it to a certain class that should have that policy implemented, and (ii) describing which methods of the class are readers and which are writers. Using such a policy is purely a matter of black-box parameterization, and totally independent of how the policy is implemented. In McHale's work we see that the policy's dependency on the set of requests (and possibly other parameters) is separated from the component.

Contracts [18] provide another example of extending the idea of genericity, and how this can be used to specify cooperation between classes separately from the classes themselves. A contract is basically parameterized by the classes that participate in such a cooperation. Contracts help to make systems more flexible since they make it easier to substitute different classes into a given cooperation pattern. They also make systems more understandable since contracts help to make system architecture explicit and manipulable.

Genericity has also been applied to Federated Database Systems [51] (an area close to Cooperative Information Systems): Generic mechanisms are offered to describe a federated database schema in terms of the external schemas of the various databases constituting the federation: one formally describes an integration contract between the various databases in terms of the data and operations they provide, without a need for programming. These ideas are also related to the ideas described in the chapter "Reflection is the Essence of Cooperation" of this book.

If we compare class parameterization — generic classes are a special form of this — and inheritance, we can again note differences in "open" vs. "closed" forms of reuse: the white-box form of reuse supported by inheritance provides more variability, whereas the black-box reuse of class parameterization provides better ease of use and robustness. On the other hand, if we consider how to realize either generic synchronization policies of contracts in existing object-oriented languages, we encounter some difficulties. In fact, both have been supported by means of software generation, since existing object-oriented languages typically support only very weak or restricted forms of genericity.

This leads us to conclude that, contrary to some early opinions concerning the relative expressive power of genericity and inheritance [35], the notion of genericity is still underestimated as a simple mechanism for providing variability in the way objects are implemented and the way they cooperate, that works well when the possible purposes of a certain software component (e.g., a synchronization mechanism) can be known ahead, as is often the case in frameworks. Furthermore, there is a need for general mechanisms to define and realize various forms of genericity and class parameterization.

3.5 Genericity and Componentware

There has been a shift in attention in industry from general object-oriented programming systems to so-called "componentware" environments [56]. Delphi [44] and Visual Basic [37] (VB) are good examples. These approaches are also of interest here due to their black-box approach and their close relationship to class parameterization.

Componentware approaches are closely related to — or may be seen as a form of — class parameterization, since components are typically parameterized by the developer so that they can be easily adapted to different applications. A component with bound parameters is instantiated — in contrast to classes in general only once — at run-time. The parameters are not normally

classes or types, but configuration values or in some cases other components. Even if a parameter is another component, this cannot be seen as a form of genericity, since the link is basically an object composition: It means that the instance of the one will be linked to the instance of the other.

Componentware environments provide a visual presentation of components. Since components are black box entities, a visual presentation is often natural. Furthermore, most components are instantiated just once in an application, making it relatively straightforward to represent applications as static configurations of components. Finally, since many components in environments such as Delphi and Visual Basic are directly concerned with user interaction, it is natural that their visual presentation correspond directly to their interface in the final application (though, of course, the behaviour of the component will differ during application construction and run-time).

Especially interesting aspects are:

- Componentware environments draw a sharp distinction between *programmers*, who implement components, and *developers*, who use components to build applications.
- Components are implemented using relatively standard object-oriented programming techniques. It is therefore possible to develop new components by (white box) inheritance from existing components. Objects that implement components adapts their behaviour at run-time by interpreting the component parameters.
- There is already quite a large market of available ready-made components for Visual Basic and Delphi.

The basic criticisms we have are the following:

- Neither Visual Basic nor Delphi provides any standard support for (visually) linking components, checking links, or creating correct links. As a result there is not much incentive for creating domain specific component frameworks: In such a framework standard cooperations between various component are supported. Such cooperations have to be "instantiated" (declaring that two components indeed have a certain cooperation) by linking the components.
- Neither Visual Basic nor Delphi provides any support for subdividing components into separate functional and non-functional aspects.
- Neither Visual Basic nor Delphi provides any extra support for creating the dynamic part (e.g., a tree editor) of a user interface.

3.6 Separating Functional and Non-Functional Concerns

In the previously described approaches a separation between functional and non-functional behaviour could be realized by delegating non-functional aspects to a specialized component or object or superclass. Two rather different approaches to separating concerns that should be mentioned are reflection, and aspect-oriented programming.

A well-known mechanism for separating functional and non-functional behaviour is the use of a explicit reflective or "meta" computation. In such approaches, some aspect of the application is explicitly "reified," or represented as an object. One can then reason about this aspect ex-

plicitly at the meta-level, and then "reflect" the desired behaviour back into the application. For example, if messages sent to an object are themselves reified as objects, then synchronization policies can be realized by explicitly examining, manipulating, and scheduling messages at the meta-level. In addition to synchronization mechanisms ([6], [3], [28]), many other non-functional aspects have been successfully modelled using reflection, such as transaction mechanisms [54], persistence [45], and request logging [12], to mention but a few.

Aspect-oriented programming [21] is a newer approach based on the idea that each functional or non-functional aspect of a application can best be described using a separate, domain-specific language. These different aspects are than "weaved together" to produce a final program. The specification of each aspect can then be altered or adapted without affecting other, independent aspects. In contrast to meta-level computation, a separation into more than two aspects is possible, and aspects can be described declaratively.

A basic criticism applicable to both kinds of approaches is that neither supports higher-order (generic) parameterization: Non-functional behaviour is still inherently described for a specific class (and, of course for its superclasses), and is not a generic mechanism (cf. generic synchronization mechanisms [29], section 3.4) that can be parameterized by a class or by some cooperating classes. We note however, that the aspect-oriented approach seems to allow for that in principle, since it is based on software generation.

4 Requirements for a Composition Environment

We have argued that the development of open systems should be based on component frameworks, and we have shown that object-oriented technology falls short in its support for compositional development in various ways. We consequently identify a set of requirements for an environment to support developers building and using component frameworks:

- Object composition is an essential ingredient of component frameworks, which is needed for creating the static object structures used at run-time. Object composition may be used for linking very large grained objects (complete encapsulated databases) as well as small objects, such as dialogue boxes, buttons etc. Objects can be instances of classes or of "class-level components."[1]
- Genericity, or "class composition" is needed to bind parameters of class-level components, and thus to adapt the behaviour and cooperation of their instances. As in "componentware" approaches (section 3.5) but in contrast to normal object-oriented approaches that support genericity, a strong separation will be kept between the (black-box) use of a class-level component by the application developer (the user of the component framework) and by the framework developer, who may implement such a component using normal (white-box) inheritance.
- The contrast between object composition vs. class composition is fundamental and necessary. This corresponds to the fundamental contrast between intentional and extensional descriptions. In an intentional description — such as in generic synchronization poli-

1. A "class level component" is a component that is used to create object instances, but may not necessarily be implemented as a class. Generic synchronization policies, for example, are class level components, but are not implemented as classes.

cies, or in approaches to schema integration — aspects of behaviour are described in terms of "things to be", that is in terms of procedural or structural interfaces. We note that this kind of intentional composition is not provided by componentware approaches.

- Different component frameworks may require different kinds of connectors. What kinds of connectors there are should therefore be extensible. For example, the link between two interface descriptions might express standard template parameterization: the structural elements of instances of the one (e.g. of lists) must be instances of the other (e.g., visual objects); another kind of link might indicate that all requests handled by instances of the one should be forwarded to instances of the other. We note that most environments and languages only provide a limited "hard-wired" set of connector types.
- Connectors should possibly cover intra-object behaviour, e.g., changing the synchronization policies of instances, as well as inter-object behaviour.
- Following the discussion in section 3.4 and section 3.6, the environment should support arbitrary kinds of generic components for non-functional behaviour.
- Both object composition and class composition need some support to ensure that compositions are correctly constructed. The framework developer should be able to define what constitutes a correct composition (the so-called component model); the application developer should be supported in creating compositions that are compatible to the component model. Compositions should of course be made persistent.
- Implementation independence is required and principally possible in a component based approach. We see compositions as "configurations" that may name components to be used (or name class level components of which objects should be instances), but due to the black-box approach, implementation choices may be delayed to link or run time. As an example, in the approach we propose (see below) we use two complementary implementation mechanisms: One more based on software generation, the other more based on parameter interpretation. Such implementation independence is of course also relevant for portability, dynamic loading etc.

A *Visual Composition Environment* [34] 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 [34] 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.

Since generic composition is one of the novel aspects of the proposed approach we shall now give two examples of generic compositions and the corresponding composition model. We shall use the fact that not only compositions but also composition models are implementation independent: A composition model is itself an intentional description, similar to a database schema, describing what kinds of components may occur and how they may be linked. It may thus also be seen as a "composition" and thus as an implementation independent configuration.

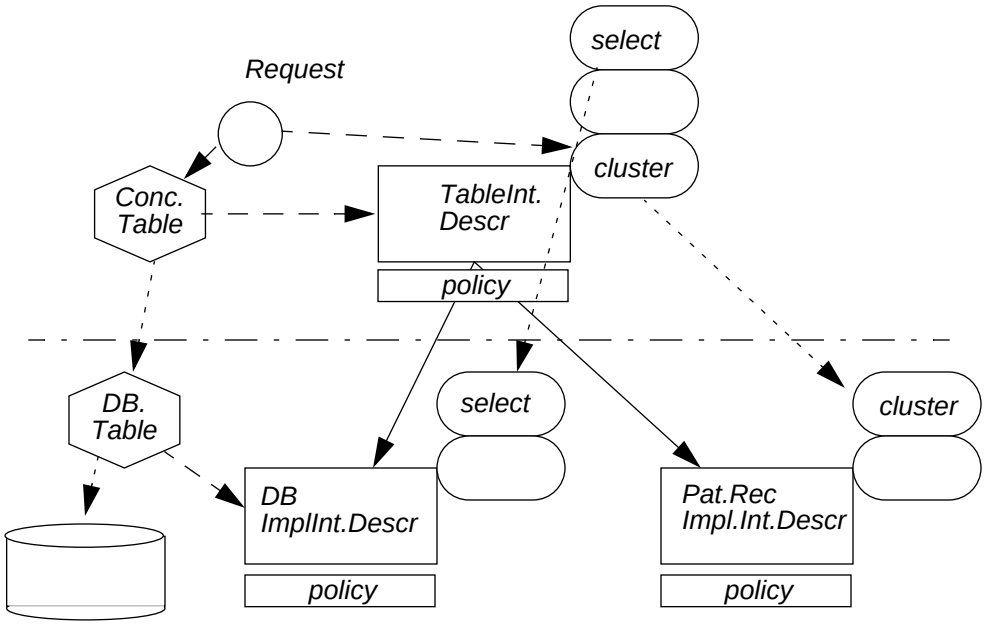

Figure 4 *A "conceptual table" is an object that provides a common interface to operations supported by several implementation objects. A "select" request, for example, is forwarded to a DB implementation, whereas a "cluster" request (i.e., to cluster table values according to some statistical properties) is forwarded to a pattern recognition component. How requests are handled is determined by a separate generic "policy" component and through the connections that exist between the elements in the interface description.*

The diagrams we used to illustrate our examples should give the reader hint as how the compositions might be represented in a visual composition environment.

4.1 Examples

We shall now consider two examples of generic compositions that make non-functional aspects explicit.

Figure 4 represents a generic composition for describing a centralized integration between a database system and a pattern recognition package used to perform statistical information analysis. This example has been taken from [31]. This form of integration is meant to hide for the user of the integrated system the fact that data and operations are located in different packages and possibly different machines. It furthermore hides the fact that in order to apply statistical pattern recognition to numerical data in a database, the data have to be copied and transferred from the database to the pattern recognition program. This means that the data as the user sees them (called the "conceptual objects") may have more than one representation in the underlying packages. Execution of a user request therefore entails relatively complex non-functional behaviour in the integrating system. The choice of a specific generic policy represents the fact that this kind of request execution is used. We note however that this policy needs certain information to be available in the generic composition, e.g., which conceptual operation corresponds to which implementation operation. Thus, this policy can only be used together with a composition model that enforces the existence of such links. Figure 5 shows the corresponding composition model.

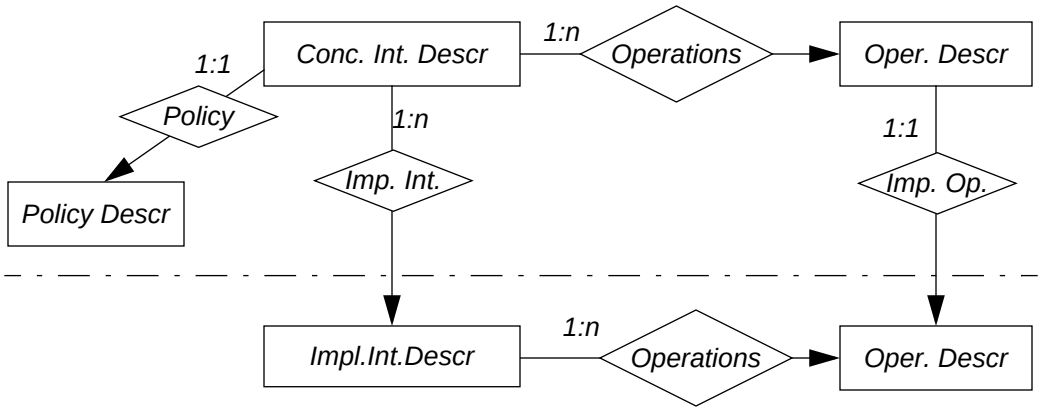

Figure 5 *Composition model for the generic composition given in figure 4. The diamonds are called property descriptors correspond to association descriptors in UML: For instance the descriptor "Policy" indicates that a Conc. Interface Descriptor must be linked to one policy component.*

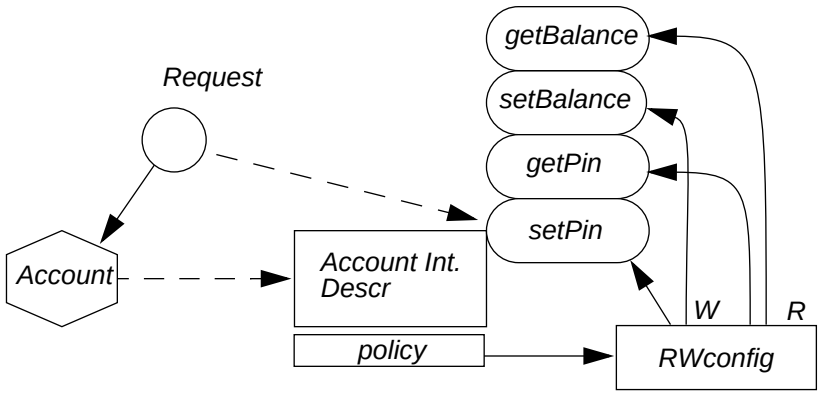

Figure 6 *Configuration of Synchronization policy for accounts: In this example a readers writers policy. The readers writers policy is configured by identifying the set of readers (in this case "getPin", "getBalance") and the set of writers ("setBalance", "setPin")*

Figure 6 represents a generic composition describing the coupling of a "Readers Writers" synchronization policy [29] to a class component of which instances are objects representing accounts. This example has been taken from [10]. In this example we see that a possibility exists to exchange policies. Figure 7 shows the corresponding composition model. We see specifically how for a specific policy as in this case the "Readers Writers" a specific part of the composition model is given that specifies how such a policy should be configured.

4.2 Towards a Visual Composition Environment

Visual formalisms are important for specifying and representing software composition because they can support multiple views of the same structures, they can provide important visual cues to aid understanding, and because they can directly represent the final application interface, and hence can conveniently support a direct manipulation paradigm during development.

Framework developers must be able to specify and represent generic architectures, components, component interfaces and glue using a high-level graphical "syntax." Application devel-

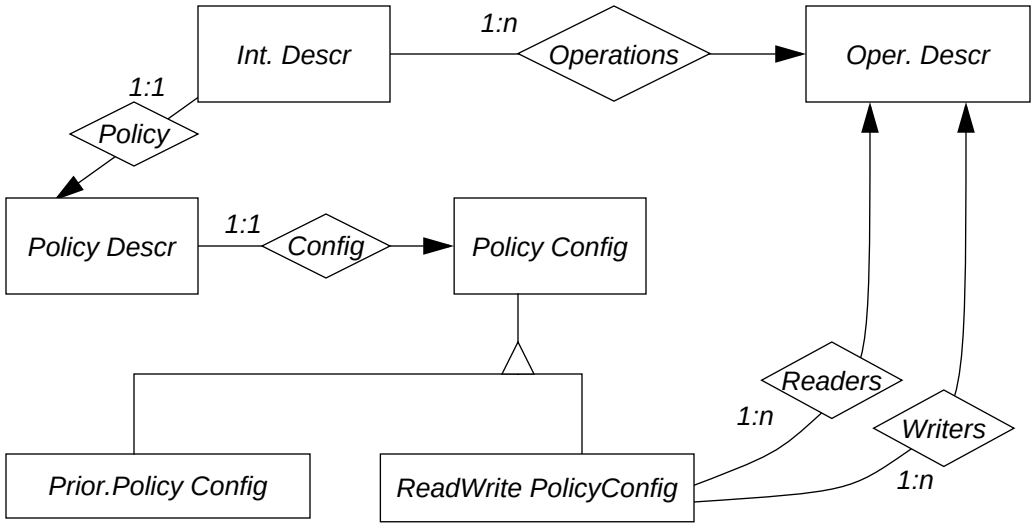

Figure 7 *Composition model for the generic composition given in figure 6.*

opers should be able to instantiate architectures by elaborating, binding and linking framework components. Abstractions must be available as explicitly manipulable (and visually represented) entities in the composition environment. Composition structures must be mappable to language sentences. The visual environment should support the user actively in creating and adapting compositions correctly. The environment should be homogeneous with respect to either object or class level composition.

The visual environment should be configurable by the composition model and to specifications how the components have to be presented. We note that this need for composition model configurability is surely one of the reasons why there has not yet been any commercial implementation of an open visual composition environment: all existing visual composition tools address a very specific application domain, typically user interface construction.

Since:

1. it is (should be) relatively simple through composition models to define new kinds of connectors (by just defining a new kind of link between class level components), and
2. a visual environment “reifies” — represents as explicit object structures — component structures including generically linked class structures, and
3. some form of run-time testing and/or debugging of compositions is needed in a visual composition environment

It is attractive for a visual composition environment to support the possibility to give run-time meaning to compositions by interpreting the information in the structure[1]. This means that the composition environment should be extensible in this sense as well: for new generic links new interpretation mechanisms must be introduced.

We now briefly illustrate each of these points for the two composition examples.

1. Another possibility would be to generate code from (compile) the class composition.

Visual Composition Support

Supporting visual composition for either one of the examples corresponds to making the diagrams shown in figure 4 and in figure 6 into explicitly manipulable structures represented as objects and links between objects. The environment can provide a separate "toolbox" window of components that may be instantiated. In the case of an interface description with its policy, the toolbox will contain, amongst others, possible different policies to be created and operations and interface descriptor components themselves. The environment checks the links the developer attempts to create between components: For example the developer can only create links from the readers property to one of the operations, and not to the interface descriptor itself. The environment provides a special "global check" operator, it checks whether all connections are consistent with each other; for example, a specific operation cannot both be a reader and a writer. The composition can only be used for run-time execution if all connections are consistent in this global sense.

Adaptability of Run-time Semantics

Run-time behaviour of a system depends on the properties (composition) of the run-time objects, and the generic information. As mentioned, in the visual composition environment there will be a run-time execution mechanism based on interpretation of that information. In such an approach, run-time behaviours of objects in the system are determined by:

1. The underlying implementation of the object in a class, which must take into account:
2. The properties of the object, and the information in the class structure

In both examples, the object to which a request is applied has an "execute" method for executing a request. In order to allow for explicit request handling, requests themselves are first class objects, as mentioned in section 3.3. Requests for an operation are instances of the corresponding interface description of that operation; the request shown in figure 4 is thus a request for a cluster operation. In both examples, most relevant information for the execution of the request is in the class structure: the policy and the generic links.

In order to allow this information to be interpreted for request execution it has to be explicitly available as an object structure. As mentioned before, this is no problem since the class structure is explicitly represented as an object structure during composition anyhow. We must ask, however, how such a structure can play both the role of an object that can be queried by the run-time objects, and that of a class that they can be asked to generate instances. For this we use the prototype design pattern [14]: each class object carries a prototype that is copied when the class object is requested for instantiation.

Interpreting the information in the class structure by the "execute" will do the following: It will delegate the knowledge of how to execute the request (the non-functional part, at least) to the same-named "execute" method in the policy object. It is this method that interprets the knowledge in the class structure, for example in the case of the centralized integration, the method will find which implementation operation has to be executed, on which kind of implementation object (in the example: cluster implementation on a pattern recognition object). It will generate the new object to which the implementation operation must be executed and create a request for that implementation operation. The execution of the implementation request is again determined by a policy.

In general, we say that the adaptation of the behaviour of the run-time objects to the class composition is done through "up-calls" to methods defined in the class objects.

Adaptability of the Visual Composition Support to the Composition Model

A visual composition environment has to adapt itself to the chosen composition model. Since the composition model is itself an intentional "class level" structure, the environment adapts itself in the same way to the composition model as (other) run-time systems adapt themselves to "normal" class compositions as described in the previous paragraph. Thus, it itself is an example of an application of giving run-time semantics through interpretation to such a class composition. This goes as follows: the composition model is explicitly represented as an object structure and attempted links are checked by querying that structure. When, for example, the developer attempts to link a read-writers policy via the "Readers" property to one of the operations, the visual composition environment can query the corresponding "Readers" property descriptor to find out if the target of the link is an instance of the right component type, in this case, whether the target is an "Operation Descriptor".

Current Research

In our latest research on visual composition environments we have achieved the following:

- The principle of interpreting generic class information has been worked out for some smaller examples [32],[33]
- The principle of having visual composition being checked on basis of a composition model has been worked out and tested [32].

Since the composition model is itself a class composition (see above), there must also be a composition model of this composition model etc. This leads to the need for a self-descriptive composition model. This and the need for dynamic adaptation —the environment should be adapted to a new composition model without having to recompile it— poses severe requirements on a kernel implementation in the form of a self-descriptive data model [31]. This kernel data model has been implemented in Self [57] and C++ [13]. Current work is focused on developing more realistic examples and further elaborating our component model.

4.3 Towards Compositional 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 [7] are not yet well-understood or widely practised.

A Component-Oriented Software Lifecycle (figure 8) must take into account that *application development* (the construction of applications from component frameworks) is a separate

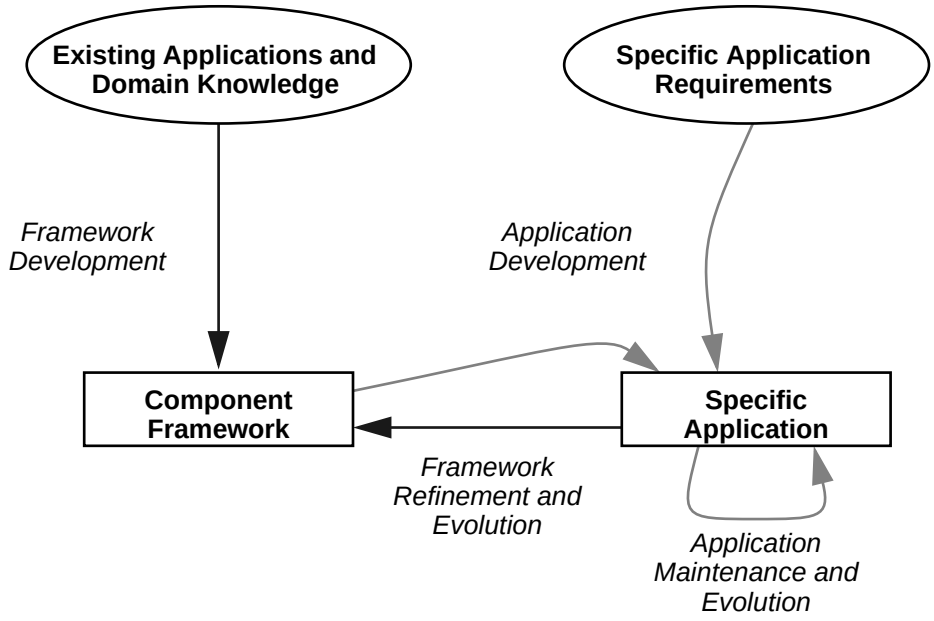

Figure 8 *A Component-Oriented Software Lifecycle*

activity from *framework development* (the iterative development of the framework itself) [40]. Framework development is capital investment whereas application development recovers the investment.

Since application development is ideally *driven* by framework development, 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.

5 Concluding Remarks

Open systems pose special requirements for software development tools and methods. Open systems must be easily adaptable to changing requirements, hence should be designed with generic requirements in mind. A component framework addresses changing requirements by providing a generic software architecture for a family of applications, and a set of components that can be configured and composed in a variety of ways.

Object-oriented languages and systems support the development of component frameworks to some degree, but suffer from a number of limitations. Object-oriented languages support both "black-box" and "white-box" components. The former are fully encapsulated and can be used in arbitrary contexts, whereas the latter may introduce implementation dependencies between components and their clients: subclasses, for example, may depend on implementation details of superclasses they inherit from, thus violating encapsulation.

Object-oriented languages also typically force one to view all kinds of components as objects, whether this model is appropriate or not. A cooperation pattern encapsulating a readers-

writers synchronization policy, for example, would be a perfectly reasonable component, but does not make much sense to represent as an object.

Finally, object-oriented systems tend to hide application architecture rather than make it explicit and manipulable. This is an obstacle to open systems comprehension and evolution. A composition environment would support open systems development by explicitly representing components and their interfaces, and by managing and guiding the composition activity.

Component-oriented software development is notably distinct from traditional development because it forces a separation between framework development and application development. These two activities are interdependent, since a component framework should drive application development, while at the same time, experiences with application development influence the iterative design of frameworks. Composition environments will provide tools needed to support these two activities, but they do not tell us what methods we should use to develop and apply software components. Most of the well-known object-oriented methods do not say anything about framework development or reuse. This is where we can expect to see the most significant advances in the near future [48].

References

[1] Mehmet Aksit and Anand Tripathi, "Data Abstraction Mechanisms in SINA/ST," *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, vol. 23, no. 11, Nov 1988, pp. 267-275.

[2] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans and Akinori Yonezawa, "Abstracting Object Interactions Using Composition Filters," *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, R. Guerraoui, O. Nierstrasz and M. Riveill (Ed.), LNCS 791, Springer-Verlag, 1994, pp. 152-184.

[3] Mehmet Aksit, Jan Bosch, William van der Sterren and Lodewijk Bergmans, "Real-Time Specification Inheritance Anomalies and Real-Time Filters," *Proceedings ECOOP'94*, M. Tokoro, R. Pareschi (Ed.), LNCS 821, Springer-Verlag, Bologna, Italy, July 1994, pp. 386-407.

[4] Gilad Bracha, "The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance," Ph.D. thesis, Dept. of Computer Science, University of Utah, March 1992.

[5] Jean-Pierre Briot and Rachid Gerraoui, *A Classification of Various Approaches for Object-Based Parallel and Distributed Programming*, february 1996, Technical Report, Ecole Polytechnique Federale de Lausanne & University of Tokyo, 1996.

[6] Jean-Pierre Briot, "An Experiment in Classification and Specialization of Synchronization Schemes," *LNCS*, vol. 1049, Springer Verlag, 1996, pp. 227-249.

[7] 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.

[8] 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.

[9] Luca Cardelli, Florian Matthes and Martin Abadi, "Extensible Syntax with Lexical Scoping", *SRC Research Report digital*, Feb. 1994.

[10] Juan Carlos Cruz and Sander Tichelaar, "A Coordination Component Framework for Open Systems," Working Paper, IAM, University of Bern, 1996.

[11] L. Peter Deutsch, *Design Reuse and Frameworks in the Smalltalk-80 System*, T.J. Biggerstaff and A.J. Perlis (Ed.), vol. II, ACM Press & Addison-Wesley, Reading, Mass., 1989, pp. 57-71.

[12] Serge Demeyer, "ZYPHER Tailorability as a link from Object-Oriented Software Engineering to Open Hypermedia," Ph.D. thesis, Vrije Universiteit Brussel Departement Informatica, 1996.

[13] Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[14] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Addison Wesley, Reading, MA, 1995.

[15] Adele Goldberg and Kenneth S. Rubin, *Succeeding With Objects: Decision Frameworks for Project Management*, Addison-Wesley, Reading, Mass., 1995.

[16] James Gosling and H. McGilton, *The Java Language Environment*, Sun Microsystems Computer Company, May 1995.

[17] Franz J. Hauck, "Inheritance Modeled with Explicit Bindings: An Approach to Typed Inheritance," *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, vol. 28, no. 10, Oct. 1993, pp. 231-239.

[18] Richard Helm, Ian M. Holland and Dipayan Gangopadhyay, "Contracts: Specifying Behavioural Compositions in Object-Oriented Systems," *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, vol. 25, no. 10, Oct. 1990, pp. 169-180.

[19] 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.

[20] Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

[21] Gregor Kiczales, "Aspect-Oriented Programming: A Position Paper From the Xerox PARC Aspect-Oriented Programming Project," *Special Issues in Object-Oriented Programming*, Max Muehlhauser (Ed.), to appear. See also: http://www.parc.xerox.com/spl/projects/aop/position.html

[22] Dimitri Konstantas, "Interoperation of Object-Oriented Applications," *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Ed.), Prentice Hall, 1995, pp. 69-95.

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

[24] R. Greg Lavender and Douglas C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," *Proc.Pattern Languages of Programs*, James O. Coplien (Ed.), September 1995.

[25] Ole Lehrmann Madsen, Birger Møller-Pedersen and Kristen Nygaard, *Object-Oriented Programming in the Beta Programming Language*, Addison-Wesley, Reading, Mass., 1993.

[26] Jeff Magee, Naranker Dulay and Jeffrey Kramer, "Structuring Parallel and Distributed Programs," *Proceedings of the International Workshop on Configurable Distributed Systems*, London, March 1992.

[27] 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.

[28] Jeff McAffer, “Meta-level Programming with CodA,” *Proceedings ECOOP ’95*, W. Olthoff (Ed.), LNCS 952, Springer-Verlag, Aarhus, Denmark, August 1995, pp. 190-214.

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

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

[31] Theo Dirk Meijler, “User-level Integration of Data and Operation Resources by means of a Self-descriptive Data Model,” Ph.D. thesis, Erasmus University Rotterdam, Sept. 1993.

[32] Theo Dirk Meijler and Robert Engel, “Making Design Patterns explicit in FACE, a Framework Adaptive Composition Environment”, *EuroPLoP preliminary Conference Proceedings*, July 1996.

[33] Theo Dirk Meijler, Serge Demeyer and Robert Engel, “Class Composition in FACE, a Framework Adaptive Composition Environment,” *Special Issues in Object-Oriented Programming*, Max Muehlhauser (Ed.), to appear.

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

[35] Bertrand Meyer, “Genericity versus Inheritance,” *Proceedings OOPSLA ’86, ACM SIGPLAN Notices*, vol. 21, no. 11, Nov 1986, pp. 391-405.

[36] Bertrand Meyer, *Eiffel: The Language*, Prentice Hall, 1992.

[37] Microsoft Corporation, *Visual Basic Programmer’s Guide*, 1993 .

[38] David R. Musser and Atul Saini, *STL Tutorial and Reference Guide*, Addison-Wesley, 1996.

[39] 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.

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

[41] 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.

[42] 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.

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

[44] Xavier Pacheco and Steve Teixeira, “Delphi Developer's Guide”, Borland Press/Sams Publishing

[45] Andreas Paepcke, “PCLOS: A Flexible Implementation of CLOS Persistence,” *Proceedings ECOOP’88*, S. Gjessing and K. Nygaard (Ed.), LNCS 322, Springer-Verlag, Oslo, August 15-17, 1988, pp. 374-389.

[46] 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.

[47] Xavier Pintado, “Gluons and the Cooperation between Software Components,” *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Ed.), Prentice Hall, 1995, pp. 321-349.

[48] Trygve Reenskaug, Per Wold and Odd Arild Lehne, *Working With Objects*, Manning Publications, 1996.

[49] Michael I. Schwarzbach, “Formal Design Constraints”, to appear in OOPSLA ‘96.

[50] Mary Shaw and David Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

[51] Amit P. Sheth and James A. Larson, “Federated Database Systems for Managing Distributed Heterogeneous, and Autonomous Databases,” *ACM Computing Surveys*, vol. 22, no. 3, September 1990, pp. 183-236.

[52] Randall B. Smith and David Ungar, “Programming as an Experience: The Inspiration for Self,” *Proceedings ECOOP ’95*, W. Olthoff (Ed.), LNCS 952, Springer-Verlag, Aarhus, Denmark, August 1995, pp. 303-330.

[53] 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.

[54] R.J. Stroud and Z. Wu, “Using Metaobject Protocols to Implement Atomic Data Types,” *Proceedings ECOOP’95*, W. Olthoff (Ed.), LNCS 952, Springer-Verlag, Aarhus, Denmark, August 1995, pp. 168-189.

[55] Dennis Tsichritzis, “Object-Oriented Development for Open Systems,” *Information Processing 89 (Proceedings IFIP ’89)*, North-Holland, San Francisco, Aug 28-Sept 1, 1989, pp. 1033-1040.

[56] James Udell, “Componentware,” *Byte*, vol. 19, no. 5, May 1994, pp. 46-56.

[57] David Ungar and Randall B. Smith, “Self: The Power of Simplicity,” *Proceedings OOPSLA ’87, ACM SIGPLAN Notices*, vol. 22, no. 12, Dec. 1987, pp. 227-242.

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