Abstract

We propose a new object-oriented programming and design technique called “Configurable Objects”. Through configurability, composite objects can be dynamically constructed by assembling compatible components. We demonstrate that this property can promote a new level of reuse. A configurable object captures a relationship between its components and represents it in a form that is reusable as these components are improved. While traditional techniques emphasize the development of components that may be useful in the future, configurability plans and prepares for the reuse of future improvements of components. We present also several cases in which reuse by configuration is more appropriate than reuse by inheritance or by composition. The implementation techniques are exemplified in C++.

1 Introduction

The rapid popularization of object oriented programming (OOP) motivated many researchers to engage in the activity of investigating the sources of its success. The goal is to apply the gained understanding as leverage in the development of various means and methods through which the paradigm could be better exploited as well as diverse extensions to it. This paper makes another step along this track, starting from the common observation that the success of OOP should be attributed to the emphasis it puts on data rather than on algorithms. A well-written object-oriented program is based on a sound model of the classification and organization of data, its attributes, and its behaviour. A new design and programming technique is presented, with the purpose of enriching the toolkit used in OOP for the construction of such models.

OOP recognizes two principal kinds of relationships between classes: inheritance and composition. Much has been written on the comparison between the two, and on the circumstances in which one should be preferred over the other (see, for example, [Johnson 1993, Meyer 1988, Rumbaugh 1991]). The literature also discusses in detail techniques and heuristics for the application of inheritance [Halbert 1987, Rumbaugh 1993, Snyder 1986].

On the other hand, composition has received much less attention in the OOP literature, most probably because it was already extensively studied by the databases com-
Current research on object-oriented analysis refer to composition as a tool for reducing complexity [Coad 1991, Odell 1992]. This is indeed the case for classes of data that model problem domain entities, such as accounts, employees, airplanes, and so on.

However, OOP excels not only in the direct mapping of the problem domain into the implementation domain, but also in helping the programmer harness purely implementation-domain entities. In this paper, we are interested mostly in the usage of composition in the implementation domain. In particular, we consider the new flavour that polymorphism may bring to composition. If a composite consists of a component of a certain class, then it makes sense to ask whether an instance of a subclass can be used as this component.

In the remainder of this section, we introduce configurable objects, give a simple example for their application, and explain how they can be used to achieve a new level of reuse. In the next section we give a concise taxonomy of configurability in the purpose of exposing the possibilities it offers. Section 3 presents some problems that arise when trying to reuse through traditional mechanisms, and shows how they are solved by configurability. In particular, we analyze several strategies for tackling the problem of enhancing the functionality of an unchangeable hierarchy. The analysis manifests some of the relative merits and weaknesses of the strategy of configurability. Section 4 is concerned with the implementation of configurable objects in C++ [Stroustrup 1991].

1.1 What are Configurable Objects?

Assembling the whole from its parts is modeled by composition (sometimes called aggregation). A composite object consists of component slots. An instance of a component class is inserted into each such slot. Consider the example depicted in Figure 1, that uses OMT notation [Rumbaugh 1991].

Then, each instance of Image is a composite object consisting of two slots. These are occupied by instances of the Matrix and Position component classes. In statically typed languages (such as C++), the component classes are completely specified at the time of the definition of the composite class. In dynamically typed languages (such as Smalltalk [Goldberg 1983]), only slot names are specified by the definition. Notwithstanding, implicit assumptions on component classes are made by the usage of the components in the methods of the composite class.

An intermediate approach between the static and dynamic extremes stems naturally from the following observation: Let C be the declared class of a certain slot of a composite object. Then, any instance x of C, a subclass of C, may be inserted into this slot. We say that x is compatible with the component slot. A configurable object is a composite object whose components may be dynamically replaced by other, compatible ones.

Software Boards Brad J. Cox in his classical book on OOP [Cox 1986] used the term “software IC” to emphasize the similarities between objects and integrated circuits, claiming that, by the use of encapsulated components, the rapid progress achieved in the hardware industry may be reproduced by software developers. Equivalently, configurable objects may be seen as “software boards”, since they are composed of slots into which several kinds of components may be plugged. Only the interfaces need be defined to assure compatibility.

Computers based on mother boards with slots, like the veteran Apple II or the more
contemporary IBM-PC, have enjoyed enormous success by enabling their users to easily build their own configuration. For example, the memory can be expanded, a new drive can be added, and the video controller can be changed. The user needs to know neither the details of how each of these devices works nor the way they are used by the computer.

Similarly, configurable objects may provide several advantages when compared to traditional designs. By enabling the client of a class to add, remove or substitute components, new levels of extensibility and flexibility are reached. The various needs of different users may generate improvements of existing components or even the creation of new kinds of components that were not foreseen by the designer of the composite. As in the case of personal computers, this can be offered without exposing implementation details and without demanding special knowledge.

Configurability allows the programmer to assemble composite objects from components of types which did not exist when the composite was originally defined. In the previous example, configurability means the possibility to instantiate the class \texttt{Image} while plugging into its appropriate slot an instance of \texttt{SparseMatrix}, a subclass of \texttt{Matrix}, which provides more efficient memory usage. In dynamically typed languages this flexibility is quite natural, while in statically typed languages it is realized by the employment of pointers.

Pointers (or references) to a base-class may point to instances of derived classes. This property, complemented by dynamic binding, is the essence of polymorphism. The power and great flexibility offered by polymorphism are well known to any OOP programmer. For example, this feature is indispensable in the crafting of a heterogeneous collection class. In an attempt to minimize the misuse which might be rendered by flexibility, we identify and investigate a specific, but yet quite general, exploitation of polymorphism. We call this \textit{configurability} and view it as a generalization of composition.

Traditionally, polymorphism is used for reducing the complexity of \textit{code}—a single code segment may obliviously treat objects of different types. In contrast to that, configurability is the employment of polymorphism as a tool for arranging \textit{data}. Proper usage of it should contribute to better data organization. The ensuing discussion, the taxonomy of configurable objects, and the examples we provide for its application, have the following objectives:

1. Offer a more conscientious approach to the usage of polymorphism.
2. Legitimize configurability as a standard reuse technique in object-oriented design.
3. Bring the issue of better support of configurability to the attention of designers of object-oriented languages.

\subsection*{1.2 Example for Configurable Objects}

We present here a very simple example whose purpose is to illustrate the concept of configurable objects, in contrast to the traditional techniques of inheritance and composition. A detailed discussion of the advantages of the new approach will be done through more elaborate examples in Section 3.

Consider a class \texttt{Set} which represents a collection of objects without repetitions. An implementation of \texttt{Set} could reuse the functionality already provided by other types of collections. This could be done by using inheritance, composition or configurability.

\textbf{Inheritance} The class \texttt{Set} may be defined as a subclass of another collection, for example, \texttt{List} (Figure 2).

This approach has the advantage of automatically granting the class \texttt{Set} with all the methods of the class \texttt{List}. Most probably, the only method that need be overridden is the one
that inserts new elements into the list. On the other hand, a set cannot be considered as a special kind of list. Methods of List that are not adequate for Set cannot be cleanly eliminated from its protocol. Actually, this kind of implementation inheritance is a common misuse of subclassing [Rumbaugh 1993].

Composition The class Set is composed by some other collection, for example, a List (Figure 3).

![Figure 2: Reuse by Inheritance](image)

Figure 3: Reuse by Composition

With this approach, some of the methods may be inherited from the abstract class Collection, while others must be overridden. The implementation of overridden methods will consist mostly of calling the methods of List. In this relationship, an instance of List is the supplier providing services for the composite Set. The main drawback in this case is the additional work necessary to implement message forwarding [Blake 1987]. We need to write several methods whose only action is resending a message to the component.

In both the inheritance and composition approaches, the class List should be viewed as part of the implementation of the class Set. The user is allowed to change the implementation of the class Set only if he or she has access to the source code. Suppose that a class, which provides the functionality of a collection more efficiently than List, becomes available. Then, it would be profitable to replace the component List in Set by an instance of this new class. However, this kind of implementation change is not commonly allowed to the client of a class.

Configurability The class Set is composed by a collection, but the actual instance of collection used in the configuration may be determined by the client (Figure 4).

![Figure 4: Reuse by Configuration](image)
sures that such instance satisfies the protocol used in the implementation of Set. If the client derives a new subclass of Collection, for example HashTable, which is more efficient than existing ones, then instances of this subclass can be plugged into the slot of the class Set.

Therefore, the client may benefit immediately from the better performance of the new subclass and does not need to know the Set's internal details, neither HashTable's internal details. Since the class Set was designed to be configurable, access to its source code is not a pre-requisite for improving its implementation.

In all the three implementation alternatives, the class Set is a descendant of Collection, either directly or indirectly through the class List. The inheritance and composition approaches make permanent the association between the classes Set and List, while configurability employs polymorphism and allows any instance of a subclass of Collection to be used as a component. This includes new subclasses that may be developed in the future.

1.3 A New Level of Reuse

Traditionally, OOP programmers have benefited from two different kinds of reuse: by inheritance and by composition. Reuse by inheritance occurs when a new subclass inherits methods from its superclass, without the need to override them. Reuse by composition occurs when a composite class uses the services of a component to perform its tasks. The subclass may be considered a client of its superclass, and the composite class may be considered a client of its components. In both cases the supplier of the services, i.e., the superclass and the component, respectively, must be known when the client is defined. The relationship between client and supplier is established statically, and can be changed only through modifications in the source code.

In contrast, configurable objects allow the reuse of code that does not yet exist. They can become clients of components that were not known at the moment of their definition, since the relationship between client and supplier may be established dynamically, at run-time, without requiring modifications. Therefore, reuse by configuration may increase the profit gained by the development of new components, providing immediate return for the resources invested in improving the efficiency of these components. Traditional design techniques emphasize the development of components that may be useful in the future, but provide little support for the construction of composites that take advantage of future improvements in their components.

1.4 Configurable Objects vs. Parameterized Classes

There are some similarities between the techniques of configurable objects and parameterized classes (templates), since both of them are intended to provide more flexibility to the clients of a class. Nevertheless, there are also some subtle, but very important differences, both conceptual and practical.

Conceptually, the relationship between a configurable object and a component is different from the one between a template and a class parameter. A component is part of the implementation since it is a supplier of services that are used by the composite to perform its tasks. A class parameter normally defines a kind of object that is manipulated by the template, but whose services are not used. In general, the class parameter received is not relevant for the data structures and algorithms used in the template. This is the reason they may be viewed as a kind of macro.

The relationship between a configurable object and a component is functional [Civello 1993], since the component contributes to the function of the composite. The relationship between a template and its components is

---

1. One can think of a template as a clever kind of macro that obeys the scope, naming, and type rules of C++ [Stroustrup 1991, page 257].
usually *non-functional*, and in many cases represents only containment. Each component in a configurable object has a name that describes its role in the association, while the elements in a container are not individually named.

Both configurable objects and parameterized classes use the composition mechanism, but the kind of association between the whole and its parts is different in each case. According to the classification of Civello [Civello 1993], a configurable object is an *assembly*, in which each component has an essential function to fulfill. Templates are normally used to represent a *group-member* association, in which the connections between whole and parts are much looser.

Practically, the relationship between configurable object and component may be established dynamically, while the relationship between template and class parameter is established statically. Moreover, the notion of default configuration has no meaningful equivalent for parameterized classes.

The differences between the two techniques should make clear that there are situations where one is more appropriate than the other. Given a specific problem, they should not be considered equivalent choices. Instead, rational criteria are to be applied for selecting one of them, based on their distinct characteristics. Conversely, the combination of both can be quite powerful, as shown in Section 4.

## 2 Concise Taxonomy of Configurability

In this section, a discussion is conducted around concepts related to the idea of configurability, providing further insight on its consequences and potentialities. We suggest several criteria for the classification of the usage of configurability. A complete taxonomy of configurability and composition in OOP is beyond the scope of this paper.

A related work is that of Odell [Odell 1994] which exposes to the OOP community the taxonomy of part-whole relations as developed by Winston, Chaffin and Herrmann [Winston 1987]. In his essay, Odell presents the diverse meanings that the phrase “part of” may have in natural language, but gives scarce attention to design issues. A more relevant work is the one of Civello [Civello 1993], which concentrates on the roles for composition when modeling problem-domain entities. However, Civello’s contribution does not cover the application of composition to pure implementation-domain objects, and the benefits that may be gained by this application.

### 2.1 Compatibility

Given a slot and a component, how does one determine if the component can be inserted into this slot? Ideally, insertion would be allowed only if *behavioral compatibility* is assured, i.e., the component has a faithful implementation of the behaviour required by the slot owner. This demand lies in a semantic level and hence cannot be guaranteed by any automatic procedure. We must therefore content ourselves with *type compatibility*, i.e., that the component is of the same type as declared for the slot, or of a sub-type of that type.

Type compatibility is easily achieved in statically typed languages for which it is checked by the compiler. In dynamically typed languages, it is more natural to use an even weaker form of compatibility, namely *protocol compatibility* [Pintado 1993]. The component is required to recognize all messages that the composite may send to an object in the slot. Protocol compatibility is checked only at runtime. In most dynamically typed languages it is possible for a composite to require type compatibility by examining at runtime the type of the component.

Sometimes the flexibility offered by protocol compatibility is preferable to the rigour of type compatibility. A case in point is the use of com-
ponents originating from two distinct hierarchies but recognizing the same protocol. In statically typed languages this similarity can not be explored without additional effort. An example of how protocol compatibility can be implemented in C++ is given in Section 4.2.

2.2 Configuration and Configurability

Definition 1 The configuration of a composite object is the set of components currently inserted in its slots. Configurability is the ability of clients of a composite to change its configuration.

The configuration is part of the implementation since a composite relies on components’ behaviours to accomplish its tasks. A configurable class is obliged to expose part of its realization. However, this exposition is such that changes to the implementation can be done in an orderly manner, respecting encapsulation, and without knowing internal details. Appropriate usage of configurability thus promotes separation of interface from implementation.

It is important to distinguish between two levels of configurability. Static configurability is the ability to create several instances of a class, each with a different configuration. Dynamic configurability is more permissive in allowing changes to the configuration during the lifetime of the object.

Composite classes with static configurability can be instantiated to produce different kinds of instances. An advantage of such classes is that they reduce the number of classes a client needs to be aware of. In this respect, they are similar to kits [Linton 1992], which are object factories devised as an instrument to reduce the size of the interface of the InterViews [Linton 1988] GUI class library. The main difference is that objects produced by kits may be of distinct types, while all instances of a configurable class are of the same type.

Our experience shows that with the absence of a compelling reason, static configurability should be preferred. The support of dynamic configuration may introduce a penalty in complexity; in whatever state the object may be in, it should be able to adapt to a change in its implementation. In many cases, it is possible to obtain an equivalent functionality by destroying the old object and constructing a new one with the desired new configuration.

2.3 Component Slots

Definition 2 A component slot is optional if the composite can exhibit complete functionality while leaving this slot empty.

Optional components may be used to extend the functionality of a composite. For example an object used for spelling check can be optionally equipped with a domain specific dictionary of jargon and acronyms. The implementation of a configurable object may be changed not only by replacing components, but also by adding or removing them. It should be understood that implementing dynamic configurability is easier when modifications are restricted to addition and removal of optional components. The operation of component replacement usually requires the complex maneuvering of saving the state of the old component and restoring it in the new one.

A configurable object should be usable even if its configurability is not exploited.

Definition 3 The default configuration is the one provided when no specific request for configuration change is made by the client.

The default configuration is a set of components sufficient for a complete implementation of the composite functionality.

There are cases in which no default configuration exists, since values for some of the component slots cannot be determined by the composite at the moment of object construction. Values
for these slots must be received as parameters by the constructor method. With the analogy of a composite object as a team of workers, such values are external personnel contracted by the client to complement the team.

Continuing with this analogy, it is possible for the same worker to be simultaneously employed in several teams. This corresponds to the same component being part of multiple composites. Sharing of this type usually leads to problems such as response to conflicting requests, memory management responsibilities, and so on.

**Definition 4** A component is exclusive to a composite if it is not a part of another composite.

Encapsulation means that all components are exclusive. Compromising it by using non-exclusive components should be done only with good reasons. The main situations in which a compromise is justified are sharing of functionality (as in windows sharing a screen driver), and sharing of data to avoid redundancy (as in several viewers displaying the same file).

An exclusive component should belong to a unique composite at any time. However, it can be removed from one composite and then added to another. A transfer of the ownership of a component suggests a different perspective of viewing configurable objects. They can be used for the implementation of sequential processing, much like a factory assembling line. From a slightly different point of view, a component moving between composites is a form of inter-object communication.

### 2.4 Configurable Inheritance

Configurable objects address the issue of replacing a component of a given class by an instance of a subclass. A similar issue is whether the superclass of a given class may be replaced by another class, conforming to some compatibility definition. This capability could be called *configurable inheritance*, since it is related to modifications in the inheritance tree, instead of changes in the whole-part hierarchy. In both cases the configurable object includes a polymorphic subobject, either a polymorphic component or an instance of a polymorphic superclass.

The questions posed by configurable inheritance are even more complex than the ones related to configurable objects. Inheritance is used not only for specialization, but also for subtyping. Hence, it is essential to know the superclass of a given class in order to determine its actual type. This would require a very restrictive definition for compatible superclasses in statically typed languages. Moreover, knowing the actual superclass may be essential to determine if the subclass is abstract or not.

Configurable inheritance would be a powerful mechanism, but it is not supported in most object-oriented programming languages. However, configurable objects may be used in order to achieve some benefits that are similar to the ones provided by configurable inheritance. The next section presents a problem that could be solved ideally by the use of configurable inheritance, but has a satisfactory implementation based on the use of configurable objects. An approximate implementation of configurable inheritance in C++ is presented in Section 4.3.

### 3 Applications of Configurable Objects

Our main interest is the application of configurability as an alternative reuse mechanism. This section presents three non-trivial cases that arise when trying to reuse existing libraries. The first is the development of new classes that are similar to existing ones, and could reuse their functionality. The second is the derivation of new leaf classes in several independent branches, in
order to add similar properties. The third is
the interface extension of every class in an un-
changeable hierarchy. Each one of these
situations illustrates some of the advantages of
the use of configurable objects when compared
to traditional techniques. Finally, Section 3.4
presents some specific applications of configurable objects, exemplified by several design pat-
terns [Gamma 1993].

3.1 Or Inheritance

In the traditional approach, whenever a new
class is created, we look for existing classes
whose functionality can be reused through in-
eritance. Sometimes this new class may be im-
plemented in several ways, i.e., it could inherit
the functionality it needs from either of several
distinct superclasses. We call this problem the
“Or Inheritance” problem, since a class \( N \) could
be defined as an \( A \) or a \( B \) or a \( C \).

A possible solution would be the definition of a distinct class inheriting from each one of
the possible superclasses. However, this would
cause undesirable code redundancy. Moreover,
if a new potential superclass is created, it would
be impossible to reuse this functionality. The
ideal solution would be the use of configurable
inheritance, since the code need be written only
once, and new classes may be used in the imple-
mentation. We will see that by the use of con-
figurable objects the same benefits are achieved.

An Example of Or Inheritance  Suppose
that we want to derive a new class \( \text{Set} \). After ob-
seving that a \( \text{Set} \) has a lot in common with col-
llections, we decide to reuse by inheritance. Sup-
pose now that the class \( \text{Collection} \) has three
subclasses, \( \text{List} \), \( \text{Tree} \), and \( \text{Array} \). Each one
of them may be used in the implementation of
the class \( \text{Set} \). We could derive three distinct
subclasses, \( \text{ListSet} \), \( \text{TreeSet} \) and \( \text{ArraySet} \), as
depicted in Figure 5. The functionality of class
\( \text{Set} \) must be reimplemented for each one of these
classes and the commonality among them can
not be explored. Moreover, if a new subclass of
\( \text{Collection} \) is added, for example \( \text{HashTable} \), a
new kind of \( \text{Set} \) must be derived as well.

![Figure 5: The problem of Or Inheritance](image)

In contrast, if \( \text{Set} \) is defined as a configurable
class, any instance of \( \text{Collection} \) can be used
in its implementation, as illustrated by Figure 6.
The methods for the configurable class \( \text{Set} \) must
be written only once and are applicable to any
configuration, thanks to polymorphism. Addi-
tionally, instances of a new class \( \text{HashTable} \)
would be compatible components, and could
be used without requiring changes nor exten-
sions. Hence, configurability allows the client of
class \( \text{Set} \) to easily improve its implementation,
achieving new levels of reuse.

![Figure 6: Or Inheritance solved by configurability](image)

3.2 Repeating Patterns

In order to reuse an existing library, we normally
derive leaf classes adding some specific proper-
ties, required by the application being de-
veloped. A common consequence of this approach
is the repetition of some patterns among distinct branches, i.e., the derivation of specialized subclasses based on the same additional properties. Given independent classes $A$ and $B$, that are not related as subclass-superclass, new subclasses are derived from each one of them, say $A_1$ and $A_2$, $B_1$ and $B_2$, such that $A_1$ and $B_1$ implement almost the same additional functionality as well as $A_2$ and $B_2$.

This repeating pattern indicates that the common functionality implemented in the subclasses should belong to an independent class. By creating this class we could avoid the subclass proliferation, reduce code redundancy, and also provide its functionality to other classes, thus improving reuse.

The solution using configurable classes would be the derivation of new subclasses for $A$ and $B$, such that they are composed by an instance of $F$, where $F$ is an abstract class. The class $F$ would have subclasses $F_1$ and $F_2$, where $F_1$ implements the functionality previously present in $A_1$ and $B_1$, while $F_2$ implements the one of $A_2$ and $B_2$. If a new subclass $F_3$ is derived, its functionality will become immediately available to $A$ and $B$. Additionally, a distinct class $C$ could also reuse the functionality of $F$.

An Example of Repeating Patterns Consider a user interface library which includes classes such as Window and Menu. If the interface is intended to work both in text mode and in graphic mode, a possible decision would be the derivation of classes GraphicWindow, TextWindow, GraphicMenu and TextMenu, as depicted in Figure 7. Each of these classes would specialize its superclass by the use of specific functions. However, subclasses working in the same mode would present a lot of code redundancy.

Using a different approach, illustrated by Figure 8, the classes Window and Menu could be made configurable. A new abstract class ScreenDriver could be created with the subclasses GraphicScreen and TextScreen. The mode of operation of each configurable object would depend on the kind of ScreenDriver that is providing the services. If a new kind of ScreenDriver were defined, for example, TerminalScreen, it could be used in the configuration without changes. Moreover, a new configurable class may be created, for example, DialogBox, which makes use of all the functionality already implemented in the class ScreenDriver and its subclasses.

3.3 Interface Extensions

An inheritance hierarchy is susceptible to refinements that can be expressed in the form of a derivation of new leaf classes. However, sometimes a need arises to add functionality to the interface of all classes in an inheritance hierarchy. Let $A$ be the root class of an inheritance tree, and let $\mu$ be a new set of methods which should be added to the protocols of $A$ and all its subclasses. If source code is available and if interface modifications are allowed, then $\mu$ may be incorporated into $A$'s protocol, causing it to be inherited by all of $A$'s subclasses.
If this incorporation is not a viable option, then there are three possible general strategies, relying on inheritance, multiple inheritance and configurability.

**Inheritance** A new subclass is derived from each non-abstract class in the hierarchy. All of these new classes need to implement the functionality of μ. This approach has the disadvantage that new subclasses derived from A, or subclasses which escaped the attention of the programmer, would have no way to reuse this functionality. Diversity of these subclasses may tempt a programmer to use slightly different implementations and ignore the danger of inconsistent behaviour.

The problem of recurring re-implementations can be solved, but at the cost of introducing multiple inheritance.

**Multiple Inheritance** Define M as a class derived from A, adding to M the functionality of μ. For each non-abstract class A′ derived from A, define a new subclass A″ derived from both A′ and M. Of course, this does not solve the problem of proliferation of subclasses. Moreover, if A has instance variables, the multiple inheritance solution raises the problem of common parents in the inheritance graph (sometimes nicknamed the “diamond problem”).

The class M must be a subclass of A in order to be able to call methods from A in its implementation. The manifestation of the “diamond problem” in this case is that instances of class A″ will have two subobjects of class A, one inherited from class A′ and the other inherited from class M. Clearly, the methods of both superclasses must share a *single* subobject of the common parent class.

**Configurability** Through configurability, the new functionality may be implemented without requiring code modifications and while permitting any subclass to use it. This is done by defining a configurable class C, composed of an instance of A, and with an interface containing μ. The implementation of μ in C relies on the A component. In addition, C is declared as a subclass of A in order to make C type compatible with A and guarantee that C recognizes the same protocol as A. However, this protocol is implemented by forwarding all messages to the A component.

Since C is configurable, any instance of a subclass of A may be inserted into its component slot. Instead of using an object of a class A′, subclass of A, a client should use an object of type C configured with an instance of A′. By doing so, the new functionality may be extended to any subclass of A, without code redundancy.

A drawback of this approach is that if A′ extends the protocol of A by introducing a new method, then this method is not accessible through an instance of C. Even if an instance of C is configured with a component of type A′ the protocol recognized by C is only the one defined by A ³. The configurability approach is therefore most suitable in cases in which A is an abstract class defining an interface, while various implementations are provided by its subclasses.

All three approaches suffer from the loss of subtype information. Let Y be a subclass of X. Let X + μ and Y + μ be X and Y augmented by μ using any of the above approaches. Then, Y + μ is not a subclass of X + μ. If it was possible to change the hierarchy, then any change to X will be reflected in all of its subclasses, and there would be no need for Y + μ to be explicitly defined. In all but the configurability approach, we generate a new set of classes that are not related by inheritance. To reconstruct the inheritance information, immediate superclasses should be compared. This is quite an involved mission.

**An Example of Interface Extensions**

*Dribble streams* [Lieberman 1986] are streams

³One possible (but not very elegant) cure for this is for the client to save an external reference to the component and invoke this kind of method through it.
that, besides producing output on some external device, also record it in a file. Consider a standard stream I/O library such as that of C++. The additional functionality of dribble streams could be useful for any kind of output stream in the library. However, it is desirable to provide it without having to change the source code, and requiring only a minimum effort to adapt existing applications.

Using configurable objects, a DribbleStream class would be a new subclass of the class OutputStream, with a slot of type OutputStream. An instance of DribbleStream sends its output to a file before sending it to its component. Such an instance can be associated with any kind of output stream, providing its services in a transparent manner. The component does not care from whom it receives the characters it outputs, while the client receives the services required and may ignore the partnership.

The methods implemented in the class DribbleStream are a subset of the protocol of the class OutputStream. If there are a few basic methods used to construct more complex ones, then only the basic ones have to be overridden. Each one of these basic methods can be implemented by forwarding the message to the enclosed component after sending the output to a file. The more complex methods are inherited and need not be changed.

3.4 Design Patterns

Design patterns [Gamma 1993] are an attempt to classify the kinds of collaboration among objects. The main goal is reusing design experience, by the identification of common patterns, "design idioms" which enrich the "vocabulary" of the software developer. Several design patterns make use of abstract coupling between classes, and therefore can be seen as specific applications of configurable objects. We mention only two of them, but many others may be found in [Gamma 1993].

The Bridge pattern separates an abstraction from its implementation. It is suitable to the problem of "single interface and multiple implementations", i.e., in the case of classes that may be realized in several ways. For example, a class implementing a stack could be constructed based on a list or an array. By the use of configurability the actual data structure used in the implementation of the stack may be dynamically changed.

The Strategy pattern is suitable to situations in which several algorithms are applicable to a given data structure. Each algorithm may be implemented in a distinct class whose instances are used in the configuration of the object that contains the data structure. For example, a class implementing a graph could make use of several algorithms to compute the maximal flow in the graph. It is well known that some of these algorithms are particularly adequate for graphs presenting special properties. By checking the graph's characteristics the most appropriate algorithm may be selected at run-time, and used to compute the flow with maximum efficiency.

4 Implementation of Configurable Objects

This section discusses some questions related to the implementation of configurable objects in C++, including protocol compatibility and configurable inheritance. Some examples illustrate the application of these techniques.

4.1 A Simple Example

Here we present a partial implementation of the class Set, as discussed in Section 3.1. Following the configurability approach, a Set is a subclass of Collection, that has a single slot which must contain an instance of some subclass of Collection. The default component class is List.

```cpp
class Set: public Collection {
```
4.3 Implementation of Configurable Inheritance

The following example shows how configurable inheritance can be used to address the problem of interface extensions, presented in Section 3.3.

template <class Type>
class Mu: public Type {
public:
    Mu(Type& prototype) :
        Type(prototype) {}
    // new methods
    mu_1(...) { ... }
    ...
    mu_n(...) { ... }
};

For any class \( X \), the template instantiation \( \text{Mu}<X> \) originates a new subclass of \( X \) which augments its functionality by the set of methods \( \{\text{mu}_1(), \ldots, \text{mu}_n()\} \).

The attention of the implementor is drawn to an intricacy which occurs in this solution. Constructor methods must receive special treatment since they are not inherited in C++ as in many other languages. In the implementation adopted here the constructor of the class \( \text{Mu} \) receives as a parameter a reference to an instance of \( \text{Type} \). This reference is used to call the copy constructor of the class \( \text{Type} \). Hence, the class \( \text{Mu} \) may be instantiated using as a parameter an object obtained by calling any of the various possible constructors of the class \( \text{Type} \). In this sense, the class \( \text{Mu} \) "inherits" all the constructors of the class \( \text{Type} \).

A disadvantage of templates is the increase of program size due to generation of many classes, each containing a copy of the methods \( \text{mu}_1, \ldots, \text{mu}_n \). This problem can be solved by the combined use of multiple inheritance, templates, and configurability:

class Mu { // A configurable class
    A *slot;
public:
    Mu( /* constructor */ );
    // new methods
    mu_1(...) { ... }
    ...
    mu_n(...) { ... }
};

The template \( \text{Composite} \) can be instantiated only with a \( \text{Component} \) which recognizes \( \text{some_message}() \). Observe that in this case, checking protocol compatibility is done at compile time and that different instantiations of the template generate disjoint composite types.
Some consequences and potentials of the new technique were discussed through the exposition of several related concepts, such as compatibility and configurability. Examples were presented to illustrate some possible applications of configurable objects, where traditional techniques are clearly unsatisfactory. These examples expose some of the merits and weaknesses of configurability. Finally, implementations strategies in C++ were presented and their applicability to these examples was demonstrated.

We observe that C++, as well as many other major OOP languages, leaves something to be desired in terms of support of configurable objects. We had to combine several features of C++ as a substitute of a single lingual construct for the definition of slots and their operations. Moreover, while implementing configurable objects, we encountered several methods whose only action is resending a message to a component. This explicit forwarding could be avoided if the programming language had a facility for a configurable to selectively export part of the interface of its components. Ideally, such a facility should also allow renaming of exported methods.

The advantages of configurable objects when compared to traditional reuse techniques, for some specific situations, support our claim that it should be recognized as a standard instrument in object-oriented design. We hope that further research on the issues of the usage of the composition mechanism and polymorphism will bring new improvements in this subject. Additionally, object-oriented languages that provide better support for configurability should be motivated, either by including new features or by the development of mechanisms to help its implementation.

Acknowledgments We thank Dan Berry, Karen Holtz, Eliezer Kantorowitz, David Lorenz, Yvonne Sagi and the anonymous referees for helpful comments on early versions of the paper.
References


