16

On the architecture of software component systems

M. Franz
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA
E-mail: franz@uci.edu

Abstract

Current object-oriented development practice is centered around application frameworks. We argue that this approach is misleading, as it detracts from the ultimate goal of composing software out of "software components" originating from different sources. In particular, we suggest a model of software composition that is based on passing of "first-class messages" rather than on inheritance.

In most object-oriented programming languages, messages and the methods that get executed in response to receiving them are only "second class citizens". In these languages, one can send a message to an object, but one cannot further manipulate the message itself as a data object. As a consequence, many of the operations that a naive observer might expect to be available are in fact not usually offered. Examples of such missing operations are the ability to store arriving messages in a data structure and execute them asynchronously later, perhaps in a different order, or the capability of forwarding a received message to another object without first having to decode it.

We are working on a system based on an experimental language that supports "first-class messages" efficiently. We argue that this additional language capability by itself suffices to simplify the design of extensible, component-oriented systems, and that it leads to a more uniform overall system architecture.

1   INTRODUCTION

The widespread deployment of object technology has brought about sweeping changes to many areas of software construction. For a variety of applications, the approach of taking an object-oriented application framework and customizing it for a task at hand is a fast and cost-effective strategy that yields good results. Similarly, reusing object-oriented components within an organization can be highly beneficial. Unfortunately, however, current object-oriented technology is less well suited for the emerging component-software paradigm, in which many independently developed and highly specialized software parts cooperate in such a manner that they appear to the end-user as one homogeneous application program.

The major hurdle to applying object-oriented technology in component-structured systems is of architectural nature. Current object-oriented tools are almost ideally suited for extending a comparatively large application framework with a comparatively small amount of user-level functionality (Figure 1). This is usually achieved by employing the *inheritance* mechanism offered in object-oriented languages: the user-level functionality is implemented by deriving specialized descendants of framework classes in which the default behavior of the framework is being overridden by the desired behavior of the application.
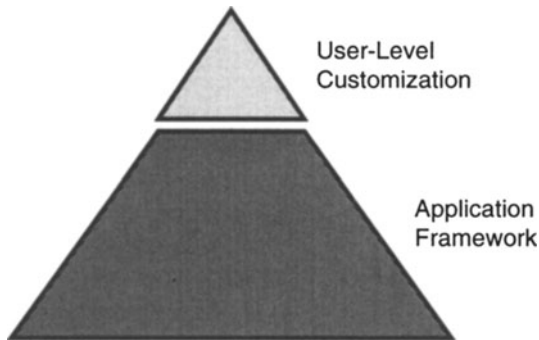
Figure 1: Framework-based application architecture

This approach is particularly beneficial in the user-interface part of highly interactive applications. Consequently, it is a common implementation strategy to use an application framework specifically for realizing user-interfaces, while the domain-specific remainders of the same applications might be implemented from the ground up (and not necessarily using object-oriented technology). In this case, Figure 1 would apply only to the user-interface part of the application.

A similar approach (Figure 2) is currently being employed in the development of Java "applet" extensions for the most popular World Wide Web browsers. Applets

are user-level customizations of an application framework called the *Java Class Library* (Chan and Lee, 1997). What is striking about the Java approach in particular is that the Java application framework is orders of magnitude larger than almost all user-level applications that have so far been implemented on top of it. Moreover, the framework is expanding rapidly with each revision.
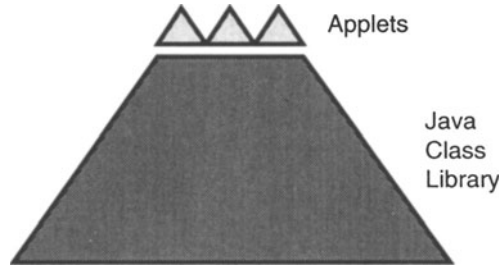

Figure 2: Java applet execution environment

This is partly due to the fact that user-level Java applets are frequently executed by a slow virtual machine interpreting an intermediate byte-code representation (Lindholm et al., 1996), while the framework is always represented as native code and hence can provide a guaranteed performance. We suspect, however, that a further contributing factor for the trend of incorporating every conceivable functionality into the framework itself is that it is just so much easier to reuse code in the application framework than it is to reuse third party code obtained elsewhere.

Why then, is it so much more difficult to reuse code outside of the application framework, and why is MacIlroy's vision of a software component industry (McIlroy, 1968) so persistently elusive? After all, Java supports *dynamic linking* of classes, which should make the composition of software out of pre-fabricated, independently developed building blocks all that much easier.

Our answer to this question is that, besides the well-known organizational obstacles to effective reuse (e.g., Card and Comer, 1994), the object-oriented concept of *inheritance* quickly becomes a burden as the number of sources to inherit from grows. Hence, taking a self-consistent application framework and customizing it in a few isolated places is different from building something new out of a variety of pre-fabricated components.

In the remainder of this paper, we elaborate some more on the difference between customizing an application framework and composing a piece of software from ready-made building blocks. We then argue that the former approach is a dead end in the long term, and that efforts should be undertaken to better support composeability of software from different sources. We suggest that "first-class messages" are a first step in this direction, since they simplify the assembly of a

consistent end-product from a host of independently developed software building blocks.

2   HIERARCHICAL DECOMPOSITION AND BLACK-BOX REUSE

One of the most successful engineering techniques is *hierarchical decomposition*. A problem is successively divided "top-down" into sub-problems, until it is small enough to be solved directly, yielding a self-contained sub-system that will become part of the eventual solution. The individual sub-systems that have been created in this manner are then combined in a "bottom up" fashion. Since the various sub-systems have been designed in isolation to each fulfil a particular task, they can later be put together in combinations that differ from the original constellation, i.e., they can be *reused*.

Hierarchical decomposition has the important structural property that when several smaller parts are combined into a larger part, the larger part "shadows" the smaller ones. One no longer needs to understand the smaller parts in order to use the larger part that has been constructed out of them. Just as importantly, one can even change the interface between the smaller parts without affecting the interface of the large part to the outside world.

Unfortunately, this structural property is lost in most object-oriented systems. Object-oriented design often differs from hierarchical decomposition because inheritance can be used not only to express *specialization*, but also *generalization* (Wegner and Zdonik, 1988; Evered et al., 1997). As a consequence, object-oriented design does not automatically create tree structures. For example, in languages that support multiple inheritance, each of several base classes can be used in the specification of several dependent ones, creating a *web* of dependencies. The *Interface* mechanism present in Java is somewhat more restrictive than general multiple inheritance, and hence lessens this problem somewhat, but still encourages the cross-dependencies that counteract hierarchical structure.

Even without multiple inheritance, object-oriented design destroys *locality*. Objects are backward-compatible with objects of their superclasses, while method overriding has the effect that the meaning of a certain piece of code can be changed in a future extension. As a consequence, programmers need to spend more time analyzing components before they are able to use them. Certification of such components is also more difficult, and even more importantly, mandates that none of the components upon which the certified part depends is changed afterwards. Because of lack of locality, traditional software review techniques are almost impossible to apply.

As a consequence of all this, reusing someone else's classes is much more difficult than reusing a traditional set of sub-systems with static procedural interfaces. One needs to know not only the definition of all reused classes, but also all possible interactions between one's own classes and the reused ones. It is no coincidence that in many object-oriented systems (Goldberg and Robson, 1983;

Goldberg, 1984; Muys-Vasovic, 1989) the source code is considered to be an integral part of the documentation ("white-box reuse").

3   REUSABLE SOFTWARE COMPONENTS

If software engineering were like every other engineering discipline, we would expect to be able to acquire ready-made components and simply "plug them in", i.e., use them without having to study their implementation. For example, if we required an abstract data type "Stack", we could most probably find one in a library somewhere, download it, and reuse it after studying the description of its interface.

In today's application-framework-oriented world, this particular goal of the "software components" idea has more or less been accomplished. Frameworks, such as the Java Class Library, are so encompassing that they contain ready-made solutions for most of the common problems. For example, the Java Class Library provides a class *java.util.Stack* with the required "stack" functionality.

However, a second, equally important aspect of the "software components" concept is *not* fulfilled by the current application framework approach: it is virtually impossible to substitute a built-in class of a framework by an alternate external one that fulfils the same interface. At closer range, we see that the framework itself is one rather large monolithic piece of software, with a multitude of non-obvious inner dependencies. For example, as is elaborated in the next section, the "Stack" class found in the Java Class Library provides much more than just the functionality of a stack. In order to duplicate its functionality externally, one would, for example, need to know also about "Vectors" and "Enumerations".

Hence, rather than encouraging an independent marketplace for freely substitutable software components, current software engineering practice encourages the accumulation of all reusable functionality in an application framework. Since the application framework is really a monolithic system, all potential cross-dependencies are under the control of a single team of architects, and need not be documented externally. Consequently, replacing internal components or policies, unless specifically anticipated in the original design, requires access to source code. We contend that while this approach is practical in today's competitive development climate, it will harm our practice in the long run.

Instead of extensible frameworks, we should strive to craft true software component architectures (Figure 3). The main characteristics of such an architecture are the following:

- a software component system is a result of *hierarchical decomposition*,
- components on the same hierarchical level communicate as *peers*,
- components are *substitutable* with equivalent ones fulfilling the same interface, and
- the *common substrate* that is shared by all components is relatively small.
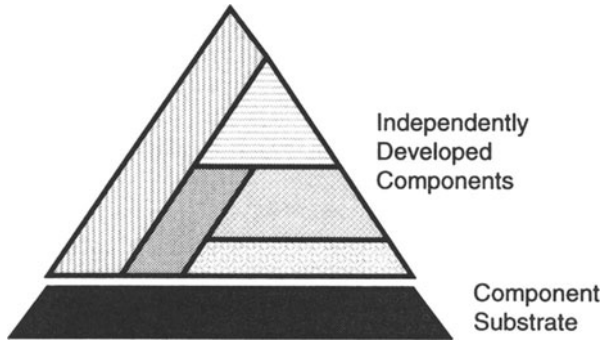
Figure 3: Software component architecture

Interestingly enough, such component architectures do exist on the *macroscopic* level. For the last few years, various standards for component interoperability have been defined, such as *CORBA* (Object Management Group), *COM/OLE* (Microsoft), and *SOM/OpenDoc* (Apple Computer, IBM, Novell), and this "coarse-grained" component structure has been demonstrated to work effectively. Why then, is the same approach not also utilized on a microscopic level, for example, for implementing the stack functionality required by an application? Mainly because it is generally assumed to be too heavy-weight.

Upon closer examination, the above-mentioned interoperability standards differ from the object-oriented programming model that forms the basis of application frameworks. Although the interfaces between components lend themselves to be modeled as message protocols, and hence the components themselves as objects, there is little "object orientation" beyond this. In particular, there is *no inheritance across component boundaries*. Instead of invoking "super", objects specifically have to request the services of other objects more prepared for fulfilling a particular task, i.e. *delegate* the request.

We are in the process of developing a system based on the experimental language *Lagoona* (Franz, 1997a), a descendant of Oberon (Wirth, 1988a), in which all component interaction is initiated by message-passing among objects. Unlike conventional object-oriented languages, however, Lagoona's messages are "stand-alone" data objects, and not subordinate to classes. For example, Lagoona's messages can be stored in data structures, such as "message queues". They can be duplicated and sent to more than just a single receiver object, and previously stored messages can be executed asynchronously.

While Lagoona, just like other object-oriented languages, provides type extension (Wirth, 1988b), polymorphic variables, and automatic method dispatch depending on the type of the receiver argument, it does away with the concept of *method inheritance* that is usually offered by object-oriented languages. Instead, objects

have the option of explicitly forwarding received messages to other objects that handle them on their behalf, using a delegation mechanism called *re-send*. Hence, the effect of inheritance can be simulated, if required, but it turns out that re-send can be used to create far more effective architectures than traditional class inheritance. In particular, Lagoona's microscopic program architecture corresponds to the macroscopic architecture of component-oriented systems.

In the next section, we first examine the "Stack" data structure of the Java Class Library in a little more detail, exposing the many dependencies a user of the class has to be aware of. Without going into the syntactic peculiarities of Lagoona, which are not the subject of this paper, we then explain how much more straightforward and elegant an implementation would be in a language that provides stand-alone messages. This brings us to the conclusion that the "compose out of parts and link by message-passing" model of software construction might be better suited as a basis for software reuse than the "inherit from framework and customize by overriding" model that is currently popular.

4   HIGHER-ORDER DATA STRUCTURES: CONTAINERS

An adequate support for user-defined data structures is an important characteristic of a good developing environment. For example, most object-oriented application frameworks provide a host of pre-defined "container" data types that implement data structures such as linked lists, stacks, hash tables, and the like, sparing the programmer the effort of re-implementing this functionality over and over. As a further benefit, the common code handling the container functionality is factored out into a single class, rather than being duplicated in many different places, reducing the overall application footprint.

Unfortunately, in object-oriented application frameworks, these container data types are often realized in a relatively heavyweight fashion. As a case study, let us take a look at how container data structures are implemented in the Java Class Library (Chan and Lee, 1997). Java's standard package *java.util* provides five container classes: *Dictionary*, *HashTable*, *Properties*, *Vector*, and *Stack*, forming the inheritance hierarchy depicted in Figure 4.
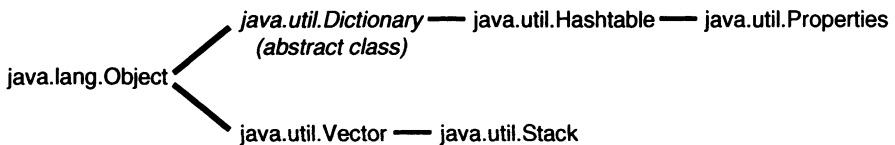
Figure 4: An excerpt from the Java class hierarchy

In particular, a Java *Vector* is an expandable indexed data structure holding an arbitrary number of objects as its elements. Elements can be added and removed at any index position. A Java *Stack* is a specialization of such a *Vector* that additionally provides simplified methods (such as *push* and *pop*) for managing data in a LIFO queue. The specification of *Stack* doesn't mandate invalidation of the methods inherited from *Vector*, hence it is unclear whether the integrity of a Stack is actually guaranteed in implementations of the Java Class Library, or whether it can be circumvented by calling an inherited *Vector* method to insert or remove elements in the middle of a *Stack*. It seems that the inheritance relationship between *Vector* and *Stack* has the sole purpose of facilitating code reuse, while creating a problematic extension relationship.

A common task required of container data structures is enumeration of their contents. For this purpose, the Java Class Library specifies an interface called *Enumeration*. The *Enumeration* interface provides two methods, *hasMoreElements()* that determines whether there are any more elements in the enumeration and *nextElement()*, which retrieves the next element in the enumeration. All five of the above-mentioned container classes provide a method *elements()* that returns an enumerator object compatible with the *Enumeration* interface*. The enumerator object can then be used for accessing the individual elements of the container data structure. For example, one might use a loop such as depicted in Figure 5 for accessing the elements of a vector.

```
for (
    Enumeration e = v.elements();  // create an enumeration
    e.hasMoreElements();  // while still some unprocessed elements left
    )
{
    Object o = e.nextElement() ;  // retrieve the next element
    o.doIt;  // do something with it
}
```

Figure 5: Iterating over a container data structure in Java

There is a considerable overhead involved in using container classes in this manner. Every time that the elements of any of our containers need to be

* Note that while *Dictionary* and *Vector* each provide a method *elements()* returning an *Enumeration*, these two instances of *elements()* are conceptually two different methods. In languages such as Java, the intent of simultaneously introducing the same message into two disjoint class hierarchies can be expressed only by providing two textually equal definitions. In Lagoona, on the other hand, messages are "stand-alone" data objects, and not subordinate to classes. This means that messages are defined globally on the package level, and not within the scope of a class; as a consequence, two otherwise disjoint classes can "share" the same message by providing method implementations for it.

enumerated, we first have to create an enumeration object. The enumeration object contains pointers to the elements of the original data structure and keeps track of which elements have been enumerated already.

Now think of how much effort is required to create an alternative class *MyStack* that could be used as a substitute for *java.util.Stack*. Not only does one have to implement the complete functionality of *Vectors*, since *Stacks* happen to be backward-compatible with them, but one also has to implement a complete private enumerator class for each container class. Further, container classes have complex interactions with their enumerators, which in the case of Java are not even clearly specified (for example, what if an object is removed from a container before an ongoing enumeration has come to a finish?).

5   USING FIRST-CLASS MESSAGES

How then, do first-class messages make this any easier? There are several contributing factors. First, if we do away with method inheritance, as in the language Lagoona, we arrive at a looser coupling between the individual software modules, increased locality, and regain what Bertrand Meyer once called *modular continuity* (Meyer, 1988): a small change in a module should not trigger a large change in the resulting system. Second, we gain *structural uniformity*: fine-grained interaction between individual objects now follows the same model as the coarse-grained interaction between *OpenDoc* parts or *ActiveX* components. Note that the abolition of code inheritance need not necessarily lead to code duplication or diminished code reuse. On the contrary, since messages are now tangible elements at the source-language level, they can be re-sent even outside of the type-extension hierarchy of the original receiver object. The main difference is that this process is now explicit. The eventual implementation can be made efficient (Franz, 1997a).

Most importantly, however, the simple model of "tangible" messages leads to sweeping architectural simplifications. For example, instead of the clumsy enumeration feature of *java.util.Stack* described above, it would be much more elegant if every container data structure simply offered a mechanism by which messages could be *generically broadcast* to all the contained elements. For example, in our paper on Lagoona (Franz, 1997a), we describe the use of a *distributor object* that can receive arbitrary messages and automatically re-send them to all elements of a private data structure (i.e., *broadcast* them). This is not only simpler than the approach taken in Java, but it is also safer, since no direct pointers to the individual contained elements are ever revealed. Note that this approach also reduces overall complexity and code-size: instead of programming a loop over the enumerator's data structure at every iteration site, the loop is now encapsulated wholly within the container.

As a case study, consider the example of an extensible graphics editor. By extensible, we mean that the graphics editor is supplied with a number of pre-

defined graphical shapes, and the user has the ability to later add components that implement further shapes. The addition of such late extension needs to be possible without requiring that any part of the editor or any already existing extensions be updated.

In a traditional framework-based system, this is solved by letting the graphical editor communicate with all of its shapes through an *abstract interface*, and by using dynamic loading to add classes implementing additional shapes to the already executing system (Franz, 1997b). This is a good solution to the problem, but only as long as the abstract message-interface of "shape" objects is considered immutable.

Now imagine that we want to create a new kind of shape extension for the existing graphics editor that represents a *clock*, a circle with two hands that display the current time in analog form within the graphics-editor window. Just like any other shape, clocks need to be instantiable: users may create arbitrarily many clocks within each graphics document.

Hence, the question becomes: how do we control the periodic update of each clock's display? A naive solution is to attach a separate thread to each clock-object that handles the update; however, this is of course a very uneconomic use of processor resources. Ideally, there should be only a single thread that is responsible for updating *all* the clocks on the screen, no matter how many of them are present. (This also has the added advantage that clock updates become synchronous)

A "good" solution therefore uses just one thread, located in the same module as the clock class, which periodically sends *tick* messages to all clock-objects, instructing them to update their respective displays. Unfortunately, in a conventional object-oriented system, this means that the thread sending the *tick* messages needs to keep track of all the clock objects, because *tick* messages can be sent only to clock objects, and not to other graphical shapes: At the time that the abstract message protocol of "shape" was defined, it wasn't anticipated that we would eventually need *tick* messages, and hence no provision was made for them.

However, there is a considerable bookkeeping effort associated with keeping track of the clock objects: it means that every time a new clock is created, it needs to register with the *tick* thread, and before any clock is destroyed, it needs to unregister. Since the latter can occur also as a side-effect of user actions such as closing a window containing such a clock, implementing all of the bookkeeping operations is no trivial task. In fact, it is feasible in practice only if a finalizing garbage collector is available that can perform the unregister operation automatically.

Besides requiring substantial programming effort, the outlined solution has aesthetic shortcomings: It requires a separate data structure specifically for linking together all the clock objects. This is in addition to the link already maintained in the graphics editor, which groups together all the objects belonging to a graph. One can easily imagine that many such separate data structures are required once that

the object hierarchy grows to include lots of object types whose message protocol cannot completely be accommodated by the abstract "shape" interface.

In traditional object-oriented languages, the only way of avoiding all of this is by exposing the graphic editor's underlying data structure that links the individual objects in a graph. Then, instead of maintaining separate data structures, the extension modules can iterate over the full graph, sending messages only to selective objects. However, this violates the concepts of information hiding and safety, as it gives the programmer of one extension access to objects created by another one.

This is where first-class messages come in. First-class messages allow a complete separation of concerns while greatly simplifying the construction of such extensible systems. If the above example were programmed in Lagoona, the graphics editor could offer a procedure that *generically broadcasts* any message to all objects in a graph. In this case, the iterator is wholly contained in the "main part" of the editor, which also completely encapsulates the manner in which individual objects in a graph are linked together. As a consequence, no object need ever be exposed to any code written in another module. Moreover, the protocol of messages that can be broadcast need not be defined a priori; only those messages that relate to the interaction between the editor itself and the shapes it contains need to be specified abstractly.

Hence, our *tick* thread would instruct the graphics editor to tell every object that the time had advanced. This message would be broadcast to all objects, but only the clocks would actually have an implementation associated with the message; all other objects would simply ignore it. The added cost of this scheme is that the iterator needs to touch every object, including objects that don't "understand" the message being sent (which can be determined by a simple table look-up). In return, a significant amount of bookkeeping effort is avoided elsewhere and great architectural simplifications are gained.

6   RELATED WORK

While lacking direct language support for messages and methods, the *Oberon System* (Wirth and Gutknecht, 1992), which originally inspired the Lagoona language, has an architecture that makes extensive use of generic message broadcast and message re-send. For example, all editing applications in the Oberon System are structured after the *Model-View-Controller* pattern (Krasner and Pope, 1988), and update-events that result from model changes are distributed to the respective viewers by a broadcast mechanism. Hence, rather than keeping pointers from the model back to the views that display them, models notify their viewers of a change by sending a message to the root of the display hierarchy. The message then "trickles down" the display hierarchy, automatically forwarded by every container-object to each of its children. Hence, for example, a window-object passes all received messages to the objects representing its contents. Note that the

messages forwarded in this manner may include messages that are not included in the protocol of the container.

The main difference between Oberon and the system we are building is that the Oberon language (Wirth, 1988b), in which the Oberon system is written, doesn't automate type-dispatch. This places a burden on the programmer and simultaneously also makes efficient implementation more difficult. Our project starts off from the general architecture of the Oberon System with the aim of improving it further by the addition of a small amount of language support.

Another object-oriented programming language that explicitly renounces method inheritance is *Emerald* (Raj et al., 1991). Similar to Lagoona, Emerald stresses the concept of *locality* (which its authors call *object autonomy*) by forcing all behavior to be encapsulated within the definition of each individual object class. In Emerald, the domain of encapsulation is the class; unfortunately there is no separate package concept. An interesting aspect of Emerald is the fact that it bases object substitutability on *interface conformity* (rather than common type-ancestry); hence multiple implementations of the same class are possible. Emerald is targeted towards distributed systems and hence has slightly different goals than Lagoona; in particular, Emerald has no equivalent to Lagoona's re-send mechanism and hence, in the absence of inheritance, makes code-sharing difficult.

Several recent papers (e.g., Odersky and Wadler, 1997; Krall and Vitek, 1997) propose to extend and improve upon the original definition of the Java language. Some of the proposed constructs, such as *Higher-Order Functions* (Odersky and Wadler) and *Iterators* (Krall and Vitek) present alternative solutions to the "loose coupling" idea that is approached in Lagoona through the mechanism of message forwarding. However, while these additional constructs raise the expressive power of Java, we feel that they make an already complex programming language even more difficult to master. The primary design goal of Lagoona, on the other hand, is to replace method inheritance altogether by a simpler and more flexible construct that can be used to emulate inheritance if required. We contend that Lagoona is a simpler language than Java, and that message forwarding is a "natural" paradigm for component-based systems since it uniformly applies to intra-component messaging as well as inter-component communication.

7   SUMMARY AND CONCLUSION

Elevating messages to stand-alone status on the programming language level, alongside classes and variables, gives rise to new system architectures. These architectures lead to elegant and small implementations, and since they duplicate the macroscopic structure of software component systems on a microscopic level, provide structural uniformity. For these reasons, we think that languages that directly correspond to such architectures are better suited for the implementation of component-oriented systems than conventional object-oriented languages.

8 REFERENCES

Arnold, K. and Gosling, J. (1996). *The Java Programming Language*; Addison-Wesley.

Card, D. and Comer, E. (1994) Why Do So Many Reuse Programs Fail? *IEEE Software*, **11**, 5, 114-115.

Chan, P. and Lee, R. (1997). *The Java Class Libraries: An Annotated Reference*; Addison-Wesley.

Evered, M., Keedy, J.L., Schmolitzky, A., and Menger, G. (1997). How Well Do Inheritance Mechanisms Support Inheritance Concepts?, in Hanspeter Mössenböck (Ed.), *Modular Programming Languages, Proceedings of the Joint Modular Languages Conference, JMLC'97*, Springer Lecture Notes in Computer Science No. 1204, 252-266.

Franz, M. (1997a). The Programming Language Lagoona: A Fresh Look at Object-Orientation. *Software – Concepts and Tools*, 18:1, 14-26.

Franz, M. (1997b). Dynamic Linking of Software Components *IEEE Computer*, 30:3, 74-81.

Goldberg, A. (1984) *Smalltalk-80: The Interactive Programming Environment*; Addison-Wesley.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and its Implementation*; Addison-Wesley.

Krall, A. and Vitek, J. (1997) On Extending Java, in Hanspeter Mössenböck (Ed.), *Modular Programming Languages, Proceedings of the Joint Modular Languages Conference, JMLC'97*, Springer Lecture Notes in Computer Science No. 1204, 321-335.

Krasner, G.E. and Pope, S.T. (1988) A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-89. *Journal of Object-Oriented Programming*, 1:3, 26-49.

Lindholm, T., Yellin, F., Joy, B., and Walrath, K. (1996) *The Java Virtual Machine Specification*; Addison-Wesley.

McIlroy, M.D. (1976) Mass Produced Software Components, in *Software Engineering, Concepts and Techniques, Proceedings of the NATO Conferences*, New York, 88-98.

Meyer, B. (1988) *Object-Oriented Software Construction*; Prentice-Hall.

Muys-Vasovic, J.-D. (1989) MacApp: An Object-Oriented Framework Application, in *Tutorial Notes, Technology of Object-Oriented Languages and Systems (TOOLS) '89*.

Odersky, M. and Wadler, P. (1997) Pizza into Java: Translating theory into practice, in *Proceedings of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, 146-159.

Raj, R.K., Tempero, E., Levy, H.M., Black, A.P., Hutchinson, N.C., and Jul, E. (1991) Emerald: A General-Purpose Programming Language. *Software-Practice and Experience*, 21:1, 91-118.

Wegner, P. and Zdonik, S.B. (1988). Inheritance as an Incremental Modification Mechanism, or, What Like Is and Isn't Like, in *ECOOP'88 Proceedings*, Springer Lecture Notes in Computer Science, No. 322, 55-77.

Wirth, N. (1988). The Programming Language Oberon. *Software-Practice and Experience*, 18:7, 671-690.

Wirth, N. (1988). Type Extensions. *ACM Transactions on Programming Languages and Systems*, 10:2, 204-214.

Wirth, N. and Gutknecht, J. (1992). *Project Oberon: The Design of an Operating System and Compiler*; Addison-Wesley.

9   BIOGRAPHY

Michael Franz is an assistant professor in the Department of Information and Computer Science at the University of California, Irvine. He holds a Doctorate in Technical Sciences and a Diploma in Computer Engineering, both from the Swiss Federal Institute of Technology (ETH) in Zurich. Further information about Franz and his research can be found at *http://www.ics.uci.edu/~franz* .