A Framework for Building Extensible C++ Class Libraries

Arindam Banerji, Dinesh Kulkarni, David Cohn
Distributed Computing Research Laboratory
University of Notre Dame
Notre Dame, IN 46556
axb@cse.nd.edu

Abstract

Extensibility leads to better designed and more reusable software. Traditionally, implementors have built extensible C++ software using ad hoc mechanisms built from scratch. This paper identifies specific characteristics that constitute extensible software. A framework for building extensible C++ libraries has been defined and constructed on AIX 3.2. Finally, the paper gives guidelines for implementors of extensible software through a discussion of an on-going application of the framework.

1. Introduction

A critical problem in designing software libraries is the difficulty of predicting possible future uses. Designers of C++ class libraries attempt to avoid this problem by *designing in* implementation choices. However, basing libraries on most probable uses and using run-time checks to select an implementation cannot accommodate unforseen future uses. Nowhere is this problem more acute than in the construction of system services to handle mobility or workstation clustering. Three examples are presented below.

Consider a library for managing information access in a mobile computing environment [Banerji, 93a]. The caching behavior and access mechanisms are critical to the performance of such a system. Optimum caching performance is highly dependant on usage patterns which can neither be accurately predicted at design time nor reasonably calculated at run-time. Rather, performance will improve if the implementation can respond to client-provided run-time control directives. Run-time client control over the implementation, or just the ability of a client to provide usage hints, could better optimize performance.

In a clustered workstation environment [Banerji, 93b], on-line updating of software is desirable. Ideally, a new implementation could be added to running software without disturbing existing applications. This implies the ability of a client to dynamically replace or add implementations of either member functions or even entire classes.

Finally, suppose that a large company has developed a class library to manage its critical corporate data. The library includes mechanisms to handle all inquires in use at the time of its creation. Later, after the developers have moved on to other projects, a new need arises. If this need cannot be handled by the existing library, either the developers must return to this project or new programmers must wade through the original source code. If the library were extensible, new functionality could be added without reference to the original code or its developers.

Although the situations presented above may seem far fetched, they are real problems in mobile computing, clustered computing and management of object library management [Banerji, 94]. Moreover, these problems have counterparts in system and application software for stand-alone workstations [Krueger, 93]. As discussed below, some solutions have been suggested in the past. However, many of these solutions are ad hoc and lacked the genericity needed to construct extensible class libraries for other application domains. This paper addresses genericity by discussing the design, implementation and use of a *framework* that allows C++ class library designers to build extensibility into their subsystems.

This framework has various specializable and easily replaceable components. In addition, certain programming guidelines that aid in gluing together the components of an extensible library are discussed. The existing libraries for this framework and sample test cases have been built for AIX 3.2 running on IBM RS/6000s. A custom port of the AT&T cfront 3.01 was used to build the components.

The next section discusses extensibility and its implications. The terms and technologies pertaining to the framework implementation are then detailed. Related work is discussed in the subsequent section. Section 4 lays out the overall structure of the framework as well as the details of the design principles. Section 5 discusses the programming guidelines that implementors need to follow and presents a concrete example. Section 6 describes the user's view of software, built using this technology. The paper ends with a summary of its contributions.

2. What Extensibility really means

This section attempts to demystify the term *extensibility.* In order to do this, it is necessary to list the features that usually characterize extensible software. The identification of such features leads to a better understanding of the techniques and mechanisms necessary to support extensibility.

Extensibility implies different things to different people. Instead of using some preconceived notion of this software property, we choose to define the essential features of extensible software. Some of these features are obvious, while others are not. The features are:

- ***Separation of interface from implementation*** is necessary to ensure that the visible functionality of a software library is not cluttered with non-relevant implementation details. Such a feature is considered good practice, since it allows for implementation changes with little or no client-code recompilation. Furthermore, it is a fundamental requirement for supporting other features of extensible software.
- ***Tunable implementations*** allow design decisions to be based upon actual run-time data. Best case scenario implementations are replaced by designs that can act upon client-provided hints and directives. Thus, application-dependant usage information, which can never be predicted at design time, may be used by clients to fine-tune software implementations. This feature may appear to conflict with interface-implementation separation. However, as discussed later, it is possible to simultaneously support such conflicting features.
- ***Multiple coexisting implementations*** for one particular interface allow clients to choose an implementation that closely matches their needs. Hence, conflicting design choices may be reflected in separate implementations, thus affording implementors the luxury of application-specific optimizations. Clients on the other hand, are left with the responsibility of selecting an appropriate implementation at run-time. Furthermore, this separation allows for the independent development of different implementations of an interface. Thus, bug fixes for exist-

ing implementations can be made available as new implementations of an existing interface.

- ***Addition or substitution of implementations*** allows clients to effect major reconfigurations without recompilation or even application restart. Constituent classes and member functions of an implementation may be dynamically replaced. Similarly, whole new implementations for existing interfaces could be loaded at run-time. Typically, this is used to dynamically load new implementations of existing interfaces, in order to fix bugs or update services.
- ***Dynamic addition of member functions to interfaces*** allows clients to add services without requiring recompilation of existing code. Usually implementations are expected to support only those public member functions that correspond to a particular interface. Should a new implementation support additional member functions, the clients can effectively integrate such services by extending the interface dynamically. Such integration does not affect existing clients who still can use the old unextended interface.

These features of extensible software point to specific base technologies that are required to implement them. These base technologies have been adopted by the flexibility framework, described in this paper. These technologies, in no particular order, are:

- ***Explicitly separate interface and implementation hierarchies:*** Most well designed software systems separate out interfaces from implementations. Typically, such separations are performed in an adhoc manner and the interaction between interfaces and implementations is decided on a case-by-case basis. In order to support separate implementations, it is necessary to completely separate out interfaces and implementation hierarchies. The inter-dependence of these hierarchies, may be minimized by ensuring that interface classes only interact with an abstract base class representing the implementation hierarchy. All concrete implementations of an interface inherit from this base class. However, the interface classes do not depend upon the symbols of any specific implementation; only on those of the abstract base class.
- ***Run-time access to type information:*** Interactions between interface and implementation hierarchies involve passing pointers to abstract base class of the implementation hierarchy. This is necessary to ensure that the interface classes do not depend upon any symbols available in concrete implementations. Thus, downcasting to derived class pointers is often required. Classes that are used in the interaction between interfaces and implementations are thus, associated with Run-Time Type Information [Lajoie, 93].
- ***Dynamic linking and loading:*** Run-time addition of implementation requires that object modules be loaded and linked into running code. Similar facilities are necessary to enable on-the-fly substitution of implementation classes. Whatever mechanism is used has to deal with C++ specific problems, such as, mangled names and initialization of static constructors and destructors. Invariably, such a mechanism is highly dependant upon the exact nature of the dynamic linking services provided by the target operating system.
- ***Class objects:*** Flexibility to control implementations or manipulate the member functions that belong to an interface requires a level of indirection greater than that provided by C++. Class objects are used to provide this indirection. As in Smalltalk [Goldberg, 89], class objects also afford a mechanism for calling C++ constructors, based upon a given set of properties, instead of the name of the class being initialized. These objects may be associated with interface and implementation objects to act as a run-time interpreter of some usually implicit entity.
- ***Indirection in name resolution:*** Dynamic addition of member functions to an existing class interface requires that it be possible to map a given member-function call to a generic

per-class function-call dispatcher [Coplien, 91]. This technology of indirection in name resolution or dynamic dispatch, already exists in many pure object-oriented languages [Ungar, 87]. However, making it available for C++ with reasonably good performance, is quite another matter.

- ***Dual interfaces:*** If implementation details are removed from interfaces, how can clients of extensible software fine-tune implementations? The technology of opening up implementations through dual interfaces is used. The idea of open implementations [Kiczales, 92] with one interface to access the functionality of a subsystem and another to optionally control the implementation, is not new. It allows clients to provide usage information and implementation directives through a second interface. The second interface or the meta interface should be made available to clients on a need-to-know basis.

All these base technologies have existed for a while. However, we believe no one has integrated them to provide an environment explicitly geared to developing extensible C++ libraries. This work integrates these existing technologies into an easily usable form.

3. Related Work

The need for extensibility in software has been stressed for both operating systems and languages [Kiczales, 92]. The authors of the Meta-object protocol for CLOS [Kiczales, 91] have been instrumental in discussing open implementations and dual interfaces. Apertos (formerly Muse) [Yokote, 92] has applied reflection and meta-object protocols to operating systems. Choices [Campbell, 93], an object oriented operating system has done yeoman work in supporting frameworks for dynamic code loading and stepwise refinement. Recently, Open C++ [Chiba, 93] has used translator directives for redirecting method invocations to metaobjects to implement object groups in a distributed system.

Many of the basic mechanisms, such as changing type interfaces on the fly, have also been discussed in previous work. These include preprocessor-generated class objects to instantiate dynamically loaded subclasses [Dorward, 90] and type-set interfaces for schema manipulation in databases [Skarra, 86]. Of course, the RTTI extensions of ANSI C++ have been discussed in great detail [Stroustrup, 92], [Vines, 93]. Finally, it is important to note the numerous idioms mentioned in the last few chapters of [Coplien, 91] that deal with flexibility, indirection and its effective use.

Finally, it is pertinent to mention some recent industry initiatives. Although, efforts such as CORBA [OMG, 91] effectively separate out implementations from interfaces, they are mainly geared towards facilitating object interactions. Thus, objects created by two different languages can be made to cooperate as in IBM SOM [IBM, 91] or object interactions can be transparently made to span machine boundaries. It is possible to associate some flexible properties to CORBA compliant software through the use of meta-objects and indirections. However, very little support is provided for structuring object systems for flexibility.

4. The Extensibility Framework

This framework represents a significant set of collaborating classes and class hierarchies, that capture the patterns and mechanisms needed to implement extensible C++ software[1]. Each of the components or class-hierarchies of this framework can be further specialized by using inheritance.

1. Paraphrased from [Firesmith, 1993]

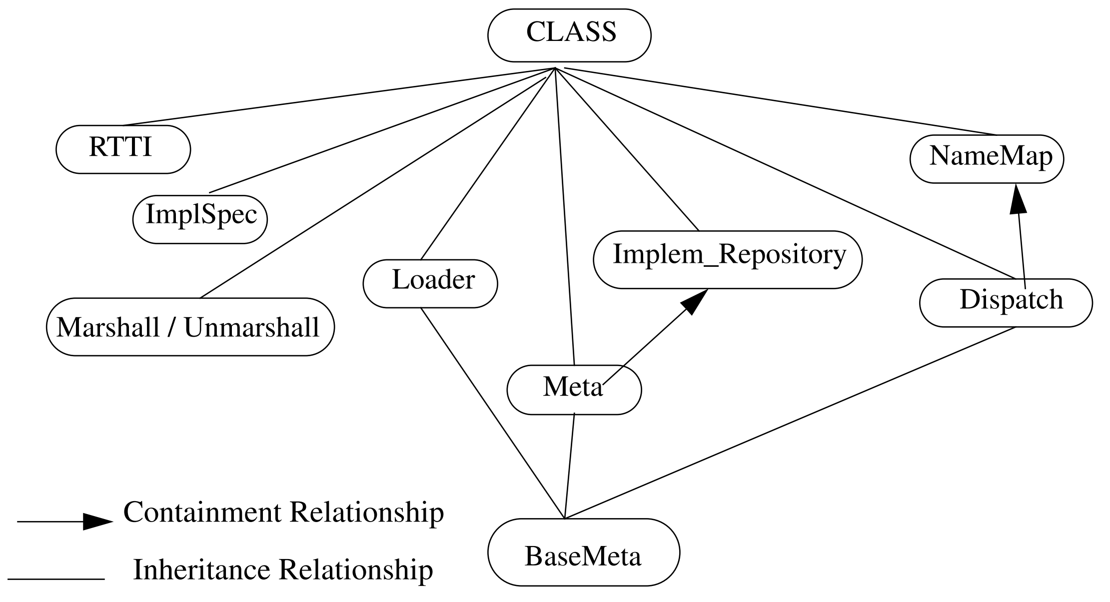

Figure 1. Components of the Framework

This allows programmers to tailor the services provided by this framework according to their specific needs. In addition to these components, there are certain programming guidelines both for implementors who use this framework and for clients[1] who use the software built with this framework. This section discusses the details of the various components of the framework shown in Figure 1. The next section presents how such a framework may actually be used.

Before reviewing the components of the framework, it is important to understand the three distinct relationships between the entities discussed here. The first one is the *instance-of* relationship between a class and its object instance. The former is represented only in the C++ source while the latter also has a run-time representation. The second one is *inheritance* which applies to both interfaces (subtyping) as well as objects (implementation). The third relationship is more subtle: that between an object and its metaobject. The latter provides a run-time representation of some of the implementation aspects of the former. Since a metaobject makes an otherwise implicit aspect explicit, it is said to *reify* that aspect. For simplicity, a metaobject that reifies a class will be referred to as a *class object*.

Figure 1 shows the various component hierarchies of the extensibility framework. There are three kinds of components - those that are incorporated into class-objects, those that are part of every object and those that realize some helper services. Typically, as mentioned below, the BaseMeta component is part of every class object. On the other hand, RTTI is part of every object. The Marshall/Unmarshall component and the ImplSpec component facilitate interactions between interfaces and implementations. The components in Figure 1 are:

- ***CLASS***, a base class from which every other framework class inherits, is a place-holder which ensures the safety of type-casts.
- ***RTTI***, the set of Type_info and typeid classes that implement ANSI compliant run-time type information for appropriately instrumented classes.

---

1. Henceforth, "implementor" refers to programmers who use the framework directly to build software and "client" refers to those who use the software developed in this manner.

- ***ImplSpec***, a set of classes, that allow flexible specification of implementation characteristics and are especially useful in cases where the name of the target class is unknown.
- ***Implem_Repository***, a repository that maintains information about available implementations, their class objects and their chracteristics.
- ***Marshall/Unmarshall***, an extensible class hierarchy that allows clients to specialize the marshalling of parameters.
- ***Meta***, a base class for class objects, that maintains the implementation repository and provides an indirection mechanism for constructor calls, when the exact name of the target class is unknown at compile-time.
- ***Loader***, a front-end class that provides a user-friendly interface to run-time linking and loading facilities.
- ***NameMap***, a repository that maps function keys and parameter type-tags to a target generic function.
- ***Dispatch***, a class hierarchy that maintains the NameMap and handles the mapping of dynamically loaded interfaces to per-class generic functions.
- ***BaseMeta***, a class that aggregates the properties of Meta, Loader and Dispatch class hierarchies, thus forming a direct parent to all class objects for concrete implementations.

Each of these components is actually a class hierarchy that can be specialized to get the precise functionality desired. Implementors incorporate elements of these class hierarchies to effect extensible software. Although, implementors and clients only see these components, the actual realization of the framework uses a few libraries to effectively implement some services. There are primarily two such libraries - the run-time linking library for AIX and the library that implements RTTI. These libraries acts as implementations tools for this framework. The next few subsections discuss some of the basic elements shown in the figure. The section ends with a brief description of the implementation tools or libraries used by this framework.

4.1. Identification of Different Implementations

As mentioned before, a particular interface may be supported by any number of different implementations. Hence there must be a mechanism to identify these different implementations. Since, it is possible that implementations may be loaded at run-time, names of implementation classes cannot be used for such a purpose. Instead, the **`ImplSpec`** hierarchy provides the mechanism necessary to identify some particular characteristics of an implementation. **`ImplSpec`**, as shown below, defines a common protocol for comparison of implementation characteristics. Subclasses of the class **`ImplSpec`**, may use any characteristic such as the implementation-name to identify implementations, but have to support the comparison protocol defined by **`ImplSpec`**. Thus, typically every piece of extensible software defines its own subclass of **`ImplSpec`** in order to distinguish between different implementations.

```
// The following class defines the common comparison protocol, to be supp
// ported by all characterizations of implementations. Different exten
// sible software libraries may characterize implementations differ
// ently.Some may do it through implementation-name strings, while oth
// ers may use integer constants.

class ImplSpec : virtual public CLASS // CLASS is a global base, associ
// ated with properties, common to all objects of an extensible library.
       {
        public : // some operations have been omitted for brevity
```

```
        ImplSpec(const char *); // all characterizations are finally
                                  // converted into strings, for simplicity.
        virtual ~ImplSpec() ; // destructor
        // The comparison protocol follows
        virtual int operator == (ImplSpec &) ;
        virtual int operator == (ImplSpec *) ;
        virtual int operator != (ImplSpec &) ;
        virtual int operator != (ImplSpec &) ;
        ... other comparison operators ...
  private :   // mainly responsible for maintaining pointer to a
              // global repository, which catalogs all available
              // implementation characteristics.
        impl_map_t *table; // implementation repository front-end
        ...other private data...
  } ; // Base class of the Implementation Specification hierarchy
```

4.2. Indirection as an Architectural Tool

The key to building extensibility is indirection. For example, in C++ virtual functions provide a level of indirection that allows a call of the form `base_object_ptr->foo()` to be dynamically resolved to the function `foo`, in an appropriate derived class object. However the semantics of C++ limits the target of the resolution to be a similarly named function within the class hierarchy. If static type-safety were not a concern, another level of indirection could have removed the shackles of a fixed resolution mechanism. Thus, it is very important to figure out exactly where an indirection should be added and what may be gained from adding it. Based on this, the framework adds the following indirections:

- Indirection in constructor calling
- Indirection in name-resolution during function-dispatch
- Indirection in marshalling/unmarshalling parameters

4.3. Constructor Calling

Constructors for dynamically loaded classes need to be called indirectly, since class-name-based constructor calls would cause unresolved externals during compilation. The class `Meta` and `Implem_Repository` cooperate to provide this indirection. All classes which need this degree of flexibility are associated with a class object, that is a specialization of `Meta`. `Meta` in turn contains a pointer to a globally available instance of `Implem_repository`, as in

```
class Meta: public CLASS {
            public: // ignore constructors and destructors
/*
In the following code ImplSpec is used to specify some characteristic of
the implementation. In the simplest case, it may simply be a string con-
taining the name of a class.
*/
                  virtual void register_class_object(ImplSpec *,...);
                   // function that allows addition of the class-object
                   // of a subclass[1] to the repository.
                  virtual void unregister_class_object(ImplSpec *,...);
```

---

1. Subclass here refers to the subclass of the class that the class that Meta is associated with. This, if `foo` is associated with `fooMeta`, a specialization of `Meta`, then the subclass here refers to `childof_foo`, a class derived from `foo`.

```
            // removes the entry for class-object of a sub-class
            // from the repository.
           virtual CLASS *instantiate(ImplSpec *);
            // an indirect constructor called with some form of a
            // of a specification of which particular subclass
            // needs to be instantiated.
      private:
            static Implem_Repository *repository;
      };    // specification of the Meta class.
```

Typically, dynamically loaded code is in the form of a subclass of an existing base class. Assuming that the base class and the subclass are associated with class-objects derived from Meta, then the sub-class automatically registers its class-object with the super-class repository. Thus, if **foo** and **fooMeta** are respectively the superclass and its associated class object and **childof_foo** and **childof_fooMeta** are the derived class and its associated class object, then the following happens during static initialization:

```
childof_fooMeta childof_foo::meta = new childof_fooMeta;
```

This call to the constructor of **childof_fooMeta** causes it to register itself with **fooMeta**, as the class object for **childof_foo**. At this point, instantiation requests for **childof_foo** when sent to **fooMeta**, are automatically forwarded to the **instantiate** in **childof_fooMeta**. This **instantiate** function, in turn calls the constructor of **childof_foo**. If there are additional parameters that need to be passed to the constructor of **childof_foo**, then the code gets a more complicated, but the principle remains the same.

4.4. Name Resolution and Function Dispatch

Quite often, a subclass with an additional non-inherited member function, needs to be dynamically loaded. This implies that the supported interface is extended by the addition of this subclass and clients using this new subclass should be able to access this new member function. Since, in C++, a named function call always gets resolved to a similarly named function, an extra level of indirection is needed. In this case, the name resolution mechanism during function-dispatch needs to be extended. This indirection is provided by the classes **Dispatch** and **NameMap**. These classes are quite similar to the two classes discussed above. Instead of the function **instantiate**, the function **generic_func** of the following form is used.

```
CLASS *generic_func(int function_key, CLASS *,...);
```

The dispatch object of the loaded subclass, in a manner similar to the one shown above, registers itself with the **NameMap** of the superclass dispatch object. This ensures that when a client directs an extended call to the newly loaded subclass, the generic function of the dispatch object of the subclass is ultimately called. The generic function then calls the appropriate function in the subclass.

4.5. Marshalling and Unmarshalling

For distributed systems, which form a large part of our research focus, the client and the implementation of the member function may be separated by machine boundaries.This typically requires marshalling and unmarshalling of parameters. However, factors such as relative alignments of the target and source architectures and the kind of communication links available may have a tremendous impact on the way marshalling and unmarshalling is done. In order to ensure that best possible performance is guaranteed, a slightly different form of the generic function is used.

```
Marshalled_Parm *generic_func(int, Marshalled_Parm *);
```

In this case, the class **Marshalled_parm** is an implementor specialized class that allows custom parameter marshalling and unmarshalling schemes[1]. These architectural components are aided in great part by two major implementation tools that are discussed below.

4.6. Implementation Tools

The dynamic linking and loading tool consists of a code loading library and a **Loader** class. The code-loading library, dynamically links in and loads C++ libraries into running programs. At present, it only supports the XCOFF file format of AIX 3.2. The interface supported by the library mimics the SUNOS run-time linker calls of **dlopen, dlsym and dlclose**. In order to be dynamically linkable, the set of object modules pertaining to the loaded subclass are archived into a custom shared library. Initially, the appropriate object modules of the subclass and its class objects are linked into a single object module with all outstanding externals unresolved. This object module is passed through **Munch** [USL, 92], to create a list of static constructors and destructors. A set of entry-points are created to enable calling these destructors and constructors[2].- Finally, a shared library that archives the single object module, the static initializers and the generated entry-points, is created. As is obvious, the compiler driver of AT&T's **cfront** had to be changed somewhat to support this process.

During the call to **dlopen,** all unresolved externals within this library are bound to the appropriate symbols of the running client program. The **dlopen** and **dlclose** calls also ensure that the entry-points for calling static constructors and destructors, get called automatically.

The **Loader** class, in turn provides a simplistic interface that hides some of the complexity of dlopen. Furthermore, this class implements an automatic lookup of certain directories to locate the requested loadable library. The main interface function of this class is:

```
int add_impl(char *BaseClassName, char *LoadTargetCharacteristic);
```

The RTTI tool is a library-version[3] of the ANSI-C++ language extension. The Type_Info and typeid classes are closely based upon the implementation detailed by Bjarne Stousroup [Stroustroup, 91]. An extensive set of macros have been added that ease the chore of instrumenting classes. For example, the declaration of a class needs to include a line of the form:

```
RTTI_SCAFFOLDING_DECL(NAME_OF_CLASS)
```

The definition of the class needs to include a macro of the form:

```
RTTI_SCAFFOLDING_IMPL1(NAME_OF_CLASS, NAME_OF_PARENT_CLASS)
```

The macros for handling template classes are equally easy to use. A set of macros have been provided to automate narrowing of classes, in the presence of virtual inheritance. The set of RTTI classes themselves may be easily specialized as per the requirements of the ANSI standard.

---

1. Marshalling/Unmarshalling is not discussed in detail, since it is only of interest in case of distributed computing.
2. These correspond to **_main** and **__dtors** in the USL **libC.a**, except that they have different names.
3. It is available for ftp from invaders.dcrl.nd.edu:/pub/software/rtti.tar

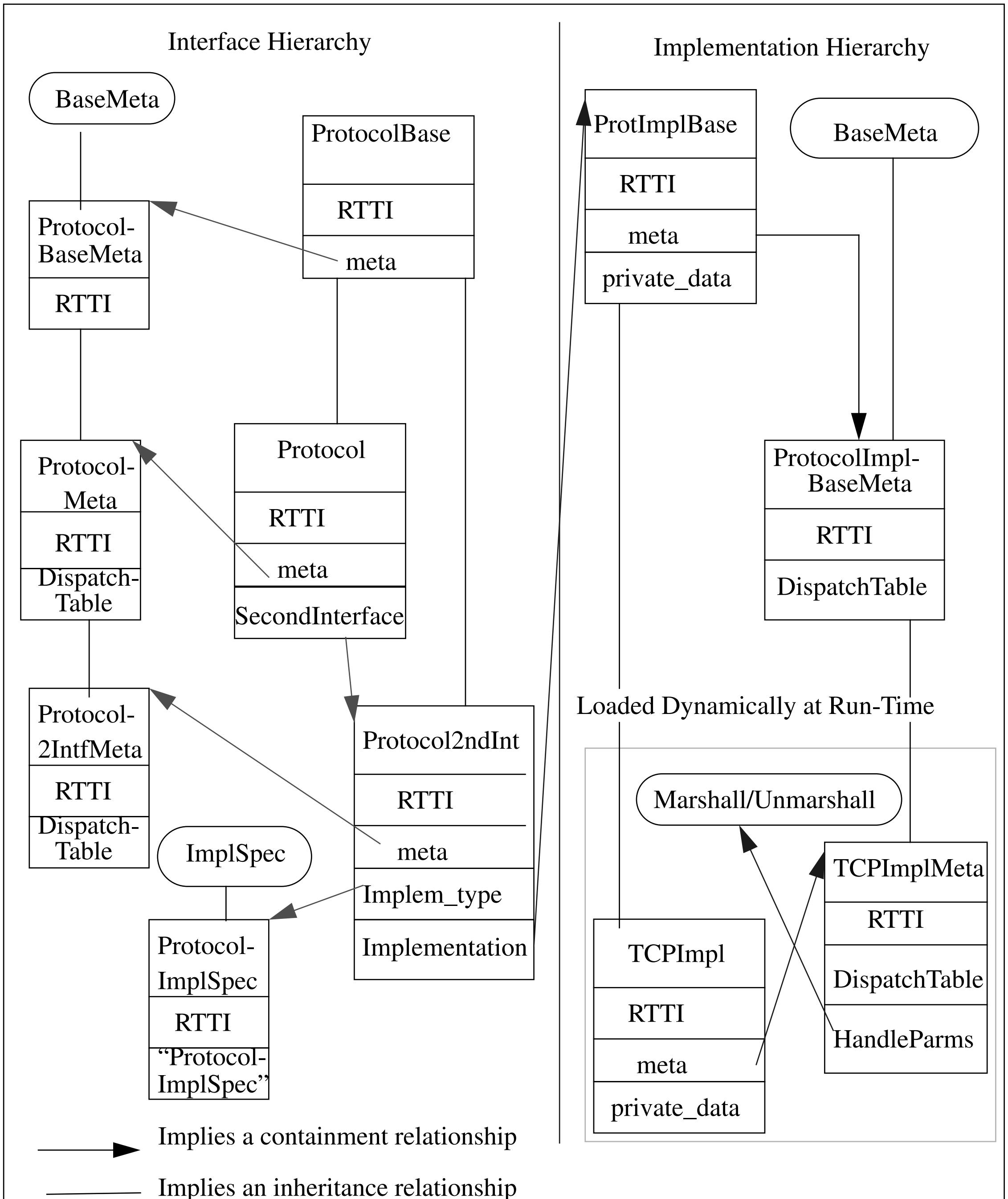

Fig 2. Implementor's View of Communication Protocol

5. Implementor's view

Having discussed the internal design of the framework components, this section sheds light on the overall structure of extensible software, that may be constructed using this framework. One of the target subsystems, currently under construction is an implementation of a user-level communication protocol library. The idea is to provide a dynamic object-oriented framework for building communication protocols, a problem similar to that addressed by x-kernel [Peterson, 90]. The layout of the hierarchies that constitute the framework is shown in Fig 2.

Figure 2, clearly shows two hierarchies - the implementation hierarchy and the interface hierarchy. On the interface side, there are two main objects - one representing the primary interface of network protocols and the other, an interface for controlling implementations. Each of these objects are associated with class-specific meta-objects that provide facilities such as function dispatch. One the implementation side, all concrete implementations inherit from a single base class. Again, the base class of the implementation as well as concrete realizations that inherit from this base class, are all associated with class-specific meta-objects. The functionality of the meta-objects on the implementation side is slightly different from those on the interface side. As can be clearly seen from the figure, only a single arrow crosses the interface-implementation barrier. This arrow represents a polymorphic pointer to the implementation tree. All interactions with implementations are exacted through this polymorphic pointer. Two final points need to be made about the figure. Although not shown, all classes in the subsystem inherit directly or indirectly from the class **CLASS**. Concrete implementations, that is subclasses of the implementation base as well as any associated meta-objects, may be attached to the implementation hierarchy at run-time through the dynamic loading services.

The main interface of the communication protocol library is provided by the class **Protocol**. It actually provides two interfaces, the first one related to the basic functionality and the second one for manipulation of the implementation, or a meta-interface which is reified by an object of class **Protocol2ndInt**.

```
class Protocol: public ProtocolBase {
            public:
                  // constructors and destructors
                  // open a passive session
                  virtual Protocol &OpenSession(...);
                  // open an active session
                  virtual Protocol &OpenSessionEnable(...);
                  // passes certain calls to the second interface.
                  Protocol2ndInt operator->();
                        ...
                  /* member functions for comm. protocols. */
                        ...
                  // macro that generates RTTI scaffolding
                  RTTI_SCAFFOLDING_DECL(Protocol)
                  // pointer to class object that controls
                  // the behavioral metacomputation...
                  static ProtocolMeta *meta;
            private:
                  // Pointer to the optional second interface
                  Protocol2ndInt *SecondInterface;
            }; // specification for the Protocol Interface
```

As can be seen in the code fragment above, apart from providing support for the functionality provided by a regular communication protocol, the class contains pointers to two other entries - a *second interface object* and a meta-object. Typically, the member functions of **Protocol** just pass on the calls to appropriate member functions of the **Protocol2ndInt** class. The meta-objects, build upon the various components of the framework, and are discussed in the next subsection.

An object of the class **Protocol2ndInt** implements the second interface. For a communication protocol the dual interface allows users to provide such directives as window-sizes, distribution of small communication buffers vs. large communication buffers etc. These features are similar to those offered by the unstructured ioctl system call. In addition, the client gets to choose which particular implementation of the protocol he/she would like to use. This information is specified in the form a ImplSpec class, which allows various kinds of implementation options to be specified.

```
class Protocol2ndInt: public ProtocolBase[1] {
      public:
             // ignore constructors and destructors...
            void set_impl(ImplSpec *,...);
             // choose a particular implementation - such as TCP.
            void set_window_size(int);
             // set the window size.
            ...
            /* other functions to support an open implementation*/
            ...
            /* If necessary, member functions to handle forwarded
                  calls from the Protocol class. */
            static Protocol2ndIntfMeta *meta;
            // class object that allows behavioral manipulation
            RTTI_SCAFFOLDING_DECL(Protocol2ndInt);
      private:
            ProtocolImplSpec *implem_type;
             // identifies which particular implementation was
             // chosen, e.g.: TCP or UDP.
            ProtImplBase *implementation; // the implementation
      }; // specification of the dual interface.
```

As can be seen from Figure 2, the only symbols from the implementation hierarchy that are visible to the interface hierarchy, are those of the class ProtImplBase - the root of the Implementation hierarchy. ProtImplBase has two main functions. Firstly, it acts as an abstract placeholder, from which any number of prototype implementations may inherit. This ensures that dynamically loaded subclasses of ProtoImplBase are used in a type-safe manner. Secondly, ProtImplBase through its meta-object manages the extensible properties of the real prototype implementations. With the help of the Implem_repository and the BaseMeta classes, it provides the services necessary to afford run-time extensibility.The message protocol between the two hierarchies is that supported by the member functions of ProtImplBase and the meta-object of ProtImplBase i.e.: ProtImplBaseMeta.The well-specified nature of the interaction between the two hierarchies, ensures that multiple implementations can co-exist.

```
class ProtImplBase: public CLASS {
      public: // ignore constructors and destructors
            // some communication protocol pertinent functions
            void openSession(...);
            void openSessionEnable(...);
            ...
            /* some communication related member functions */
            ...
            ProtocolImplBaseMeta *operator->();
```

1. The class ProtocolBase simply encapsulates, the shared characteristics of the Protocol2ndInt and Protocol classes, and thus forms a placeholder for structuring the interface hierarchy.

```
        // returns pointer to the implementation meta-object
private:
      ProtocolImplBaseMeta *meta; // the meta-object
}; // specification of the root of the Impl. hierarchy
```

5.1. An Implementor's Manual

Having shed some light on the overall structure of software that uses the extensibility framework, it is time to specify the exact steps that a programmer must take to be implement such software. For most of the implementation steps that need to interact with the framework, template files are used to guide the implementor. Generic makefiles make the task of building these executables, even simpler. Finally, it must be mentioned that certain conventions need to be followed while naming shared libraries of interfaces and loadable implementations. These conventions, not mentioned here, allow the dynamic loading facility to easily find a shared library that matches a certain implementation characteristic. The steps themselves are:

- Step 1. Create a primary interface and a secondary interface class for the software library that needs to be constructed.
- Step 2. Create a base class for the implementation hierarchy, which essentially handles the sum of all the member functions represented in the two classes from Step1. The definitions of these member functions may be kept empty.
- Step 3. Associate each of these classes with meta-objects or class-objects, based on simple name substitution of available template classes. The instantiate function of the meta-objects on the implementation side must be updated to match the actual constructors supported by the primary interface.
- Step 4. Declare a few standard static objects that allow for automatic creation of instances of the meta-objects for both the implementation and interface objects. The declaration of these static objects is facilitated through easy-to-use macros.
- Step 5. Update available generic makefiles to use the actual file names used for this particular software library.
- Step 6. Use the makefiles from Step 5 to create shared libraries containing the interface hierarchy, the base of the implementation and associated meta-objects. Thus, a binary form of the interface provided by the software library is now available. At this point, an implementation must be created.
- Step 7. Design and create an implementation that supports at least all the member functions provided by the base class of step 2. If no extra member functions are to be supported, jump to Step 11.
- Step 8. Create an implementation-specific dispatcher, based upon available templates. This dispatcher is called by generic_func, when the new member function is called by a client.
- Step 9. Create a header file that defines macros to map this new function call to a call to `generic_func`. This header file can also be set up using available template header files.
- Step 10. At this point create the required meta-object for this implementation and declare the requisite static objects, as in Step 4.
- Step 11. Use generic makefiles, and create an implementation-specific makefile. Ensure that naming standards for implementation libraries are followed.
- Step 12. Finally, place the shared library created in Step 12, in an appropriately named

directory, following certain naming conventions.

As mentioned, the objects in the interface hierarchy, the root of the implementation hierarchy and a meta-object that controls the implementation hierarchy root are archived into a shared library. A client program can then link in the shared library corresponding to the interface, that he/she needs to use. An implementor creates a new implementation, creates a loadable shared library that inherits from the implementation base class. This loadable library, can be automatically loaded at the request of the client (as discussed in Section 6). Similarly, any number of new implementations may be created and loaded.

6. Client's View

A typical client of extensible software, thus engineered may want to use the interfaces provided for three particular purposes:

- Use the direct functionality of the interface e.g.: OpenSession
- Use the second interface to control the implementation e.g.: set_window_size
- Use the meta-functionality to load new implementations of the interface.

The following piece of code demonstrates this for the protocol class, discussed above. Initially, a client programmer allocates a protocol object and selects the implementation to be used. For example, in this case the programmer decides to use the UDP implementation of the protocol. After setting the implementation, the programmer may use the UDP implementation as desired. At this point, for some hypothetical and fictitious reason, the programmer decides to use TCP, instead of UDP. Assuming, that the TCP implementation is not pre-loaded, the programmer asks for it to be loaded through the **`add_impl`** call. Subsequently, the TCP implementation is selected and the TCP protocol realization is ready for use. Finally, just as a demonstration sample, the programmer chooses to change the sliding window size used by the TCP realization. This is achieved through the smart-pointer which provides access to the secondary interface. It is perhaps important to mention that the following code is meant to demonstrate the use of flexibility and not necessarily to present semantically correct use of the TCP protocol.

```
 // Allocate a protocol object - the default implementation is
 // used - there may or may not be a default implementation.
Protocol *obj = new Protocol;

// Actually set the implementation to be used to be udp
 (*obj)->set_impl("UDP");
 /*the pointer operator is used to get at the second-interface*/

// Use the functionality of the protocol object now.
 obj->OpenSession(...);
... /* some code here */...

// At this point the client decides to add the tcp implementation
// to the running program. It calls the interface provided by
// the loader class, thru the meta pointer in the Protocol class.
 obj->meta->add_impl("ProtImplBase","TCP");

// Now the object may change the implementation type
(*obj)->set_impl("TCP");

// Now, the object can be called regularly, as in..
```

```
(*obj)->set_window_size(...);
```

Occasionally, the client may want to load an implementation that extends the prescribed interface. Let us assume that the TCP implementation is being used. At this point the programmer decides to use MobileTCP, a realization of TCP that supports mobility of connections. This extra functionality is supported by an additional member function **migrateconn**. One possible mechanism would be to create an new interface and use it to access the new implementation. However, sometimes recompilation of interface classes is not an option. In such cases, the programmer may dynamically extend the interface of the class, as shown below. The steps to be taken are as follows:

```
// Assumes that the appropriate header files for MobileTCP are actu
// ally pulled in by the client programmer.
Protocol *obj = new Protocol;
// Add the new implementation first
obj->meta->add_impl("ProtImplBase", "MobileTCP");

// Add the interface to the interface hierarchy - here the name of
// the function to be added is "migrateconn"
obj->meta->add_intf("ProtImplBase", "MobileTCP", "migrateconn");
// At this point migrateconn is ready for use.
obj->migrateconn(...) ;
```

At this point, other clients may directly call the migrateconn function, as long as they are using the Mobile TCP implementation. Existing clients do not need to recompile any code. However, new clients that intend to use this new function must pull in the header files specific to this extended interface, so that a call to migrateconn, automatically gets expanded into a call to generic_func. It is expected that this kind of extensibility will not be used very often.

7. Conclusion

The framework described here, represents a critical step in structuring extensible software. In addition to identifying the specific characteristics of extensible software, it provides a good set of tools for dynamic flexibility. Its services are geared towards the construction of more reusable and better designed software. Perhaps more importantly, it is a key step towards developing class libraries that can be tailored without access to source code.

8. Availability

The extensibility framework is not yet available for general distribution. Please contact the primary author at axb@cse.nd.edu, for latest availability information.

9. References

[Banerji, 93a] A. Banerji et. al. Mobile Computing Personae, *Proc. Workshop on Workstation Operating Systems IV,* Napa, California, Oct. 93, pp. 14-20.

[Banerji, 93b] A. Banerji et. al. The Substrate Object Model and Architecture, *Proc. IWOOOS '93*, pp31-43.

[Banerji, 94] A. Banerji et. al. Design, Distribution and Management of Object-Oriented Software, *Proc. USENIX Applications Development Symposium*, to appear.

[Campbell, 93] R. Campbell et. al., Designing and Implementing Choices: An Object-Oriented System in C++, *Communications of the ACM*, 36(9), Sept, 93, pp. 117-126.

[Chiba, 93] S. Chiba & T. Masuda, Designing an Extensible Distributed Language with a Meta-Level Architecture, *ECOOP '93 - Object-Oriented Programming, LNCS 707*, Springer Verlag, pp. 482-501.

[Coplien, 91] J. Coplien, *Advanced C++ Programming*, Addison Wesley.

[Dorward, 90] S. Dorward, R. Sethi, J. Shopiro, Adding New Code to a Running Program, *USENIX C++ Conference*, pp. 279-292.

[Firesmith, 93] D. Firesmith, Frameworks: The Golden Path to Object Nirvana, *Journal of Object-Oriented Programming*, 6(6), Oct. 93, pp. 6-8.

[Goldberg, 89] A. Goldberg & D. Robson, *Smalltalk-80 The Language*, Addison Wesley, 1989.

[IBM, 91]IBM (1991) *OS/2 2.0 Technical Library, System Object Model and Reference*, Version 2.00, IBM.

[Kiczales, 91] G. Kiczales et. al., *The Art of Metaobject Protocol*, MIT Press.

[Kiczales, 92] G. Kiczales, Towards a New Model of Abstraction in the Engineering of Software, *Proc. Workshop on Reflection and Meta-level Architectures, IMSA '92*.

[Krueger, 93] K. Krueger et. al., Tools for the Development of Application-Specific Virtual Memory Management, *Proc. OOPSLA '93*, ACM, pp.48-64.

[Kulkarni, 93] D. Kulkarni et. al., *Information Access in Mobile Computing Environments*, Tech. Report 93-11, Dept. of Computer Science & Engg., University of Notre Dame, Notre Dame, Indiana.

[Lajoie, 93] H. Lajoie, Standard C++ Update - The New Language Extensions, *C++ Report,* July-Aug. 93, pp. 47-52.

[OMG, 91] *The Common Request Broker: Architecture and Specification,* OMG Document No. 91.12.1, Object Management Group, Framingham, MA.

[Peterson, 90] L. Peterson, N. Hutchinson, S. O'Malley & H. Rao, The x-kernel: A Platform for Accessing Internet Resources, *IEEE Computer*, 23(5), May 1990, pp. 23-33.

[Skarra, 86] A. Skarra & S. Zdonik, The Management of Changing Types in an Object-Oriented Data Base, *Proc OOPSLA '86*, ACM, pp. 483-495.

[Stroustrup, 91] B. Stroustrup, *The C++ Programming Language*, 2nd Edition, Addison Wesley.

[Stroustrup, 92] B. Stroustrup, D. Lenkov, Runtime Type Identification for C++, *C++ Report,* 4(3), March-April 92, pp. 32-42.

[Ungar, 87] D. Ungar & R. Smith, Self: The Power of Simplicity, *Proc. OOPSLA '87*, pp. 227-242.

[USL, 92] *C++ Language System*, Unix System Labs.

[Vines, 93] D. Vines, Z. Kishimoto, Smalltalk's Runtime Type Support for C++, *C++ Report,* 5(1), Jan. 93, pp. 44-52.

[Yokote, 92] Y. Yokote, The Apertos Reflective Operating System: The Concept and its Implementation, *Proc. OOPSLA '92*, ACM, pp. 414-434.