Chapter 1

Component-Oriented Software Technology*

Oscar Nierstrasz and Laurent Dami

Abstract Modern software systems are increasingly required to be open and distributed. Such systems are open not only in terms of network connections and interoperability support for heterogeneous hardware and software platforms, but, above all, in terms of evolving and changing requirements. Although object-oriented technology offers some relief, to a large extent the languages, methods and tools fail to address the needs of open systems because they do not escape from traditional models of software development that assume system requirements to be closed and stable. We argue that open systems requirements can only be adequately addressed by adopting a *component-oriented* as opposed to a purely object-oriented software development approach, by shifting emphasis away from programming and towards generalized software composition.

1.1 Introduction

There has been a continuing trend in the development of software applications away from closed, proprietary systems towards so-called open systems. This trend can be largely attributed to the rapid advances in computer hardware technology that have vastly increased the computational power available to end-user applications. With new possibilities come new needs: in order to survive, competitive businesses must be able to effectively exploit new technology as it becomes available, so existing applications must be able to work with new, independently developed systems. We can see, then, that open systems must be "open" in at least three important ways [49]:

1. *Topology*: open applications run on configurable networks.
2. *Platform:* the hardware and software platforms are heterogeneous.
3. *Evolution:* requirements are unstable and constantly change.

* In *Object-Oriented Software Composition*, O. Nierstrasz and D. Tsichritzis (Ed.), Prentice Hall, 1995, pp. 3-28. For more information, please see: http://iamwww.unibe.ch/~oscar/OOSC/

Object-oriented software development partially addresses these needs by hiding data representation and implementation details behind object-oriented interfaces, thus permitting multiple implementations of objects to coexist while protecting clients from changes in implementation or representation. Evolution is only partially addressed, however, since changes in requirements may entail changes in the way that the objects are structured and configured. In fact, to address evolution, it is necessary to view each application as only one instance of a *generic class* of applications, each built up of reconfigurable software components. The notion of component is more general than that of an object, and in particular may be of either much finer or coarser granularity. An object encapsulates data and its associated behaviour, whereas a component may encapsulate *any* useful software abstraction. Since not all useful abstractions are necessarily objects, we may miss opportunities for flexible software reuse by focusing too much on objects. By viewing open applications as compositions of reusable and configurable components, we expect to be able to cope with evolving requirements by unplugging and reconfiguring only the affected parts.

1.1.1 What Are Components?

If we accept that open systems must be built in a component-oriented fashion, we must still answer the following questions: What exactly are components, and how do they differ from objects? What mechanisms must programming languages and environments provide to support component-oriented development? Where do components come from in the software development lifecycle, and how should the software process and methods accommodate them?

In attempting to answer these questions, we must distinguish between methodological and technical aspects. At a methodological level, a component, we will argue, is a component because it has been *designed* to be used in a compositional way together with other components. This means that a component is not normally designed in isolation, but as part of a *framework* of collaborating components. A framework may be realized as an abstract class hierarchy in an object-oriented language [23], but more generally, components need not be classes, and frameworks need not be abstract class hierarchies. Mixins, functions, macros, procedures, templates and modules may all be valid examples of components [3], and component frameworks may standardize interfaces and generic code for various kinds of software abstractions. Furthermore, components in a framework may also be other entities than just software, namely specifications, documentation, test data, example applications, and so on. Such components, however, will not be discussed in detail in this paper: we will mainly concentrate on some technical aspects related to software components.

At a software technology level, the vision of component-oriented development is a very old idea, which was already present in the first developments of structured programming and modularity [32]. Though it obtained a new impulse through the compositional mechanisms provided by object-oriented programming languages, component-oriented soft-

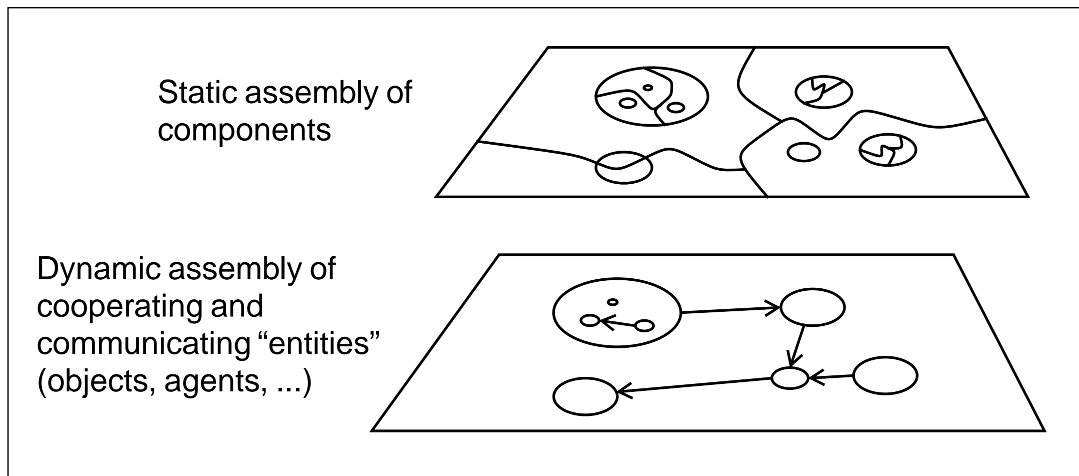

Figure 1.1 *Static and dynamic views of an application.*

ware development is not easy to realize for both technological and methodological reasons. For a programming language to support component-oriented development, it must cleanly integrate both the *computational* and the *compositional* aspects of software development. An application can be viewed simultaneously as a computational entity that delivers results, and as a construction of software components that fit together to achieve those results (figure 1.1). A component *per se* does not perform any computation, but may be combined with others so that their composition does perform useful computations, much in the way that the parts of a machine do not necessarily perform any function individually, but their composition does. The integration of these two aspects is not straightforward, however, since their goals may conflict. To take a concrete example, concurrency mechanisms, which are computational, may conflict with inheritance, which is a a compositional feature, in that implementation details must often be exposed to correctly implement inheriting subclasses [26] [31] (see chapter 2 for a detailed discussion of the issues). To complicate things even further, the distinction between "composition time" and "run time" is not always as clear as in the picture above: with techniques such as dynamic loading, dynamic message lookup or reflection, applications can also be partially composed or recomposed at run-time.

In order to achieve a clean integration of computational and compositional features, a common semantic foundation is therefore needed in which one may reason about both kinds of features and their interplay. As we shall see, the notions of *objects*, *functions* and *agents* appear to be the key concepts required for such a foundation. In consequence, we will adopt a definition of software component which is sufficiently abstract to range over these various paradigms.

In short, we say that a component is a *"static abstraction with plugs"*. By "static", we mean that a software component is a long-lived entity that can be stored in a software base, independently of the applications in which it has been used. By "abstraction", we mean that a component puts a more or less opaque boundary around the software it encapsulates.

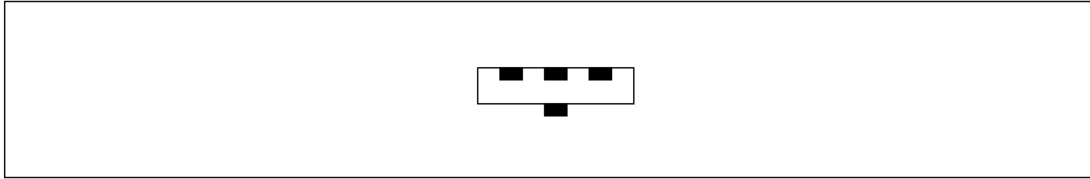

Figure 1.2 *A software component and its plugs.*

"With plugs" means that there are well-defined ways to interact and communicate with the component (parameters, ports, messages, etc.). So, seen from the outside, a component may appear as in figure 1.2: a single entity, which may be moved around and copied, and in particular may be instantiated in a particular context, where the plugs (the small black rectangles) will be bound to values or to other components. In fact, such visual representations of components can be very convenient for supporting interactive composition of applications from component frameworks (see chapter 10). *Software composition*, then, is the process of constructing applications by interconnecting software components through their plugs. The nature of the plugs, the binding mechanisms and the compatibility rules for connecting components can vary quite a bit, as we shall see, but the essential concepts of components, plugs, plug-compatibility and composition remain the same.

1.1.2 Where Do Components Come From?

Once the programming language and associated tools support the development of components, we are still left with the question, "Where do the components come from?" Although we argue that a component-oriented approach is necessary to deal with evolving requirements, it turns out that components themselves only emerge through an iterative and evolutionary software lifecycle. This is reasonable, if we consider that components are only useful as components if they can be easily used in many contexts. Before a "reuseful" component can be designed [23], one must first collect, understand and analyze knowledge about these different contexts to determine how their different needs can be addressed by some common frameworks. When component frameworks are put to use, they must be evaluated with respect to how easily they can be applied to new problems, and improvements must then be introduced on the basis of new experience. Component-oriented development is therefore a *capital-intensive activity* that treats component frameworks as capital goods (or "reusable assets"), and requires investment in component development to achieve economic benefits in the long-term [53]. This means that not only must the programming language technology and support environment address the technical requirements of component-oriented development, but the entire software process, including the analysis and design methods, must incorporate the activity of "component engineering" into the software lifecycle.

Udell, who has provocatively proclaimed the "failure of object-oriented systems to deliver on the promise of software reuse," [50] supports this view by arguing that sets of

components, such as those delivered with VisualBasic are a much more successful example of software reuse than object-oriented programming. An animated discussion followed on the Internet* which finally came to the obvious agreement that successful software reuse is a matter of methodology and design, more than technology; so object-oriented systems cannot be taken as responsible for lack of reusability: they are more likely to help in producing reusable software, provided that the right design decisions are taken in the first place. Additional arguments on the same line can be found in [22], where various authors discuss software reuse not only in terms of technology, but above all in terms of economical, human and organizational factors.

Our position is that both software methods and development technology need to undergo some significant changes in order to take advantage of component-oriented development. We will first focus on some of the foundational issues concerning the difference between objects and components, and their integration in programming languages and environments; then we will briefly survey related technological and methodological problems to be resolved; finally, we will conclude with some prospects for the future of component-oriented development.

1.2 Objects vs. Components

Object-oriented programming languages and tools constitute an emerging software technology that addresses the development of open systems in two important ways:

1. as an *organizingprinciple;*
2. as a *paradigm for reuse.*

In the first case, one may view an object-oriented application as a collection of collaborating objects. The fact that each object properly encapsulates both the data and the corresponding behaviour of some application entity, and that one may only interact with this entity through a well-defined interface means that reliability in the face of software modifications is improved, as long as client–server interfaces are respected. In the second case, one may view applications as compositions of both predefined and specialized software components. Application classes inherit interfaces and some core behaviour and representation from predefined abstract classes. Interactions within an application obey the protocols defined in the generic design. Inheritance is the principle mechanism for sharing and reusing generic designs within object-oriented applications.

Despite these two significant advantages of object-oriented development, it is still true that present-day object-oriented languages emphasize *programming* over *composition*, that is, they emphasize the first view of applications to the detriment of the second. In general, it is not possible to reuse classes without programming new ones — one cannot simply compose object classes to obtain new classes in the way that one can compose

---

* The discussion took place during September 1994 in the newsgroup comp.object, under the subject heading “Objects vs Components.”

functions to obtain new functions. Furthermore, one is either forced to define a given component as a class, whether or not the object paradigm is an appropriate one, or, if other kinds of components are supported, the list is typically *ad hoc* (for example, mixins, macros, modules, templates).

If we consider the various dimensions of programming languages supporting some notion of objects, we discover a mix of features concerned with computational and compositional issues. Wegner [54] has proposed a classification scheme with the following seven "dimensions": objects, classes, inheritance, data abstraction, strong typing, concurrency and persistence. According to the criterion that sets of features are orthogonal if they occur independently in separate programming languages, it turns out that objects, abstraction, types, concurrency and persistence are orthogonal. But this does not tell us how easy or difficult it is to cleanly integrate combinations of features within a single language.

In fact, if we consider just objects, classes and inheritance, it turns out that it is not at all straightforward to ensure both object encapsulation and class encapsulation in the presence of inheritance [47]. One way of explaining this is that classes are overloaded to serve both as templates for instantiating objects and as software components that can be extended by inheritance to form new classes. Typically, these two roles are not cleanly separated by the introduction of separate interfaces. Instead, various *ad hoc* rules must be introduced into each object-oriented programming language to determine what features of a class may be visible to subclasses. Since these rules cannot possibly take into account the needs of all possible component libraries, the net effect is that encapsulation must often be violated* in order to achieve the desired degree of software reusability.

A reasonably complete programming language for open systems development should not only support objects and inheritance, but also strong typing and concurrency. Types are needed to formalize and maintain object and component interfaces, and concurrency features are needed to deal with interaction between concurrent or distributed subsystems. (Fine-grain parallelism is also of interest, but is not an overriding concern.) Though types and concurrency are supposedly orthogonal to objects and inheritance, their integration is not a simple matter.

One source of difficulty for types is that objects are not simply values taken in isolation, like integers, strings, higher-order functions, or even more complex constructs such as abstract datatypes. Objects typically belong to a global context, and may contain references to other objects in that context. Furthermore, since they are dynamic entities, they may change behaviour or state, and hence the meaning of references changes over time. Hence, extracting static type information from such dynamic systems is considerably more difficult. Modelling inheritance is also problematic, due to the two different roles played by classes. Many difficulties in early attempts arose from efforts to identify inheritance and subtyping. It turns out, on the contrary, that subtyping and inheritance are best considered

---

* We say that encapsulation is violated if clients of a software component must be aware of implementation details not specified in the interface in order to make correct use of the component. In particular, if changes in the implementation that respect the original interface may affect clients adversely, then encapsulation is violated. If the inheritance interface cannot be separately specified, then encapsulation can be violated when implementation changes cause subclasses to behave incorrectly.

as independent concepts [1] [7]. It may even be convenient to have a separate notion of type for the inheritance interface [28].

When concurrency is also brought into the picture, the same conflicts are seen to an exaggerated degree:

1. Concurrency features may conflict with object encapsulation if clients need to be aware of an object's use of these features [45] (see chapter 2).
2. Class encapsulation may be violated if subclasses need to be aware of implementation details [26] [31].
3. Type systems generally fail to express any aspect of the concurrent behaviour of objects that could be of interest to clients (such as the requirement to obey a certain protocol in issuing requests — see chapter 4).

The source of these technical difficulties, we claim, is the lack of a sufficiently component-oriented view of objects. Components need to be recognized as entities in their own right, independently of objects. A class as a template for instantiating objects is one kind of component with a particular type of interface. An object is another kind of component with an interface for client requests. A class as a generator for subclasses is yet another kind of component with a different kind of interface. Each of these components has its own interface for very different purposes. It is possible to provide syntactic sugar to avoid a proliferation of names for all of these different roles, but the roles must be distinguished when the semantics of composition is considered.

The other lesson to learn is that each of these dimensions cannot simply be considered as an "add-on" to the others. An appropriate semantic foundation is needed in which to study the integration issues. If state change and concurrency are modelling requirements, then a purely functional semantics is not appropriate. As a minimum, it would seem that a computational model for modelling both objects and components would need to integrate both *agents* and *functions*, since objects, as computational entities, can be viewed as particular kinds of communicating agents, whereas components, as compositional entities, can be seen as abstractions, or functions over the object space. Moreover, since components may be first-class values, especially in persistent programming environments where new components may be dynamically created, it is essential that the agent and function views be consistently integrated. From the point of view of the type system, both objects and components are typed entities, although they may have different kinds of types.

1.3 Technical Support for Components

Component-oriented software development not only requires a change of mind-set and methodology: it also requires new technological support. In this section, we will review some of the issues that arise:

- What are the *paradigms* and *mechanisms* for binding components together?
- What is the *structure* of a software component?

- At which stage do composition decisions occur, i.e. how can we characterize the *composition process*?
- How do we formally model components and composition, and how can we *verify* that fragments are correctly composed?
- To which extend does a *concurrent* computational model affect software composition?

These questions obviously are interrelated; moreover, they depend heavily on the composition paradigm being used. We have argued that, ideally, a complete environment for software composition should somehow provide a combination of objects, functions and agents. So far, these paradigms have evolved quite independently. In order to combine them into a common environment, considerable care must be taken to integrate them cleanly. In the following, we examine the specific contributions of each paradigm to software composition, we discuss how they may be integrated, and we summarize the principle open research problems.

1.3.1 Paradigms for Assembling Components

Probably the most fundamental composition mechanism to mention is *functional* composition. In this paradigm one entity is first encapsulated and parameterized as a functional abstraction, and is then "activated" (instantiated) by receiving arguments that are bound to its parameters. Obviously this compositional mechanism occurs in nearly every programming environment, and is by no means restricted to functional programming languages. Many languages, however, do not allow arbitrary software entities to be treated as values, and therefore do not support functional composition in its most general form. Parameterized modules, containing variables that can be bound later to other modules, for example, are still absent from many programming languages. At the other end of the spectrum, functional languages use functional composition at every level and therefore provide *homogeneity:* any aspect of a software fragment can be parameterized and then bound to another component, thereby providing much flexibility for delimiting the boundaries of components. Furthermore, functional programming supports *higher-order* composition, i.e. components themselves are data. In consequence, composition tasks themselves can be encapsulated as components, and therefore some parts of the composition process can be automated. Finally, functional composition has the nice property of being easily verifiable, since functions can be seen externally as black boxes: under some assumptions about the parameters of a function, it is possible to deduce some properties of the result, from which one can know if that result can safely be passed to another function. Current functional programming languages have developed sophisticated type systems to check correctness of composed software [37][21].

Functional composition is a local composition mechanism, in the sense that it only involves one abstraction and the values passed as parameters. By contrast, agent environments typically use a global composition mechanism, often called a *blackboard*. A blackboard is a shared space, known by every component, in which information can be put

and retrieved at particular *locations*. For systems of agents communicating through channels, the blackboard is the global space of channel names. Even without agents, global memory in traditional imperative programming also constitutes a kind of blackboard. Blackboard composition supports *n*-ary assemblies of components (whereas local composition mechanisms are mostly binary); furthermore, free access to the shared space imposes less constraints on the interface of components. The other side of the coin, however, is that blackboard composition systems are much more difficult to check for correctness because interaction between components is not precisely localized. As a partial remedy to the problem, blackboard composition systems often incorporate encapsulation mechanisms for setting up boundaries inside the global space within which interference is restricted to a well-known subset of components. By this means, at least some local properties of a blackboard system can be statically verified. The π-calculus [35], for example, has an operator to restrict the visibility of names; in the world of objects, *islands* [19] have been proposed as a means to protect local names and avoid certain traditional problems with aliasing.

Finally, object-oriented systems have introduced a new paradigm for software composition with the notion of *extensibility* — the possibility of adding functionality to a component while remaining "compatible" with its previous uses. Extensibility, typically obtained in object-oriented languages through inheritance or delegation, is an important factor for smooth evolution of software configurations. The delicate question, however, is to understand what *compatibility* means exactly. For example, compatibility between classes is usually decided on the basis of the sets of methods they provide, possibly with their signatures; in the context of active objects, this view does not take into account which *sequences of methods invocations* are accepted by an object. Chapter 4 studies how to capture this aspect through so-called regular types. Moreover, compatibility can be meaningful not only for classes, but for more generalized software entities; in particular, object-oriented systems based on prototypes and delegation need to understand compatibility directly at the level of objects. Chapter 6 investigates a functional calculus in which compatibility is defined at a fundamental level, directly on functions.

Figure 1.3 is an attempt to represent visually the different paradigms. Functional composition is pictured through the usual image of functions as boxes, with parameters represented as input ports and results of computation as output ports. Connections between components are established directly and represent bindings of values to formal parameters. The blackboard paradigm has an addressing scheme that structures the global space; it sometimes also uses direct connections, but in addition, components are put at specific locations, and they may establish connections with other components through their locations. Here locations are pictured as coordinates in a two-dimensional space for the purpose of the visual illustration. In practice, the common space will most often be structured by names or by linear memory addresses. Finally, extensibility is pictured by additional ports and connections added to an existing component, without affecting the features that were already present. Seen at this informal level, it is quite clear that some cohabitation of the paradigms should be possible, but it is also clear that many details need

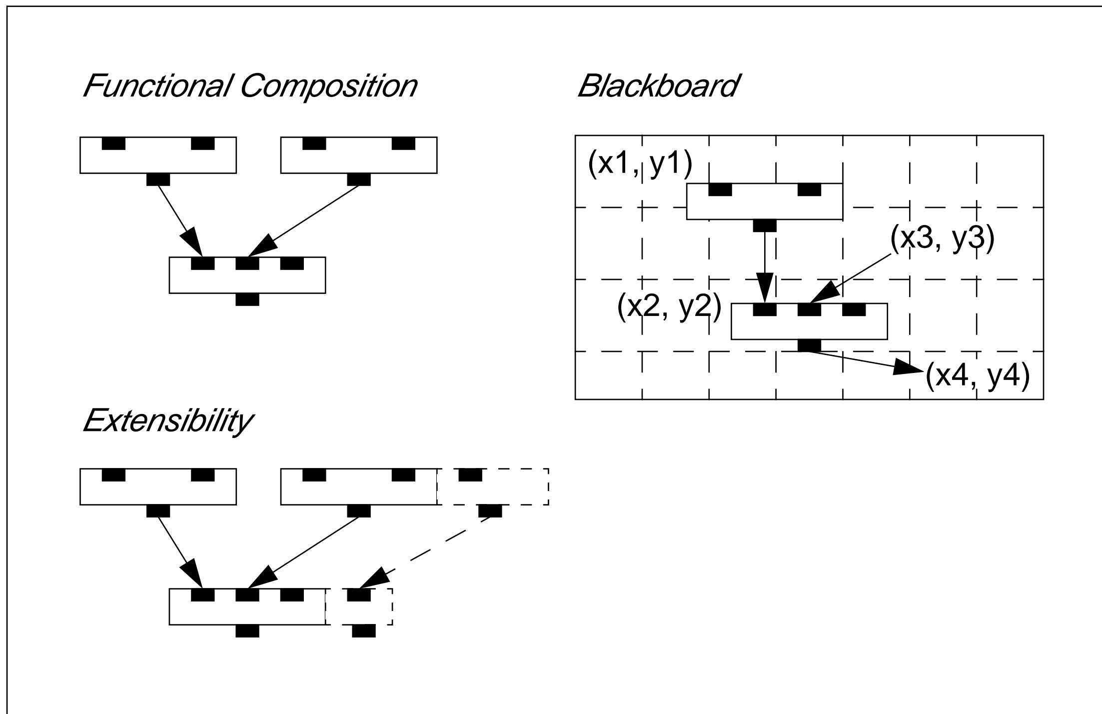

Figure 1.3 *Composition paradigms.*

careful study. The next subsections discuss the notions of components (the boxes), mechanisms (the arrows), and software configurations (the assemblies).

1.3.2 Components as Static Abstractions

In the introduction, we described components in terms of their usage: a software fragment is a component if it is designed for reuse and is part of a framework. This does not tell much about the structural aspects of a component. Some global invariants seem to be valid within any composition paradigm: components typically are *static entities*; moreover, they always consist of some kind of *abstraction.* Both notions, however, deserve more careful examination.

There are many different kinds of static software entities: procedures, functions, modules, classes and so on. In each case, they have a persistent existence independent of their surrounding context, allowing them to be manipulated and stored individually. Once assembled into a program, these static entities control the creation and evolution of dynamic entities, which in current languages are usually *not* components (procedure activations, objects, dynamic data structures). Several examples can be found, however, of dynamic entities that could be interesting as reusable software fragments, but cannot directly participate in a composition because of limitations of the software environment. For example, in most object-oriented languages the classes are static, but the objects (instances) are not.

In such languages various strategies are typically used by programmers to have objects as composable entities, such as defining a class that encapsulates a single object (instance). Another strategy, heavily used in the NeXTStep environment [39], is to define complex archiving procedures so that groups of objects can be stored into files (so-called "nib" files); the corresponding files can then be composed and the resulting configuration used to recreate at run-time the collection of objects defined in the individual groups. In cases like this, where the structure of the objects composing a user interface is known statically and does not evolve at run-time, the ability to directly store objects would be much more convenient than writing programs or description files that will dynamically recreate a configuration of objects.

Another limitation to composition occurs in exactly the reverse situation: saying that components are static entities does not mean that they should be always assembled statically. Open systems have an increasing need to dynamically manipulate and exchange components, and dynamically link them with a running application. Recent languages for distributed agents such as Telescript [56] or *Obliq* [5] are good examples of this new direction. Dynamic assembly means that software can be configured at the latest stage, according to user's needs, or that several running applications can dynamically collaborate to exchange information.

The notion of a component is also closely related to that of an *abstraction*, a self-contained entity, with some kind of boundary around it, which can later be composed with other entities. A procedure is an abstraction for a sequence of instructions; a class is an abstraction for a collection of objects; a module is a set of named abstractions. The fact that abstractions have boundaries is crucial for software composition, since it provides a means for structuring software, controlling interaction between components, and verifying proper assembly. Unfortunately, most software environments impose some restrictions on the use of abstractions: boundaries cannot be drawn arbitrarily, according to user's needs, but must follow specific patterns. For example, in most object-oriented systems, boundaries cannot cross inheritance paths, i.e. a class cannot be defined without explicitly referencing its superclass. Only CLOS [27] supports a notion of inheritance through *mixins* in which the superclass need not be known and can be bound later. Full flexibility for drawing abstraction boundaries requires all software components to be treated as *first-class values* that can be passed as parameters to other components. As discussed above, the languages that are most advanced in that direction are functional languages, where "everything is a function," and functions are data. Since functional abstraction is the only abstraction mechanism, programmers have great flexibility in choosing which aspects to fix in a function definition and which aspects to leave open in parameters.

Besides treating components as values, another property of abstractions that has a great impact on compositionality is *scalability*, namely the possibility to use the same abstraction and composition mechanisms at every level of a configuration. Again this is obviously the case with functions, where an assembly of functions is a function again. The advantage is the economy of concepts, and the fact that there is no limit on the granularity of components. Through their inheritance interface, classes can be seen as scalable, since the incre-

mental modifications of a subclass, together with the parent class, form a class again. By contrast, modules are usually not scalable: an assembly of modules is not a module itself. An environment without scalability imposes a fixed granularity of composition (modules can only be assembled into programs), and therefore restrict reusability of components. Furthermore, the absence of scalability often creates problems for formal studies of programming and composition environments, because formal theories are most successful when they can rely on a small set of universal operators. A striking example can be observed in the area of concurrency, where theoreticians typically use process calculi with scalability (a pool of agents or processes is itself a process), while most practical implementations involving concurrency clearly distinguish between a process and a system of processes.

1.3.3 The Composition Process

In traditional environments for software development the various phases for building an application are well-defined and distinct: first one has to write a collection of modules, possibly with some interdependencies, and with some dependencies to predefined modules stored in libraries; then one has to *compile* the modules, in order to generate machine code and, in strongly typed systems, to check type correctness of the modules; finally, one has to *link* the various pieces of machine code together, using a global name space to resolve all cross-references. This, of course, is the schema for compiled languages, but it accounts for the great majority of development environments in current use. Therefore, in such systems, the granularity of components seen by programmers is basically the same as the granularity of units manipulated by the development environment.

In order to get more flexible composition environments, this well-established scheme of program development has to be reviewed. There are several reasons why a component-oriented lifecycle is needed, and there are several tendencies in modern languages that demonstrate the possibility of improving the traditional three-phase assembly of software.

We discussed above the necessity for open systems to be able to dynamically link new agents into a running system. This implies that the information that is normally discarded at link-time, namely the association between global names and memory addresses, needs to be kept both in the running system and in the agent that will be added to it. In other words, even a complete system can no longer considered to be totally closed: names may be locally resolved, but they still need to be considered as potential free variables that can be linked later to a dynamic entity.

In some object-oriented systems, this is true to a further degree: not only the linkage information, but also a major part of compile-time information is required at run-time — this is necessary to implement features such as delegation or even reflection. Early advocates of object-oriented programming were often arguing in favour of the high level of flexibility offered by fully dynamic object-oriented systems, even if they admitted that such choices have a cost in terms of resources: dynamicity typically consumes more memory and more computing power than statically optimized code. Later, some thought they

had found the adequate compromise with C++: use objects and classes, but compile away a maximum of information, only keeping what is strictly necessary (namely tables for dynamic binding of virtual functions); this is one of the main reasons why the C++ community grew so rapidly. Indeed, C++ has been and is very successful for a large number of applications, but one could say that the original target of proponents of object-oriented programming has shifted: C++ is being used as a replacement for C, for applications in which interaction with operating system, efficient use of resources, tractability for large-scale projects are essential. We are slowly rediscovering, however, that if flexibility, openness, fast prototyping are really important issues, then the choice of C++ is no longer justified. In the recent years, demand for qualified Smalltalk programmers has been steadily increasing, and large-scale high-level platforms for application development like *OpenStep* [40] are being based on Objective-C instead of C++; both languages differ from C++ in that they maintain full information about objects, classes and methods in the run-time environment. So the market is progressively acknowledging that efficiency is not necessarily the most important feature in any case, and that it also has its cost in terms of lack of openness and flexibility.

We are not saying that the future of software components is necessarily in fully interpreted languages, but that flexible open systems need to deal with components in many possible forms, ranging from source code to machine code through several intermediate representations, partially compiled and optimized. Some modern languages in various areas already demonstrate this tendency, and show that much progress has been done for such implementation strategies. For example, both the scripting language Perl [52] and the functional language CAML-Light [30] are compiled into an intermediate form that is then interpreted; actually, interpreted Perl programs are sometimes faster than equivalent compiled programs written in C, and the implementation of the CAML-Light interpreter is faster than compiled versions of the original CAML language! Another example is the Self language [51], which provides a very high level of run-time flexibility, and yet has efficient implementations based on the principle of *compile-by-need*: the run-time system includes a Self compiler, and methods are compiled whenever needed. Static compilation of a method in an object-oriented system is sometimes complicated, because one has to make assumptions about the context in which it will be called (taking inheritance into account); if, instead, the method is compiled at run-time, then more information is known about the context (i.e. which actual object the method belongs to), which allows for a more efficient compilation of the method. In other words, the time lost to compile the method at run-time may be quickly recovered through subsequent calls to the same method.

Ideally, the responsibility of switching between high-level, human-readable representations of components and low-level, optimized internal representations should be left to the composition environment. In practice, however, programmers still often need to guide these choices. This means that the granularity of components manipulated by the system is visible to programmers. In itself, this is not necessarily a disadvantage, but the problem is that this granularity is often identified with the granularity of logical components of a software system. In other words, programmers are forced to think in terms of “compila-

tion units," instead of thinking in terms of "modules." Leroy [29] explained very clearly the distinction:

> *Modularization* is the process of decomposing a program in[to] small units (*modules*) that can be understood in isolation by the programmers, and making the relations between those units explicit to the programmers. *Separate compilation* is the process of decomposing a program in[to] small units *(compilation units)* that can be type-checked and compiled separately by the compiler, and making the relations between these units explicit to the compiler and linker.

Identifying the two concepts is very common, and yet is limiting, as Leroy points out in the context of the SML language [37]. Modules — i.e. logical units of a program — may be structurally much more complex than compilation units, especially if, as discussed above, one wants to be able to treat them as first-class values and to perform higher-order module combinations, either statically or even dynamically. In this respect, SML has probably the most sophisticated module system for an existing programming language, yet it does not support separate compilation. Several researchers are currently working on removing this limitation [29][16].

1.3.4 Verification of Composition

Whenever components are assembled to perform a common task, there is always an implicit contract between them about the terms of the collaboration. In order to be able to verify the correctness of a configuration, the contracts need to be made explicit and to be compared for eventual discrepancies. This issue can be addressed by a type system. However, conventional type systems cannot capture in general all the aspects of a contract, because of their limited expressiveness. Two approaches can be taken for dealing with this problem. One approach, taken by Meyer in the Eiffel language [33], is to enrich the interfaces of components with additional constraints expressing the expectations and promises of each partner in the contract. Part of the constraints are checked by the type system, and part of them are verified at run-time, each time that an actual collaboration (control passing) between two components takes place. The other approach is to improve the expressiveness of type systems. Much research has been done in this direction, especially in the area of functional programming languages. Polymorphic type inference in languages such as ML or Haskell [21] actually provides a level of security that is much higher than in a traditional language like Pascal, without putting any additional burden on the programmer. However, as soon as one leaves the functional model, such results are no longer applicable: in systems with blackboard composition (imperative programming languages, concurrent systems) one cannot infer much type information. As far as object systems are concerned, this is still an open question, examined in detail in a survey by Fisher and Mitchell [11]. The addition of subtyping makes both type inference and type checking considerably harder, so despite important progress made over the recent years, no object-oriented language with an ML-like type system has yet been developed.

To capture the recursive semantics of objects at a type level, most researchers use explicitly typed systems with either recursive types or existential quantification; such solutions have improved the state of the art for object typing, but are not likely to be applied soon in real languages, since the complexity of the resulting type expressions would probably appal most programmers not familiar with type theory. Therefore we believe that practicability of object typing will be achieved through type inference rather than through explicit typing; preliminary results in that direction are discussed in [18]. The difficult point, however, is to be able to infer types that are both "minimal" in the sense of subtyping, and "principal" in the sense of Curry type schemes (a type scheme is principal for a term if and only if it can generate all other types of that term by substitution of type variables). To our knowledge, this is still an open problem; but some recent results on principal types for objects are collected in [15].

Coming back to the problem of explicit contracts between components, we should mention another family of solutions that puts the contract, not inside components, but outside. For interlanguage composition, this is even the only possibility, since it would be quite difficult to compare contracts specified in different languages and models. An example of a contract being outside of the components is a database schema that specifies the conditions under which a common database may be accessed, and which must be respected by every program doing transactions on the database.While providing a glue between heterogeneous components, this kind of solution has the disadvantage of being quite rigid: the terms of the contract are specified from the beginning and can hardly be changed later; moreover, this approach cannot support scalability, since components are clearly distinct from configurations of multiple components. Contracts outside of components are also found in *module interconnection languages*, whose job is precisely to perform composition of software components. The amount of information handled in such languages varies from one system to the other; Goguen, for example, advocates an algebraic approach to capture semantic information about the components [13]. It should be noted, however, that module interconnection languages seem to have lost part of their importance in the literature in favour of more homogeneous approaches in which the distinction between components and component assemblies is less strict. Object-oriented approaches fall into that category, as do functional approaches to an even greater degree.

Type systems and algebraic specifications aim at verifying correctness in a machine-checkable way by statically looking at a software configuration. They belong, therefore, to the world of static semantics. By contrast, a number of techniques have been developed for studying the dynamic behaviour of programs, like denotational, algebraic, operational or axiomatic semantics. Since such techniques deal with dynamic information, and are therefore not decidable in general, they are commonly used for studying programming languages and environments rather than particular software configurations. It is therefore not our purpose here to discuss them in detail. It should be noted, however, that several of the points discussed above for the evolution of component-oriented software development will have some impact on these analysis techniques. For example, most of these semantics are compositional, but they are not modular (for denotational semantics, this is acknowledged by Mosses [38]). In the scenario of iterative compositional development, it should

be possible to progressively refine the semantics of a component according to the available knowledge about its context: we know more about a component inserted into a given configuration than about this component seen in isolation. Instead of the usual distinction between static semantics, dynamic semantics, and what Jones [25] calls "binding time analysis," we should again have a whole range of intermediate steps, corresponding to the various intermediate stages of assembly.

Finally, it should be noted that traditional semantic techniques induce an equivalence relationship over software components — they have been designed to be able to state whether two components are equal or not. In the context of object-oriented programming, this is no longer sufficient, since the idea is to extend components — to produce new components that are not just "equal" to previous ones (plug-compatible), but in some sense are "better" (extended). To deal with this aspect, theoreticians of object-oriented languages have developed the notion of *partial equivalence relationships (PERs)* [4], which equates components not universally, but relative to a given type: for example the records {x=1, y=3}, {x=1, y=4, z=10} are equivalent as type {x:Int}, but not as type {x:Int, y:Int}. An alternative approach is proposed in this book in chapter 6, in which components are this time universally related, but by a *compatibility* partial order instead of an equivalence relationship.

1.3.5 Objects as Processes

Earlier in this chapter we argued that *components* and *concurrency* are both fundamental concepts, and cannot be considered as "add-ons" to programming languages. Furthermore, the semantic issues are sufficiently subtle and complex that it is essential to have a formal object model and a semantic foundation for reasoning about all language features. What, then, should the object model look like, and what would be an appropriate semantic foundation?

Let us consider the features we would need to model in a language that supports component-oriented development:

1. *Active Objects:* objects can be viewed as autonomous agents or processes.
2. *Components:* components are abstractions, possibly higher-order, over the computational space of active objects.
3. *Composition:* generalized composition is supported, not just inheritance.
4. *Types:* both objects and components have typed interfaces, but, since objects are dynamic entities and components are static, the type system must distinguish between them.
5. *Subtypes:* subtyping should be based on a notion of "plug compatibility" that permits both objects and components to be substituted if their clients are satisfied [55].

An object model must therefore cope with both objects and components. Objects encapsulate *services*, and possess *identity*, *state* and *behaviour**. The services are obtained through the behaviour according to some client/server *protocol*. Components, on the other hand, are *abstractions* used to build object systems, e.g., they are functions over the object/process space. Although functions are fundamental, we cannot model objects as functional entities because they are long-lived and concurrent. Since input and output are on-going, and the same input may produce different results at different times, objects are essentially non-functional. Ideally, an *object calculus* [41] would merge the operational features of a process calculus with the compositional features of the λ calculus.

Interestingly, recent progress in the study of process calculi addresses many aspects of the semantics of concurrent object-oriented systems. The original work by Milner on a Calculus of Communicating Systems (CCS) [34] resulted in a highly expressive process calculus that nevertheless could not be used to model "mobile processes" that can exchange the names of their communication ports in messages. This, of course, is essential to model objects. Work by Engberg and Nielsen [10] borrowed and adapted concepts from the λ-calculus to deal with this, and Milner [36] refined and simplified their results to produce the π-calculus, a true "calculus for mobile processes." In the meantime, Thomsen [48] developed the first "Calculus for Higher-Order Communicating Systems" (CHOCS) which essentially added term-passing to CCS. From an object systems point of view, this should allow one to model objects and components as values at run-time. Milner extended the π-calculus to a polyadic form [35], which allows one to express communication of complex messages, and he introduced a simple type system for the calculus. Following on work by Milner, Sangiorgi [46] developed a higher-order process calculus (HOπ), whose semantics can be faithfully preserved by a mapping to the unadorned π-calculus, and Hennessy [17] has developed a denotational model of higher-order process calculi. Honda [20] has also developed the ν-calculus, a process calculus based on asynchronous communication, whose semantics is obtained by a *reduction* of the features of the π-calculus. Going in the opposite direction, Dezani *et al.* [9] have investigated synchronous parallelism and asynchronous non-determinism in the classical λ-calculus. In the object-oriented community, there have been several other attempts to develop object calculi that take their initial inspiration from either process calculi or the λ-calculus, or both [8] [20] [41].

We propose that a formal model of objects and components based on recent developments in process calculi and λ-calculi should form a good basis not only for understanding and explaining abstraction and composition in a component-oriented software development method, but can actually serve as an abstract machine for developing a new generation of component-oriented languages [43] [44], much in the same way that the λ-calculus has served as a semantic foundation for modern functional programming languages.

---

* The distinction between "state" and "behaviour" is admittedly artificial, but is useful for conceptual reasons, since state is thought of as hidden and behaviour as visible. In fact, the notions are dual, and one can consider the "state" of an object to be its "current behaviour."

1.3.6 Summary of Research Topics

In this section we have listed some very ambitious wishes for the future of component-oriented development environments, but we have also shown that several directions already present in modern programming languages can give us some confidence about fulfilment of that program. To summarize, here are the points that we consider as most important research issues:

- Merge current notions of abstraction in process calculi, functional languages and object-oriented languages into a single notion of *component*, which should be a firstclass, storable entity equipped with the notions of parameterization (leaving some aspects of the component "open") and instantiation (ability to generate a "copy" of the component in a given run-time context), and furthermore should support scalability (possibility to encapsulate a partial configuration of components as a new component).
- Develop software manipulation tools that are able to deal with partial configurations and support an iterative assembly process, by using various levels of intermediate representations of components. Current tasks of type checking, compilation to machine code and linkage will be replaced by incremental change of intermediate representation.
- Find expressive, yet decidable type inference/partial evaluation systems, that will be able to statically decide about the correctness of a partial configuration, in a way that is transparent to (or requires minimal typing information from) programmers.

It can be seen that these research directions require a tight integration between current research being done both at a theoretical level (semantics and types of programming languages) and at a practical level (implementations, compiler/interpreter design).

1.4 Component Engineering

Once we have a language and environment that permits us to develop software component frameworks, there remains the question how these components should be developed, maintained and applied. With traditional software development, applications are in principle designed to meet very specific requirements. Component frameworks, on the other hand, must be designed to meet many different sets of requirements, and should even be built to anticipate unknown requirements.

Consider the following scenario* [42] for application development: an application developer has access to a *software information system (SIS)* that contains not only descriptions of available component frameworks, but domain knowledge concerning various application domains, descriptions of requirements models, generic designs, and guidelines for mapping requirements specifications in the problem space to designs and imple-

* This scenario was elaborated as part of the ITHACA project (described briefly in the preface).

mentations in the solution space (see chapter 7 for a description of a such a system). A software information system is closer in spirit to an expert system than to a repository; in fact, the principle of a SIS is that it should encode and present the knowledge acquired by a domain expert.

To use the SIS, the application developer first enters into a dialogue to identify the relevant application domain. The information pertaining to this domain can be referred to as a *Generic Application Frame (GAF)*. The GAF determines the context for application development. The next step in the dialogue is to specify the requirements. Since the GAF includes domain knowledge and requirements models, the requirements specification is largely performed according to existing patterns. The specific requirements will then lead the SIS to suggest, according to stored guidelines, generic designs and component frameworks that can be used to build the application. The guidelines may also suggest how components should be instantiated or specialized to meet specific requirements. (Chapter 10 contains a brief description of RECAST, an interactive tool for requirements collection and specification, based on this scenario.)

The process of completing requirements specifications, making design decisions and refining and composing components results in a new information structure that we will call a *Specific Application Frame (SAF)*. The SAF consists not only of the completed application, but all the information that was generated along the way. When application requirements evolve, the SIS is again used, but in this case the dialogue results in pre-vious decisions being reconsidered and a new SAF being built from the old.

This scenario is very appealing, but suggests more questions than it answers. How is domain knowledge to be captured and represented in the SIS? How are generic designs and component frameworks developed and described? How are guidelines determined and encoded? Who is responsible for maintaining the SIS and its contents, and how are the contents evaluated and maintained? Is the scenario even realistic? How much will the SIS need to be supported by human experts? We believe it is, because successful generic applications and component frameworks do exist, but nobody knows how far this scenario can be pushed to work well in practice. Will it only work for very restricted and well-understood application domains, or is it also valid for more complex and evolving domains?

This suggests that the role of *component engineering* is fundamentally different from the more traditional role of *application development*. Although the same person may in some cases play both roles, it is important to separate them in order to keep the different sets of requirements distinct. In particular, the clients for each are very different. The clients of an application are (ultimately) the end-users, whereas the clients of a component framework are the application developers.

Why is it necessary to elevate component engineering to a distinguished activity? Should it not be possible to find reusable components by scavenging existing object-oriented applications? A plausible scenario might have application developers use traditional methods to arrive at an object-oriented design, and then search for reusable objects that would at least partially meet the specifications. The “found” objects would then be tailored to fit the task at hand.

The problem with this scenario is that you do not get something for nothing. Software components are only reusable if they have been *designed* for reuse. A repository of software objects from previous applications is like a "software junkyard" that, more likely than not, will not contain just what you are looking for. The cost of searching for and finding something that approximately meets one's needs, and the additional cost of adapting it to fit may exceed the cost of developing it from scratch. Worse, the tailored components are not maintainable, since such an approach will encourage a proliferation of hacked-up, incompatible versions of somewhat similar components, none of which is ultimately reusable. *Systematic* rather than accidental software reuse requires an investment in component framework development and in software information management [53].

1.4.1 Benefits and Risks

A component that has been designed for reuse always forms part of a framework of components that are intended to be used together, much in the way that modular furniture is made of components that can be combined in many ways to suit different needs. Clearly the development of a component framework represents an investment that must be evaluated against the expected return. The benefits can be measured in two ways: a component framework should make it easier (i) to fill (at least partially) the needs of many different applications, and (ii) to adapt a given application to changing needs. (These are also the main selling points of modular furniture.) If either or both of these requirements are present to a sufficient degree, it may be worthwhile developing a component framework, or investing in the use and possible adaptation of an existing framework.

In fact, one can easily argue that component frameworks should *always* be used: long-lived applications necessarily undergo changes in requirements with time that can be more easily met with the use of a framework, and short-lived applications must typically be developed under tight time constraints, which can also be facilitated by the use of an existing framework. The risks, however, must also be considered:

1. A steep learning curve can be associated with the use of a framework. Developers must be willing to invest time and effort into learning a framework before the benefits can be realized. The *not invented here* syndrome can be difficult to overcome.
2. Development of new frameworks is a costly and long-term activity. The long-term benefits must be justified in terms of the opportunities for recovering the investment.
3. Individual projects have short-term goals and deadlines that conflict with the long-term goals of component-engineering. Management must commit to developing a service-oriented infrastructure to support the provision of frameworks to projects [14]. If the use of frameworks introduces too much overhead, projects will not adopt them.

4. New frameworks evolve rapidly in the beginning, and may undergo several complete redesigns before they stabilize. The costs of re-engineering client applications of a redesigned framework may be quite high, though the long-term benefits of re-engineering can be significant. In principle one should not use unstable frameworks for a large base of client applications, but on the other hand, a framework will not evolve to the point that it stabilizes unless it is applied to many different kinds of applications.

The reason that each of these points can be considered a risk is that present software engineering practice actually *discourages* component-oriented development by focusing on the individual application rather than viewing it as part of a much broader software process. To address these points we need to rethink the way software is developed and introduce new activities into the software lifecycle.

If we reject the "software junkyard" model of software reuse, we can still consider it as a starting point for component engineering. A component engineer processes and digests the results of previous development efforts to synthesize (i) domain knowledge and requirements models [2], (ii) design patterns [12] and generic architectures, (iii) frameworks [24] and component libraries, (iv) guidelines to map from problem to solution domains (i.e. from requirements to designs and implementations). The result of component engineering, therefore, resembles a well-designed cookbook — it is not just a collection of prepackaged recipes, but it contains a lot of background information, generic recipes, suggestions on how to combine and tailor recipes, and advice on how to meet specific needs. The "cookbook" is intended to compensate for the fact that not everyone can afford the time and expense required to become an expert, and so the acquired expertise is reduced to a standard set of guidelines and rules. Naturally one cannot hope to answer all possible needs with such an approach, but a large class of relatively mundane problems can be addressed.

Note that component engineering is not concerned only with developing software components, but touches all aspects of software development from requirements collection and specification, through to design and implementation. The point is that the most beneficial artefacts to reuse are often not software components themselves but domain knowledge and generic designs. Software reuse is most successful if one *plans* for it in advance. By waiting until after requirements are specified and the systems are designed, many opportunities for reuse may have been wasted, and one may not even be able to find suitable components to reuse.

Component engineering can only be considered successful if the results are used to build more flexible applications. Ideally, these results actually *drive* the application development process: an application developer should be quickly positioned in the software information space to some GAF, and the activities of requirements collection and specification, application design, component selection and refinement should follow from a flexible dialog between the developer and a software information system on the basis of the contents of the GAF.

1.4.2 How to Get There from Here

However attractive such a software information system might be, little is known about how one should build one that would be successful in practice. (See chapter 7 for a discussion of some of the issues.) Good results have been achieved by introducing a so-called "Expert Services Team" of individuals who are responsible for introducing reusable assets into projects [14]. In this way, some of the domain expertise is formalized in terms of reusable assets, but the knowledge of how to apply them to particular situations remains a responsibility of this team. The hard parts remain: (i) how to identify the reusable assets applicable to a given situation (identifying the GAF), (ii) mapping the results of analysis to available architectures and designs, (iii) elaborating missing subsystems and components, (iv) adapting frameworks to unforeseen requirements.

More generally, there are various aspects of component-oriented development that can only be considered open research problems. Some of the more significant problems are:

1. *Domain knowledge engineering:* how should domain knowledge be captured and formalized to support component-oriented development?
2. *Synergy between analysis and design:* traditional software engineering wisdom would keep design issues separate from analysis, but opportunities for reuse can be missed unless one plans for it. How can analysis benefit from the knowledge that frameworks will be used in system design?
3. *Framework design:* what methods apply to framework design? Object-oriented analysis and design methods do not address the development of frameworks. Guidelines exist, but no methods [23].
4. *Framework evolution:* frameworks evolve as they stabilize. What principles should be applied to their evolution? How do we resolve the technical difficulties of maintaining applications based on evolving frameworks? [6]
5. *Reuse metrics:* traditional software metrics are of limited use in the development of object-oriented software. Less is known about measuring the cost of developing component-oriented software. How does one measure potential for reuse? The size and cost of framework-based applications? The cost of developing and maintaining reusable assets? [14]
6. *Tools and environments:* what software tools would facilitate component-oriented development? How can the software information space be managed in such a way as to provide the best possible support both for application developers and component engineers?

1.5 Conclusions

Component-oriented software development builds upon object-oriented programming techniques and methods by exploiting and generalizing object-oriented encapsulation and

extensibility, and by shifting emphasis from programming towards *composition*. Present object-oriented technology is limited in its support for component-oriented development in several ways. First and foremost, the notion of a *software component* is not explicitly and generally supported by object-oriented languages. A component, as opposed to an object, is a static software abstraction that can be composed with other components to make an application. Various kinds of components can be defined with object-oriented languages, but their granularity is typically too closely linked with that of objects — in addition to classes, both more finely and coarsely grained abstractions are useful as components.

Supporting both components, as software abstractions, and objects, as run-time entities, within a common framework requires some care in integrating corresponding language features within a common framework. In particular, it is not so easy to devise a satisfactory type system that captures "plug compatibility" in all its useful forms and guises. Concurrency and evolving object behaviour pose particular difficulties, as is seen in chapters 2, 4 and 5. For these reasons, we argue, it is necessary to establish a suitable semantic foundation of objects, functions and agents that can be used to reason about software composition at all levels.

Foundational issues, though important, address only a small part of the difficulties in making component-oriented development practical. Even if we manage to produce computer languages that are better suited to expressing frameworks of plug-compatible software components, there is a vast range of technological and methodological issues to be resolved before we can expect that component-oriented development will become widespread. The most fundamental question — where do the components come from? — is the hardest to answer. In a traditional software lifecycle, application "components" are tailor-made to specific requirements. In a component-oriented approach, the activity of *component engineering* must be explicitly incorporated into the lifecycle, and supported by the software process, the methods and the tools. "Software reuse" is not something that can be achieved cheaply by arbitrarily introducing libraries or "repositories" into an existing method. In fact, rather than focusing on software reuse, we must concentrate on reuse of design, of architecture and of expertise. Component engineering is the activity of distilling and packaging domain expertise in such a way as to make component-oriented application development possible.

References

[1] Pierre America, "A Parallel Object-Oriented Language with Inheritance and Subtyping", *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, vol. 25, no. 10, Oct. 1990, pp. 161–168.

[2] Roberto Bellinzona, Mariagrazia Fugini, Vicki de Mey, "Reuse of Specifications and Designs in a Development Information System", *Proceedings IFIP WG 8.1 Working Conference on Information System Development Process,* ed. N. Prakash, C. Rolland, B. Pernici, Como, Italy, Sept. 1–3 1993, pp. 79–96.

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

[4] Kim B. Bruce and Giuseppe Longo, "A Modest Model of Records, Inheritance, and Bounded Quantification," in *Information and Computation*, vol. 87, 196–240, 1990.

[5] Luca Cardelli, "Obliq: A Language with Distributed Scope," preliminary draft, March 1994.

[6] Eduardo Casais, "An Incremental Class Reorganization Approach," *Proceedings ECOOP '92*, ed. O. Lehrmann Madsen, *Lecture Notes in Computer Science*, vol. 615, Springer-Verlag, Utrecht, June/July 1992, pp. 114–132.

[7] William Cook, Walter Hill and Peter Canning, "Inheritance is not Subtyping," *Proceedings POPL '90*, San Francisco, Jan. 17–19, 1990, pp. 125–135.

[8] Laurent Dami, "Software Composition: Towards an Integration of Functional and Object-Oriented Approaches," Ph.D. thesis no. 396, University of Geneva, 1994.

[9] Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, Adolfo Piperno, "Fully Abstract Semantics for Concurrent Lambda-calculus," in *Proceedings TACS '94*, *Lecture Notes in Computer Science*, vol. 789, Springer-Verlag, 1994, pp. 16–35.

[10] Uffe Engberg and M. Nielsen, "A Calculus of Communicating Systems with Label Passing," DAIMI PB-208, University of Aarhus, 1986.

[11] Kathleen Fisher and John C. Mitchell, "Notes on Typed Object-Oriented Programming," in *Proceedings TACS 94*, *Lecture Notes in Computer Science*, vol. 789, Springer-Verlag, Utrecht, 1994, pp. 844–885.

[12] Erich Gamma, Richard Helm, Ralph E. Johnson and John Vlissides, "Design Patterns: Abstraction and Reuse of Object-Oriented Design", *Proceedings ECOOP '93*, *Lecture Notes in Computer Science*, Springer-Verlag, vol. 707, 1993, pp. 406–431.

[13] Joseph A. Goguen, "Reusing and Interconnecting Software Components," in *IEEE Computer*, Feb. 1986, pp. 16–27.

[14] Adele Goldberg and Kenneth S. Rubin, *Succeeding with Objects: Decision Frameworks for Project Management*, Addison-Wesley, 1995, forthcoming.

[15] Carl A. Gunter and John C. Mitchell, *Theoretical Aspects of Object-Oriented Programming*, MIT Press, Cambridge, Mass. , 1994.

[16] Robert Harper and Mark Lillibridge, "A Type-Theoretic Approach to Higher-Order Modules with Sharing," in *Proceedings POPL '95*, ACM Press, 1995, pp. 123–137.

[17] Matthew Hennessy, "A Fully Abstract Denotational Model for Higher-Order Processes," in *Information and Computation*, vol. 112(1), pp. 55–95, 1994.

[18] Andreas Hense, "Polymorphic Type Inference for Object-Oriented Programming Languages," Dissertation, Saarbrücken, Pirrot, 1994.

[19] John Hogg, "Islands: Aliasing Protection in Object-Oriented Languages," in *Proceedings OOPSLA 91*, *ACM SIGPLAN Notices*, vol. 26, no. 11, pp. 271–285, Nov. 1991.

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

[21] Paul Hudak, Simon Peyton Jones and Philip Wadler (eds), "Report on the Programming Language Haskell — A Non-Strict, Purely Functional Language (Version 1.2)," *ACM SIGPLAN Notices*, vol. 27, no. 5, May 1992.

[22] IEEE Software, "*Software Reuse*", vol. 11, no. 5, Sept. 1994.

[23] Ralph E. Johnson and Brian Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, vol. 1, no. 2, 1988, pp. 22–35.

[24] Ralph E. Johnson, "Documenting Frameworks using Patterns", *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, vol. 27, no. 10, Oct. 1992, pp. 63–76.

[25] Neil D. Jones, "Static Semantics, Types, and Binding Time Analysis," *Theoretical Computer Science,* vol. 90, 1991, pp. 95–118.

[26] 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.

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

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

[29] Xavier Leroy, "Manifest Types, Modules, and Separate Compilation," in *Proceedings POPL'94*, ACM Press, 1994, pp. 109–122.

[30] Xavier Leroy and Pierre Weiss, *Manuel de Référence du Langage CAML*, InterEditions, 1994. Also available by WWW at http://pauillac.inria.fr/doc-caml-light/refman.html.

[31] 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.

[32] M.D. McIlroy, "Mass Produced Software Components," *Software Engineering*, ed. P. Naur and B. Randell, NATO Science Committee, Jan. 1969, pp. 138–150.

[33] Bertrand Meyer, *Object-Oriented Software Construction,* Prentice Hall, Englewood Cliffs, NJ, 1988.

[34] Robin Milner, *Communication and Concurrency*, Prentice Hall, Englewood Cliffs, NJ, 1989.

[35] Robin Milner, "The Polyadic pi Calculus: a tutorial," ECS-LFCS-91-180, Computer Science Dept., University of Edinburgh, Oct. 1991.

[36] Robin Milner, Joachim Parrow and David Walker, "A Calculus of Mobile Processes, Part I/II," *Information and Computation*, vol. 100, 1992, pp. 1–77.

[37] Robin Milner, Mads Tofte and Robert Harper, *The Definition of Standard ML*, MIT Press, Cambridge, Mass., 1990.

[38] Peter D. Mosses, "Denotational Semantics," in ed. J. van Leuwen, *Handbook of Theoretical Computer Science,* vol. B, Elsevier, Amsterdam, 1990, pp. 575–631.

[39] *NeXTstep Reference Manual*, NeXT Computer, Inc., 1990.

[40] Next Computer, Inc. and SunSoft, Inc., *OpenStep Specification*, 1994. Available at ftp address ftp://ftp.next.com/pub/OpenStepSpec/.

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

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

[43] 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.

[44] Oscar Nierstrasz and Theo Dirk Meijler, "Requirements for a Composition Language," *Proceedings of the ECOOP '94 Workshop on Coordination Languages*, ed. P. Ciancarini, O. Nierstrasz, A. Yonezawa, *Lecture Notes in Computer Science*, Springer-Verlag, 1995, to appear.

[45] Michael Papathomas and Oscar Nierstrasz, "Supporting Software Reuse in Concurrent Object-Oriented Languages: Exploring the Language Design Space," *Object Composition*, ed. D. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, June 1991, pp. 189–204.

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

[47] 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.

[48] Bent Thomsen, ‘‘Calculi for Higher Order Communicating Systems,’’ Ph.D. thesis, Imperial College, London, 1990.

[49] 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.

[50] Jon Udell, “Componentware,” in *Byte*, vol. 19, no. 5, May 1994, pp. 46–56.

[51] 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.

[52] Larry Wall and Randal L. Schwartz, *Programming Perl*, O’Reilly & Associates, Inc., 1990.

[53] Peter Wegner, ‘‘Capital-Intensive Software Technology,’’ *IEEE Software*, vol. 1, no. 3, July 1984.

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

[55] Peter Wegner and Stanley B. Zdonik, ‘‘Inheritance as an Incremental Modification Mechanism or What Like Is and Isn’t Like,’’ *Proceedings ECOOP ’88*, ed. S. Gjessing and K. Nygaard, *Lecture Notes in Computer Science*, vol. 322, Springer-Verlag, Oslo, Aug. 15–17, 1988, pp. 55–77.

[56] J.E. White, *Telescript Technology: The Foundation for the Electronic Marketplace,* White Paper, General Magic, Inc.