

# The Design and Implementation of a Framework for Extensible Software

**Stuart M. Wheeler and Mark C. Little**

Department of Computing Science, The University of Newcastle upon Tyne,  
Newcastle upon Tyne, NE1 7RU UK

*Software systems are composed of numerous software components. These components could be developed as a part of the system, or provided by a separate software development environment (e.g., a library which provides a graphical user interface). However, over the lifetime of the software, additional components or modifications to those currently available, may be required. For example, new versions of software components may be necessary due to bug fixes. Therefore, software is often required to be extensible, enabling modifications to occur with minimal effect on existing users. To allow this extensibility, components should only be available through interfaces that are clearly separated from their implementations, allowing users to be isolated from any implementation changes. Object-oriented programming techniques offer a good basis upon which this separation can be provided. This paper describes a model for constructing extensible software based upon this separation, and illustrates this with a software development system we have implemented which supports these ideas in C++. This software development system has been designed so that no modifications are required to either the compiler or operating system, hence making it portable.*

## 1 Introduction

Software systems are composed of numerous software components. These components could be developed as a part of the system, or provided by a software development environment (e.g., a library which provides a graphical user interface). However, over the lifetime of the software, additional components or modifications to those currently available, may be required. For example, new versions of software components may be necessary due to bug fixes. Therefore, software is often required to be extensible, enabling modifications to occur with minimal effect on existing users. However, most software is not designed with the ability to easily modify individual components: minor changes in a component can require changes in many other components and applications. Obviously this affects the ease with which software components can be reused.

We believe that the extensibility required from software can be obtained by providing an “open system” framework. In this framework, components becomes *units of encapsulation*, allowing them to be modified and replaced in isolation, without affecting existing components or applications. The selection of software components to be used by an application is flexible, allowing it to be modified within the application's lifetime, without changing either the application or components. In addition, by grouping components into modules, we can further improve the encapsulation of software, with the capabilities of an application defined by the modules available to it. We

believe that object-orientation provides a natural framework within which this software development model can be realised, where software components are mapped into sets of classes, providing the required encapsulation.

This paper presents the software model that we have developed to allow the construction of extensible software, and the design and implementation of a development system that supports this model in C++. We shall illustrate the advantages of using this development system, and contrast it with other work that has been performed in this area.

## 2 The software design model

This section describes in a language independent manner the software design model which supports “open system” implementations. Section 2.2 will then illustrate how this can be modelled using object-oriented techniques.

### 2.1 Model

In the model, software components are constructed from two separate entities: the *interface component* and the *implementation component*. (Where there is no ambiguity, in the rest of this section we shall refer to interface and implementation components as *interfaces* and *implementations*, respectively).

The interactions between implementations can only occur through interfaces. A single interface can provide access to multiple implementations, and a single implementation can be accessed through multiple interfaces. The necessity of providing multiple interfaces to implementations has long been realised [1][2][3]. However, we take this further by allowing the bindings of interfaces to implementations, and the interfaces an implementation can be accessed through, to be configurable. New implementations that provide additional functionality may be used through existing interfaces, which may not be able to benefit from these features, but which are available through new interfaces.

Typically it is the implementation of a service that changes more frequently than its interface. Since implementations can only be accessed through an interface, this can hide changes to the implementation, allowing the effects of most software changes to remain local. A core part of this model is that the binding between interface and implementation is flexible, and can be changed during the lifetime of the interface. Applications are written only in terms of interfaces, and although an application can request a specific implementation from an interface, it occurs in a way that allows this request to be changed without modifying the application. The capabilities of an application are thus defined by the implementations available to it, allowing the same application to function differently between users. For example, a demonstration version of an application can be provided by simply removing a subset of the available implementations.

This separation of interfaces from implementations is not new, with much work done on Interface Definition Languages (IDL) [4]. However, most IDLs are used in the context of distributed applications, with their support structures generating client and server stub code from the interface definition. Their support structures are not powerful enough to reflect the more general interface and implementation bindings which we believe are necessary. In our model, the interfaces can be specified in an IDL or as a part of the

programming language being used. Where necessary, the support structure will then generate appropriate language specific interfaces to interact with the implementations. (A client stub would simply be another implementation which the interface can use).

Having considered the model of component separation, the following section will describe how we can use object-orientation techniques to model this separation of interface from implementation.

## 2.2 Separation of interface and implementation

In an object-oriented programming language, objects are instances of *abstract types (classes)*. A class consists of an interface, which defines the operations provided by the class, and an implementation of those operations. Because we want to strongly separate interfaces from implementations, this is best achieved by mapping the them into separate classes: *interface classes* and *implementation classes*.

Object-orientation allows us to specify the binding between interface class and implementation class in the following ways:

- *Class-based inheritance*: whole classes are related by inheritance. The pattern of inheritance is fixed when the classes are created [5].
- *Delegation*: objects can be individually related, enabling each object to make its own decision as to when, and to what, it delegates. The pattern of inheritance can vary dynamically, making delegation a more flexible and powerful way of organising objects [6].

Section 2 discussed the desirability of being able to control each binding of interface class to implementation class to improve software flexibility. Therefore, implementation delegation best matches our requirements: interfaces classes are typically simple, defining the public operations for a conformant set of implementation classes, and delegating much of the functionality to the implementation class. *Interface inheritance* is still possible, providing dynamic *implementation inheritance*.

Therefore, to provide this flexibility we require the binding between interface classes and implementation classes to be evaluated when the interface class is instantiated. Because we wish to leave this binding until runtime, we must specify it as data, and not within the code of the interface class. The instance of the interface class (*interface object*) uses this data to create and bind to the correct instance of the implementation class (*implementation object*).

Because interfaces can be bound to different implementations, the operations provided by the interface class may not reflect all of the operations provided by an implementation class. For example, an interface class to a message passing layer may not provide operations for changing the time-out and retry values, although an implementation class may provide this functionality. Therefore, to allow access to implementation specific operations, an implementation class can provide *control class(es)*, that provide corresponding operations [2][7]. Control classes, which can be common to a set of implementation classes, allow the manipulation of the non-functional characteristics of an implementation. Interface classes possess an operation through which an instance of this control class (*control object*) can be obtained.

In the following section the **Gandiva** software development platform will be described, which supports this model of separation of interface classes and implementation classes for C++.

### 3 The **Gandiva** software development system

**Gandiva** is a software development system that provides support for the construction of C++ software systems using the ideas presented in section 2. It provides a set of C++ classes to facilitate the separation of interface classes from implementation classes. An important part of our design was to provide a portable system, and therefore we have not modified the language in supporting these features. In addition, **Gandiva** has been written using the same design model, so software components have been implemented using the same separation techniques.

#### 3.1 Support for interface and implementation separation in C++

Although C++ is an object-oriented language, one of its non-object-oriented features means it does not lend itself naturally to the separation of interfaces from implementations: implementation specific information, such as private member variables and functions, appears in class definitions, tying interfaces to implementations. Changes to this private information require all code that depends on the class to be rebuilt, even if the public interface does not alter. Therefore, to provide a strong separation between interface and implementation without modifying the language, restrictions must be placed on interface classes, e.g., no public variables or friends, which are implementation specific.

To provide the separation of interface and implementation requires changing what would have been a single C++ class into a number of classes:

- The *interface class*: users interact with instances of this class, that defines the public operations that can be invoked on the implementation. The only implementation specific information present in the class definition is a single member variable: a pointer to an instance of an *implementation interface class*, to which the interface delegates all operations performed upon it.
- The *implementation interface class*: this class provides the interface class with an interface to the *implementation classes*, which are derived from the implementation interface class. The operations of implementation interface classes are pure virtual functions, which means that they must be defined in a derived class.
- The *implementation class*: instances of this class represent the implementation of an object. All implementation classes to be used by a specific interface are derived from the corresponding *implementation interface class*. Implementation classes can be derived from multiple implementation interface classes.
- The *control class*: this class provides access to operations that manipulate the non-functional characteristics of an implementation class. Implementation classes provide an operation that returns a specific instance of this control class. Interface classes provide an operation that can be used to request an instance of the implementation's control class.

When an object is instantiated by a user this results in at least two objects being created: an *interface object*, and an *implementation object*. An interface object interacts with its implementation object as an instance of the implementation interface class, relying upon inheritance to invoke the correct operation. This indirection means that the interface has no implementation

specific information, and the same interface can be used to bind to any conformant implementation.

Figure 3.1 shows an object structure formed by the above classes, where the implementation specific objects are shown in grey.

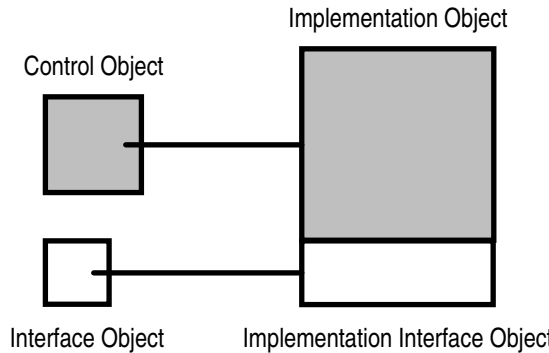


Figure 3.1, Interface, Implementation and Control Objects.

As we have mentioned, it is possible for an implementation class to be derived from many different implementation interface classes. As a result an implementation object can provide the implementation for many interface objects. Figure 3.2 illustrates this and also how an implementation object may provide multiple control objects.

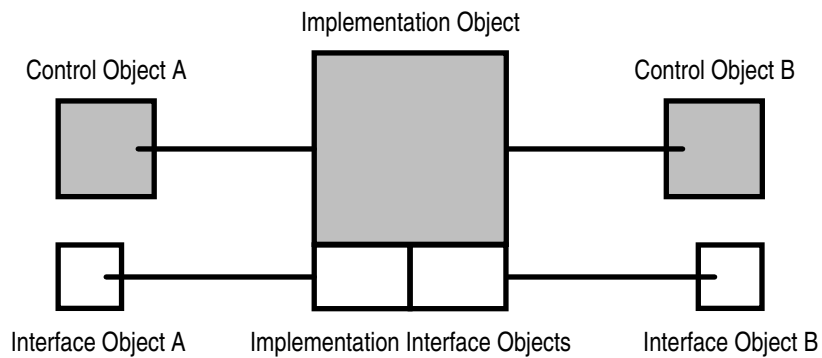


Figure 3.2, Multiple Interfaces to a single Implementation.

### 3.1.1 Example interface class

In this section we shall briefly illustrate some of the basics of building a software component in **Gandiva**. To do this we shall examine the construction of a name service object, the use of which will be described in section 3.1.2.

Construct the interface class `NameService`, and implement the operations through delegation. The code fragment below shows some of the interface class, and the implementation of the `getAttribute` method. For simplicity, the constructors which bind the interface to the implementation are not shown.

```

class NameService : public virtual Resource
{
public:
    Boolean getAttribute(char *objName, char *attrName, int &val);

    . . .
private:
    NameServiceImple*_imple;
};

Boolean NameService::getAttribute(char *objName, char *attrName,
                                int &val)
{
    if (_imple != NULL)
        return _imple->getAttribute(objName, attrName, value);
    else
        return FALSE;
}

```

Construct the implementation interface class, which specifies the operations that must be provided by all implementations. The operations of the class, which we will call `NameServiceImple`, are pure virtual functions.

```

class NameServiceImple : public virtual Resource
{
public:
    virtual Boolean getAttribute(char *objName, char *attrName,
                                int &val) = 0;

    . . .
};

```

Construct the implementation class(es), by inheriting from `NameServiceImple`. In this example a local database implementation is being provided, so the implementation class `LocalDBNameServiceImple` is shown.

```

class LocalDBNameServiceImple : public NameServiceImple
{
public:
    virtual Boolean getAttribute(char *objName, char *attrName,
                                int &val);

    . . .
};

```

In the following section we shall examine the classes which **Gandiva** provides to aid in the construction of classes using this model.

### 3.1.2 Gandiva support classes

**Gandiva** provides a set of classes to support the construction and use of interface and implementation classes. The resulting class hierarchy is shown in figure 3.3.

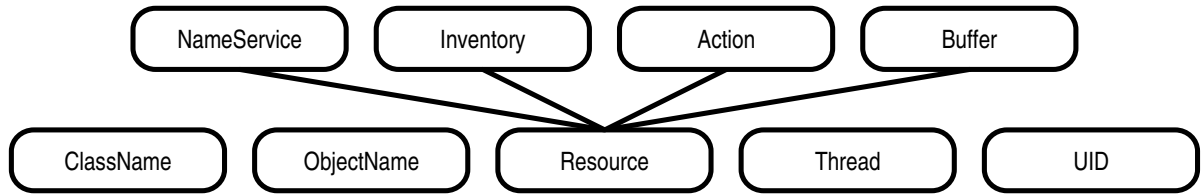


Figure 3.3, The **Gandiva** class hierarchy.

The classes `UID` and `Thread` are not of importance to this discussion, providing unique identifiers and parallel threads of execution, respectively. We shall now examine each of the remaining classes, and indicate their roles in supporting our design model. Some of the classes shown are actually multiple classes, representing interfaces and implementations.

- `ClassName`: in order to provide support for the separation of interfaces and implementations we require a run-time type system, which is provided by instances of this class. Each class is represented by an instance of `ClassName`, and these objects support basic operations such as equality and inequality. (We are currently investigating the use of the new run-time type systems in C++ [8]). This class is primarily used by the interface classes for run-time binding to implementation classes.
- `ObjectName`: we require a means whereby the mapping of interface classes to implementation classes can be specified and stored between successive instantiations of interfaces. This is provided by an instance of `ObjectName`, which is an abstract name and an associated resolution mechanism. This resolution mechanism uses the `NameService` class. The mappings are stored according to this resolution mechanism, and can be retrieved when required by invoking appropriate operations on the `ObjectName`.
- `Resource`: the lack of garbage collection in C++ means that it can be difficult to know when objects are no longer required and can be destroyed. The `Resource` class provides a means of reference counting instances of classes derived from it, and only allows deletion when they are no longer used. In addition, because the `Inventory` can be used to create instances of any class, it must treat these objects as instances of the `Resource` class. Therefore `Resource`, and the classes derived from it, provide *castup* operations to enable objects to be safely cast up their inheritance hierarchy. The `Resource` class has some correspondence to the `Object` class of the NIH library [9] and the `Resource` class of `InterViews` [10].
- `Action`: this class is not directly related to the separation of interfaces from implementations, but will be used in a later section to illustrate our model. Instances of this class are used to define scopes within an application. This class is intended to be applicable for a large range of actions, such as display update actions, resource acquisition actions, and, as we shall show later, atomic actions (atomic transactions).
- `Buffer`: used to support the conversion of a series of basic types and objects to and from a form that can be transferred across the network or placed in secondary storage. There is an interface class and several corresponding implementations, and is used by `UID`, `ClassName` and `ObjectName`.

- **Inventory:** this is an interface class and a single implementation class. An instance of the implementation class represents the core of the system which supports the interface and implementation separation. It provide a mechanism for the dynamic creation of objects based upon their `ClassName`.
- **NameService:** the interface class uses one of its implementation classes to provide access to a name resolution mechanism.

In summary, the inventory maintains an association of `ClassName` to object creation mechanism. Hence the inventory is the core component in **Gandiva** that supports the separation of interface from implementation. Interface objects are typically created using an instance of an `ObjectName`. The interface then interrogates the `ObjectName` to determine the `ClassName` of the desired implementation class. By then presenting this class name to the inventory, an instance of this implementation class can be created and bound to the interface. The `ObjectName` manipulates data obtained via the name service interface component, and therefore to modify the binding only requires changing this data.

### 3.2 Support for modules

By grouping related components into modules, we can further improve the flexibility and extensibility of software systems, making components more generally useful. **Gandiva** also provides support for the structuring of components into modules, and the construction of applications from these modules. Application builders select the set of modules that they require for an application, and then makefiles, which transparently provide access to these modules and their components, are automatically generated using `imake`. The application code is written in a way that does not reflect the number of modules available, which means that the application builder does not need any specific information about the environment in which the application will eventually be built.

## 4 Case study: the design of an atomic object support system

The motivation behind the development of this software design model is the construction of extensible fault-tolerant distributed applications. One of the areas we are examining is the provision of an atomic object support system, and we will use this as the case study to illustrate the extensibility of our approach. Before we describe the details of our design, it is necessary to explain the purpose of an atomic object support system.

An atomic object support system allows the construction of fault-tolerant applications, containing atomic objects. The operations on these objects are performed as atomic actions (atomic transactions), and groups of these operations can also be performed as atomic actions. Atomic actions have the well known properties of serialisibility, failure atomicity, and permanence of effect. Applications constructed using atomic actions can therefore maintain the consistency of atomic objects despite node failures and concurrent accesses.

To implement these properties, the atomic object support system must monitor the operations performed on atomic objects and the beginning and ending of atomic actions. If an operation on an object will compromise one of the above properties, the

system either informs the operation that no further accesses to the object should be made, or prevents any effects from the operation becoming visible.

#### 4.1 Design of the atomic object support system

The atomic object support system is required to be extensible for several reasons; we enumerate some of them here:

- An application, designed initially for a single node, may need to be distributed, permitting uniform access to local and remote objects.
- Objects have different concurrency control requirements, so the system should be able to support these, e.g., pessimistic and optimistic.
- The atomic action structure may need to be extended to provide more flexible structures, such as split transactions [11], glued actions and coloured actions [12].

Therefore, the first stage in the design of the atomic object support system was to design the interface components which will isolate applications from the implementation components which make up the support system. This will allow us to alter the support system implementation without affecting applications. To design these interface components we must first examine the monitoring role played by the support system on atomic objects and atomic actions.

- The events of interest resulting from atomic actions are their beginning and ending. In effect, the ending of an atomic action may result in multiple events due to the use of the two-phase commit protocol, i.e., prepare, commit or abort events.
- The events of interest resulting from atomic objects are their creation or deletion, and attempts to: examine, update or overwrite their states. To maintain serialisability, the support system must prevent the simultaneous updating of an object from different atomic actions. Therefore, in some circumstances the support system is required to block such operations, or in the case of an “optimistic” implementation, to check for conflicts when the action attempts to commit.

In response to these events, the support system may also generate events, such as: loading the state of an object from stable store, saving the state of an object to stable store, restoring the state of an object and obtaining the state of an object.

Figure 4.1, illustrates the structure of the atomic object support system and the events that can occur within it. The *atomic event manager* co-ordinates the requests to examine, update and overwrite atomic objects, with the commit processing of atomic actions. The atomic event manager is also responsible for saving and restoring the states of objects for recovery purposes, and saving and loading the states of objects to and from stable store to ensure state changes are persistent.

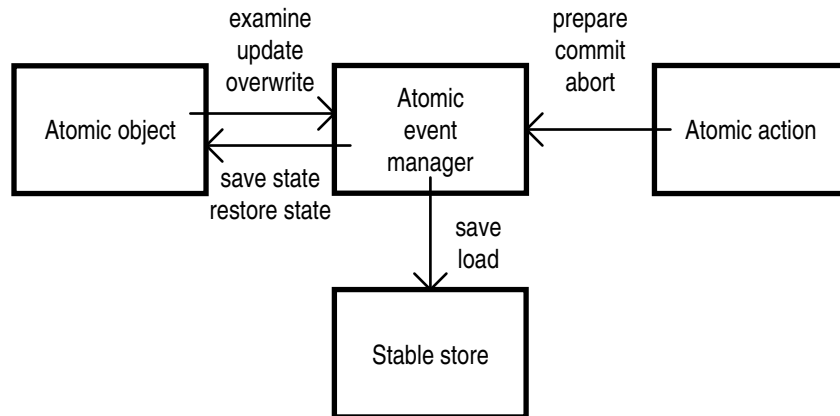


Figure 4.1, Atomic object support system events.

Because we want to support many implementations of the support system, we require that the way in which atomic actions and atomic objects interact with the atomic event manager is independent of its implementation. Therefore, to support this interaction, the atomic event manager is composed of two interface components, the *atomic action manager* and *atomic object manager*. These interface components are mapped into two classes: `AtomicActionManager` and `AtomicObjectManager`.

The following sections will describe these classes and the classes that provide atomic actions and atomic objects, `AtomicAction` and `AtomicObject`.

## 4.2 AtomicActionManager and AtomicAction classes

The `AtomicAction` class, which is derived from the **Gandiva** `Action` class described in section 3.1.2, is used by an application to define the scopes of atomic actions, by using the operations `begin()`, `commit()` and `abort()`.

```

class AtomicAction : public Action
{
public:
    Boolean begin();
    Boolean commit();
    Boolean abort();
    . . .
};
  
```

Shown below is an example of the use of the `AtomicAction` class, where both nested and top-level atomic actions are created:

```

AtomicAction a, b;
Boolean res1, res2;

a.begin(); // begin top-level action
  b.begin(); // begin nested action
    res1 = oper1();
  if (res1)
    res1 = b.commit(); // commit nested action
  else
    b.abort(); // abort nested action
  res2 = oper2();

if (res2)
  res2 = a.commit(); // commit top-level action
else
  a.abort(); // abort top-level action
  
```

The work that is carried out when an atomic action ends is dependent upon the events that have occurred over its lifetime. Therefore, the `AtomicAction` object maintains a list of `AtomicActionManager` objects, which are processed when the atomic action ends. During the execution of an atomic action instances of `AtomicActionManager` may be added to the atomic action list in response to specific events. The processing that is performed on the list when the action ends differs depending on whether the action commits or aborts. If it commits, the processing of this list takes the form of a two-phase commit protocol, using the `prepare()` and `commit()` operations. If it aborts the `abort()` operation is called.

```
class AtomicActionManager : public Resource
{
public:
    Boolean prepare();
    Boolean commit();
    Boolean abort();

    . . .
};
```

Because the `AtomicActionManager` interface hides the actual implementation, the `AtomicAction` class does not need to know either the reason that an `AtomicActionManager` object was added to its list or what task the `AtomicActionManager` object must perform when that atomic action ends. This enables us to provide extensibility for other events which we do not yet know about.

### 4.3 AtomicObjectManager and AtomicObject classes

The purpose of the `AtomicObject` class is to provide a means by which an application object can be made "atomic". The `AtomicObject` class supports state based recovery and persistence of objects. Any application class that is required to be atomic must be derived from `AtomicObject`. The `AtomicObject` class provides `examine()`, `update()` and `overwrite()` operations that are called to indicate to the atomic event manager that the object is about to be examined, updated or overwritten, respectively. The request can be blocked if the operation returns *false*. The support for saving and restoring the object's state is performed via the `saveState()` and `restoreState()` operations, which must be redefined by the application object.

`AtomicObject` provides two constructors: the default constructor is used when creating new atomic objects. The `objectName()` operation is used to obtain the object name of the newly created object for later recreation. The second constructor is used to recreate an atomic object which had been created before, with the `objectName` being used to identify the particular object.

```
class AtomicObject : public Resource
{
public:
    const ObjectName &objectName();

    virtual Boolean saveState(ObjectState &os) = 0;
    virtual Boolean restoreState(ObjectState &os) = 0;

    . . .
protected:
    AtomicObject();
    AtomicObject(ObjectName &objectName);
```

```

        Boolean examine();
        Boolean update();
        Boolean overwrite();

        . . .
};

```

To allow alternative implementations of the `examine()`, `update()` and `overwrite()` operations, the `AtomicObject` class contains an instance of the `AtomicObjectManager` class, through which these operations are indirected. To allow the atomic event manager to monitor the creation and deletion of atomic objects, when the atomic object is created it must *connect* to the `AtomicObjectManager` and *disconnect* when it is deleted.

```

class AtomicObjectManager : public Resource
{
public:
    AtomicObjectManager();
    AtomicObjectManager(ObjectName &objectName);

    Boolean connect(AtomicObject *atomicObject);
    Boolean disconnect(AtomicObject *atomicObject);

    Boolean examine();
    Boolean update();
    Boolean overwrite();

    . . .
};

```

To summarise, the `AtomicObject` and `AtomicAction` classes are used to generate events which are handled by the atomic event manager, which comprises the `AtomicActionManager` and the `AtomicObjectManager` classes. To allow extensibility the application should not make assumptions about how these events are processed. The support system interacts with the atomic event manager through interfaces, which allow events to be dealt with in a generic manner.

#### 4.4 PersistentObjectState class

The atomic event manager is responsible for the loading and saving of an object's state to and from stable storage. The implementation of this is isolated from the atomic event manager by the persistent object state interface. Implementations for this interface may be based on a variety of techniques, for example: simple files, replicated files and commercial databases.

```

class PersistentObjectState : public Resource
{
public:
    enum Outcome { done, notDone, unknown };

    Outcome save(int index, Buffer *buffer);
    Outcome load(int index, Buffer *&buffer);
    Outcome synchronize();

    . . .
};

```

Figure 4.2, illustrates the resulting class structure of the interface components of the atomic object support system. The `AtomicObjectManager` class and `AtomicActionManager` class share many of the same implementation classes. These implementation classes are referred to as `AtomicManager` classes, and are shown in grey in the figure.

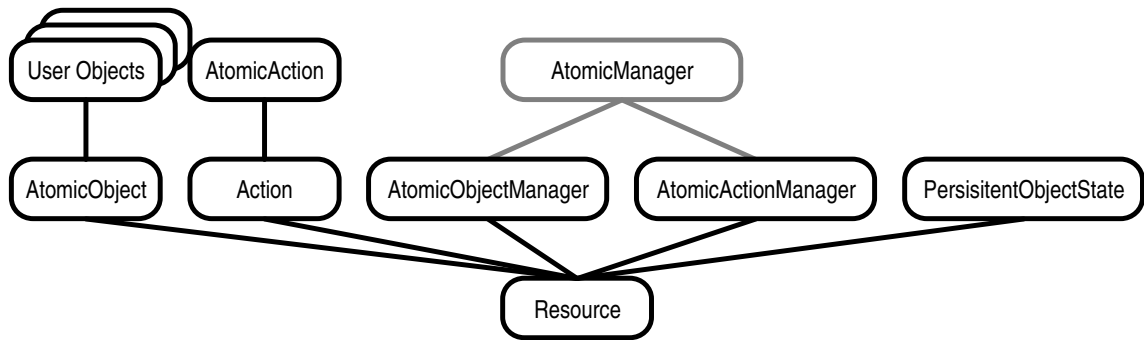


Figure 4.2, Atomic object support system class inheritance.

#### 4.5 Initial implementation

This approach to the designing of the atomic object support system allows a wide variety of implementations to be provided. Applications can be constructed that use multiple implementations, so allowing applications to pick the most suitable implementation of the support system for their needs.

One of the most important aspects of an implementation of the support system is the *object model*, which specifies the relationship between passive persistent object states (on stable storage) and active objects (objects in memory which are capable of having operations performed on them). The object model an implementation supports has a significant effect on the availability and performance of the atomic objects. Described below are some possibilities:

- For each persistent object state there exists at most a single active object: this means that no co-ordination is required to maintain the properties of serialisability, failure atomicity and permanence of effect. This model can provide high performance, but the service will become unavailable if the process which contains the active object fails. This model will be referred to as the *solo model*.
- For each persistent object state there can exist many active objects, co-located on the same node: the co-ordination required can be performed via fast single node inter-process communication mechanisms such as shared memory. This model can tolerate the failure of a process containing an active object, but not the failure of the entire node. This model will be referred to as the *multiple model*.
- For each persistent object state there can exist many active objects, arbitrarily located as the application desires: the co-ordination required must be performed via relatively slow inter-node communication mechanisms, such as message passing. This model can tolerate the failure of multiple nodes containing the active objects. This model will be referred to as the *arbitrary model*.

The solo and multiple object models have been implemented using pessimistic concurrency control. The object structure of the resulting implementations is illustrated in figure 4.3. The application object (grey) is shown derived from `AtomicObject`, which contains an instance of `AtomicObjectManager`, that forms the interface to the atomic event manager. Note that in these implementations, the concurrency control (CC), persistence (P) and recovery (R) management have been placed in separate objects. This structure increases the extensibility of the implementation, allowing selective replacement. The co-ordinating atomic object

manager simply calls each in turn to see if, for example, an update request should be allowed. The atomic action object is shown containing references to the three atomic manager objects within its list.

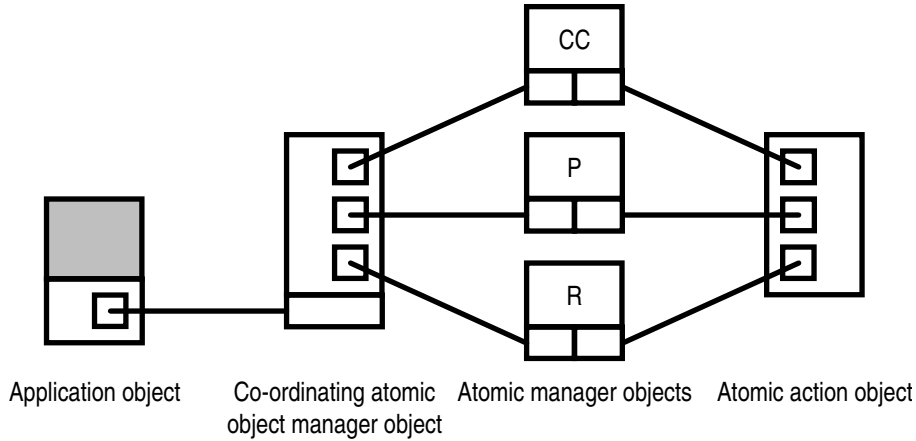


Figure 4.3, Object structure of the atomic object support system.

#### 4.6 Assessment

The performance figures from the implementations of the solo and multiple object models have been obtained, to evaluate the differences between the two object models. The figures show the rate at which examine, update and overwrite operations can be performed, within a top-level atomic action and a nested atomic action. The result are presented in the table below:

Model	Environment	examine / sec	update / sec	overwrite / sec
Solo	Top-level atomic action	1330	23.5	23.5
	Nested atomic action	925	450	445
Multiple	Top-level atomic action	270	23.5	23.5
	Nested atomic action	550	345	340

All performance figures were obtained on a lightly loaded SPARCstation LX running Solaris 2.3, for a small atomic object (the state consisted of a single integer).

The performance figure show that, as expected, the solo object model provides either better or identical performance to that obtained from the multiple object model. But as stated earlier, the solo object model has worse availability characteristic than the multiple object model. Therefore, an application designer can choose the atomic object support system implementation that suits the atomic objects within the application. If high availability is important the support provided by multiple object model implementation is appropriate. If high performance is important the support provided by solo object model implementation is more appropriate.

The atomic object support system described in the previous section is intended for the next version of the Arjuna system [13][14]. The emphasis on extensibility in the new design is because the atomic object support system in Arjuna was found to be restrictive. Arjuna uses inheritance for the construction of atomic objects, and this has proven a powerful mechanism for the construction of fault-tolerant applications. However this takes the form of implementation inheritance, making it difficult to provide any flexibility in the atomic object support system. Arjuna was constructed in a modular manner [15], but the granularity of modularity is generally too coarse and in some cases strong inter-dependencies exist between modules.

## 5 Related work

There are a number of systems that have been developed based upon similar ideas to those we have presented in this paper. In the following sections we attempt to compare and contrast some of them with our work.

### 5.1 Static model

In [16], Coplien proposes a separation of interface from implementation for C++. The model is based upon combining the interface and the conformant base class, and encoding the types of implementations as an enumeration type. When users create instances of the interface they specify, as a parameter to the constructor, the implementation type by the appropriate enumeration value. The code for the interface constructor then maps the enumeration value to an explicit creation of the required implementation. Although this does allow dynamic binding of interface object to implementation object, it suffers from the problem that all implementation classes must be known to the interface class. The code for them must be available when the application is built and adding new implementations requires modification of the enumeration type, the interface, and the applications.

InterViews is an object-oriented programming system that supports the construction of graphical user interfaces in C++ [10]. The InterViews system is designed to allow the programmer to deal with “abstract” graphical entities such as buttons, labels, menus, scroll bars, etc., without knowledge of the details of their “look and feel”. The actual implementation of the entities is defined by the environment in which the application is executed. For example, if a graphical entity is to be displayed on a black and white screen, then an implementation will be chosen to provide the rendering that is suitable for black and white. This flexibility is achieved through the use of “kits”. Kits are used to obtain instances of the objects that correspond to the graphical entities, whose implementations suit the environment. The problem with kits is that they are designed to support only a small set of object classes, and are not extensible.

### 5.2 Language modifications

In [17], Martin describes the separation of interfaces and implementations with the aid of a modified C++ pre-processor. New language keywords of `interface` and `implements` are provided by the pre-processor and are used by programmers to specify interfaces and implementations respectively. The `reuses` keyword is also provided to allow implementations to be used in other class hierarchies. However, interfaces are simply a means of ensuring conformance of implementations, and are not used by the programmer, who

must still explicitly instantiate objects of the correct (real) type. The author states that objects should be accessed through their base type and the code which uses them isolated to improve modularity, but no support for this is provided in the scheme. Changes to implementation classes, interface classes, or application requirements, will still require the rebuilding of these applications.

The OpenC++ system described in [18] achieves an “open system” architecture through *reflection* [19]. Classes can be *reified* and method invocation controlled through a *meta-object protocol (MOP)* which can be redefined by users as desired. For example, a MOP can be defined which causes invocations of a specific method to be executed on a distributed replica group, rather than on a single local object. The system is based upon a modified pre-processor, with the meta-object protocol hiding the object (method) implementation between two pre-processor created stub objects. These marshal and unmarshal parameters and call the correct implementation method based upon the controlling meta-object. This is a large overhead on method calls in the case of local invocations (the authors quote at least 10 times slower than a virtual method call). The authors claim that since OpenC++ is intended for distributed applications it should not affect overall performance. There is no direct support for dynamic binding of interfaces to implementations, and modular software construction is not addressed. The MOP is statically created for a specific purpose, and changing the MOP would require rebuilding the application and/or the component to be controlled.

The Shared Object Model (SOM) provides a limited form of interface and implementation separation in C++ [20]. By modifying the compiler and linker, and introducing the `extern "shared"` linkage directive, applications can be compiled against one version of a class definition (essentially the interface), and a different version of the class (the implementation) could be provided by one of the libraries. The linker then performs the necessary binding between the two. The interfaces are “traditional” C++ definitions, containing both public and private information. Therefore, to guarantee that applications continue to work despite the linking of different versions of classes, limitations are imposed on how classes can vary between versions, e.g., class changes must be “upwardly” compatible. Although SOM attempts to improve the modularity of C++, the restrictions it must impose on class development limits its overall usefulness: because an interface is still tied to a (range of essentially similar) implementations, it would not be possible for it to bind to a completely different type, even though the implementation may possess a conformant public interface.

### 5.3 Operating systems

The Spring system is an experimental distributed environment developed by Sun Microsystems [3]. It includes a distributed operating system and a support framework for distributed applications. The main focus during its development was on the evolution and extensibility of the system using the separation of interface and implementation. Although the system is written in C++, all key interfaces are defined in a separate interface definition language [4]. The IDL is object-oriented, including support for multiple inheritance. The support structure for this language generates surrogate objects (essentially C++ interface classes) from these IDL descriptions. All interfaces in Spring are versioned, with major and minor version numbers, and the system supports a

limited form of dynamic binding of interface object to implementation object: when an interface requests a binding to a specific implementation it will either be given that implementation or one which is version compatible. However, since the IDL interface is tied to specific versions of a class, binding to completely different implementation types would not be possible, and once bound, an interface/implementation relationship cannot be changed during the lifetime of the interface.

## 6 Conclusions

Separating software components into their interface and implementation components provides flexibility and extensibility in the design and implementation of software components. This separation model is independent of a specific language, but object-orientation provides a natural framework in which it can be realised, by separating object interfaces from their implementations. We have shown how this model can be translated into C++, by converting what would originally have been a single class into several classes: the interface and implementation classes. Although we have talked in terms of C++, it would also be possible for software developers to specify interfaces in an IDL, and use an appropriate code generator to create the required C++.

## Acknowledgements

We would like to thank our colleagues on the Arjuna project, Santosh Shrivastava, Graham Parrington, Steve Caughey, David Ingham, and Jim Smith, for commenting on earlier drafts of this paper. The work reported here has been supported in part by grants from the UK Ministry of Defence, Engineering and Physical Sciences Research Council (Grant Number GR/H81078) and ESPRIT project BROADCAST (Basic Research Project Number 6360).

## References

- [1] "Advanced Network Systems Architecture (ANSA) Reference Manual", Volume A, Release 1.00, Part VI, Computational Projection, March 1989.
- [2] International Standard ITU-T Recommendation, "Open Distributed Processing Reference Model Part 3: Architecture", Draft of 27<sup>th</sup> February 1995.
- [3] G. Hamilton and S. Radia, "Using Interface Inheritance to Address Problems in System Software Evolution", Proceedings of the ACM Workshop on Interface Definition Languages 1994.
- [4] OMG, "Common Object Request Broker Architecture and Specification", OMG Document Number 91.12.1.
- [5] A. Snyder, "Inheritance and the development of encapsulated software components", Research directions in object-oriented programming, MIT Press, Cambridge, Massachusetts, 1987, pp. 165-188.
- [6] M. Wolczko, "Encapsulation Delegation and Inheritance in Object-oriented Languages", Software Engineering Journal, pp. 95 - 101, March 1992.
- [7] G. Kiczales, "Towards a New Model of Abstraction in Software Engineering", Proceedings of the International Workshop on Reflection and Meta-Level Architecture, Tama-City, Tokyo, November 1992.
- [8] B. Stroustrup, "The Design and Evolution of C++", Addison Wesley, 1994.
- [9] K. Gorlen, "An Object-Oriented Class Library for C++ Programs", Software-Practice and Experience Vol. 17, No. 12, pp. 899-922, 1989.

- [10] M. A. Linton *et al*, “InterViews Reference Manual Version 3.1”, Stanford University, December 1992.
- [11] C. Pu, G. Kaiser and N. Hutchinson, “Split-transactions for open-ended activities”, Proceedings of the 14<sup>th</sup> VLDB Conference, Los Angeles, pp. 26-37, September 1988.
- [12] S. K. Shrivastava, and S. M. Wheeler, “Implementation Fault-Tolerant Distributed Applications using Objects and Multi-Coloured Actions”, Proceedings of the Tenth International Conference on Distributed Computing Systems, pp. 203-210, Paris, France, May 1990.
- [13] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington, “An Overview of Arjuna: A Programming System for Reliable Distributed Computing”, IEEE Software, Vol. 8 No. 1, pp. 63-73, January 1991.
- [14] G. D. Parrington, S. K. Shrivastava, S. M. Wheeler and M. C. Little, “The Design and Implementation of Arjuna”, BROADCAST Project Technical Report, 65, September 1994. (Available from the Department of Computing Science, University of Newcastle upon Tyne, UK).
- [15] S. K. Shrivastava and D. L. McCue, “Structuring Fault-tolerant Object Systems for Modularity in a Distributed Environment”, IEEE Transactions on Parallel Distributed Systems, Vol. 5, No. 4, pp. 421-432, April 1994.
- [16] J. O. Coplien, “Advanced C++ Programming Styles and Idioms”, Addison Wesley, 1992.
- [17] B. Martin, “The Separation of Interface and Implementation in C++”, Proceedings of the USENIX C++ conference, pp. 51-63, Washington D.C., April 1991.
- [18] S. Chiba and T. Masuda, “Designing an Extensible Distributed Language with a Meta-Level Architecture”, Proceedings of ECOOP 93, 1993.
- [19] R. Stroud, “Transparency and Reflection in Distributed Systems”, Operating Systems Review, Vol. 27, pp. 99-103, April 1993.
- [20] T. C. Goldstein and A. D. Sloane, “The Object Binary Interface - C++ Objects for Evolvable Shared Class Libraries”, SMLI TR-94-26, June 1994.