On the Interaction of Object-Oriented Design Patterns and Programming Languages

Technical Report CSD-TR-96-020

Gerald Baumgartner* Konstantin Läufer** Vincent F. Russo*

* Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, U.S.A.
{gb, russo}@cs.purdue.edu

** Department of Mathematical and Computer Sciences
Loyola University
Chicago, IL 60626, U.S.A.
laufer@math.luc.edu

February 29, 1996

Abstract

Design patterns are distilled from many real systems to catalog common programming practice. We have analyzed several published design patterns and looked for patterns of working around constraints of the implementation language. Some object-oriented design patterns are distorted or overly complicated because of the lack of supporting language constructs or mechanisms. We lay a groundwork of general-purpose language constructs and mechanisms that, if provided by a statically typed, object-oriented language, would better support the implementation of design patterns and, thus, benefit the construction of many real systems. In particular, our catalog of language constructs includes subtyping separate from inheritance, lexically scoped closure objects independent of classes, and multimethod dispatch. The proposed constructs and mechanisms are not radically new, but rather are adopted from a variety of languages and combined in a new, orthogonal manner. We argue that by describing design patterns in terms of the proposed constructs and mechanisms, pattern descriptions become simpler and, therefore, accessible to a larger number of language communities. Constructs and mechanisms lacking in a particular language can be implemented using *paradigmatic idioms*.

1 Introduction

Design patterns [21, 18, 37, 17] are a distillation of many real systems for the purpose of cataloging and categorizing common programming and program design practice. With the help of a design pattern catalog, a programmer can reuse proven design solutions and avoid reinvention. Many of the design patterns described in the literature, however, are influenced by the point of view of a particular programming language. Gamma et al. [21], the authors of a popular design patterns catalog, make this observation (page 4):

The choice of programming language is important because it influences one's point of view. Our patterns assume Smalltalk/C++-level language features, and that choice determines what can and cannot be implemented easily. If we assumed procedural languages, we might have included *design patterns called "Inheritance," "Encapsulation," and "Polymorphism." Similarly, some of our patterns are supported directly by the less common object-oriented languages.*

We believe the influence is two-way and that an understanding of the interactions can benefit both the pattern and the programming language communities. To understand how an analysis of patterns gives insight to the language designer, we first need to consider the influence programming languages have on patterns.

Influence of Languages on Patterns

The choice of language has a two-fold effect on a pattern collection. First, as Gamma et al. observed, some low-level patterns can be omitted if the programming language supports them directly through a language construct or mechanism. Second, some patterns are distorted or overly complicated because of the lack of a supporting language construct or mechanism.

The Singleton pattern [21] is an example of a pattern that merely implements a missing language construct. A Singleton is a class with only one instance and code that ensures that no other instances will be created. If a language allows the definition of a single object without a class, or if it offers a module construct as in Modula-3 [9], there is no need for the Singleton pattern.

For examples of patterns that work around a defect in the programming language, consider that most of the complexity of Coplien's Handle-Body and Envelope-Letter idioms [17] is due to the need to implement reference counting since C++ [26] lacks garbage collection. Explicit storage management also complicates several other idioms in Coplien's collection. In a language with a garbage-collected heap, these idioms would be greatly simplified or not needed at all.

Observe that not all design patterns are influenced by their implementation language. Some patterns, such as Producer-Consumer [6, 30], can be described independent of the programming language. The reason the Producer-Consumer pattern is language-independent is that the language constructs and mechanism it relies on (arrays or lists and functions) are basic enough that they can be found in any language.

Many design patterns are not so basic and rely on language mechanisms such as inheritance, encapsulation, and polymorphism. This does not mean that such a pattern is not generally applicable, but rather that in a non-object-oriented language the missing language constructs and mechanisms need to be implemented in the form of other patterns. The Composite pattern [21], for example, relies on the ability to use the two classes `Leaf` and `Composite` polymorphically. To implement subtype polymorphism in C, all that is needed is to maintain a dispatch table containing function pointers [39]. The Composite pattern can, therefore, be implemented in C by defining structures representing `Leaf` and `Composite` together with a function dispatch table implementation of the Polymorphism pattern. We term a pattern that implements a missing language construct or mechanism (such as Polymorphism and Singleton) a *paradigmatic idiom*.

We argue that to increase the audience, object-oriented design patterns should be described in terms of a *richer* set of object-oriented language constructs. To implement a pattern in a programming language that does not offer the full set of language constructs, it has to be combined with the proper paradigmatic idioms, as illustrated above.

Increasing the number of constructs taken for granted will simplify many patterns by eliminating distracting implementation details. Some existing patterns might even be supported by language constructs directly and, thus, become paradigmatic idioms.

Influence of Patterns on Languages

From the programming languages point of view, patterns are interesting in that they are a reflection of current programming practice. An analysis of patterns, therefore, can indicate which language constructs or mechanisms would be useful in practice and should be provided by new object-oriented languages.

Returning to the examples above, the Singleton pattern and its frequent appearance in other patterns indicates that a module construct would be a useful addition to object-oriented programming languages. Likewise, the complexity of Coplien's idioms to deal with the lack of an automatically managed heap indicates the need for garbage collection as a storage management mechanism.

For this paper, we have analyzed design patterns from several sources [21, 18, 37, 17] and looked for patterns of working around constraints of the implementation language. Based on this analysis, we catalog general-purpose language constructs and mechanisms that, if provided by a statically typed, object-oriented language, would benefit design patterns and, transitively, a large body of real systems. We do not invent radically new language features but rather propose combining language constructs and mechanisms from a variety of languages in an orthogonal manner.

The language constructs and mechanisms we address in this paper are all related to the object model of a language and its type system. In particular, we propose that ideal object-oriented languages

- separate code reuse from subtyping by making inheritance a pure code-reuse mechanism and use interface conformance to define the subtype relationship,
- include syntax for specifying lexically scoped closure objects independent of the class construct, and
- provide both single dispatch and multimethod dispatch.

By separating the subtyping mechanism from the code reuse mechanism, we can strengthen the power of both. In particular, we advocate a separate language construct for specifying interface types independent from the construct for defining implementations (the class). We also advocate a separate language mechanism for determining the conformance between classes and interface types (subtyping) independent from the mechanism for code reuse (inheritance).

In addition to objects that are instances of classes, we propose lexically scoped, classless closure objects. Such objects fulfill the roles that modules, packages, and closures play in other languages. Furthermore, this notion of objects allows the uniform treatment of a class's metaclass fields and methods as an object. By allowing abstraction over all kinds of objects with explicit interfaces, polymorphism is obtained in a way that is uniformly applicable to both objects and class instances.

Single dispatch is the appropriate mechanism for adding functionality to software in the form of new classes while multimethod dispatch is more appropriate for adding new behavior to a set of existing classes. By providing both forms of dispatch, software can be evolved in both of these ways.

Other important mechanisms and constructs, including garbage collection, synchronization constructs to support concurrency, and language mechanisms for persistence, distribution, and migration of objects are not directly related to the object model and type system of a language and, therefore, are beyond the scope of this paper.

The following catalog of language constructs and mechanisms is presented roughly in a pattern style. Each section starts by describing a common structuring problem found in patterns, followed by examples of patterns exhibiting the problem. We then present a solution in the form of a language construct or mechanism and conclude by showing how the solution simplifies or replaces existing patterns. The paper does not intend to present a complete language design; rather, code samples and illustrations of proposed language constructs are presented in pseudo-C++ syntax for illustrative purposes.

2 Explicit Interface Descriptions

It is often desirable to develop a hierarchy of interface types independent from the class hierarchy, which provides concrete implementations of these interfaces [8, 15]. With respect to design, two major problems arise when class inheritance is co-opted into implementing both the interface and implementation hierarchies for a system. First, it becomes difficult to separate logical abstractions (interfaces) from classes implementing those abstractions and, second, it becomes difficult or impossible to abstract over existing code for reuse purposes.

The problem is that most object-oriented languages provide only one abstraction mechanism: the class. Implementations of an abstract interface type must explicitly state their adherence to the interface by inheriting from a class that declares the abstract interface. The need to inherit from such *abstract classes* often constrains or forces alterations in implementation hierarchies in order to introduce new interface types.

A separate language construct for abstraction that does not rely on classes would leave classes free to be used solely for implementation specification. If the adherence of a particular class to an abstract interface type is inferred from the class specification and does not need to be explicitly coded in the class, a cleaner separation of interface and implementation can be achieved. Stating this adherence explicitly might still be useful for documentation purposes, but should not be required. A clean, language-supported separation of interface from implementation also allows the flexibility of inheritance for code reuse to be strengthened, as discussed in Section 3.

Examples

The need for explicit interface specifications is particularly evident in the Bridge, Adapter, Proxy, and Decorator patterns [21].

Bridge and Adapter

The general idea of the Bridge pattern is to support the construction of an abstraction hierarchy parallel to an implementation hierarchy and to avoid a permanent binding between the abstractions and the implementations of these abstractions.

For example, given the implementation hierarchy,

```
class Implementor {
public:
    void MethodImp1() = 0;
    void MethodImp2() = 0;
};

class ConcreteImplementorA : public Implementor;
class ConcreteImplementorB : public Implementor;
```

we might want to create new classes that delegate parts of their behavior to classes from the implementation hierarchy:

```
class Abstraction {
private:
    Implementor * imp;
public:
    void Method1() { imp->MethodImp1(); }
    void Method12() { imp->MethodImp1(); imp->MethodImp2(); }
};

class RefinedAbstraction : public Abstraction;
```

The Adapter pattern is essentially the same as the Bridge pattern. The difference is that the Bridge pattern is used in designs that separate abstractions and implementations, while in the Adapter pattern the abstraction is added *retroactively*. We identify the two patterns equating `Adaptee` with `Implementor` and `Adapter` with `Abstraction` and choose the name Bridge for both.

The Bridge pattern makes the assumption that all implementation classes are subclassed from the common abstract superclass `Implementor` used to define their interface. This can cause problems in the presence of component libraries that are provided in “binary-only” form, or where we desire to use components that are already part of a library intended for a different application.

The straightforward solution of subclassing the implementation classes from the abstract superclasses defined by the pattern fails if only header files and binaries, but no source code, are available for the two libraries since introducing a new superclass would require access to the source code of the implementation classes. The only choices remaining are to use a discriminated union in the application that uses the abstractions, to use multiple inheritance to implement a new set of leaf classes in each implementation hierarchy, or to use a hierarchy of forwarding classes. The former solution is rather inelegant, and the latter two clutter up the name space with a superfluous set of new class names. Mularz [35] makes a similar observation when discussing building wrappers to access legacy code.

Even with source code available, if the component classes are already part of another application, altering their inheritance relationships could break that application. Again, we are forced to derive new classes through multiple inheritance or to use forwarding classes.

Proxy and Decorator

A more realistic scenario for retroactive abstraction is abstracting the type of an existing class that is only given in compiled form and providing an alternate implementation of this type. If the original application was not designed with this form of reuse in mind, or if the alternative implementation uses different data structures, we end up with a similar problem as above. The Proxy and Decorator patterns illustrate this problem.

The purpose of the Proxy pattern is to provide a placeholder for an object. For example, an object on a remote machine might be represented by a proxy on the local machine. The way this is achieved in the pattern is by subclassing both the proxy class and the subject from the same abstract superclass.

```
class AbstractSubject;
class Subject : public AbstractSubject;

class Proxy : public AbstractSubject {
private:
    Subject * realSubject;
public:
    void Request() { /* ... */ realSubject->Request(); /* ... */ }
};
```

The Decorator pattern is identical to the Proxy pattern equating `Decorator` with `Proxy`, `Component` with `AbstractSubject`, and `ConcreteComponent` with `Subject`. The only difference is that while `Proxy` forwards to a concrete implementation, `Decorator` forwards to the abstraction `Component`. The Decorator pattern simply allows us to "proxy" multiple possible implementations. We identify the two patterns and use the name Proxy for both. What distinguishes Proxy from the Bridge pattern is that the proxy has the same interface as the object it stands in for (the subject) while the Bridge pattern defines separate interfaces.

The Proxy pattern would benefit from interfaces separate from classes since we would not need to introduce `AbstractSubject` as an abstract superclass. Instead, if the clients of the subject class used an interface to describe it, a proxy would simply have to implement that interface and delegate methods through a reference to the subject. The proxy and the subject would not be constrained to the same class inheritance hierarchy. This can be especially important in cases where the subject class is already in an implementation hierarchy that might provide fields to the proxy that it did not need or methods that the client does not ever need to call.

Solution

As we have alluded to, the standard paradigmatic idiom that implements an interface type hierarchy is a hierarchy of abstract classes. Achieving interface hierarchies separate from implementation hierarchies is problematic in traditional object-oriented programming languages. In statically typed languages like C++, the required interface classes are usually defined as abstract base classes [20]. Multiple inheritance (from both an implementation class and an interface) can be used to establish the needed type relationships in the implementation hierarchy.

Linking implementation and interface hierarchies together can lead to type conflicts. Consider an abstract type `Matrix` with two subtypes `NegativeDefiniteMatrix` and `OrthogonalMatrix`. Assume we wish to have several different implementations of these abstract interface types, namely `DenseMatrix`, which implements matrices as two-dimensional arrays, `SparseMatrix`, which uses lists of triples, and `PermutationMatrix`, which is implemented as a special case (subclass) of sparse matrices that takes advantage of permutation matrices having only one element in each row and column.

If we try to model these types and implementations with a single class hierarchy, we end up either duplicating code or violating the type hierarchy. While `DenseMatrix` can be made a subclass of the abstract classes `Matrix`, `NegativeDefiniteMatrix`, and `OrthogonalMatrix` by using multiple inheritance, we cannot do the same for `SparseMatrix`. Since class inheritance normally implies a subtype relationship between child and parent classes, doing so would make `PermutationMatrix`, which is a subclass of `SparseMatrix`, an indirect subclass and, therefore, a subtype, of `NegativeDefiniteMatrix`. Since permutation matrices are positive definite, this would violate the type hierarchy. The alternative of having a separate class `SparseNegativeDefiniteMatrix` is not satisfying either since it causes code replication.

Similar arguments have been given in the literature to show that the `Collection` class hierarchy of Smalltalk-80 [22] is not appropriate as a basis for subtyping. While the problem does not arise with dynamic typing, it becomes an issue when trying to make Smalltalk-80 statically typed while retaining most of its flexibility.

Explicit support for object interfaces has been introduced into some object-oriented languages. For example, interfaces are supported in Java [41] as an explicit syntactical construct. A type hierarchy can be created using interfaces, and an implementation hierarchy can be created through class (single) inheritance. However, Java interfaces are in effect only syntactic sugar for abstract superclasses since implementation classes must explicitly state which interfaces they implement (akin to inheriting from an abstract superclass). Since this type relation is passed down through the implementation hierarchy, it still can cause the difficulty described above.

The object form of the Adapter pattern [21] proposes another idiomatic pattern to solve the problem by using a forwarding class with a single instance variable that references an implementation class. This solution is workable in most object-oriented languages but introduces the added burden of having to code the forwarding explicitly along with the associated run-time overhead.

In [3, 2], a conservative extension to C++ is proposed that gives both syntactic and semantic support for separating interfaces from implementations. The `signature` construct is much like an abstract superclass in C++, or an interface in Java, except that a class's conformance to a signature is inferred by the compiler rather than having to be explicitly declared. This allows much more flexibility in the implementation hierarchies. Also, since the conformance of a class to a signature is inferred rather than explicitly declared, it would be possible to change the semantics of inheritance (*not* to imply a subtype relationship) and prevent problems like the one in the computer algebra example above. We discuss this further in Section 3.

Thus, the ideal language construct for separating interfaces from implementations would allow classes and interfaces both to be specialized through inheritance, and would support inferred subtyping. In other words, a method invocation on a variable of interface type should be redirected automatically to an instance of a class implementing the interface, and the conformance of a class to an interface should be checked at compile time.

For example, given declarations of the form

```
interface I {
    void h();
    int g(int);
};

class C {
    // ...
public:
    void h();
    int g(int);
};
```

```
class D {
    // ...
public:
    void f(char, int, float);
    int g(int);
    void h();
};

I * ip = new C;
ip->h();
ip = new D;
int j = ip->g(7);
```

both class C and class D conform to the interface I and, therefore, the assignments are valid. Invocations should automatically be redirected to the proper method implementing the method.

Stating of the conformance to an interface type explicitly might still be useful for documentation and compile time checking of completeness. For example, given

```
class E : implements I {
    // ...
public:
    void h();
    int g(int);
};
```

a compiler could check that E does indeed implement all the methods defined in I, as it is documented to do so. However, for the reasons described above this should not be mandatory.

Another useful feature would be to support method renaming, perhaps with some form of cast notation. For example, in the Bridge pattern [21], one of the libraries may have chosen different method names.

```
interface Graphic {
public:
    void draw();
    void move(int, int);
};

class Graphic1 {
public:
    void render();
    void move(int, int);
};

Graphic * g = (rename render to draw) new Graphic1;
```

Renaming is useful since it can obviate the need for the Adapter pattern in cases when the implementation class simply chose a different name for the method in question than the interface. When a simple renaming is not sufficient, class or object Adapters [21] can still be used.

Uses

The need for type abstraction separate from implementation is pervasive throughout numerous patterns (in particular the Structural Patterns in Gamma et al. [21]). The general observation can be made that any abstract class that contains no code should be replaced by an interface definition instead. For example, in the Abstract Factory pattern, the `AbstractProduct` classes could all be interfaces as would be the `AbstractFactory` class itself; in the Builder pattern, the `Builder` class would be an interface; in the Bridge pattern, the `Implementor` class would be an interface; in the Decorator pattern, the `Component` class should be an interface; etc. Using interfaces instead of classes also better documents the uses of the abstractions.

3 Improved Code-Reuse Mechanisms

The construction of a class can frequently be simplified by reusing code from existing classes. Inheritance is the characteristic reuse mechanism provided by object-oriented languages. In fact, the possibility for code reuse offered by inheritance is one of the reasons for the popularity of object-oriented languages.

Code reuse by inheritance can be grouped into four different patterns: specializing an existing class, filling in missing pieces in a framework, composing a class from existing classes, and creating a new class from *pieces* of existing classes.[1]

Specialization For specializing an existing class, a subclass can add new fields and methods or override existing methods. In C++ terminology, the mechanisms used for this purpose are *public inheritance* and redefinition of *virtual* member functions [26]. In Smalltalk-80, all inheritance is public and all methods are virtual [22].

Template Method A *framework* is a skeleton of an application or algorithm that implements the control structure of a class of related applications or algorithms. The framework defines an interface for pieces to be filled in to create a specific instance of such an application or algorithm. Aside from overriding virtual methods, a mechanism commonly used in frameworks is an *abstract* method. An abstract method is a method that can be called by other methods of a framework but is not implemented in the framework class itself. An implementation has to be provided by a subclass. The Smalltalk-80 term for this mechanism is *subclass responsibility.* The style of refining certain steps of an algorithm is described in the Template Method pattern in [21].

Mixin In the mixin programming style [34], inheritance is used to add functional components to a class. The same effect could be achieved by making the components fields of the class. However, making the methods of a component available to clients of the class would require writing forwarding methods. Mixin inheritance simplifies the reuse of the components whose public methods should be made available. The standard mechanisms used for this style of code reuse is *multiple inheritance* of (mostly) *non-virtual* methods. The import statement in Modula-3 [9] serves a similar purpose, except that imported functions are not automatically re-exported.

Theft In some cases, a class might only inherit part of its superclass(es) or rename inherited methods purely for reusing existing code without intending any semantic relationship between the superclass(es) and the subclass. In C++, the mechanism used for this purpose is *private* inheritance. In Modula-3, it is possible to import functions from a module selectively.

Since inheritance in most object-oriented languages also defines a subtype relationship, the possibilities for code reuse have to be restricted such that the subtype relationship is not broken. In particular, the definition of public inheritance in C++ is slightly limited, which limits the applicability of the Specialization pattern. Since private inheritance does not define a subtype relationship, the Theft pattern is often avoided.

[1]To our knowledge, the only form of code reuse that has been published as a pattern is Template Method. The other forms of code reuse seem to be considered directly supported by a language's inheritance mechanism. Since inheritance is assumed to be a basic language mechanism [21], these code reuse patterns are not found in object-oriented design pattern collections. We suggest to include all four code reuse patterns in pattern collections as a guide for programmers to use inheritance properly.

Examples

Since we have interface conformance for defining a subtype relationship, we can strengthen inheritance for code-reuse purposes by breaking the subtype relationship it usually defines. This makes the Specialization pattern more flexible, as the following example demonstrates. Not having to use inheritance for defining a subtype relationship also allows programmers to use the Theft pattern more often.

To motivate the limitation of public inheritance, suppose we have an implementation of coordinate points and would like to extend them to color points (this example is based on examples from [16, 15]). Class `ColorPoint` inherits class `Point`, adds color support, and redefines equality to compare the color as well.

```
class Point {
protected:
    int x, y;
public:
    virtual double dist() { return sqrt (x * x + y * y); }
    virtual int equal(Point & p) { return (x == p.x) && (y == p.y); }
    virtual Point & closer(Point & p) { return (p.dist() < dist()) ? p : *this; }
    // ...
};

class ColorPoint : public Point {
protected:
    Color c;
public:
    virtual int equal(ColorPoint & p) { return (x == p.x) && (y == p.y) && (c == p.c); }
    // ...
};

ColorPoint & p = // ...
ColorPoint & q = // ...
int i = p.equal(q.closer(p));            // type error
```

The last line results in a compile-time type error, since `equal()` expects its argument to be of type `ColorPoint&` but the return type of `closer()` is of type `Point&`. By comparing whether `q` or `p` is closer to the origin, we have lost the information that both are color points.

What we would like is a form of inheritance that results in `ColorPoint::closer()` having the type

```
virtual ColorPoint & closer(ColorPoint &);
```

That is, we would like occurrences of type `Point` in an parameter type or return type to be changed to `ColorPoint` when a method is inherited. C++ does not allow this since changing parameter types in this fashion would violate the contravariance rule required for subtyping.

Solution

If subtyping is achieved by testing the conformance of a class to an interface, we do not need inheritance for subtyping purposes. Assume class `D` is a subclass of `C` defined by C++-style public inheritance and overriding of virtual methods. Since `D` conforms to any interface that `C` conforms to, an instance of `D` can be assigned to an interface reference wherever an instance of `C` was assigned before. The run-time dispatch for method calls through an interface reference gives the same polymorphism as a method dispatch. That is, interface conformance subsumes any subtype relationship defined by inheritance in which only virtual methods are overridden. The only case where interface conformance cannot be used instead of class inheritance to define a subtype relationship is when the subclass overrides a *non-virtual* method of the superclass. This use of inheritance, however, is usually considered a programming error [14].

If inheritance does not define a subtype relationship, it can be made more flexible for code reuse purposes in two ways. By introducing the notion of `selftype`, it is possible to override methods covariantly. Furthermore, private inheritance can be used more frequently for code reuse.

The type `selftype` refers to the receiver's class type. If a declaration involving `selftype` is inherited, `selftype` is rebound to refer to the subclass. For example, in

```
class Point {
protected:
    int x, y;
public:
    virtual double dist() { return sqrt(x * x + y * y); }
    virtual int equal(Point & p) { return (x == p.x) && (y == p.y); }
    virtual selftype & closer(selftype & p) { return (p.dist() < dist()) ? p : *this; }
    // ...
};
```

`selftype&` is synonymous with `Point&`. In class `ColorPoint`, inheriting class `Point` results in the parameter and return types of `closer` to be synonymous with `ColorPoint&`. The call to `equal` in

```
ColorPoint & p = // ...
ColorPoint & q = // ...
int i = p.equal(q.closer(p));
```

now type-checks correctly. In interface inheritance, `selftype` refers to the interface type.

Using `selftype` together with virtual methods gives the programmer more flexibility in specializing an existing class at the expense of losing the subtype relationship. For some applications, however, this is the desired behavior. If a subtype relationship is needed, it can be achieved by declaring parameter and return types to be of an interface instead of a class type.

For implementing frameworks using the Template Method pattern [21], a subtype relationship between the abstract class and the concrete class is not required. The concrete class inherits the code of the abstract class and fills in the missing methods. The result is that the concrete class conforms to any interface the abstract class conforms to.

Since both for specializing an existing class and for defining a framework virtual methods are more common, we suggest to use syntax similar to Java's [41]: methods are virtual by default unless they are explicitly declared `final`. Also for specifying abstract methods, a keyword, such as Java's `abstract` keyword, could be used.

To support the mixin programming style, we suggest that object-oriented languages support an import mechanism related to that found in Modula-3. Importing a class works similar to inheritance, except that all methods of the imported class are treated as non-virtual methods, whether they were declared final or not. In other words, importing a class `C` is operationally the same as defining a field of type `C` and writing forwarding methods for all of `C`'s public and protected methods.

To support unstructured code reuse, i.e., the Theft pattern, we can allow renaming of inherited methods or only inheriting part of a superclass. For example, with a syntax such as

```
class CDE :
    private C only f1, f2;
    public D except f;
    public E only f, g rename g to E_g;
{
public:
    // ...
};
```

we could inherit the methods `C::f1` and `C::f2` without re-exporting them, inherit everything of class `D` except the method `D::f`, and inherit `f` and `g` from class `E` while renaming `E::g` to `E_g` to avoid a name conflict with `D::g`. The same syntax can be used for selectively importing a class. Similar syntax has been proposed in [33].

Note that in the above example, the interface of class `CDE` will conform to the interface of class `D` if `D::f` and `E::f` have the same type. Interface conformance, therefore, enables us to define subtype relationships that could not be achieved with inheritance.

If classes `D` and `E` in the example above both inherit from class `A`, an instance of class `CDE` would have two copies of the fields of class `A`. For classes `D` and `E` to share the same `A` part, in C++ they would need to use *virtual inheritance* from class `A`. We suggest to use *sharing constrains* similar to those in ML [32, 31] instead. In the above example, we could specify the sharing constraint

```
sharing D::A == E::A;
```

in class `CDE`. The advantage over virtual inheritance is that its use does not have to be anticipated when defining classes `D` and `E`.

Although the proposed forms of inheritance and import do not preclude a traditional dispatch table implementation of inheritance, they can be implemented more efficiently and simpler than C++-style inheritance. Since these forms of inheritance do not allow class instances to be cast up and down the inheritance hierarchy, and since non-virtual methods cannot be overridden, a compiler can simply *copy the code* of inherited classes and recompile it. The advantages of such an implementation are fourfold. Dispatch tables are not needed anymore. Object migration in a distributed environment is easier to implement; all the code that needs to be shipped is readily available without walking the class hierarchy. Generating code for *path expressions* [7] or similar mechanisms for synchronizing concurrent method calls becomes simpler since a method might have different synchronization constraints depending on the class of the receiver. Finally, the semantics are cleaner than with the complicated table lookup in C++.

An implementation of actually copying and recompiling source code has three disadvantages. First, it is expensive at compile time since inherited methods are repeatedly compiled. Second, it introduces additional space overhead since the multiple compiled versions of each method are almost identical. Finally, it prohibits inheriting classes for which only header files and object files are provided.

By copying byte code or object code, however, we still have all the advantages compared to an implementation using dispatch tables without the overhead of recompilation and the necessity of having source code available. An advantage of copying source code or byte code rather than object code is that it enables some compiler optimizations such as eliminating unused methods and fields or more aggressive inlining.

Uses

In most patterns in Gamma et al. [21], inheritance is used to define a subtype relationship by inheriting an abstract superclass. In these cases, the proper language construct to use would be an interface.

The most common form of code reuse found in design patterns is a framework. The Template Method pattern explains how to build a framework. Further frameworks are found in Factory Method, Singleton, Adapter, Mediator, and Observer. The `Component` and `Decorator` classes in the Composite and Decorator patterns also represent frameworks.

The only cases in Gamma et al. of using inheritance to specialize an existing class are in the Decorator and Mediator patterns. The refined abstractions in the Bridge pattern would be implemented as specializations of an interface.

Only the class adaptor in the Adaptor pattern uses private inheritance as a short-hand for composition. The object adaptor is structurally much cleaner.

In all cases where inheritance is used both for type abstraction and for code reuse, using interfaces for the abstractions simplifies the inheritance structure and makes it unnecessary for inheritance to define a subtype relationship.

4 Objects Without Classes

Many applications contain only a single copy of certain components. For example, a compiler contains only one parser. Even though there might be multiple network interfaces, an operating system contains only one TCP/IP stack. In the Abstract Factory pattern [21], there is only one product factory.

In languages that only provide classes for constructing objects, it is necessary to ensure that certain classes get instantiated at most once. This is the purpose of the Singleton pattern [21].

In addition to constructing singleton software components, there is a need for packaging components for program delivery. Some object-oriented languages provide constructs specifically for this purpose. Examples are namespaces in C++ [26] or packages in Java [41]. Shaw [40] lists module as a pattern for component packaging.

Namespaces in C++ and packages in Java provide rudimentary support for packaging but are not flexible enough to be used as singleton components. They cannot be parameterized, passed to functions, or specialized through inheritance.

Examples

The Singleton Pattern

Instead of defining a singleton object, using the Singleton pattern [21] allows the programmer to define a class that has only one instance. To ensure that no more than one instance can be created, it is necessary to intercept requests to create new objects. Using a named class allows the singleton object to be passed to functions or to be specialized in subclasses.

In C++, a typical implementation of the Singleton pattern is (from [21]):

```
class Singleton {
public:
    static Singleton * Instance();
protected:
    Singleton();
private:
    static Singleton * _instance;
};
```

By not making the constructor public, the only way for clients to create an instance is through the method `Instance()`:

```
Singleton * Singleton::_instance = 0;

Singleton * Singleton::Instance() {
    if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}
```

Making `Instance()` a class method and storing the single instance in a class field ensures that only one instance can be created. Accessing the object indirectly through the `Instance()` method has the disadvantage that it precludes static resolution of method calls to the singleton object.

Component Packaging

The Abstract Factory pattern [21] provides an interface for creating families of products. Suppose we need to create scrollbars and windows of either the Presentation Manager or the Motif product families. A skeleton of an implementation for these products might look as follows:

```
interface AbstractScrollBar;
class PMScrollBar : implements AbstractScrollBar;
class MotifScrollBar : implements AbstractScrollBar;

interface AbstractWindow;
class PMWindow : implements AbstractWindow;
class MotifWindow : implements AbstractWindow;
```

A concrete factory, would create products from one of the two families, e.g., `MotifScrollBar`s and `Motif-Window`s. `AbstractScrollBar` and `AbstractWindow` define common interfaces for both product families.

If an application has to deal with many different product families, the naming conventions for the different types of products quickly become unmanageable. What is needed is a packaging mechanism that introduces a new scope and allows grouping, e.g., `ScrollBar` and `Window` into a `Motif` package.

Solution

The bookkeeping effort of explicitly managing the creation of singleton objects can be avoided if the language provides syntax for constructing objects without requiring a class to be declared. The same syntax would allow packaging components.

Singleton Objects

A simple language design solution to support the singleton pattern would be to introduce an *object construct* with the same syntax as a class construct but without constructors or destructor. In pseudo-C++ syntax, the declaration

```
object Singleton1 {
public:
    // public methods
private:
    // private data
};
```

could be used to define and initialize the constant `Singleton1`.

With such syntax, it is guaranteed that only one object will be created and, therefore, references to it can be statically resolved. In addition, unlike namespaces in C++ or packages in Java, these objects can be extended by inheritance.

```
object Singleton2 : public Singleton1 {
public:
    // additional public methods
private:
    // additional private data
};
```

To abstract over singleton objects, we can use the interface construct described earlier. It is not necessary to provide separate interface constructs for objects and classes. Given an interface type `T`, a reference of type `T&` can then be assigned either a singleton object that provides all the methods specified in `T` or an instance of a conforming class:

```
interface T {
    // ...
};

T & p = Singleton1;
T * q = new C;
```

Interface types also allow us to define polymorphic functions that can take as argument any singleton conforming to the interface.

```
int f(T &);
int i = f(Singleton1);
int j = f(Singleton2);
```

If clients require access to only one of several subobjects of `Singleton1`, all objects could be encapsulated in an outer object that exports a reference to only one of the subobjects.

Packages

The object construct described above is related to *module* constructs as found in Modula-3 [9] or ML [32, 31]. Unlike modules in these languages, objects are first-class values, i.e., they can be passed to and returned from functions or assigned to variables. The advantage of modules is that they allow packaging and exporting of types in addition to variables and functions. Modules can therefore be considered *higher-order* objects.

Both Modula-3 and ML provide different constructs for module interfaces (called `INTERFACE` in Modula-3 and `signature` in ML) and for the implementation of a module (called `MODULE` in Modula-3 and `structure` in ML). The components exported from the module are those listed in the interface.

In C++-style syntax, the interface and implementation parts could be combined into one construct and labeled with `public` and `private`, respectively. Syntactically, a package then looks the same as a singleton object:

```
object Motif {
public:
    class ScrollBar : implements AbstractScrollBar { /* ... */ };
    class Window : implements AbstractWindow { /* ... */ };
    // other public types, data, and methods
private:
    // private types, data, and methods
};
```

The main difference is that packages also export types, i.e., packages are higher-order objects. (We use the terms "package" and "singleton object" to distinguish between the two uses of the `object` construct.) A package interface can be defined using the same interface construct as for classes.

In addition to modules, ML provides parameterized modules called `functors`. In C++-like syntax, a parameterized package could be defined as a template:

```
template <T1 ComponentClass, T2 Package1> object Package2 : implements T3 {
    // ...
};

Package2<C, P> pkg;     // C must conform to T1, P must conform to T2
```

where `T1` and `T2` are interface types constraining the possible argument classes and packages, respectively, and `T3` is the interface of the resulting package. For type-checking template parameters and results, package interfaces can also specify the types that a package must export. Such higher-order interfaces are only useful for type checking templates.

To allow information hiding, both Modula-3 and ML allow exported types to be *opaque.* An opaque type is a type whose name is declared in the module interface but whose definition is only given in a module implementation. Clients of a module, which only get access to the name of an opaque type, can declare variables of the type, initialize them by calling a function of the module, and pass them to other functions. Clients cannot inspect or alter values of an opaque type, except through functions exported by the module.

The style of programming with modules and opaque types is known as the *abstract data type (ADT) style.* The role of opaque types in this programming style corresponds to private fields in an object-oriented style. In this pseudo-C++ syntax, an opaque type would be a type whose *name* is declared `public` but whose definition is `private`.

Sometimes it is useful for packages to be passed to a function or method by reference (for an example, see Section 6). However, exporting types complicates the assignment to an object reference. Given a package interface `T`, suppose the function '`int f(T& P)`' takes as argument a package that exports the non-opaque type `t`. Inside the function, it would then be possible to declare a variable of type `P.t` and, since `P.t` is not opaque, it would be possible to inspect a value of this type. However, the exact type may not be known at compile time since it can depend on the actual package being passed. Allowing a function to inspect values of a non-opaque type would, therefore, require types to be first-class values and some type checking to be done at run-time.

The solution for keeping the language statically typed has traditionally been to make modules second class, i.e., to disallow module references. Neither ML nor Modula-3 allows passing modules to functions.

The solution for getting the best of both worlds, exported types *and* first-class packages, is to consider exported types opaque when a package is accessed through an object reference. Such a solution was proposed as an object-based extension of ML modules by Mitchell et al. [33] and for supporting separate compilation of ML modules by Harper and Lillibridge [23].

As with classes, it is often useful to define new objects by reusing the code of existing objects, as shown in the definition of `Singleton2` above. Since all of the code reuse patterns in Section 3 apply to objects as well, all forms of inheritance and import can be used for defining objects. The only exception is `selftype`, since singleton objects and packages do not have an implementation type. An object containing an `abstract` method would need to be considered an abstract object and could only be used as a parent for object inheritance.

In summary, to benefit packaging and pattern implementations, we suggest adding to object-oriented languages an ML-style module system consisting of (parameterized) packages and package interfaces. As with classes and class interfaces, the conformance of a package to a package interface could be tested either structurally or by name. A package that does not export types, i.e., a singleton object, is a first-class value and can be assigned to references or passed to functions. A package that does export types is a higher-order object. When passing such as package to functions, all exported types become opaque to make the package first-class. Like classes, packages or singleton objects can be refined by inheritance.

The proposed language construct would replace the Singleton pattern and provide superior facilities for packaging components as compared to C++'s namespaces or Java's packages.

Uses

In a language with singleton objects as described above, the Singleton pattern is no longer needed. An extended Singleton pattern that allows a variable number of instances can be modeled by controlling the instantiation of parameterized objects. For languages without classless objects, the Singleton pattern would be the necessary paradigmatic idiom.

Any pattern that uses the Singleton pattern can be expressed directly using the `object` construct. In the Abstract Factory pattern [21], the abstract factory and the abstract product become interfaces, and the concrete factory becomes an object. Only the concrete products remain classes. Similarly in the Builder pattern [21], builder and concrete builder become an interface and an object, respectively.

The purpose of the Facade pattern [21] is to package software components. Since usually only one facade object is required, it could be implemented as a package. Similarly, a Mediator [21] would typically be a package.

Both the Strategy and the Visitor patterns [21] package only methods. They do not define any new data structure. Concrete strategies and concrete visitors would therefore be singleton objects, with the abstract classes `Strategy` and `Visitor` being replaced by interfaces. Similarly, each concrete state in the State pattern [21] would be a singleton object.

5 Lexically Scoped Closure Objects

Some programming situations call for a *(lexical) closure* mechanism for creating behavior on the fly that can be invoked at a later time but has access to the lexical environment current when this behavior was created. Common situations that could benefit from a closure mechanism are the parameterization of an object by behavior, the state change of an object from one behavior to another, and the creation of new behavior by partial application of existing behavior.

Many statically typed object-oriented languages such as C++ [20] or Java [41] neither allow behavior to be created on the fly, nor do they give functions or objects access to the surrounding local environment. By contrast, Smalltalk-80 [22] provides *blocks* as a limited form of closures.

Examples

Specialized behavioral patterns such as Command, Iterator, State, and Strategy [21] are workarounds for missing language support for lexical closures. The abundance of such patterns has probably emerged due to a lack of closure support in many object-oriented languages and a failure to identify closures as the underlying generalized mechanism. We first illustrate the need for lexical closures by examining the Iterator pattern. We then argue that the remaining three patterns can be unified under the concept of behavior encapsulated in a closure object that has access to the lexical environment current when the object was created.

The Iterator Pattern

The purpose of the Iterator pattern is to support efficient iteration over a collection without exposing its internal structure. This pattern allows a collection to be traversed in different ways without requiring the collection to provide a method for each kind of traversal. Furthermore, the Iterator pattern supports multiple simultaneous traversals over the same collection.

The Iterator pattern actually describes an *external* iterator, i.e., the control flow for the traversal is outside of the iterator or container operations. By contrast, an *internal* iterator is usually provided as a method of the container class that takes a visitation function as its argument. For example, the `do:` method in the Smalltalk-80 collection class hierarchy [22] is an internal iterator parameterized by a block.

The following example shows a typical application of the Iterator pattern for multiple traversal of a container in C++. Each of the two iterators keeps track of one position in the traversal. The `ListIterator` class provides operations for initializing the traversal, accessing the current item, going to the next element, and checking whether more items follow. The `ListIterator` class is a friend of the `List` class to allow direct access to the representation of the `List` class.

```
template <class Item> class List {
public:
    // ...
    friend class ListIterator<Item>;
};

template <class Item> class ListIterator {
public:
    void First();
    void Next();
    bool IsDone();
    Item & CurrentItem();
    // ...
};

List<int> aList;
// ...
```

```
ListIterator<int> i(aList), j(aList);
for (i.First(); ! i.IsDone(); i.Next()) {
    for (j.First(); ! j.IsDone(); j.Next())
        cout << setw(8) << i.CurrentItem() * j.CurrentItem();
    cout << endl;
}
```

This pattern has a number of weaknesses. First, encapsulation is sacrificed for efficiency reasons, since the iterator class is a friend of the collection class. Second, the user is required to write his or her own control structure instead of using a method that already encapsulates the control flow appropriate for the traversal. This means that the user must strictly follow a protocol for the order in which the iterator methods are invoked although this protocol cannot be expressed in the interface of the iterator. For example, invoking `Next` after `IsDone` returns true is probably undefined. Finally, robustness becomes an issue with external iterators because the user can change the container by adding or deleting elements during iteration [27].

Internal iterators do not have these problems, but are hard to write in languages like C++ because of the lack of lexical scoping. External iterators are mostly a workaround for the lack of internal iterators in such languages [1]. The only other advantage of external iterators is that they allow the user to control the progress of the iteration. This is useful in situations in which it is not desirable to traverse an entire collection at once (see also the pairwise iteration example in Section 7).

The State, Strategy, and Command Patterns

The purpose of the State, Strategy, and Command patterns is to encapsulate behavior in an object.

Specifically, the State and Strategy patterns support configuring objects with one of several possible behaviors. Both patterns consist of a client class called `Context` and a behavioral class called `State` or `Strategy`, respectively. The only difference between `State` and `Strategy` is that `State` allows dynamic configuration, while `Strategy` usually provides static configuration. There is also a minor stylistic difference in the C++ code given for the two patterns [21]. In the `State` pattern, the `Context` class does not use the `State` class in its public interface. In the `Strategy` pattern, by contrast, the constructor of the `Context` class is explicitly parameterized by a `Strategy` class or object. Given the behavioral and structural similarities, we identify the two patterns and choose the name Strategy for both.

The Command pattern applies the Strategy pattern to encapsulate a request with its receiver. The pattern consists of a client class called `Invoker` and a behavioral class called `Command`.

In all three patterns, the behavioral classes have subclasses `ConcreteState`, `ConcreteStrategy`, and `ConcreteCommand`, respectively.

Gamma et al. do not establish any relationship between the three patterns other than relating State and Strategy to the Flyweight pattern [21]. Zimmer recognizes the commonalities between these behavioral patterns and tries to capture them in the Objectifier pattern [42], which is not significantly different from the Strategy pattern.

The following example (adopted from [21]) illustrates the Strategy pattern. A composition maintains a collection of textual and graphical components of a document. Upon creation, a composition can be parameterized with the desired layout strategy to be invoked in the `Layout()` method.

```
class Composition {
public:
    Composition(Compositor *);
    void Layout();
private:
    Compositor * _compositor;
    Component * _components;
};

void Composition::Layout() {
        CompositionData theCompData;
        // ...
        _compositor->Compose(theCompData);
        // ...
    }
```

The `Compositor` interface describes layout strategies that determine how the components of a document should be arranged into lines. A `SimpleCompositor` looks at components one line at a time to decide where line breaks should go. An `ArrayCompositor` breaks the components into lines each containing a fixed number of components.

```
interface Compositor {
    int Compose(CompositionData & data);
    // ...
};

class SimpleCompositor : implements Compositor { /* ... */ };
class ArrayCompositor : implements Compositor { /* ... */ };
```

Finally, when a new document is created, the desired layout strategy is passed as an argument.

```
Composition * quick = new Composition(new SimpleCompositor);
Composition * table = new Composition(new ArrayCompositor);
```

In this example, implementations of the `Compositor` interface are used as behavioral arguments, that is, as a replacement for closures, which are not directly supported by languages such as C++.

Solution

The problems addressed by the patterns discussed in this section and other specialized behavioral patterns are naturally solved by introducing a closure mechanism for creating new behavior that captures the environment current at the time this behavior was created. Such behavior comes either in the form of a single function or an entire object bundling several methods. Functions or objects can then be defined in any scope and use identifiers defined in the current environment, which consists of definitions in outer scopes. The key idea is that the function or object might be passed around and used elsewhere, but retains access to portions of the current environment. This kind of mechanism is known as *(lexical) closure* and is a standard feature of functional or applicative languages and of some object-oriented languages, including Smalltalk-80 [22], CLOS [4, 25, 36], and Cecil [11, 12].

Coplien [17] and Läufer [29] developed a paradigmatic idiom that simulates closures in C++ using classes. Their approaches fall short of true lexical closures in that closures cannot be anonymous and must capture explicitly the portions of the environment they use. Breuel [5] describes an extension of C++ that supports efficient named lexically scoped functions. We propose further generalizing those ideas by introducing lexically scoped closure objects.

We use the object construct previously introduced in Section 4 to express closure objects. A closure object can be named or anonymous. A closure object can be created on the fly and captures the surrounding lexical environment current at the time the object was created. A closure object is allowed to escape the lexical scope in which it was created and takes the captured environment with it. As in most functional languages, supporting closures requires allocating on or moving to the heap the activation records of any functions or methods that allow a closure to escape their scope.

The following example includes an interface `Counter` for simple counter objects and a function `MakeCounter` that creates counter objects with a given initial setting. When a counter object is created, it captures the variable `value` from the environment current at that time.

```
interface Counter {
    void Next();
    int Current();
};

Counter & MakeCounter(int value) {
    return object {
        void Next() { value++; }
        int Current() { return value; }
    };
}

Counter & from100 = MakeCounter(100);
from100.Count();
cout << from100.Current();      // prints "101"
```

We observe that a function closure is merely a closure object with a single function call method (e.g., `operator()` in C++). It might be useful to allow function notation as *syntactic sugar* for this common case. The next example shows a counter function that increases the counter each time the function is invoked.

```
typedef int (* Counter) ();

Counter MakeCounter(int value) {
    return int (*) () { return value++; };
}

Counter from100 = MakeCounter(100);
cout << from100();              // prints "100"
cout << from100();              // prints "101"
```

The following is a version of the multiple traversal example that uses internal iterators and closures instead of external iterators. It is usually hard to perform nested iterations over the same collection using internal iterators. Using lexical closures, such iterations become easy and natural. The key observation is that we can pass to the inner iteration a closure that has access to the state of the outer iteration.

```
template <class Item> interface Action {
    void Apply(Item i);
};

template <class Item> class List {
public:
    void ForEach(Action<Item> & a);
    // ...
};

List<int> aList;

aList.ForEach(
    object {
        void Apply(int i) {
            aList.ForEach(
                object {
                    void Apply(int j) { cout << setw(8) << i * j; }
                }
            )
            cout << endl;
        }
    }
)
```

Using function notation, the previous example can be rewritten as

```
aList.ForEach(
    void (*) (int i) {
        aList.ForEach(void (*) (int j) { cout << setw(8) << i * j; })
        cout << endl;
    }
)
```

Similar to the iterator example above, many applications of the Strategy pattern and its variations can be expressed more naturally by creating a function that invokes a method from an object defined in the current environment. The function itself can be invoked from elsewhere.

```
MyClass * receiver = new MyClass;
// ...
void aCommand() { receiver->MyAction(); }
// ...
otherFunction(aCommand);
```

Uses

Workarounds for the lack of lexical closures in a language are as pervasive [17, 29] as attempts to introduce them into languages that lack them [5, 19]. Closures and closure objects are useful for any type of behavioral parameterization, including callbacks in event-driven systems and parameters to applicative versions of iterators as used in functional languages [28].

6 Metaclass Objects

Many patterns rely on the ability either to abstract over classes or to parameterize an operation based on an object's class. For example, it would be useful to allow object construction to be parameterized by a class chosen at run time. It is also often desirable to abstract over methods that operate on the class itself rather than on an instance of the class. An example of such a method is a method accessing a class instantiation count.

To achieve abstraction over classes, we need an object representing the class at run time. Such *metaclass objects* would have fields and methods associated with the class, such as instantiation counts or constructors. We could then abstract over metaclass objects with regular interfaces, treat classes as values, and even allow parameterization by a class.

Examples

Abstract Factory and Factory Method

The Abstract Factory pattern [21] provides an interface for creating families of products without specifying their concrete classes. Suppose we want to configure an application with either the Presentation Manager or the Motif look-and-feel. When creating widgets such as scrollbars or windows, the application should not hard-code the names of the widget classes.

Given the following interfaces and classes,

```
interface AbstractScrollBar;
class PMScrollBar : implements AbstractScrollBar;
class MotifScrollBar : implements AbstractScrollBar;

interface AbstractWindow;
class PMWindow : implements AbstractWindow;
class MotifWindow : implements AbstractWindow;
```

we need to separate from the application the code for instantiating the classes. The application only knows that the generated objects conform to the interfaces `AbstractScrollBar` and `AbstractWindow`, respectively.

The solution suggested by the Abstract Factory pattern is to introduce an interface for factories creating the widgets.

```
interface WidgetFactory {
    AbstractScrollBar & CreateScrollBar ();
    AbstractWindow & CreateWindow ();
};
```

In C++, `WidgetFactory` would be defined as an abstract class. For each product family, a concrete factory class is needed to generate the proper widgets:

```
class PMWidgetFactory : implements WidgetFactory {
public:
    AbstractScrollBar & CreateScrollBar () { return new PMScrollBar; }
    AbstractWindow & CreateWindow () { return new PMWindow; }
};

class MotifWidgetFactory : implements WidgetFactory {
public:
    AbstractScrollBar & CreateScrollBar () { return new MotifScrollBar; }
    AbstractWindow & CreateWindow () { return new MotifWindow; }
};
```

At startup time, one of the concrete factories is selected. The application refers to the factory through a reference of type `WidgetFactory`.

The Factory Method pattern [21] is similar to the Abstract Factory pattern but refers to only one creation method. In the above example, both `CreateScrollBar()` and `CreateWindow()` are factory methods. The structure of the Factory Method pattern is often to make the application a framework class, called `Creator`, with the factory method as an abstract method that needs to be supplied by a subclass `ConcreteCreator`.

The disadvantage of both patterns is that the different product families need to be mirrored by different concrete factories and concrete creators, respectively. With a run-time representation of the product classes, we could parameterize both the abstract factory and the creator, with the product class(es) to be instantiated.

Solution

Smalltalk-80 [22] supports classes as objects directly and, as discussed in Gamma et al. [21], such metaclasses greatly simplify cases like the one described above. C++ only supports a limited form of metaclass by allowing fields and variables to be declared `static`. For example,

```
class A {
private:
    static int i;
public:
    static int geti() { return i; }
    A() { /* ... */ }
    // ...
};
```

declares that all instances of `A` share a common field `i` and the method `geti()`. Such metaclass data is not encapsulated in an object but rather accessed using the scope resolution syntax. In this example, `i` and `geti()` are accessed as `C::i` and `C::geti()`, respectively.

The problem with static members as an approximation of metaclasses is that C++ does not allow the use of classes as values. If the static parts were encapsulated in an object, we could assign metaclass objects to references, invoke metaclass methods through these references and, most importantly, abstract over metaclass objects using interfaces.

Ideally, any syntax for declaring metaclasses should attempt to group the methods and fields of the metaclass object and the methods and fields of instances of the class into one construct in order to localize class documentation and simplify maintenance. A simple C++-like syntax for merging metaclass definitions and instance definitions into one construct might be

```
class A {
meta private:
    int i;

meta public:
    A & new();
    A & new(int i, String s);
    int geti() { return i; }

public:
    int f();
    void g(int);
};
```

This construct would simultaneously define both the metaclass object and the implementation type of class instances.

Note that the object constructor becomes the method `new()` of the generated metaclass object. Making the constructor a method of the metaclass object with a well-known name allows us to abstract over object creation, as needed for the Abstract Factory and Template Method patterns, and to statically type-check this abstraction.

Abstracting over metaclass objects can be accomplished with interfaces that name the metaclass methods but not the instance methods. This abstraction is unambiguous. For example, while

```
interface instanceA {
    int f();
    void g(int);
};
```

abstracts over the instances of class `A` above,

```
interface metaA {
    instanceA & new();
    instanceA & new(int, String);
};
```

abstracts over its metaclass object. Note that in this interface the return type of `new` is `instanceA` and not `A`. This is necessary to enforce the encapsulation provided by class `A` and to allow the `metaA` interface type to abstract over other implementations of the same interface as well.

Consider again the Abstract Factory pattern. Since the `ConcreteFactory` classes do nothing but create instances of products from the proper product families, it would be possible to replace them completely with an `AbstractFactory` class parameterized by the product family's classes. With a statically type-checked metaclass object (as with the object interfaces discussed in Section 4), it is also possible to check that only the correct `Product` classes are created by methods of `AbstractFactory`.

For example, we can define interfaces for the products,

```
interface AbstractScrollBar {
    void handleClick();
};
```

```
interface AbstractWindow {
    void move(int x, int y);
};
```

and then define interfaces for classes that generate objects of types `AbstractScrollBar` and `AbstractWindow`, respectively:

```
interface ScrollBarMaker {
    AbstractScrollBar & new(int x, int y, String label);
};

interface WindowMaker {
    AbstractWindow & new(int x, int y, int w, int h);
}
```

Implementations of the product classes can either implicitly or explicitly implement the product and meta-class interfaces, as in

```
class PMScrollBar {
meta public:
    PMScrollBar & new(int x, int y, String label) { /* ... */ }
public:
    void handleClick() { /* ... */ }
};

class PMWindow : implements AbstractWindow {
meta public:
    PMWindow & new(int x, int y, int w, int h) { /* ... */ }
public:
    void move(int x, int y) { /* ... */ }
};

class MotifScrollBar : implements AbstractScrollBar {
meta public:
    MotifScrollBar & new(int x, int y, String label) { /* ... */ }
public:
    void handleClick() { /* ... */ }
};

class MotifWindow {
meta public:
    MotifWindow & new(int x, int y, int w, int h) { /* ... */ }
public:
    void move(int x, int y) { /* ... */ }
};
```

A factory is then just a singleton object that is initialized with the product classes to instantiate. For example,

```
object WidgetFactory {
private:
    ScrollBarMaker & scrollbarMaker;
    WindowMaker & windowMaker;

public:
    void init(ScrollBarMaker & scrollbars, WindowMaker & windows) {
        scrollbarMaker = scrollbars;
```

```
            windowMaker = windows;
        }

        AbstractScrollBar & makeScrollBar(int x, int y, String label) {
            return scrollbarMaker.new(x, y, label);
        }
        AbstractWindow & makeWindow(int x, int y, int h, int w) {
            return windowMaker.new(x, y, h, w);
        }
    };

    WidgetFactory.init(MotifScrollBar, MotifWindow);
    AbstractWindow & win = WidgetFactory.makeWindow(0, 0, 100, 100);
```

In this example, the singleton object `WidgetFactory` fulfills the roles of both abstract and concrete factories in the pattern. The static type-checking of metaclass interfaces guarantees that only the proper types of classes will be used as the products. For example, a window class could not be passed in as the scrollbar class to instantiate in `makeScrollBar` since a window class would not match the interface required by `ScrollBarMaker`.

Using objects to package implementations, we can simplify this pattern even further and allow whole component libraries to be checked. For example, we can package together all related product implementations:

```
    object Motif {
    public:
        class ScrollBar {
        meta public:
            ScrollBar & new(int x, int y, String label) { /* ... */ }
        public:
            void handleClick() { /* ... */ }
        };

        class Window {
        meta public:
            Window & new(int x, int y, int w, int h) { /* ... */ }
        public:
            void move(int x, int y) { /* ... */ }
        };
    };
```

and then provide an interface description of such a package:

```
    interface ProductLibrary {
        AbstractScrollBar & ScrollBar.new(int x, int y, String label);
        AbstractWindow & Window.new(int x, int y, int w, int h);
    };
```

using syntax that allows to specify methods of nested classes.

This avoids many problems. `ProductLibrary` establishes an interface for all component libraries to be used with the factory. It prevents, for example, Motif scrollbars from being used together with Presentation Manager windows. The factory simply becomes a singleton object parameterized by the product library from which to instantiate.

```
    object WidgetFactory {
    private:
        ProductLibrary & P;
```

```
public:
    void init(ProductLibrary & lib) {
        P = lib;
    }

    AbstractScrollBar & makeScrollBar(int x, int y, String label) {
        return P.ScrollBar.new(x, y, label);
    }
    AbstractWindow & makeWindow(int x, int y, int h, int w) {
        return P.Window.new(x, y, h, w);
    }
};

WidgetFactory.init(Motif);
```

Uses

The Builder pattern [21] can benefit from metaclasses objects in a similar way as the Abstract Factory pattern. Instead of writing product-specific concrete builder classes, we only need to parameterize the `Builder` class by the concrete product to be built.

Interfaces for metaclass objects not only allow us to abstract over object creation but over any method of the metaclass object. Suppose, we want to maintain instantiation counts for several unrelated classes. We would define an interface for the methods to access the instantiation count. A function to print or analyze these instantiation counts could then take as an argument any metaclass object conforming to the interface.

7 Method Dispatching on Multiple Parameters

Typical object-oriented programming languages such as Smalltalk-80 [22] and C++ [20] use *single dispatching* to determine the method invoked. When a method is invoked on a receiver object, a suitable method implementation is selected dynamically according to the class of the receiver. Other arguments do not influence method selection and are simply passed to the method.

This approach works well for many kinds of methods, especially when the receiver is more important than the other arguments for determining which method implementation to select. However, for some kinds of methods, there might be several equally important arguments, and the asymmetry of the single-dispatch style is not appropriate. Furthermore, it is often cumbersome to add a new method to each class in an otherwise stable class hierarchy. Dispatch based on multiple arguments would allow defining the new method without modifying the class hierarchy by dispatching on an argument whose type is a class from this hierarchy.

Methods whose selection is based on several arguments are called *multimethods* as supported in CLOS [4, 25, 36] and Cecil [11, 12]. Typical multimethods include binary arithmetic operations, binary equality testing, and simultaneous iteration over several collections.

Examples

Multimethods are not directly supported by single-dispatching object-oriented languages, but can be simulated by invoking several methods such that each argument that participates in method selection acts as the receiver once. As soon as the class of an object is known, the class name is encoded in the name of the next method invoked. This technique is called *multiple dispatching* [24] or, in the common case of dispatching on two arguments, *double dispatching*. Double dispatching is exemplified in the Visitor pattern [21].

The Visitor Pattern

In the Visitor pattern, a visitor represents an operation to be performed on the elements of a structure. This allows defining new operations without changing the classes of the elements to be operated on.

As a typical C++ example of the Visitor pattern, consider abstract syntax trees and operations on these trees in the context of a compiler (adopted from [21]). Abstract syntax trees are built from nodes for assignments, variable references, expressions, and so on. Operations on abstract syntax trees include type checking, code generation, flow analysis, etc.

If these operations were defined as methods for each node class, the code implementing the operations would be distributed over the node class hierarchy, making it hard to understand and maintain. Furthermore, adding a new operation would be tedious and would usually require recompiling all node classes.

Instead, the Visitor pattern expects the node classes to have a single method called `Accept` that takes a visitor object as an argument. The pattern further relies on separate visitor classes corresponding to the operations. For each node class *X*`Node`, each visitor has a visitation method `Visit`*X* that is invoked from within the `Accept` method in the node class. This makes it easy to add a new operation, such as a new code optimization scheme, in the form of a new visitor class.

When the `Accept` method is invoked, method selection is first based on the node class; the name of the node class is then encoded in the visitation method invoked on the visitor object; finally, the visitation method is selected based on the actual visitor class. Hence the Visitor pattern provides the effect of double dispatching.

First, we define the `Node` interface with various implementation classes.

```
interface Node {
    void Accept(NodeVisitor & v);
    // ...
};

class AssignmentNode : implements Node {
public:
    void Accept(NodeVisitor & v) { v.VisitAssignment(this); }
    // ...
};

class VariableRefNode : implements Node {
public:
    void Accept(NodeVisitor & v) { v.VisitVariableRef(this); }
    // ...
};
```

We define the `NodeVisitor` interface with implementations `TypeCheckingVisitor` and `CodeGenerating-Visitor`.

```
interface NodeVisitor {
    void VisitAssignment(Node & n);
    void VisitVariableRef(Node & n);
    // ...
};

class TypeCheckingVisitor : implements NodeVisitor {
public:
    void VisitAssignment(Node & n);
    void VisitVariableRef(Node & n);
    // ...
};
```

```
class CodeGeneratingVisitor : implements NodeVisitor {
public:
    void VisitAssignment(Node & n);
    void VisitVariableRef(Node & n);
    // ...
};
```

Several problems arise when using the Visitor pattern. First, it is difficult to add new element classes. Each new element class X`Node` requires defining a method `Visit`X in the interface `NodeVisitor` and corresponding method implementations in each visitor class to be added. This makes the visitor class library difficult to maintain when not only operations, but also element classes are added frequently.

Second, the Visitor pattern assumes that the public interface of the `Node` class is large enough so that visitors can do their job. This often leads to larger interfaces than otherwise desirable and potentially defeats encapsulation.

Third, double dispatching is error-prone because it requires method selection to be implemented manually. This makes it hard to see which argument combination causes the execution of which method.

Solution

Simulated double or multiple dispatching is a standard paradigmatic idiom in languages that do not support multimethods directly. The Visitor pattern is actually an instance of this idiom.

The problems of the multiple dispatching idiom can be avoided if the language provides multimethods as a built-in construct. Such a construct allows the programmer to indicate on which arguments method selection should be based. Instead of specifying method lookup procedurally (as in the idiom), multimethods provide a way of specifying method lookup declaratively [11].

We argue that multimethods make it easy to define new visitation methods or extend the element and visitor class hierarchies alike. Using C++-like syntax, multimethods can be defined as methods in a class or object. We suggest to perform dynamic dispatching on all arguments whose type is a class reference or class pointer. Unlike static overloading resolution, dispatching is performed at run-time, although multimethods can be statically typed [38, 10, 13].

By combining multimethods and packages, we can simplify the abstract syntax tree example by bundling the visitation methods for each tree operation in a package.

```
object TypeChecker {
public:
    void Visit(AssignmentNode & n);
    void Visit(VariableRefNode & n);
    // ...
};

object CodeGenerator {
public:
    void Visit(AssignmentNode & n);
    void Visit(VariableRefNode & n);
    // ...
};

Node & root = // ...

TypeChecker.Visit(root);
CodeGenerator.Visit(root);
```

Both node classes and visitor objects can now be added easily. Adding a node class usually requires adding the corresponding `Visit` methods in the visitor classes; methods can also be added to existing visitor classes by using inheritance. Adding a visitor class does not require any change to the node classes; however, the new visitor class must provide a `Visit` method for each existing node class.

Alternatively, the multimethods for type checker and code generator could have been put into a single package by dispatching on two arguments. While not useful for this example, having a single package makes sense in other cases, for example, when binary methods are involved.

The proposed multimethod mechanism follows the general rule that every method invocation must have a receiver. For a multimethod defined in a class, the receiver is an instance of the class. For a multimethod defined in an object, the receiver is the object itself. Syntactically, multimethods appear like statically overloaded methods in C++ [20]. Semantically, multimethods reduce to overloaded methods only when passed a class pointer or reference.

We propose an *exact-match* multimethod selection scheme that operates in two steps. First, the method is dispatched with respect to the receiver; this step determines in which class or object the method is defined. Second, from all methods within the class of the receiver or within the receiver object, a method is selected based on an exact match on the classes of all arguments. This approach allows the uniform treatment of all methods as multimethods.

To avoid the complexity of best-match algorithms for multimethod selection, we suggest dispatching only on parameters that have class reference or class pointer types. If the type of an argument in such a parameter position is an *interface* reference or pointer type, then the *run-time* dispatch mechanism narrows down the set of suitable method implementations to methods whose parameter type in this position matches the current class of the argument value. If the type of an argument is a *class* reference or pointer type, then the set of suitable method implementations can be narrowed down at *compile time* to methods with this type in the given parameter position. This process is repeated for each argument until finally a single method is selected to be called.

If all values passed to a method have class types, the method implementation can be fully selected at compile time. In this case, method selection reduces to static overloading. To avoid the need for complicated disambiguating rules for which method to select, parameters that have interface reference or pointer types should not be used for dynamic method selection, and all multimethods with the same name should have the same type in this parameter position. Since method dispatch is only performed for parameters of a class type, there can be only one method with a given name whose parameters are all of interface types.

To guarantee that all method invocations can be handled, it is necessary to check statically whether all combinations of parameter classes and interfaces are provided by a multimethod. Since the proposed object system achieves subtyping via interface conformance instead of inheritance, an explicit subtype hierarchy is no longer available at compile time (see Sections 2 and 3). Therefore, determining statically whether a multimethod is defined for all parameter type combinations is slightly more complicated than in systems with an explicit subtype hierarchy such as Cecil [13] and would require linker support. The necessary information could be collected by the linker from the compiler-generated method dispatch tables.

To maintain an object-oriented or data-abstraction-oriented view of multimethods, we could conceptually consider a multimethod as part of each class for which the method dispatches. It would thus be tempting to give each multimethod access to the non-public fields of the parameters on which dispatching is performed. However, this would compromise encapsulation since anyone could now gain access to the implementation of an existing class by writing a multimethod that dispatches on a parameter of that class. We propose separating dispatching and access as a solution to this problem. To grant the multimethod access to non-public fields, the class should declare the multimethod or an entire package containing the multimethod as its `friend`. Encapsulation problems as encountered in the Visitor pattern thus no longer occur.

Uses

Typical uses of multimethods include binary arithmetic operations, binary equality testing, simultaneous iteration over several collections, and displaying a shape on an output device (see also [11]).

For example, a multimethod for iterating simultaneously over two collections could be defined as follows. This method would traverse the two collections in lock step and apply a visitation function to each pair of items visited in each step. Here, we present only the case for iterating over two lists.

```
void PairDo(List & c1, List & c2, PairVisitor v) {
    for (ListIterator i(c1), j(c2); ! i.IsDone() && ! j.IsDone; i.Next(), j.Next())
        v.Apply(c1.CurrentItem(), c2.CurrentItem());
}
```

8 Conclusion

We have examined design patterns, as representatives of common programming practice, and analyzed the influence the choice of implementation language has on pattern design. We believe this analysis benefits both the patterns community and the language communities.

Impact on Design Patterns

Based on our analysis, we have presented several *general-purpose* language constructs and mechanisms that would simplify patterns if they were available in object-oriented languages. We argue that even if they are not available in the chosen implementation language, design patterns should be described in terms of these constructs and mechanisms. This makes patterns simpler, more succinct, and applicable to a larger set of language communities.

To use design patterns written in terms of a larger set of constructs and mechanisms than the implementation language supports, it is necessary to combine these patterns with *paradigmatic idioms*, which are patterns implementing the missing language constructs and mechanisms for a particular implementation language.

With our set of language constructs and mechanisms, it is clearly possible to express all existing patterns since these constructs and mechanisms are a superset of those found in typical object-oriented languages. Additional constructs or mechanisms might be beneficial but would likely be specific to individual patterns and of less general-purpose value.

Impact on Language Design

We suggest that future object-oriented languages provide the language constructs interface, object, and class and the mechanisms interface conformance, code reuse, lexical scoping, and multimethod dispatch.

The proposed language mechanisms are all orthogonal; none can be simulated by a combination of the others. These mechanisms extend traditional object-oriented mechanisms. Multimethod dispatch is an extension of single dispatch. We propose separating the subtyping and code-reuse aspects of inheritance and, as a result, could strengthen both.

Of the proposed language constructs, interface and object are orthogonal. A class is not orthogonal to objects since it also creates the metaclass object. Rather a class is syntactic sugar for a combination of the metaclass object together with a representation type. For orthogonality, it might be desirable to add a language construct for specifying a representation type directly. Such a construct would be similar to `record` in Pascal or `struct` in C, except that it can also contain methods. However, since the combination of a representation type together with a metaclass object containing at least the method `new` is that common, the `class` construct subsumes the functionality of a `record`. Since the class syntax is terser than manually defining a metaclass object together with a representation type, and since it is more conventional, it should be included with any object-oriented language.

It is up to the language designers to integrate the proposed constructs and mechanisms in an orthogonal way into the design of an actual language. To our knowledge, there does not yet exist a programming language that supports the full range of our language constructs and mechanisms. Modern functional languages, such as ML, have closures and packages but lack classes. Most contemporary object-oriented languages do not have closures and have only a rudimentary form of packages (e.g., namespaces in C++).

A language based on our constructs and mechanisms would benefit a larger community than just the object-oriented community. For example, using lexically scoped closure objects allows us to write programs in an abstract data type style or in a functional style as well as in an object-oriented style.

References

[1] Henry G. Baker. Iterators: Signs of weakness in object-oriented languages. *ACM OOPS Messenger*, 4(3):18–25, July 1993.

[2] Gerald Baumgartner and Vincent F. Russo. Implementing signatures for C++. Technical Report CSD-TR-95-025, Department of Computer Sciences, Purdue University, August 1995.

[3] Gerald Baumgartner and Vincent F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software—Practice & Experience*, 25(8):863–889, August 1995.

[4] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp object system specification: X3J13 document 88–002R. *ACM SIGPLAN Notices*, 23(Special Issue), September 1988.

[5] Thomas M. Breuel. Lexical closures for C++. In *Proceedings of the 1988 USENIX C++ Conference*, pages 293–304, Denver, Colorado, 17–21 October 1988. USENIX Association.

[6] Frank Buschmann and Regine Meunier. A system of patterns. In Coplien and Schmidt [18], chapter 17, pages 325–343.

[7] Roy H. Campbell and A. Nico Habermann. The specification of process synchronization by path expressions. In E. Gelenbe and Claude Kaiser, editors, *Proceedings of the International Symposium on Operating Systems*, volume 16 of *Lecture Notes in Computer Science*, pages 89–102, Rocquencourt, France, 23–25 April 1974. Springer-Verlag, Berlin, New York.

[8] Peter S. Canning, William R. Cook, Walter L. Hill, and Walter G. Olthoff. Interfaces for strongly-typed object-oriented programming. In *Proceedings of the OOPSLA '89 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 457–467, New Orleans, Louisiana, 1–6 October 1989. Association for Computing Machinery. *ACM SIGPLAN Notices*, 24(10), October 1989.

[9] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–43, August 1992.

[10] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 182–192, San Francisco, California, 22–24 June 1992. Association for Computing Machinery. *Lisp Pointers*, 5(1), January-March 1992.

[11] Craig Chambers. Object-oriented multi-methods in Cecil. In O. Lehrmann Madsen, editor, *Proceedings of the ECOOP '92 European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56, Utrecht, The Netherlands, 29 June - 3 July 1992. Springer-Verlag, Berlin, New York.

[12] Craig Chambers. The Cecil language: Specification and rationale. Technical Report 93–03–05, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, March 1993.

[13] Craig Chambers and Gary T. Leavens. Typechecking and modules for multimethods. *ACM Transactions on Programming Languages and Systems*, 17(6):805–843, November 1995.

[14] Marshall P. Cline and Greg A. Lomow. *C++ FAQs: Frequently Asked Questions.* Addison-Wesley, Reading, Massachusetts, 1995.

[15] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, California, 17–19 January 1990. Association for Computing Machinery.

[16] William R. Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Proceedings of the OOPSLA '89 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 433–443, New Orleans, Louisiana, 1–6 October 1989. Association for Computing Machinery. *ACM SIGPLAN Notices*, 24(10), October 1989.

[17] James O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, Reading, Massachusetts, 1992.

[18] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design.* Addison-Wesley, Reading, Massachusetts, 1995.

[19] Laurent Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches.* PhD thesis, Université de Genève, Genève, Switzerland, April 1994.

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

[21] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.

[22] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[23] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages (POPL)*, pages 123–137, Portland, Oregon, 17–21 January 1994. Association for Computing Machinery.

[24] Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 347–349, Portland, Oregon, 29 September - 2 October 1986. Association for Computing Machinery. *ACM SIGPLAN Notices*, 21(11), November 1986.

[25] Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, Bedford, Massachusetts, 2nd edition, 1990.

[26] Andrew R. Koenig, editor. *Working Paper for Draft Proposed International Standard for Information Systems — Programming Language C++.* Accredited Standards Committee X3, Information Processing Systems, American National Standards Institute, X3J16/95–0087, WG21/N0687, 28 April 1995. Available from http://www.cygnus.com/misc/wp/index.html.

[27] Thomas Kofler. Robust iterators for ET++. *Structured Programming*, 14(2):62–85, 1993.

[28] Thomas Kühne. Inheritance versus parameterization. In Christine Mingins and Bertrand Meyer, editors, *Proceedings of the 1994 Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific '94)*, pages 235–245, Melbourne, Australia, 1995. Prentice-Hall. The article in the proceedings was corrupted in typesetting; a correct version can be obtained from the author (kuehne@isa.informatik.th-darmstadt.de).

[29] Konstantin Läufer. A framework for higher-order functions in C++. In *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 103–116, Monterey, California, 26–29 June 1995. USENIX Association.

[30] Regine Meunier. The pipes and filters architecture. In Coplien and Schmidt [18], chapter 22, pages 427–440.

[31] Robin Milner and Mads Tofte. *Commentary on Standard ML.* The MIT Press, Cambridge, Massachusetts, 1991.

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

[33] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pages 270–278, Orlando, Florida, 21–23 January 1991. Association for Computing Machinery.

[34] David A. Moon. Object-oriented programming with flavors. In *Proceedings of the OOPSLA '86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–8, Portland, Oregon, 29 September - 2 October 1986. Association for Computing Machinery. *ACM SIGPLAN Notices*, 21(11), November 1986.

[35] Diane E. Mularz. Pattern-based integration architectures. In Coplien and Schmidt [18], chapter 23, pages 441–452.

[36] Andreas Paepcke. *Object-Oriented Programming: The CLOS Perspective.* MIT Press, Cambridge, Massachusetts, 1993.

[37] Wolfgang Pree. *Design Patterns for Object Oriented Software Developers.* ACM Press, New York, New York, 1994.

[38] François Rouaix. Safe run-time overloading. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 355–366, San Francisco, California, January 1990. Association for Computing Machinery.

[39] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[40] Mary Shaw. Patterns for software architectures. In Coplien and Schmidt [18], chapter 24, pages 453–462.

[41] Sun Microsystems, Mountain View, California. *The Java Language Specification*, Version 1.0 Beta, 30 October 1995. Available from ftp://ftp.javasoft.com/docs/javaspec.ps.

[42] Walter Zimmer. Relationships between design patterns. In Coplien and Schmidt [18], chapter 18, pages 345–364.