Rutgers University Department of Computer Science Technical Report 387
Submitted 24 Jun 1999

Reconciling Encapsulation and Dynamic Dispatch via Accessory Functions *

C. Benjamin Flynn
Haverford College
Haverford, PA 19041
cflynn@haverford.edu

David Wonnacott
Department of Computer Science
Haverford College
Haverford, PA 19041
davew@cs.haverford.edu
(610) 896-4973
(610) 896-4904 (FAX)

Revised to April 1, 1999

Abstract

Many object-oriented languages allow encapsulation of data. We identify a conflict between data encapsulation and dynamic dispatch, and propose that it be resolved via the addition of "accessory functions". We discuss the implementation of accessory functions for C++, and show that this system extends naturally to allow multiple dispatch in C++.

1 Introduction

Many object-oriented languages allow (or require) some degree of data encapsulation. Encapsulation allows a programmer to focus on a single programming task and, upon its successful completion, to re-use the resulting code without ever again considering the details of its implementation. Since the language allows direct access to data only from those functions included in the class itself, we can rest assured that later uses of the class will not corrupt any properties guaranteed by the class as it was originally written. Encapsulation also allows for the substitution of more efficient data structures and algorithms after the initial code has been written.

Implicit in the idea of data encapsulation is the principle that programmers will not rampantly add operations to a completed class: New operations that work with objects of this class must instead be written outside the class, and manipulate objects of this class using the interface defined by the class. Adding operations "inside" the abstraction both weakens encapsulation and produces a software management problem – if many programmers are editing a class, how are these extensions combined? In languages like C++, new operations may either be included in new classes (possibly subclasses of the class being extended), or they may be "plain" functions (not part of any class). In other languages (e.g. Smalltalk and Java), new operations must be part of some class.

Data encapsulation can conflict with other aspects of object-oriented programming. Other work has focused on the problems arising when subclass operations access superclass data or private operations [Sny86, Lis87], or when a class' superclasses are considered an implementation detail of the class in a system that allows multiple inheritance without replicating common su-

*This work is supported by NSF grant CCR-9808694.

perclasses [Sny86].

In this article, we identify a conflict between encapsulation and the conventional use of dynamic dispatch. This conflict arises when a programmer needs to add functionality to multiple classes related by inheritance. In Section 2 of this article, we discuss the problems with maintaining encapsulation when operations are added to a group of classes. We propose, in Section 3, a mechanism for resolving this conflict via the addition of *accessory functions*. We then discuss the implementation of accessory functions for C++ in Section 4, noting in Section 4.5 that accessory functions can be used to add "multi-methods" [BKK+86] to C++. In Section 5, we discuss related work. We present our conclusions in Section 6.

2 Encapsulation and Dynamic Dispatch

Dynamic dispatch improves procedural encapsulation by eliminating the need to edit existing functions when new classes are added to an object-oriented system. Consider for example the task of representing nodes in a compiler's abstract syntax tree, as in the exercise in Chapter 1 of [App98a]. In a language like C, the programmer generally distinguishes among the different kinds of nodes by including a `kind` field in the data structure, and using a `switch` statement to select code that is appropriate for each kind of A.S.T. node. If new kinds of nodes are later added to the system, the programmer must edit each function that contains such a `switch`, introducing a new `case` for the new kind of node.

Dynamic dispatch lets the programmer add new kinds of nodes without violating a procedural encapsulation. A programmer building the above A.S.T. in Java (as in [App98b]) or C++ could create a superclass for A.S.T. nodes, and a subclass for each kind of node. A fragment of the C++ code for this example is shown in Figure 1. Instead of one central function for

```
class Exp {  // abstract superclass
public:
   virtual void print() = 0;
private:
   ...
};

class Num : public Exp {
public:
   int value();
   virtual void print();
private:
   ...
};

void Num::print()
{
   cout << value();
}

class Plus : public Exp {
public:
   Exp &lhs();
   Exp &rhs();
   virtual void print();
private:
   ...
};

void Plus::print()
{
   cout << ``('' << lhs().value()
        << ``+'' << rhs().value()
        << ``)'';
}
```

Figure 1: Dynamic Dispatch Example

all nodes, the programmer provides a function for each subclass (`print` in our example). The language automatically selects the appropriate function body via dynamic dispatch (assuming the function is declared `virtual` in C++). New kinds of nodes can be added as new subclasses, without the need to edit any existing code.

Unfortunately, this organization requires us to violate encapsulation when we add new operations to the system. Following our A.S.T. example, assume we now wish to provide code to interpret the tree. We must choose between violating encapsulation immediately by adding an operation to each class in the system, or creating a single `interpret` function that uses a `switch` to select the correct code for each kind of node (and then violating encapsulation later when we edit this function as a new class is added).

Object-oriented languages also allow the possibility of using inheritance to produce new subclasses that add an interpret operation. However, this introduces spurious uses of multiple inheritance, which we consider highly undesirable (though we have no problem with legitimate uses of multiple inheritance): If we are to apply `interpret` to each new kind of node through an `Exp` reference, then the new node classes must share a common superclass with an `interpret` function, which introduces a second superclass for the new node classes. Even in a language like Smalltalk, which does not require the use of a common ancestor to introduce the new operation, we will be faced with unnecessary multiple inheritance when we attempt to combine two sets of extensions (perhaps one adds interpretation and the other code generation for a compiler).

Another possible response to the need to add the `interpret` function to our compiler example is to claim that this operation should have been foreseen by the class designer, and thus belongs inside the encapsulation in the first place, so by adding it later we are simply fixing an oversight in the original class, not violating a correct encapsulation. However, this operation does not need to modify (or even access) any of the private data that maintain the consistency of the tree, and thus does not need to be "inside" the abstraction, regardless of when it is written. It should therefore be created outside the abstraction, on the grounds that only a minimal set of operations should be placed inside the abstraction (see [LG86, Section 4.9.3] or [Str97, Section 11.5.2] for a discussion of this principle).

```
// "pure virtual" accessory function
// for superclass
int interpret(virtual Exp &) = 0;

int interpret(virtual Num &n)
{
   return n.value();
}

int interpret(virtual Plus &p)
{
   return interpret(p.lhs())
        + interpret(p.rhs());
}
```

Figure 2: Accessory Function Example

3 Accessory Functions

We resolve this conflict between encapsulation and dynamic dispatch by introducing *accessory functions*. Accessory functions provide dynamic dispatch based on the type of an argument rather than the type of the receiver object (i.e. `this`). Since these functions are not part of the class on which dispatch is based, they do not have access to the private section of that class. Thus, we can add a set of accessory functions to a group of classes without editing or otherwise gaining access to any of their implementations. Figure 2 shows how accessory functions could be used to add a dynamically dispatched `interpret` function to our A.S.T. example of Figure 1.

The rest of this section describes various subtle points in the definition of accessory functions. We focus on finding definitions that are appropriate in the context of C++, although we believe accessory functions could also be implemented for other object-oriented languages in which the compiler is given partial information about argument types, such as Java (for Java, we would not allow accessory functions to be defined in dynamically loaded code or applied to dynamically loaded classes).

3.1 Syntax

We identify an accessory function in C++ by using keyword `virtual` in the declaration of one or more parameters. We consider `virtual` to be an attribute of a parameter (or the receiver object, in the case of the traditional use of `virtual` in C++), rather than an attribute of the function itself.

Accessory functions for C++ may be created outside of any class, as in Figure 2, or they may be created as member functions of one or more classes. Accessory functions may also be friends of one or more classes. As we will see in Section 4.5, we can use algorithms for multiple dispatch to create accessory functions that are dynamically dispatched based on the types of multiple arguments.

A function that has a single virtual parameter and is a friend of the class of its virtual parameter is similar to a traditional virtual member function, except for the syntax used to define and call it and the effect of combining inheritance and overloading (see Section 3.5). We have not resolved all issues involving the combination of a virtual parameter with a virtual receiver object (a traditional virtual function), so we currently do not allow this combination.

When an accessory function for a subclass needs to make use of the superclass function, it gives explicit type information for the virtual parameter(s), to prevent dynamic dispatch. We use syntax that is similar type casting for this purpose, as type casting would accomplish this goal for a statically dispatched function. To avoid introducing a new keyword, we reuse the word "virtual" for this purpose, i.e. the `interpret` function for `Num` could call the superclass function (were it not pure virtual) with the syntax `interpret( (virtual Exp) n )`. This is only legal if the new type is a public superclass of the argument type; its effect is analogous to using `interpret( (Exp &) n )` for a statically dispatched function. To avoid accidental mutual recursion for functions with more than one virtual parameter, we believe it is appropriate to require that explicit type information be given for all arguments (or at least all that might be virtual) in such calls. (This restriction may prove too onerous, but it is generally easier to loosen such restrictions than tighten them later.)

3.2 Restrictions

There are several restrictions on the use of virtual parameters:

- A program may not contain two functions that differ only in the "virtualness" of their arguments: We cannot have `f(Exp &)` and `f(virtual Exp &)`.

- The keyword `virtual` may only be applied to parameters from classes that already have at least one virtual function: we cannot have `f(virtual int &)`.

- Accessory functions with a single parameter cannot have parameters that change in their virtualness as we move up and down an inheritance graph. Note that this is not the case for traditional virtual functions – see the example in Section 3.4.

  For example, we must use a virtual parameter for either all or none of the `interpret` functions in Figure 2; we could, however, combine a function `int interpret(X &x)` with these functions as long as there is no `interpret` function defined on any class that is a superclass of both `X` and `Exp`.

- We generalize the above restriction for multi-parameter functions as follows: For any two functions f and g with the same name and number of parameters, either (a) there is some parameter position i such that the types of parameter i for f and parameter i for g are fundamentally incompatible (that is, there is no type that can be converted into both of these types), or (b) for every parameter position either both f and g have a virtual parameter or neither does.
- If two functions f and g have the same name and number of parameters and no fundamentally incompatible parameter, then all non-virtual parameters must have identical types: We cannot have `f(int, virtual Exp &)` and `f(float, virtual Exp &)`.

3.3 Type Casting

The compiler will not produce a virtual argument by applying an implicit type cast to a value (though it may still convert a subclass type reference (or pointer) to a superclass reference (or pointer)). This is an extension of the existing C++ rule that the compiler will not apply an implicit cast to produce the receiver object. `Virtual` is generally used in the declaration of a pointer or reference type parameter: When applied to the declaration of a value parameter, it affects type casting, but not dispatch (since complete type information must be present at compile time).

The lack of casting for virtual arguments means that adding `virtual` to a parameter of an existing function may interfere with the compilation of code had used this function: It may be necessary to add an explicit cast where an implicit one had been used previously to produce the (non-virtual) argument. We believe it would be no more difficult to implement a system that allows implicit casting for accessory functions, but that such a system could produce highly confusing results, as the casting system is based on the functions that are in scope, but the dispatch system is based on all compatible functions in the final program.

3.4 Function Selection Semantics

Function dispatch based on the types of multiple arguments, whether static or dynamic, raises two challenges: We must specify which function body is considered the correct choice for any given set of arguments (and when there will be errors because no single candidate can be chosen), and we must provide a way for the program to branch to this correct code efficiently. In this section, we consider the first of these two challenges, leaving the second for Section 4.

The specification of the appropriate function is generally based on some definition of *Most Specific Applicable* (MSA) [AGS94] function: Among all functions with the correct name and number of arguments, we try to identify a unique function with parameter types that match each argument at least as well as every other function, and with at least one parameter that matches an argument better than any other function. The "goodness" of a match is defined in terms of subtyping [AGS94], [AG96, Section 5.14] and possibly other factors such as (in C++) standard or user-defined conversion rules [Str97, Section 7.4]. Recall however that type casting for value types does not play a part in the selection of the type for a virtual argument.

When dispatch is performed by the compiler using static type information, the set of functions is generally restricted to those that are in scope at the point of the call. When dispatch is performed dynamically based on run-time type information, the set of candidate functions includes all functions in the final program that have the right name and number of parameters. In C++, the situation is made even more complicated by the fact that the compiler must use static information to select a dispatch mechanism (based on the presence or absence of the keyword `virtual` in the declarations of some of the functions being selected).

```
class b {
public:
  void test();
};

class d : public b {
public:
  virtual void test();
};

void call_test(b &o1, d &o2)
{
  o1.test();
  o2.test();
}
```

Figure 3: Choice of Dispatch in C++

For example, in Figure 3, the call `o2.test()` is dispatched based on the run-time type of the object referred to by `o2`, and this call may branch to a function from a class that was not even written at the time `call_test` was compiled. However, the call `o1.test()` always calls `b::test`, as it is dispatched based on static information about the type of the reference `o1`, even if the actual type of the object referred to is `d`. This sort of coding seems to run counter to the intent of the C++ language, and should perhaps generate (at least) a compiler warning. Note that it is explicitly illegal to do this with accessory functions (an error will be produced during dispatch table generation).

Our general rule for function selection is that there will be a call to the function that is most specific based on the static type information about non-virtual arguments and the run-time types of any virtual arguments. The restrictions given in Section 3.2 ensure that we can always identify at compile-time which arguments are being passed to virtual parameters, and that we will not have to choose between a value

3.5 Inheritance

The above rules imply that a superclass accessory function will be used (unless it is "pure virtual") if an accessory function is not provided for some subclass. This is similar to the inheritance of member functions, except that (in C++) a subclass member function overrides all superclass functions of that name, not just superclass functions with the same name and argument types.

3.6 Default Arguments

The above discussion has ignored the issue of default arguments in C++. We believe these can be handled by treating a declaration with a default argument as if it were a group of declarations of overloaded functions, all but one of which are simple inline functions that supply extra arguments and call the original function.

4 Implementation for C++

Implementation of accessory functions for C++ is mostly straightforward: In many cases, we simply need to move a step normally performed during the compilation of a single file to link-time. The presence of accessory functions affects several steps in the compilation process. The rest of this section consists of a review of some of the important points of dynamic dispatch in C++, and a discussion of how the presence of accessory functions affects various steps of this process.

4.1 Review of Virtual Functions

For an ordinary function call, the compiler may produce code that branches to a specific address (or label), or substitute the function body inline. In either case, static information about the body of the function being called may be used in the optimization of the call site. In the case of inlining or procedure cloning [Muc97], information about the call site may also be used in the optimization of the called function.

Dynamic dispatch affects the compilation of a function call in two significant ways: The mechanism used for the call is somewhat more complex, and the optimizer must make conservative approximations about which function will be called. Here, we focus only on the mechanism used to perform the actual call.

In traditional C++, virtual functions are generally implemented as follows. During the processing of each class, the compiler builds a table of pointers to each virtual function for that class. Note that the association of a given offset into the table with a given function name is known at compile time, though the entries in the virtual function table may not actually be resolved until the separate files that comprise the program are linked together.

All virtual functions must be listed in the class, and the compiler must process a superclass before any of its subclasses. Thus, the compiler can enforce the rule that a given virtual function has the same offset in the class where it is first introduced and all subclasses of this class. The compiler then ensures that a pointer to such a table of functions is included in each object that has virtual functions; Thanks to the rule above, the compiler can translate a call to a virtual function into a lookup at a known offset into this table and an indirect function call to the resulting address.

The dispatch mechanism is somewhat more complicated in the case of multiple inheritance, but retains the basic principle of generating tables at compile time and performing a lookup to a known offset to identify the function's address at run-time. In some cases, the table of function pointers must be located via an indirect reference.

Multiple inheritance also creates new opportunities for ambiguity at the time of a function call. When static dispatch is used, or dynamic dispatch is combined with single inheritance, the compiler can detect all potential ambiguities at the point of a function call. However, the combination of dynamic dispatch and multiple inheritance raises the possibility that ambiguity will arise at run-time. If a class C inherits a virtual function f from two or more superclasses, and does not provide its own f, any call to f for objects of class C will be ambiguous. We cannot statically prove that such a call will not occur, as calls to f for objects of class C may occur through pointers or references of C's superclasses. C++ prevents this from causing run-time errors by requiring that any such potential ambiguity be resolved by the programmer: An f must be provided for C.

4.2 Checking Accessory Functions

The previous section gave several requirements for accessory functions. Since it is possible to introduce any number of accessory functions without ever having more than one in a single file, these requirements must be tested at link time rather than compile time. Most of these tests are trivial if the compiler produces enough information for the linker to make the test (much of this information is already produced in a standard compiler).

4.2.1 Changes to "Virtualness"

As stated in Section 3.2, a program may not contain two functions that differ only in the virtualness of their arguments, and functions cannot change the "virtualness" of an argument between superclass and subclass parameters. These properties may be easily tested at link time if the compiler produces a list of all functions and their types for each file compiled.

4.2.2 Instantiation of an Abstract Class

To prevent run-time errors when a pure virtual function is called, no object may be created by an abstract class. However, when we compile an object creation, we do not know whether the class is made abstract by some pure virtual accessory function. Thus, the compiler must produce a list of all classes that are instantiated in each file, and a list of all pure virtual accessory functions (and their parameter types) in each file. The linker can then detect the creation of an object of an abstract class.

4.2.3 Ambiguities

Accessory functions raise new possibilities for ambiguity in the presence of multiple inheritance. For multiple inheritance, two superclasses of class C may each be given a version of an accessory function f in a file that is not visible when C is compiled. If C is not given a version of this function, the linker must produce an error to ensure that there is no run-time call to the (ambiguous) f for C.

4.3 Compiling

Now consider the task of compiling of a function call in the presence of accessory functions. The function name may correspond to several functions, only some of which have virtual parameters, and only some of which are in scope. The compiler must decide which arguments, if any, are to be treated as virtual in this call, and generate code for the function call.

We start by applying the C++ rules for overloaded function selection [Str97, Section 7.4] to the set of functions that are in scope and have the correct name and number of parameters, with the restriction that type casting of values cannot be used to create a match with a virtual parameter. If this process produces a unique best match, we generate either a regular function call (if the best match has no virtual parameters) or a dynamic function dispatch. If there is not a unique match, a compile-time error is produced. Note that the restrictions on the use of virtual given in Section 3.2 ensure that we must be able to correctly identify each argument as virtual or non-virtual: We cannot "mistake" an argument that would be virtual for a function that has simply not been scanned, for a non-virtual argument of a function in scope.

If the best match had exactly one virtual parameter, the dynamic dispatch is very much like a C++ virtual function call. We presume that the virtual argument will contain a pointer to a table of dynamically dispatched functions, and generate a load of a function pointer from this table, and a branch to this address. The presence of accessory functions means that we no longer know the size or layout of this table when compiling a single file, but we handle this by treating the offset into the table as an undefined reference that will be filled in later by the linker.

Note that we will need a separate dispatch table for each parameter position for a given set of functions. Calls to the functions `f(virtual Num &, int)` and `f(virtual Plus &, int)` must be handled with a different table from calls to `f(double, virtual Num &)` and `f(double, virtual Plus &)`.

If the function that best matches the call has multiple virtual arguments, we apply an algorithm for dynamic dispatch (see Section 4.5), using only the virtual arguments.

4.4 Building Dispatch Tables

We cannot build a dispatch table without a complete list of all functions that may be dynamically dispatched for a given type. Since this includes now functions that are separate from the class definition, we must wait until all functions are visible, at link time, to build the tables.

We rely on the compiler to give the linker a complete description of the DAG describing the class inheritance structure, and a list of all functions (complete with parameter types and information about which parameters are virtual). To handle the case in which only the first parameter may be virtual, we simply apply the algorithm used to build C++ virtual function tables after topologically sorting the inheritance DAG. The offsets are determined at this stage, and we can resolve the undefined references from Section 4.3.

4.5 Accessory Multimethods

There are several basic techniques for handling multiple dispatch:

- Retain a description of all functions (grouped by name) and the argument types for each; apply the algorithm for identifying the correct function body each time there is a call. (Caching may improve the speed of this technique.)
- Build a k-dimensional table for each function of arity k, in which the index for dimension k identifies the type used in the function call. At the time of the call, select the proper function body from the table. This table may be represented as a compressed sparse matrix or a decision tree.
- Treat multiple dispatch as a sequence of single dispatches, each of which identifies the type of one parameter.

Many of the data structures for multimethod dispatch grow significantly in size as the number of parameters involved in the dispatch increases. Since a programmer using accessory functions must identify which parameters are virtual, no special compiler techniques are needed to eliminate unnecessary dynamic dispatch on arguments that do not require it.

We do not believe that any new dispatch algorithms need to be developed for accessory functions. Existing algorithms are discussed in [AGS94, Section 3.2] and [CC98, Section 3.7].

5 Related Work

Other work on uncovering and resolving conflicts between encapsulation and other features of object-oriented languages has focused on rules for subclass access [Sny86, Lis87], and the visibility of the inheritance structure itself [Sny86]. We do not know of any other work directed toward resolving the conflict we outline in Section 2.

Several techniques have been developed for introducing multimethods into single-dispatch languages. Leavens and Millstein [LM98] view multiple dispatch as a dispatch on a tuple of types. This approach requires that all arguments that participate in the dynamic dispatch be grouped into a tuple class. This system, as we understand it, does not allow dispatch on classes that are not part of the receiver tuple. To add a new dynamically dispatched function, such as the `interpret` function for the code in Figure 1, we would have to create a new tuple class containing the `Exp` class, and define `interpret` there. As in the case of adding new functions via subclasses, it is not clear how multiple independent extensions would be merged in this system without creating unnecessarily complex class structures.

Boyland and Castagna's "parasitic methods" for Java [BC97] also provide dynamic dispatch based on multiple arguments. Leavens and Millstein [LM98, Section 4.2] show that this approach sometimes requires the modification of existing classes when new subclasses are defined. Thus, we do not believe it supports encapsulation as strongly as our accessory functions do.

A discussion of other techniques for multimethod dispatch can be found in [AGS94, Section 3.2] and [CC98, Section 3.7]. These papers also propose new techniques of their own.

6 Conclusions

Dynamic dispatch improves encapsulation of functions in object-oriented languages by letting the programmer extended a function to handle new types without editing the existing function definition. New operations can be provided for classes without editing the classes, as long as they make use of the interfaces defined by the classes, and do not require dynamic dispatch. However, if a new group of functions require dynamic dispatch, most object-oriented languages require the editing of the definitions of the classes on which dispatch will occur (Systems that do not require this, such as CLOS [GLS90], generally do not have strong support for data encapsulation.)

Accessory functions let a programmer create dynamically dispatched functions without editing or gaining access to the classes that participate in the dispatch. When a program is written as a set of classes an accessory functions, we can add either new classes or new functions without editing any existing code.

References

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 1996.

[AGS94] Eric Amiel, Olivier Gruber, and Eric Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *OOPSLA'94 – Object Oriented Programming Systems, Languages and Applications*, pages 244–258, October 1994. Published as SIGPLAN Notices Vol. 29, No. 10.

[App98a] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[App98b] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[BC97] John Boyland and Giuseppe Castagna. Parasitic methods: an implementation of multimethods for java. In *OOPSLA'97 – Object Oriented Programming Systems, Languages and Applications*, pages 66–76, October 1997. Published as SIGPLAN Notices Vol. 32, No. 10.

[BKK+86] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. Commonloops: merging lisp and object-oriented programming. In *OOPSLA'86 – Object Oriented Programming Systems, Languages and Applications*, pages 17–29, 1986.

[CC98] Craig Chambers and Weimin Chen. Efficient predicate dispatching. Technical Report UW-CSE-98-12-02, Dept. of Computer Science and Engineering, University of Washington, 1998.

[GLS90] Jr. Guy L. Steele. *Common Lisp: The Language (second edition)*. Digital Press, 1990.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. The MIT Press, Cambridge, Mass., 1986.

[Lis87] Barbara Liskov. Data abstraction and hierarchy (keynote address). In *OOPSLA'87 – Object Oriented Programming Systems, Languages and Applications (Addendum)*, pages 17–34, October 1987. Published as SIGPLAN Notices Vol. 23, No. 5.

[LM98] Gary T. Leavens and Todd D. Millstein. Multiple dispatch as dispatch on tuples. In *OOPSLA'98 – Object Oriented Programming Systems, Languages and Applications*, pages 374–387, October 1998. Published as SIGPLAN Notices Vol. 33, No. 10.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA'86 – Object Oriented Programming Systems, Languages and Applications*, pages 38–48, October 1986.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1997.