Inheritance Decomposed[⋆]

Position Paper

Peter H. Fröhlich
phf@acm.org

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract. Inheritance is often portrayed as a fundamental ingredient of object-oriented programming, one which is essential to building extensible software systems. However, inheritance is also a controversial mechanism with many competing and often contradictory interpretations, whose use can even impede extensibility in certain scenarios. Decomposing inheritance into the more basic mechanisms of object composition and message forwarding solves many of these problems. The resulting programming model is simpler yet more flexible than one based on inheritance, and illustrates that inclusion polymorphism is actually the more fundamental ingredient.

1 Introduction

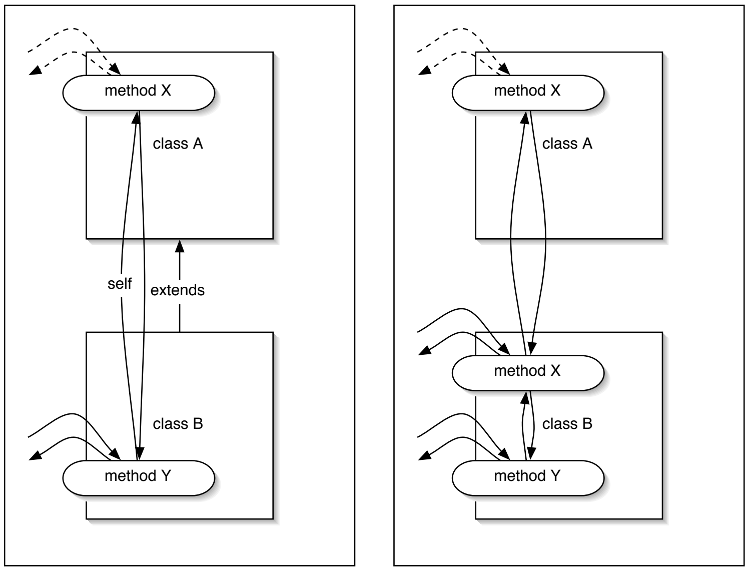
The paradigm of object-oriented programming is often characterized by the use of *inheritance* between classes. Wegner popularized this view [22] and it seems to persist to this day, for example in Moby [5]. In the sense of object-oriented programming, inheritance is an *incremental modification* mechanism that transforms an "ancestor" class into a "descendent" class by augmenting it in various ways. While this basic understanding of inheritance is almost universally accepted, there are still many competing interpretations regarding the details of the mechanism [21]. Several of these interpretations imply contradicting guidelines for the application of inheritance in the design and implementation of software systems. Some well-known examples illustrate this problem:

– The "classic" notion of inheritance confuses the relations of *subtyping*, which concerns substitutability, and *subclassing*, which concerns code reuse [1, 4, 15, 19]. Surprisingly, even relatively modern languages such as Java [10] separate these relations only partially.
– From a modeling perspective, inheritance can be used for either *classification* or *implementation* purposes. The former implies statements such as "Every square is a rectangle for which width = height holds." while the latter implies statements such as "Given squares, we need an additional dimension to implement rectangles."
– Different programming languages offer numerous variations of the "subclassing aspect" of inheritance. In Java and Smalltalk [9], descendent methods control whether ancestor methods are called. In Beta [16], ancestor methods control whether descendent methods are called. In Eiffel [17], descendents can *cancel* or *rename* ancestor features, neither of which is possible in Java, Smalltalk, or Beta.

If we follow established convention and consider inheritance the "hallmark" of object-oriented programming, we have to admit that there are as many different "paradigms of object-oriented programming" as there are interpretations for inheritance. This is quite unsettling, especially since there is a comparatively undisputed concept which is central to object-oriented programming as well: inclusion polymorphism [3].

The thesis of this paper is that inclusion polymorphism (and therefore subtyping), not inheritance (and therefore subclassing), is the "essence" of object-oriented programming.[1] To justify this claim—and hopefully also to "rekindle" the inheritance debate—we argue in three main steps: In Sect. 2 we show to what extent various forms of inheritance can be decomposed into the more basic mechanisms of object composition and message forwarding, relying only on inclusion polymorphism. In Sect. 3 we analyze a number of common object-oriented design patterns to illustrate that subtyping is more commonly used than subclassing in object-oriented programming. In Sect. 4 we briefly discuss and evaluate the programming style induced by a focus on composition and forwarding. In Sect. 5, we summarize our discussion, outline several avenues for future work, and offer our conclusions. While some of the points we make in this paper have been made before [1, 11], we believe that they benefit from our more focused discussion. Also, we feel that these points need to be restated at least once every decade, making this workshop a perfect opportunity.

[⋆] Decompose, v.: To become resolved … from existing combinations; to undergo dissolution; to decay; to rot. [12]
[1] Incidentally, inclusion polymorphism—or *implementation polymorphism* [6]—is also the mechanism worth preserving as we move from object-oriented to component-oriented programming [20].



(a) Inheritance (b) Forwarding

Fig. 1. Call patterns for adding methods using inheritance and fowarding.

2 Decomposing Inheritance

Our goal for this section is to decompose various forms of inheritance into the more basic mechanisms of object composition and message forwarding. In some sense, this is similar to the debate on prototypes vs. classes and inheritance vs. delegation which raged in the 1980s [14]. However, we target a class-based object model [7] that is based on separating messages (abstract operations) from methods (concrete operations). Our model defines *interface types* as sets of messages and *implementation types* as sets of methods with associated storage declarations.[2] Interface types are related by structural subtyping based on the message sets they denote. However, implementation types are never related in any way, and in particular not through inheritance. Thus, reuse or adaptation of existing implementation types has to be lifted to the instance (object) level.

To show to what extent our object model can replace inheritance-based models, we discuss several examples revolving around the major issue that arises in the presence of inheritance: the typical call patterns produced using *super*, *inner*, and *self*. We do not discuss the problem of shared state since it can always be expressed in terms of message sends. Furthermore, we ignore the problem of object identity and the related question of transparency to clients; the recently proposed *generic wrappers* mechanism already solves this problem very well [2]. Finally, we can safely disregard issues such as encapsulation, aliasing, and further details about the type or module system.

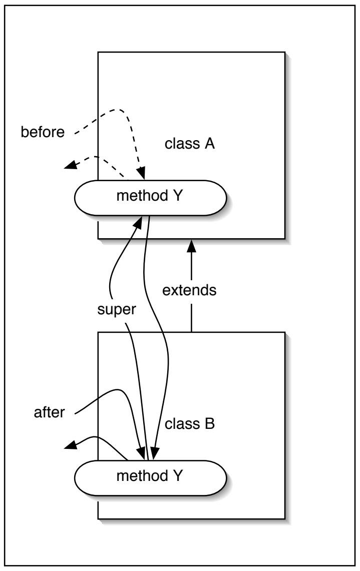
The “base case” for our discussion of call patterns is a single class A that has neither ancestors nor descendents. Since inheritance is not used at all, we (trivially) do not need to decompose it either.

Next, we consider a class A that defines a method X and a descendent class B that extends A and defines a Y method. This scenario is depicted in Fig. 1 where we also show the resulting decomposition.[3] On the one hand, since the message Y was not known when the method X was implemented, the call pattern resulting when we send a message X to an instance of B can not involve the Y method. This case can thus easily be decomposed by implementing X in B to forward to an instance of A. On the other hand, if we send a message Y to an instance of B, the method Y *could* send a X message to *self*. However, this case is already handled correctly by the forwarding method X in B. Thus we can accurately decompose inheritance relationships that *add* methods in descendent classes.

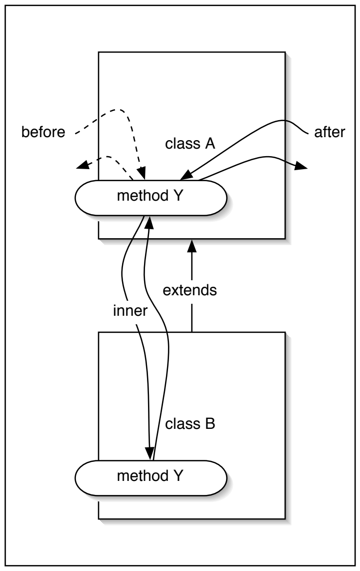
For the next two examples, we consider a class A that defines a method Y and a descendent class B which extends A and *also* defines a Y method. This scenario is depicted in Fig. 2 for two different inheritance mechanisms.

Fig. 2(a) illustrates the well-known “overriding” semantics of inheritance used in Smalltalk, Eiffel, Java, and most other object-oriented languages. If we send the message Y to an instance of A, the method $A.Y$ will be

[2] Somewhat surprisingly, our model is similar to Smalltalk’s original model [13] with the addition of static typing.
[3] For space reasons, we have to omit similar figures for decomposed versions in our object model from now on.



(a) Override (e.g. Java)



(b) Augment (e.g. Beta)

Fig. 2. Call patterns for overriding or augmenting methods.

invoked ("before" in Fig. 2(a)). However, if we send the same message to an instance of B, the method $B.Y$ will be invoked ("after" in Fig. 2(a)). Inside the $B.Y$ method, we can invoke $A.Y$ by sending the message Y to *super*. To decompose the "overriding" semantics, the class B has to hold a reference to an A instance, and to emulate the receiver *super* we send messages to that instance.

Fig. 2(b) illustrates the somewhat obscure "augmentation" semantics of inheritance used in Beta. As before, if we send the message Y to an instance of A, the method $A.Y$ will be invoked ("before" in Fig. 2(b)). However, if we send the same message to an instance of B, the method $A.Y$ will *still* be invoked ("after" in Fig. 2(b)). Inside the $A.Y$ method, we can invoke $B.Y$ by sending the message Y to *inner*.[4] To decompose the "augmentation" semantics, the class A has to hold a reference to a B instance, and to emulate the receiver *inner* we send messages to that instance.

A number of comments are in order at this point. First, note that our decomposition of these two inheritance mechanisms makes their differences more explicit. The "overriding" semantics allow us to redefine methods in ways that the author of the ancestor class did not anticipate. The "augmentation" semantics rely on the author of the ancestor class to provide suitable "hooks" for extension. We can thus view these inheritance mechanisms as corresponding to *wrappers* and *plugins* in the paradigm of component-oriented programming [20]. Next, note that while the two inheritance mechanisms can not easily emulate each other, we can emulate both through composition and forwarding, even concurrently. That is, we can "extend" a class A through "overriding" by wrapping it in an instance of a class B and through "augmenting" by supplying it with a plugin instance of a class C as well. Finally, note that in our model, the concepts of *super*, *inner*, and *self* are very transparent and therefore easy to understand. Sending a message to *self* always invokes a method in the same class, sending a message to *super* always invokes a method in the "closest" ancestor class, and sending a message to *inner* always invokes a method in the "closest" descendent class. This is not the case in inheritance-based models, which will complicate the remaining decomposition.

Consider the example in Fig. 3. The class A initially defines two methods $A.X$ and $A.Y$ and the method $A.Y$ sends the message X to *self*, resulting in the dotted call pattern. The descendent class B extends A and overrides $A.Y$ with a $B.Y$ method. This method does not call $A.Y$ through *super* but instead sends X to *self*, resulting in the dashed call pattern. We have already seen how to decompose the scenario up to now: we give B a reference to an A instance and defined $B.X$ as a forwarding method. However, we will now add a third class to the mix. The descendent class C extends B (and therefore A as well) and overrides $A.X$ with a $C.X$ method, which also calls $A.X$ through *super*. The resulting call pattern can not be easily decomposed. If we define a forwarding method $C.Y$ the send of X to *self* in $B.Y$ would cause in $C.X$ to be ignored. We cannot make C into a plugin for B either

[4] In Beta, *inner* is actually a keyword and no message sending is involved; the semantics are identical however.

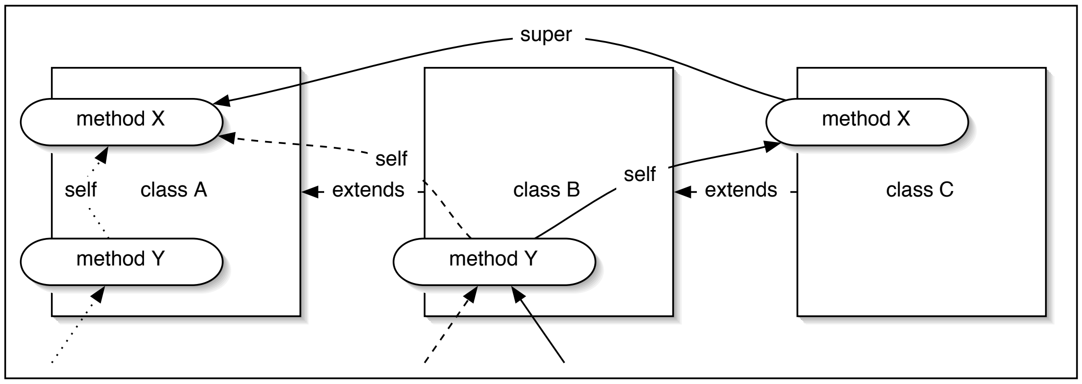


Fig. 3. The intricacies of inheritance with overriding semantics for *self* message sends.

since $C.X$ uses *super* whereas plugins only work for *inner*. Also, this would make B dependent on a plugin even when none is required, i.e. when we only need B and not C instances.

The only viable approach is to pass the "right" *self* as an explicit parameter with each message. To see how this works, we need to "follow the message" through the resulting call pattern. Assume we send Y to an instance c of C with the parameter *self* identifying the (arbitrary) source instance. The $C.Y$ method forwards this message an instance of B and also replaces the existing *self* parameter with c instead. In $B.Y$ we send X to the *self* = c we were passed, but in turn we replace the *self* parameter with b again. Instead of the call through *super*, $C.X$ will send X to the *self* = b parameter, passing *self* = c once again. The $B.X$ method simply forwards to $A.X$ and we have recreated the call pattern induced by inheritance in this example.

It should be quite obvious that the resulting implementation is very sensitive to changes in the composition. One possible conclusion to draw from this example is that inheritance should not be decomposed at all. However, we believe that the opposite is true. Our decomposition simply sheds some light onto the complexity of call patterns that object-oriented programs sometimes exhibit. Put another way, if a call pattern is complex to decompose, chances are that it is complex to understand as well. This complexity remains hidden from the programmer by "virtue" of the inheritance mechanism, which in turn makes systems exhibiting these call patterns difficult to understand. Finally we get to the question of how common these complex call patterns are in practice, which leads us to the next section.

3 Design Patterns

Object-oriented design patterns are "elements of reusable object-oriented software" that capture proven solutions to recurring software development problems. Somewhat unintentionally, design patterns also provide examples for typical uses of object-oriented programming languages. Given that something is described as a design pattern, it must have occurred often enough to be identified as such. Thus, if many design patterns utilize a certain language mechanism, we can be reasonably sure that many "real world" software systems use the mechanism as well.

In Table 1, we list the design patterns from [8] and classify them regarding their use of inheritance. A "+" in the column "Abstract" means that inheritance from a fully abstract ancestor class is used to establish a common interface in the sense of subtyping. A "+" in the column "Concrete" means that inheritance from a (partially) concrete ancestor class is used in the sense of subclassing. Somewhat surprisingly, only three out of 23 design patterns critically depend on inheritance for subclassing. Two of these, *Factory Method* and *Template Method*, use inheritance to provide "hooks" that descendent classes are expected to override. As we saw in Sect. 2, the resulting call patterns can be decomposed using the "plugin" approach. Only one variation of the *Adapter* pattern resists any attempt at decomposition. A *Class Adapter* uses multiple inheritance for efficiency reasons: it allows adapting an existing class without the need for auxiliary objects. Obviously, we can not decompose this particular use of inheritance while staying true to the intent of the pattern.

Our sample of design patterns illustrates that most uses of inheritance can be decomposed easily. While it would be a fallacy to conclude that because some mechanism is *not* used in design patterns, it is also unused in real systems, we still get the impression that the importance of inheritance might be overrated to some extent.

Pattern	Abstract	Concrete	Notes
Abstract Factory	+	−	
Builder	+	−	
Factory Method	−	+	Replacing “inner” (Sect. 2)
Prototype	+	−	
Singleton	−	−	
Adapter	+	+/−	Class Adapter / Object Adapter
Bridge	+	−	
Composite	+	−	
Decorator	+	−	
Facade	−	−	
Flyweight	+	−	
Proxy	+	−	
Chain of Responsibility	+	−	
Command	+	−	
Interpreter	+	−	
Iterator	+	−	
Mediator	+	−	
Memento	−	−	
Observer	+	−	
State	+	−	
Strategy	+	−	
Template Method	−	+	Replacing “inner” (Sect. 2)
Visitor	+	−	

Table 1. Use of inheritance in design patterns [8] for interface (fully abstract ancestor) or implementation (partially concrete ancestor) reasons.

4 Programming Style

At this point, it is natural to ask “What remains of object-oriented programming if we take inheritance away?” As can be expected by now, “Almost everything!” is an appropriate answer. Without inheritance in the sense of subclassing or code reuse, we still approach object-oriented design with many of the same techniques. Consider, for example, the development of a container class such as `Bag`. We start by designing suitable operations (messages) and their associated semantics. The two most obvious operations are `Add(element: ANY)` to add an element and `Remove(element: ANY)` to remove one. Operations such as `Choose(): ANY` to select a “random” element, `Clear()` to remove all elements, and `Empty(): BOOLEAN` to check whether a `Bag` is empty might be useful as well.

Once we have decided on an interface (set of messages), we survey existing interfaces to find messages with identical semantics that might already exist. If there are no such messages, or none are suitable for some reason, we declare the messages and the interface type as shown in Fig. 4. Before embarking on the actual implementation, it is advisable survey existing implementations for potential reuse. For example, we might already have a `List` container class, and we could reuse it (by *composition*!) for the `Bag` implementation. To avoid creating a concrete

```
MODULE com.lagoona.papers.ecoop02.Bags;
MESSAGE
  Add(element: ANY);
  Remove(element: ANY);
  Choose(): ANY;
  Clear();
  Empty(): BOOLEAN;
TYPE
  Bag = { Add, Remove, Choose, Clear, Empty };
END com.lagoona.papers.ecoop02.Bags.
```

Fig. 4. An interface for the `Bag` container class.

dependency, we could also design a plugin interface for `Bag` that describes the operations required of the class we use for implementation.

Now consider the process of adding a variant of `Bag`, for example one that returns the number of elements when it receives the message `Size(): INTEGER`.[5] Using inheritance, it would be impossible to define a descendent `CountingBag` in a modular fashion, i.e. without knowledge about the `Bag` implementation. The problem is that the `Clear` message might be implemented either in terms of `Choose` and `Remove`, or it might access the underlying data structure directly. In the former case, we have to override `Add` and `Remove` only, whereas in the latter case we also have to override `Clear` to count accurately. As we alluded to in Sect. 2, the intricate call patterns resulting from sends to *self* in the presence of inheritance are to blame. Using composition and forwarding, we are "forced" to add the counting functionality through a wrapper that "overrides" all relevant operations, making our extension modular by construction.

5 Conclusions

We have discussed an approach for decomposing inheritance mechanisms as found in many object-oriented programming languages into the more basic mechanisms of object composition and message forwarding. Our analysis of common object-oriented design patterns also showed that inheritance might not be as central to object-oriented programming as is often believed. We have also illustrated the programming style induced by a focus on composition and forwarding using several examples.

One of our reasons for investigating the exact relationship of inheritance to composition and forwarding can be found in our work on the programming language Lag*oo*na [7]. Lag*oo*na does not provide an inheritance mechanism because of its negative impact on component-oriented programming. Having developed a clearer understanding of the tradeoffs involved, we are now confident that this approach is indeed viable. While there are certain limits to the kinds of inheritance relations that can be expressed in a straightforward way through composition and forwarding, it seems that we exclude exactly the "dangerous" uses of inheritance. For the design and implementation of components and frameworks, we thus view these limitations as advantages.

We hope to provide a more formal argument along the lines of this paper in the near future, mainly to ensure that we did not "forget" any interesting cases. Also, we intend to investigate how the use of composition and forwarding can be made more attractive for developers by providing suitable abstractions and "shortcuts" in programming languages. Finally, we are interested in designing a development methodology that emphasizes programming without inheritance.

Acknowledgements: I thank Kimberly Haas, Fermin Reig, Michael Shafae, and the anonymous referees for valuable comments on earlier versions of this paper. Discussions with Ziemowit Laski improved my understanding of inheritance, and Nils Gruschka pointed out the importance of design patterns in this context. I am also indebted to Erik Ernst, Michael Franz, Andreas Gal, and Clemens Szyperski for many fruitful discussions. My work was partially supported by the National Science Foundation under grants EIA-9975053 and CCR-0105710.

References

1. P. America. Inheritance and subtyping in a parallel object-oriented language. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 276 of *Lecture Notes in Computer Science*, pages 234–242. Springer-Verlag, 1987.
2. M. Büchi and W. Weck. Generic wrappers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 201–225, Sophia Antipolis and Cannes, France, June 2000.
3. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, Dec. 1985.
4. W. R. Cook, W. Hill, and P. S. Canning. Inheritance is not subtyping. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 125–135, San Francisco, CA, 1990.
5. K. Fisher and J. Reppy. The design of a class mechanism for Moby. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 37–49, Atlanta, GA, May 1999.
6. P. H. Fröhlich and M. Franz. On certain basic properties of component-oriented programming languages. In D. H. Lorenz and V. C. Sreedhar, editors, *Proceedings of the First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, pages 15–18, Tampa Bay, FL, Oct. 15 2001. Technical Report NU-CCS-01-06, College of Computer Science, Northeastern University, Boston, MA 02115.

[5] This example was inspired by [18] where the fragile base class problem is discussed in much more detail.

7. P. H. Fröhlich, A. Gal, and M. Franz. On reconciling objects, components, and efficiency in programming languages. Technical Report 02-12, Department of Information and Computer Science, University of California, Irvine, CA 92697-3425, USA, Mar. 2002. Revised May 2002.
8. E. Gamma, J. Vlissides, R. Johnson, and R. Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
9. A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
10. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.
11. F. J. Hauck. Inheritance modeled with explicit bindings: An approach to typed inheritance. In A. Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 231–239, Washington D.C., Sept. 1993.
12. http://www.dict.org/. *Query for "decompose," definition provided courtesy of Webster*, Feb. 16 2002.
13. A. C. Kay. The early history of Smalltalk. In T. J. Bergin and R. G. Gibson, editors, *History of Programming Languages*, pages 511–597. ACM Press, 1996.
14. H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 214–223, Portland, OR, Nov. 1986.
15. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.
16. O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley (ACM Press), 1993.
17. B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.
18. L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In E. Jul, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382, Brussels, Belgium, July 1998. Springer-Verlag.
19. A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 38–45, Portland, OR, Nov. 1986.
20. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
21. A. Taivalsaari. On the notion of inheritance. *Computing Surveys*, 28(3):428–479, Sept. 1996.
22. P. Wegner. Dimensions of object-based language design. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 168–182, Orlando, FL, Oct. 1987.