The Design and Implementation of a Framework for Configurable Software

Stuart M. Wheater and Mark C. Little
Department of Computing Science,
The University of Newcastle upon Tyne,
Newcastle upon Tyne,
NE1 7RU, UK.

Appeared in the Proceedings of the IEEE Third International Conference on Configurable Distributed Systems, May 1996

Abstract

Software systems are typically composed of numerous components, each of which is responsible for a different function, e.g., one component may be responsible for remote communication, while another may provide a graphical user interface. Different implementations of a component may be possible, with each implementation tailored for a specific set of applications or environments. Being able to reconfigure software systems to make use of these different implementations with the minimum of effect on existing users and applications is desirable. Configurable software systems are also important for a number of other reasons: additional components or modifications to those currently available, may be required. For example, new versions of software components may be necessary due to the discovery of design flaws in a component; a RPC which provides unreliable message delivery may be suitable for an application in a local area network, but if the application is to be used in a wide area network, a different RPC implementation, which guarantees message delivery, may be necessary. Therefore, software is often required to be configurable, enabling modifications to occur with minimal effect on existing users. To allow this configurability, 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 configurable software based upon this separation, and illustrates this with a software development system we have implemented which supports these ideas in C++.

Keywords: configurability, extensibility, modularity, object-oriented.

1. Introduction

Software systems are typically composed of numerous components, each of which may perform a different function, e.g., a RPC component which is used for remote communication, and an atomic action component to guarantee consistency of data in the presence of failures. There may be many different ways of implementing the functionality provided by a component, each implementation may be useful for a specific set of applications. Although initially an application may be built to use a specific implementation, it is possible that over the lifetime of the application a different implementation may be required. For example, a RPC which provides unreliable message delivery may be suitable for an application in a local area network, but if the application is to be used in a wide area network, a different RPC implementation, which guarantees message delivery, may be more suitable. Errors in implementations may also require changes for the application. Ideally we would require that changes in the components and/or application requirements would not necessitate rebuilding the applications.

We believe that the configurability required from software can be obtained by providing a framework which supports the development of extensible software components. 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 configurable, 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 configurable 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 we have developed. Section 2.2 will then illustrate how this can be modeled using object-oriented techniques.

2.1 Model

In the model software components are split into 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, which are independent software components. A single interface can be used to access multiple implementations simultaneously, 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, but these interfaces may not be able to benefit from the additional features, 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 configurable, 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 removing an appropriate subset of the available implementations, possibly replacing them with dummy implementations which return error messages to the user.

This separation of interfaces from implementations is not new, with much work done on Interface Definition Languages (IDL) [4]. However, IDLs are typically used in the context of distributed applications. The interface to a remote service is specified in terms of the IDL which is then used to generate appropriate client and server stub code (the implementation). The applications are then written in terms of this (static) implementation. In our model, the interface to the interface component 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 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 the binding of interface class to implementation class to improve software configurability. 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 most 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 run-time, 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 reliable 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 construction 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 support the construction 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 four classes:

(i) The *interface class*: users interact with instances of this class, which 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.

(ii) 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.

(iii) 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.

(iv) 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.

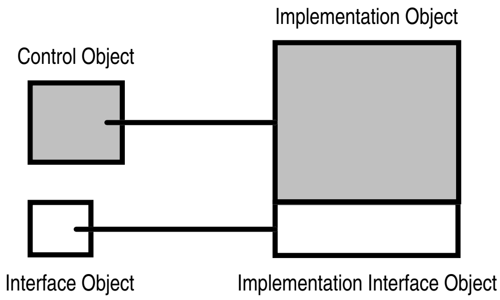

Figure 3.1, Interface, Implementation and Control Objects.

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

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.

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.

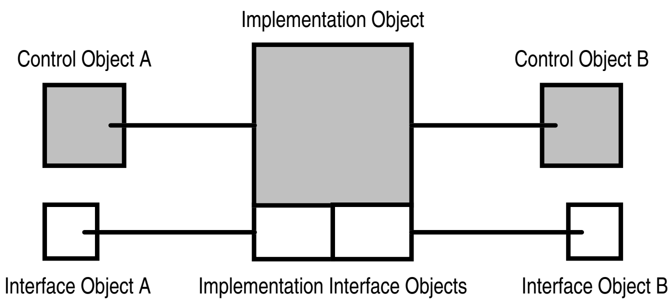

Figure 3.2, Multiple Interfaces to a single Implementation.

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

3.1.1 `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.

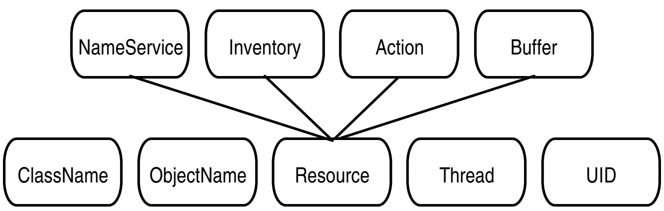

*Figure 3.3, The **`Gandiva`** class hierarchy.*

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 there 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, i.e., a means of saving this configuration information. 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`. Instances of `ObjectName`s represent the main store for configuration information. An object uses the attributes of an `ObjectName` to determine the type of its implementation; this implementation will also use the `ObjectName` to determine the `ObjectName`s of the objects the implementation contains. By changing the attributes of an `ObjectName` the configuration of the corresponding object can be altered.

- `NameService`: the interface class uses one of its implementation classes to provide access to a name resolution mechanism.

- `Inventory`: this is an interface class and a set of implementation classes. 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`.

- `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).

- `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].

- `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`.

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 motivation behind the development of this software design model is the construction of configurable 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 our main case study to illustrate the configurability of our approach. However, we shall first describe the implementation of our configurable remote communication mechanism, which will help illustrate the use of the `ObjectName` in our system.

4.1 Remote communication

In a distributed environment objects may communicate with each other using a remote procedure call mechanism (RPC). The purpose of a RPC mechanism is to maintain, with appropriate client and server stub code, the abstraction of local procedure calls across address space boundaries. There are a number of RPC implementations providing different functionality, e.g., reliable message delivery, or group communication [21]. In most distributed systems only a single implementation is provided, which all objects must use [1][14]. Because of different application requirements, systems such as ISIS [22] provide a number of different implementations, e.g., reliable causally ordered, or reliable globally ordered. However, each of these implementations has a different interface and application programmers must choose the correct interface when building applications. Therefore modifications in application requirements require changes in application code.

We believe that it is necessary for distributed systems to support different communication semantics and that application requirements may alter, necessitating a change in the communication mechanism used, possibly on a per object basis. We require that, as far as possible, these changes should not mean changes in the application code. This has obvious advantages, allowing programmers to build applications which can be used in a range of environments, and to simply experiment with different implementations. This configurability requirement means that objects should interact with the communication mechanisms through an interface which does not imply a specific implementation. This is achieved through the `Dispatcher` interface class.

```
class Dispatcher : public Resource
{
public:
 enum OutCome { DONE, NOTDONE, UNKNOWN };

 Dispatcher(const ObjectName &objectName);
 virtual ~Dispatcher();

 OutCome dispatch(Array<Buffer*> work,
                  Array<Buffer*> &result);
  ...
};
```

Users call the `dispatch` method with an array of work `Buffers` and expect to be returned an array of result `Buffers`. Each dispatcher implementation can interpret the `Buffers` differently. The interface conveys no information about how the implementation works: no network communication need be involved at all. This interface encourages a layered (hierarchical) design of RPC, where each layer is represented by a dispatcher providing a specific functionality. Each dispatcher in the hierarchy performs some implementation specific work and then forwards the message `Buffers` to another dispatcher.

Figure 4.1 illustrates this with a basic dispatcher hierarchy consisting of the following dispatcher implementations: an *Operation Dispatcher*, which packs/unpacks information identifying the remote method to invoke, and a *Network Dispatcher* which is responsible for sending/receiving messages. The direction of the messages for a client sending a request to the object is also shown.

Using this model, distributed services are represented by a dispatcher hierarchy. These hierarchies are configured when created, through an appropriate `ObjectName` instance, and can be reconfigured dynamically. As an example, we shall consider a basic RPC hierarchy, and two possible dispatcher implementations: `TCPDispatcher` which guarantees messages delivery provided sender and receiver do not crash, and `UDPDispatcher` with which no guarantees are provided for the delivery of messages.

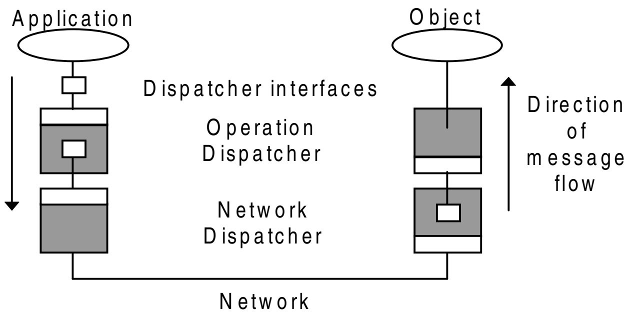

Figure 4.1, Basic dispatcher hierarchy.

When the dispatcher hierarchy is created within the stub code, the user passes an `ObjectName` to the constructor, which is used to initialise it. The attributes of this `ObjectName` which are required to select one of the above dispatcher implementations are shown below, along with their types. (It is important to understand that these attribute names and types are not imposed by the `ObjectName` class, but are specific to a set of its instances, i.e., they are maintained within data only).

```
RemoteObjectName          objectname
 DispatcherName           classname
```

```
 DispatcherObjectName    objectname
  HostName               string
  HostPort               unsignednumber
```

The `DispatcherName` attribute is used to select either the `TCPDispatcher` or the `UDPDispatcher`. When that dispatcher is created it is passed the `ObjectName` corresponding to the `DispatcherObjectName` attribute, and uses this to obtain the address (host name and port number) of the remote dispatcher. (The address format is also configurable, and can be specific to each dispatcher). These `ObjectName` instances would typically be stored within some naming service, and only obtained when the application requires them. Therefore it is possible for them to be modified without affecting the applications which use them.

The following code fragments show how an `ObjectName` may be used to create a dispatcher hierarchy for a `SpreadSheet` object, and the corresponding client stub code constructor:

```
ObjectName mySpreadSheetName("SNS:DailyWork");
SpreadSheet mySpreadSheet(mySpreadSheetName);

SpreadSheet::SpreadSheet(const ObjectName& objName)
{
 ObjectName dispatcherName(NULL);

 if (objName.getAttribute("DispatcherName",
                          dispatcherName))
 {
  /*
   * create dispatcher interface & implementation
   * based upon contents of dispatcherName.
   */
 }
}
```

4.2 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 serialisability, 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 it cannot be performed, or prevents any effects from the operation becoming visible.

4.2.1 Design of the atomic object support system

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

(i) An application, designed initially for a single node, may need to be distributed, permitting uniform access to local and remote objects.

(ii) Objects have different concurrency control requirements, so the system should be able to support these, e.g., pessimistic and optimistic.

(iii) 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 configure 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 4.2, illustrates the structure of the atomic object support system and the events that can occur within it. The *atomic event manager* co-ordinates the events which correspond to requests to examine, update and overwrite atomic objects, with the commit processing of atomic actions. The atomic event manager is also responsible for generating events corresponding to 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.

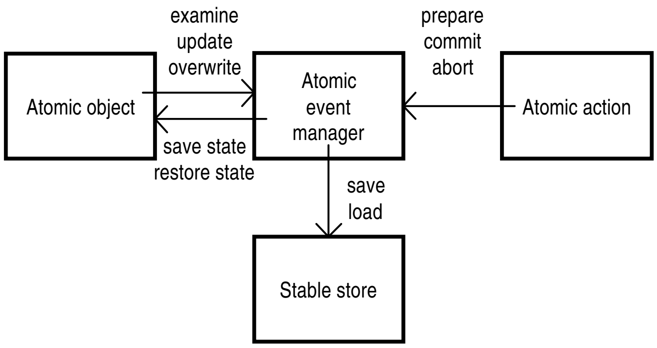

Figure 4.2, Atomic object support system events.

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`.

4.2.2 AtomicActionManager and AtomicAction classes

The `AtomicAction` class, which is derived from the **Gandiva** `Action` class described in section 3.1.1, 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;
 a.begin();           // begin top-level action
 b.begin();          // begin nested action

 if (oper1())
  b.commit();        // commit nested action
 else
  b.abort();         // abort nested action

if (oper2())
 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.

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.

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

4.2.3 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 one constructor, which is used when creating new atomic objects and for re-creating existing objects. The attributes of the supplied ObjectName are used to determine which type of object is being created. The objectName() operation is used to obtain the object name of the newly created object for later recreation.

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

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

4.2.4 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 4.3, 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.

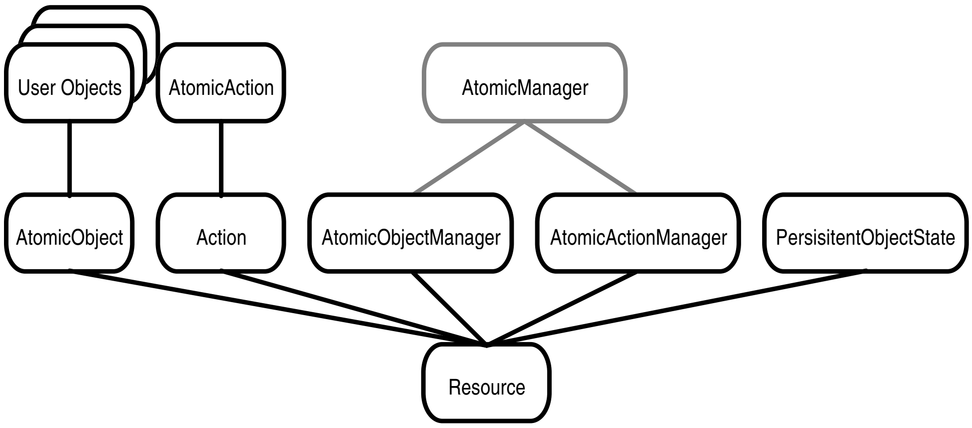

Figure 4.3, Atomic object support system class inheritance.

4.2.5 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 be configured with the most suitable implementation of the support system for their needs.

One of the most important configuration aspects for 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 4.4. 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 configurability 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.

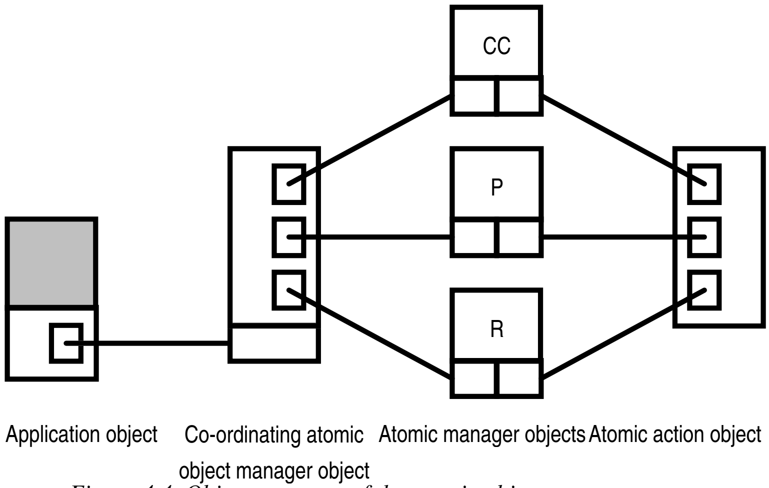

Figure 4.4, Object structure of the atomic object support system.

The object structure in figure 4.4 is configured from the attributes of the `ObjectName` passed to the atomic application object. The attribute names of this `ObjectName` and the attribute names of `ObjectNames` that it contains, along with their attributes type, are listed below:

```
DemoAtomicObjectName                    objectname
  ConfigObjectName                      unsignednumber
  Uid                                   uid
  AtomicObjectManagerObjectName         objectname
    ConfigObjectName                    unsignednumber
    ClassName                           classname
    RecoveryObjectName                  objectname
      ConfigObjectName                  unsignednumber
      ClassName                         classname
      BufferClassName                   classname
    PersistenceObjectName               objectname
      ConfigObjectName                  unsignednumber
      ClassName                         classname
      PersistentObjectStateObjectName   objectname
        ConfigObjectName                unsignednumber
        ClassName                       classname
        FileName                        string
        BufferClassName                 classname
    ConcurrencyControlObjectName        objectname
      ConfigObjectName                  unsignednumber
      ClassName                         classname
```

The value of the above attributes can be changed to alter the configuration of the atomic object. For example, by changing the “ClassName” attribute of the “ConcurrencyControlObjectName” `ObjectName`, the implementation of that part of the service can be altered; the implementation could be changed to optimistic concurrency control from pessimistic.

4.2.6 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 per second, within a top-level atomic action and a nested atomic action. The result are presented in the table below.

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).

Model	Environment	examine	update	overwrite
Solo	Top-level	1330	23.5	23.5
	Nested	925	450	445
Multiple	Top-level	270	23.5	23.5

	Nested	550	345	340

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 configurability 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 course 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 Interface and implementation separation

In [16], Coplien proposes a separation of interface from implementation for C++. However, each interface must know about all possible implementations, and so this is a static model, requiring changes to the interface code to reflect changes in the allowed set of implementations. 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 OpenC++ system described in [18] achieves a configurable architecture through *reflection* [19]. Classes can be *reified* and method invocation controlled through a *meta-object protocol* (*MOP*), which acts as a stub object, intercepting and processing appropriate invocations. 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 MOP is statically created for a specific purpose, and changing the MOP requires rebuilding the application and/or the component to be controlled. In addition, because the MOP must intercept and parse method invocations, there are significant performance overheads.

The Shared Object Model (SOM) provides a limited form of interface and implementation separation in C++ [20]. By modifying the compiler and linker, 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 performs the necessary binding between the two. However, SOM places limitations on how classes may change from the original definition, e.g., they must be “upwardly” compatible, and dynamic modification is not possible.

The InterViews graphical user interface presented in [10] allows programmers to deal with “abstract” graphical entities such as buttons and scroll bars, without knowledge of the details of their “look and feel”, which can change between run-time environments. This flexibility is achieved through the use of “kits”, which 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 there is no support for application programmers to extend this.

The Spring system is an experimental distributed environment developed by Sun Microsystems [3], and is closest in aims and functionality to our model. 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 support structure for this language generates surrogate objects (essentially C++ interface classes) from these IDL descriptions. Binding of interface to implementation is flexible and can occur at run-time. However, it is unclear what is the equivalent of `ObjectName`, to maintain this configuration information without statically tying interface to implementation.

5.2 Configuration languages

Existing systems supporting dynamic reconfiguration such as Conic [23], Durra [24], and Surgeon [25] provide facilities for module addition and deletion and often rely on special configuration languages for specifying component interconnections and changes to those interconnections. Systems such as Regis [26], Clipper [27], and the work presented in [28] also use separate configuration languages and generate C++ implementation code from these language specifications. However, at the implementation level these systems implicitly tie component interfaces to implementations and would therefore benefit from the introduction of a software engineering model similar to that which we have presented in this paper. The interface and implementation separation which we have described could be automatically generated from an appropriate specification in these configuration languages, providing greater flexibility and configurability.

6. Conclusions

Separating software components into their interface and implementation components provides flexibility and configurability in their design and implementation. 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++.

Acknowledgments

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 27th 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 14th VLDB Conference, Los Angeles, pp. 26-37, September 1988.

[12] S. K. Shrivastava, and S. M. Wheater, "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. Wheater and M. C. Little, "The Design and Implementation of Arjuna", USENIX Computing Systems Journal, Vol. 8, No. 3, pp. 253-306, Summer 1995.

[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.

[21] E. Cooper, “Replicated distributed programs”, Proc. of 10th ACM Symposium on Operating System Principles, Washington, pp. 63-78, December 1985.

[22] K. Birman, T. Joseph and F. Schmuck, “ISIS - A Distributed Programming User's Guide and Reference Manual”, The ISIS Project, Department of Computer Science, Cornell University, Ithaca, NY, March 1988.

[23] J. Kramer and J. Magee, “The evolving philosophers problem: dynamic change management”, IEEE Transactions on Software Engineering, 6 (11), pp. 1293-1306, 1990.

[24] M. R. Barbacci, *et al*, “Durra: a structure description language for developing distributed applications”, Software Engineering Journal, 8 (2), pp. 83-94, March 1993.

[25] C. Hofmeister, *et al*, “Surgeon: a packager for dynamically reconfiguring distributed applications”, Software Engineering Journal, 8 (2), pp. 95-101, March 1993.

[26] J. Magee, N. Dulay, and J. Kramer, “A Constructive Development Environment for Parallel and Distributed Programs”, Proceedings of the Second International Workshop on Configurable Distributed Systems, pp. 4-14, March 1994.

[27] B. Agnew, C. Hofmeister, and J. Purtilo, “Planning for Change: A Reconfiguration Language for Distributed Systems”, Proceedings of the Second International Workshop on Configurable Distributed Systems, pp. 15-22, March 1994.

[28] M. Zimmermann and O. Drobnik, “Specification and Implementation of Reconfigurable Distributed Applications”, Proceedings of the Second International Workshop on Configurable Distributed Systems, pp. 23-34, March 1994.

ERROR: undefined
OFFENDING COMMAND: S

STACK:

(up)