Combining Contracts and Exemplar-Based Programming for Class Hiding and Customization *

Victor B. Lortz **Kang G. Shin**

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122

{*vbl,kgshin*}*@eecs.umich.edu*

Abstract

For performance reasons, client applications often need to influence the implementation strategies of libraries whose services they use. If an object-oriented library contains multiple service classes customized for different usage patterns, applications can influence service implementations by instantiating the customized classes that match their needs. However, with many similar service classes, it can be difficult for applications to determine which classes to instantiate. Choosing the wrong class can result in very subtle errors since a customized class might use optimizations that work only over a restricted domain. In this paper, we show how client-side software contracts and exemplar-based class factories can be used to construct customized server objects. By expressing priorities and requirements in contracts, clients can delegate service class selection to the library and thereby avoid implicit dependencies on the library implementation. We have used this approach in the implementation of a real-time database system.

*The work reported in this paper was supported in part by the Office of Naval Research under grant N00014-92-J-1080, by the National Science Foundation Industry/University Cooperative Research Center at the Univ. of Michigan, and by the NSF under grant DDM-9313222.

OOPSLA 94- 10/94 Portland, Oregon USA
© 1994 ACM 0-89791-688-3/94/0010..$3.50

1 Introduction

Traditionally, designers of software services (languages, libraries, operating systems, etc.) define abstract interfaces to the services and hide all implementation details from the clients. However, this model has proven inadequate when implementation decisions bias the resulting server performance in favor of certain usage patterns and against others. For example, an LRU paging strategy for a virtual memory system is optimal for clients that exhibit locality of reference and sub-optimal for those that do not.

Kiczales and Lamping call the problem of choosing a service implementation strategy a *mapping dilemma* [1]. Since the service provider cannot control client usage patterns, successful resolution of mapping dilemmas requires prior knowledge of client needs. In some cases, it is possible to collect historical usage patterns on a per-client basis and use that data to predict future usage. However, a more general solution is to allow clients to help resolve mapping dilemmas through a meta-protocol. Kiczales and Lamping categorize meta-protocols as either *declarative*, in which clients choose among pre-defined service implementations or *imperative*, in which clients are free to override part or all of the service implementations. Well-designed meta-protocols minimize the effort required to achieve

acceptable server behavior.

In this paper, we describe a technique for constructing server objects that combines the advantages of declarative and imperative meta-protocols. The declarative part of our meta-protocol consists of client-side "contracts" that determine which customized service classes will be used and configure server objects according to client needs. Client-side contracts expose the otherwise hidden assumptions that clients make about servers, and they indicate the client's willingness to abide by any semantic restrictions required by the server. Furthermore, the same contract language can be subsequently used by the client to extract meta-level information from the server object. Therefore, our contract meta-protocol can support a dialogue between clients and servers regarding characteristics that are not captured in the base-interface to the service. Prior meta-protocols do not support bidirectional meta information flow.

The imperative part of our meta-protocol is the ability to derive new service classes and merge them with those of the original library through exemplar-based programming. This provides good incrementality since only those service methods that are being customized need to be reimplemented. Exemplar-based server construction also improves encapsulation by hiding part of the library's internal class hierarchy from applications. In our real-time database research [2], we use this exemplar-based technique to customize database services according to the real-time and semantic characteristics of applications.

The remainder of this paper is organized as follows. Section 2 discusses our use of contracts to communicate semantic information needed for server construction. Section 3 explains our exemplar-based approach for selecting service classes and extending library functionality. Section 4 presents a C++ implementation of our technique. Section 5 discusses related work. Section 6 concludes and discusses future work.

2 Contracts

By analogy to contracts employed in civil law, a "software contract" metaphor is sometimes used to describe relationships between software entities. The software entities could be two interacting processes, an application and a software library, a server object and a client object or application, or a base class and a derived class. Wirfs-Brock *et al.* [3] and Meyer [4] consider software contracts to be properties of server classes. Since the server defines the contract, there is no way for an application to add clauses or establish its own contracts. With our technique, applications can create client-side contracts to specify requirements and communicate application-dependent semantic information to servers.

Our approach to software contracts is motivated by the following analogy with contract law. When a legal contract is established between a service provider and a client, there is both an express and an implied contract. The express contract consists of the specific clauses in a contract document. The implied contract consists of the reasonable and customary duties of that kind of service provider. For example, a contract with a plumber might specify the brand of faucet to install in a kitchen. If the plumber installs the right type of faucet but the plumbing leaks, the plumber is liable for damages even if the contract does not specifically mention leaks. This is because a plumber's professional duties routinely include leak-free installation of plumbing. A leaky installation is a violation of the implied contract.

We consider the methods and any class invariants exported by a service class to constitute an implied contract between the service class and the application. By instantiating a server object from a service class, an application establishes an implied contract with the server that covers most aspects of its subsequent use.

However, just as in legal contracts, an application may want to specify explicit terms that must be fulfilled in addition to the implied terms. It may be that only certain specialized service classes

can satisfy the explicit terms. In this case, the contract can be used to select an acceptable class from the general population of service classes. In other cases, the explicit terms might relax certain constraints and thereby permit server objects to optimize various aspects of their services. For example, a server object that supports concurrent access could use simplified locking protocols if it knew the application would not perform concurrent update operations. This semantic constraint could be supplied by the application in the contract. Therefore, explicit client-side software contracts are complementary to implicit server-side contracts. In the remainder of this paper, we restrict our discussion to client-side software contracts, which we simply call "contracts."

Contracts are composed of clauses that contain either requirements or preferences. Requirements, such as "persistent," are mandatory. Preferences, such as "minimize_execution_time," can be used to decide between servers that meet the mandatory requirements. If no service classes in the library can meet the requirements of the contract, then the library can set an error flag or throw an exception.

When contracts are used to select and configure the implementations of server objects, the contract becomes a declarative meta-protocol for those services. Selecting servers through contracts resembles service specification and acquisition in distributed computing systems [5, 6], except our server objects are much lighter-weight and are constructed from local libraries rather than remote server processes. In the next section, we discuss how client-side contracts can help determine which customized service classes to use.

2.1 Contracts for Customization

Through customization, object-oriented programming can partially overcome the classic tradeoff between flexibility and efficiency in software libraries. Instead of supporting a single, general-purpose implementation of a function or abstract data type, an object-oriented library can provide a variety of specialized classes that collectively cover the same domain but are individually more efficient than a general-purpose implementation. Specialized classes can be more efficient since the best implementation of a given service often depends on the patterns of use within the application.

For example, a Set class that keeps members in a hash table would perform well if a client mainly tests for set membership. However, a linked-list internal representation might be better if memory is scarce, the set contains few members, or the application adds members frequently and rarely tests for membership. Instead of using a single compromise implementation, an object-oriented library can contain multiple compatible classes, each optimized for certain operations. This example is realistic: the library of generic container classes supplied with the gnu C++ compiler includes eleven customized **Set** classes, each using different underlying data structures and algorithms. An application using such a library can choose the customized class that best meets its needs.

However, with the flexibility of choosing from a group of similar server classes comes the burden of understanding their subtle differences and making a good choice. As libraries become larger and more complex, this problem becomes increasingly difficult. Furthermore, since many of the differences between customized classes may reflect semantic differences that are not expressible in the syntax of the language, there is a danger of mismatch between the semantics supported by a customized server class and its actual use in an application. For example, a server object used in a multithreaded environment might use a concurrency control protocol that supports a single writer and multiple readers. If the single writer restriction is violated by the application, as could happen accidentally since the restriction is only implicit in the service implementation, the object is likely to become corrupted.

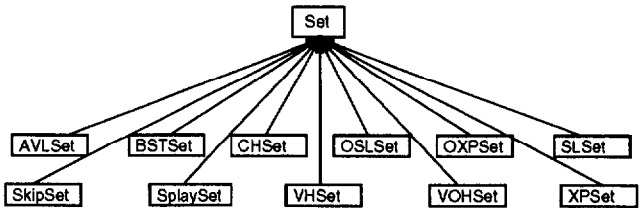
Class browsing tools are often proposed to assist application writers in selecting classes [7], but these tools expose the full complexity of the class hierarchy and do not enforce any semantic restrictions. Furthermore, applications that explicitly use

concrete service classes in a customized class hierarchy can become dependent on the internal class structure of the library. Such dependencies make future reorganizations of the library classes difficult to accomplish without propagating changes to existing applications. Therefore, applications should be kept as independent as possible from the structure of a library's internal class hierarchy. Class browsing tools do not address this need.

The most common approach to choosing a customized class is to instantiate the class by name. The customized characteristics are usually encoded in the class name. For example, the generic container class library distributed with the gnu C++ compiler contains a base class and eleven customized classes that implement sets with different data structures and algorithms: Set, AVLSet, OSLSet, VHSet, BSTSet, OXPSet, SkipSet, VOHSet, CHSet, SLSet, SplaySet, and XPSet. While this technique for specifying customization is easily understood, in practice it is unwieldy. As additional semantic attributes such as persistence or concurrency are added, the class names either become very long or very cryptic. It also becomes difficult to remember the correct order for semantic attributes in class names.

Worse yet, each combination of semantic attributes implies a unique class. Thus, the addition of new semantic attributes results in an exponential explosion in the number of classes. If persistence and two types of concurrency control (e.g., single-writer and multiple-writer) were added to the gnu Set classes, the eleven subclasses would become sixty-six (persistent and non-persistent versions of the original classes plus persistent and non-persistent versions for each type of concurrency control). Figure 1 illustrates the class hierarchy for the Set classes and how an application creates an instance of a Set class.

Because selecting an appropriate customized service class is so dependent upon application semantics, software contracts specified by applications can aid in the selection process. Instead of using class names directly, applications can use contracts to provide a mapping between application requirements and service classes. With this more flexible means of communicating semantic requirements, it is possible to avoid unnecessary proliferation of classes by supporting several combinations of semantic attributes in a single class. Exemplar-based programming, discussed in Section 3, can further reduce the number of distinct classes required to implement customized services. For instance, a single concurrent implementation of a class could support several locking protocols for single-writer or multiple-writer semantics, each protocol represented by a separate exemplar of that class.




```
VHSet<int> application_set; // application must explicitly choose the class
```

Figure 1: Generic Set Classes.

In our implementation, contracts are composed of constraint clauses encoded in character strings. These contract strings are passed by applications to the service library at runtime during server object initialization. The syntax of the contract language is implementation-specific. A typical contract string might be: "range_checked; lookup_time<=O(log n);". Using strings for contracts is simple, portable, convenient, and offers more flexibility than plausible alternatives such as defining language extensions for processing contracts at compile time. An application may need to dynamically determine contract constraints at runtime, so compile-time contract processing is not always possible. Furthermore, by leaving contract interpretation to class member functions rather than embedding it in the compiler, we preserve the ability to define new types of constraints with whatever syntax is most convenient in the context of a particular class. It is even possible to define new constraints in customized subclasses without modifying the base class. This provides great flexibil-

ity and helps keep contract constraints orthogonal to the class hierarchy. Another reason we chose to use character strings for contracts is that our real-time database performs remote object creation by passing contracts across the network using remote procedure calls. Character strings are easy to transmit via RPC, whereas more complex contract representations (i.e., contract objects) would be possible but more difficult to support.

The contract string is supplied as a parameter to an abstract service class constructor function. The abstract service class is equivalent to the "abstract factory" of [8, 9]. Multiple classes corresponding to concrete implementations of the abstract service can exist within the library, but applications need never know which concrete class will be used. The factory class constructor examines the contract string during object initialization and either creates a new object to meet the terms of the contract or rejects the construction request. We use exemplar-based programming to implement the class factory. Section 3 describes our use of exemplar objects in more detail.

For example, an application might declare a persistent array object as follows:

```
MdartsArray<int> parts_list("parts_list",
   "persistent; range_checked; sparse;
size=1000");
```

The prefix MDARTS is the acronym for our real-time database system [2]. In this case, the abstract factory class **MdartsArray** is specified along with the database name of the object and its contract string. Some constraints, such as "persistent", might be supported only by specialized subclasses that access a disk-based database system. Some, such as "size", can be implemented in the **MdartsArray** base class and be inherited by the subclasses. The mapping of contract constraint clauses onto the subclasses of the **MdartsArray** class is of no concern to the application. The library either will construct a customized, correctly-configured server object or will signal an error (return NULL or raise an exception).

2.2 Advantages of Using Contracts for Customization

There are many advantages to customization through software contracts. One advantage is that application-specific semantic attributes can be explicitly stated in object declarations. This is especially helpful when the attributes represent hints to server objects that enable various optimizations. By declaring these hints, the application is expressing a willingness to abide by whatever restrictions are implicit in the hints.

For example, single-writer concurrency control protocols can reduce locking delays compared to more general concurrency control methods [10]. However, a server object cannot control application behavior to ensure that the single-writer restriction is followed. With contracts, a single-writer service class would not be chosen unless the application explicitly indicated in the contract that it would avoid concurrent updates (by specifying a concurrency constraint such as "exclusive_update"). For software reliability and maintenance reasons, it is important that such restrictions be declared in the code itself rather than existing only in documentation or in the memory of the application developer.

Another important advantage of customization through contracts is that it allows applications to maintain a simplified view of the library's class structure. We call this "class hiding" because the class hierarchy beneath each abstract factory class is hidden from applications. The **MdartsArray** example discussed above illustrates this idea. There could be dozens of different classes in the library that support the abstract interface defined in the **MdartsArray** base class. Because the semantic attributes that determine which subclass to use are passed in the contract, the application can safely ignore the underlying complexity and still be assured that a server customized to its needs will be created. If no server can be created, the library can signal an error condition (throw an exception).

2.3 Disadvantages of Contracts

Naturally, there are also disadvantages to using client-side contracts, and the technique should not be applied indiscriminately. Contracts add complexity to the library implementation, and they impose runtime overhead as they are evaluated during server initialization. Contracts may not be worthwhile for libraries with few customized class hierarchies. The runtime overhead during initialization is amortized over subsequent server use, so the efficiency gained through customization in server functions must exceed the initialization overhead for the method to be worthwhile. If an application wants to avoid the overhead of processing contracts during initialization, it always has the option of explicitly specifying a service class. Doing so will lose the benefits of class hiding and semantic checks, but that decision can be left up to an application.

In general, contracts are best for server objects that will be used frequently by applications and that can achieve significant performance improvement through customization. However, even for cases in which efficiency gains through customization cannot justify the use of contracts, contracts may still prove useful for specifying semantic constraints to reduce errors in server usage.

Another limitation of our contract implementation is that errors in contract strings are not detected until runtime. This problem is unavoidable if dynamic creation of contract strings is allowed. Finally, since it is unknown until runtime which service classes will be used, application executables might become very large as they incorporate all of the customized classes. We address this issue in Section 3. Although contracts are not necessarily useful in all circumstances, a library need not support contracts for every service class hierarchy. If one or more of the following criterion are met, contracts may be appropriate for that hierarchy:

- Multiple service classes with similar functionality are present in a hierarchy.
- Server objects can achieve significantly better performance if they are implemented for special cases.
- Server classes depend on application usage patterns such as no concurrency or restricted concurrency (e.g., single-writer, multiple reader).
- Server characteristics important to applications are outside the language syntax or semantics (e.g., persistence, concurrency, memory requirements and real-time performance).

If contracts are restricted to static values, it might be possible to do the contract processing at compile time. This would detect errors in contract strings and eliminate the runtime overhead of selecting service classes. If the compiler itself could not be modified, a preprocessor could determine which concrete service classes should be used for each server object. The preprocessor could then replace the factory construction call with an explicit call to the concrete class constructor.

2.4 Types of Constraints in Contracts

An important design decision in implementing contracts for a class library is what types of constraint clauses will be supported. We do not believe it is appropriate to seek a universal taxonomy of constraint types, because software designers should be free to evolve contracts and constraints to express whatever semantics a library or application domain needs. This philosophy in part motivated our decision to use character strings to implement contracts. Classes in a hierarchy can interpret the constraint strings using whatever technique is appropriate, from simple string comparisons to parsing and interpreting some constraint language specific to that hierarchy. Nevertheless, it is illustrative to consider the constraints we have developed as part of our real-time database research. Our list of constraints is still evolving, but thus far we have identified the following generic constraint types:

concurrency semantics: Specify whether multiple clients will share the server object and how many concurrent writers are allowed.

persistence: Indicate if persistence is required for this server object.

staleness: Signal a problem if data accessed by a read transaction has not been updated within the specified period.

read and write transaction response times: Specify time constraints on database transactions according to needs of real-time applications.

read-only or write access: Declare access restrictions for particular database objects. Permissions are checked during initialization to reduce overhead during transaction processing.

units for numeric values: Configure the server object to scale numeric values of data it manages according to the units an application needs.

transaction priorities: Declare priority to use for this client in real-time transaction scheduling.

type constraints for service classes: Specify that the server be of a particular class or sub-hierarchy in the library class hierarchy.

2.5 Meta-level Queries

Thus far, we have described our contracts exclusively in terms of their use in choosing and constructing suitable server objects. However, it is also possible to use the contract meta-protocol to extract information from the server object after it has been instantiated. In most cases, an instantiated server object will exceed the specifications of the contract. It can be useful for a client application to discover what the actual characteristics of the server object are once it is constructed. To support bidirectional meta-level information flow, the server class can include methods for directly querying for server characteristics. Once the code for contract processing during object construction is written, relatively little effort is required to add support for direct meta-level queries.

For example, a contract in a real-time application might specify a timing constraint for reading the state of a database object. If the database object is successfully constructed, the application will know that the timing constraint will be met. However, the application might be able to relax its constraints on other objects if it knew the extent to which this object exceeded the performance specified in the contract. A real-time database class could export a method called QueryTiming() to permit such queries:

```
Time readTime =
    dbArray.QueryTiming("read(element)");
```

The class of dbArray would already need to implement a method for computing response times to evaluate contract constraints such as "read(element) <= 50usec;", so most of the work to add query support would already be done. QueryTiming() could call the same method and simply return the response time instead of performing a comparison.

3 Exemplars and Customized Classes

Exemplar-based programming, in which prototype objects play a role similar to that of classes, is often cited as an alternative to more traditional object-oriented architectures. For example, the Self language uses exemplars and delegation to dispense with classes altogether [11]. While exemplars in Self form the basis of a complete programming paradigm, exemplars can be useful in a class-based object-oriented context as well [12]. Coplien illustrates the use of exemplar-based programming in C++ [13]. In our implementation, we combine software contracts with Coplien's autonomous generic exemplar idiom (in which exemplars register themselves with a base class and object construction requests iterate over the exemplars).

Exemplars are special objects that are prototype representatives of an entire class. In general, a class can have multiple exemplars, but often only a single exemplar is used. Given an exemplar object, applications can construct copies of the exemplar by invoking a special clone() method. Because exemplars are objects, they can be stored in data structures. In some object-oriented languages, classes are first-class objects, so class objects could be used with our technique instead of exemplars.

Rather than choosing a specific service class, an application chooses a base class and specifies the rest of its requirements in a contract string. The base class is an abstract factory class that only defines the service interface. The contract is passed to the population of exemplars derived from that base class. The exemplars then bid on the contract to determine which class meets the application's requirements. The winner of the bidding process is cloned, and the clone object is returned to the application.

If a given service class can be configured for a variety of usage patterns, an exemplar can be created for each configuration of that class. This reduces the number of distinct classes needed to reflect a combination of service characteristics.

One of the key advantages of using exemplars for class selection is that the "class factory" does not need to know how many concrete classes it contains. Users needing special-purpose concrete classes that were not supplied in the original library can derive those classes from some point in the library hierarchy, override the specific methods that need tuning, and reflect those differences in the exemplar's contract processing code. Since the exemplar adds itself to the factory, this new class will automatically be considered as a candidate for future server creation with no changes required in the implementation of the abstract factory class or existing application code. Without the exemplar mechanism, existing applications can benefit from new service classes only if the application code is changed to specify the new classes or the factory object-creation function is modified to include the new classes. The abstract factory approach also allows the library implementer to restructure the internal hierarchy of concrete classes without disturbing existing application code.

For example, a library might initially contain an array base class and two customized subclasses: one that supports concurrency and one that does not. Each of these subclasses would contain various configuration options to support different combinations of semantic attributes. Suppose applications using this library are developed. Now suppose that the library developer decides to split the class that supports concurrency into three separate classes, each customized to support a subset of the attributes supported by the original concurrent class. The developer may want to do this to improve efficiency. As long as applications use the array base class and specify semantic requirements in contracts, multithreaded applications whose contracts originally mapped to the single concurrent class will now automatically use one of the new classes. No source code modifications in the applications are required. The programs need only be relinked with the new library.

Given exemplar-based object construction, there are still numerous implementation issues to consider. For instance, how are the exemplars organized?, how is the bidding process accomplished?, how can applications avoid linking in unneeded exemplars?, etc. In the remainder of this section, we consider these issues. In Section 4, we present an example C++ implementation of contracts and exemplar-based object construction.

Since we are interested in groups of exemplars derived from a common base class, the data structure containing the exemplars should belong to the base class. The simplest way to do this is to create a linked list for each abstract factory class. Exemplars of classes derived from the factory class are added to the list during exemplar initialization. Bidding can then proceed by iterating over the list and submitting the contract to each exemplar in turn. Either the first exemplar to satisfy the contract is cloned or the exemplar that best satisfies the contract is cloned. In the former case, the iteration proceeds until one of the exemplars clones itself. In the latter case, each exemplar returns a "bid" value in response to the contract. The function performing the exemplar iteration keeps track of the most attractive bid and clones that exemplar once all of the bids are examined.

Clearly, if large numbers of exemplars are associated with each base class, iteration over all of them will be slow. If the first exemplar to satisfy the contract is cloned, performance will be somewhat better. However, in this case the order of exem-

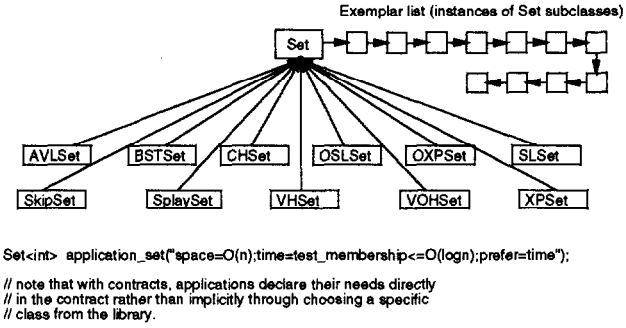
plars in the list may influence which service class is constructed. Since applications might prioritize different service characteristics, there may not be a single list ordering that is best for all applications. Nevertheless, the "first bid wins" approach does ensure that the server returned will satisfy the requirements specified in the contract.

To improve the performance of the bidding process, one could use more sophisticated techniques than iteration over a linked list. The selection of a service class can be viewed as a search process over exemplars using the contract as the key. If the exemplars are organized during library initialization into a classification network or a signature-based hash table, the search could be guided at runtime by the contract. If there are many exemplars, this could dramatically reduce the number of exemplars asked to bid on a contract.

However, the value of complex algorithms must be weighed against their cost. More complex data structures require more memory and more complex search algorithms. Since the library maintains multiple exemplar lists, each attached to a different base class, most of the exemplars in the library are eliminated from consideration when the application specifies the base class. Since the exemplar bidding process is performed only during server object construction, it is unlikely to occur inside tight application loops where efficiency is crucial. The best technique for exemplar bidding ultimately depends upon the particular class hierarchy and expected patterns of use by applications. This is an interesting research problem on its own.

A naive implementation of contracts and exemplars would include all exemplars (and their associated code) in applications using the library. If most of these exemplars are never used (cloned) by an application, which is likely, this means lots of unused code will be linked into the application. Furthermore, the presence of unused exemplars will slow the bidding process. Ideally, one would like to have each exemplar list contain only those exemplars that will be used by the application. Unfortunately, this information is not known until runtime.

If the contracts do not depend on runtime information, this problem can be solved with the following technique. For each abstract factory class there is a header file that causes all customized exemplars to be linked into an application. Applications during development and testing use this header file so all exemplars are included. If a certain mode is enabled during testing, the base class keeps track of which exemplars are actually cloned in each application run. As the application terminates, the base class exemplar (in its destructor) writes a new header file that includes only the classes of cloned exemplars. Applications ready for production can use the new header files and thereby avoid linking in unused exemplar code. With this technique, the exemplar-based approach to customization does not necessarily lead to bloated code size. Another approach for eliminating unused exemplars would be to process contracts at compile time or during a preprocessing stage. However, as memory prices fall and operating systems add support for shared libraries, code size becomes less important. Therefore, in the long run it may be unnecessary to eliminate unused exemplars.



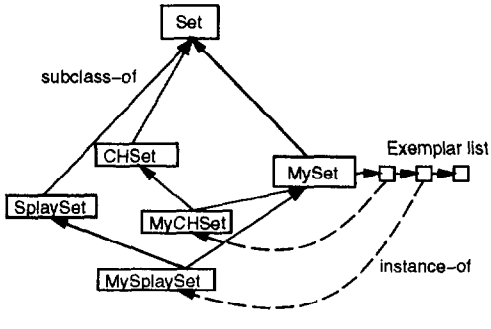

```
Set<int> application_set("space=O(n);time=test_membership<=O(logn);prefer=time");

// note that with contracts, applications declare their needs directly
// in the contract rather than implicitly through choosing a specific
// class from the library.
```

Figure 2: Generic Set Classes With Exemplars.

Figure 2 shows how exemplars and contracts could be used to simplify the application interface to the generic Set classes introduced previously. The object declaration below the diagram in Figure 2 shows that applications do not need to specify a particular service subclass if exemplars are used to create the servers. Instead, the application can use the base template class, Set<T>. This example raises an interesting question: how difficult is it to add support for contracts to an existing class li-

brary? If one has access to the library source code, it is possible to add constraint-checking methods and exemplar objects to the existing classes.

However, what if it is not feasible or not desirable to modify the library source code? Our basic contract and exemplar techniques would not work in this case, since the exemplars are instances of the classes, and they must include constraint-checking methods to bid on contracts. There are two alternatives to modifying existing library code. One possibility is to create a new class hierarchy derived from the original library using multiple inheritance to add the necessary methods. This approach is illustrated in Figure 3. The class **MySet** in Figure 3 contains the exemplar list and the methods for checking contracts and selecting exemplars. The new subclasses such as **MyCHSet** inherit the constraint-checking interface from **MySet** and also inherit the **Set** functions from the original library classes. Applications use the cloned exemplar objects through the **MySet** interface.

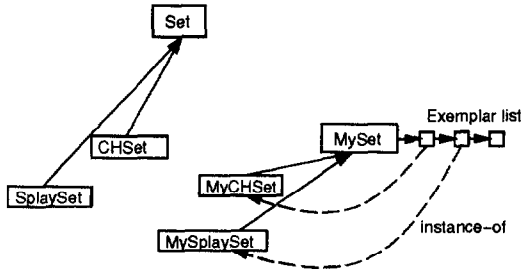



```
MySet<int> application_set("space=O(n);time=test_membership<=O(logn)");
```

Figure 3: Adding Contract Support via Inheritance.

A second approach is to create shadow classes that simply encapsulate knowledge of the existing classes and do not inherit from them. The exemplars of the shadow hierarchy process the contracts and determine if the classes they represent would satisfy them. When one of these shadow exemplars is chosen, it creates a new instance of the original class it represents instead of cloning itself. Figure 4 illustrates this approach.

Note that it is not actually necessary to create




```
Set<int> * application_set =
        MySet<int>::make("space=O(n);time=test_membership<=O(logn)");

// note that the application interface with a shadow hierarchy requires a different
// syntax: an explicit call to MySet<int>::make() is required, and a pointer to
// an original Set object is returned.
```

Figure 4: Adding Contract Support via a Shadow Hierarchy.

a complete shadow hierarchy corresponding to the original class hierarchy. A shadow exemplar could contain knowledge of multiple classes in the original hierarchy and could represent all of those classes during server construction. Furthermore, it is possible to add new, application-defined service classes without modifying the original library by merging the exemplars of the new classes with shadow exemplar(s) in the abstract factory.

4 Example C++ Implementation

Although our approach to contracts and customization is not language-specific, our prototype library includes a C++ implementation of exemplar-based customization. In this section, we present this implementation. We first discuss support for constraint processing in the base class "**Md_Base.**" We next present two example service classes: an abstract factory class called "**MdartsArray**" and a customized service class called "**RangeCheckedArray**" that supports the semantic constraint "range_checked". For brevity, part of the class hierarchy and some class member functions are omitted.

```
// The "Constraint" struct is used to store one
(constraint,operator,value)
// clause.
struct Constraint {
    Constraint(int ct, char *c, char * o, char * v);
    ~Constraint();
    int constraint_type;
    char * constraint;          // constraint name
    char * oper;                // operator string
    char * value;               // value string
};

class Exemplar { public: Exemplar(){} };

class Md_Base {
protected:

    // ConstructServer() parses the constraints and
    // tries to clone an object that meets them. If
    // successful, returns server object, else NULL.
    //
    Md_Base * ConstructServer(const char *
                contract) {
        Md_Base * ex;      // exemplar object ptr
        Plist<Constraint> cl;
        MakeConstraintList(contract,cl);

        // find an exemplar that meets all constraints
        Plist<Md_Base> & elist =
        getExemplarList();
        for (Pix p = elist.first(); p; elist.next(p)) {
            ex = elist(p);
            if ( ex->checkAllConstraints(cl) )
                return ex->clone();
            }
        return 0;
    }

  // Pure virtual functions below. These must
  // be implemented by derived service classes.
  virtual void registerExemplar(Md_Base * ptr) = 0;
  virtual int   checkConstraint(const
      Constraint& c) = 0;
  virtual void  stageConstraintCheck() = 0;
  virtual Md_Base *   clone() = 0;
  virtual Plist<Md_Base>& getExemplarList() = 0;
};
```

Code common to all exemplar-based class hierarchies is factored into the abstract base class called "Md_Base." This class contains methods for parsing contract strings and cloning server objects from lists of exemplars. Since Md_Base declares pure virtual functions needed for exemplar-based object construction, derived classes are forced to implement the required functions.

The Exemplar class is used as a dummy parameter to a special constructor in each derived service class that adds the exemplar to the base class exemplar list. By declaring a static pointer to the exemplar in the service class, C++ static member initialization can be used to automatically construct and register exactly one exemplar per class. This technique is borrowed from Coplien [13].

```
class MdartsArray: public Md_Base {
public:
    static MdartsArray * make(
        const char * contract) {
        return (MdartsArray *)
        ConstructServer(contract);
    }
    // public MdartsArray methods here ...

protected:

    // constructor for array object
    MdartsArray(int s) {
      sdatap = shared_memory_malloc(
      sizeof(shared_data)+(s-1)*sizeof(int) );
      sdatap->theSize = s; }

    // define basic constraints recognized by
    // MdartsArray classes.
    enum ConstraintType { unknown = -1, size,
    range_checked, expandable, sparse,
    persistent, concurrency };

    // array representation: size and first element in
    // array, stored together in shared memory. A
    // more realistic class would use templates
```

```
struct shared_data {
   int    theSize;
   int    theArray;      // start of array
};
shared_data * sdatap;

// list of derived exemplars – subclasses add
// their own exemplars to this list by calling
// registerExemplar().
static Plist<Md_Base>   TheExemplarList;

Plist<Md_Base> & getExemplarList() { return
TheExemplarList; }

// Implementation of pure virtual functions
// defined in class Md_Base.
void registerExemplar(Md_Base * ob) {
   TheExemplarList.prepend(ob); }
void stageConstraintCheck() {
   sdatap→theSize = 1; } // initialize state

// check a single constraint
int  checkConstraint(const Constraint& c) {
switch (c.constraint_type) {
   case size:
      sdatap→theSize = atoi(c.value);
      return 1; // success
   default: // only "size" supported by base
      return 0;
   }
}
};

// definition of static (one per class) exemplar list
Plist<Md_Base> MdartsArray::TheExemplarList;
```

The constraint checking function in each service class consists of a switch statement over the enumerated constraint type. This design permits relatively efficient processing of constraints. The contract string is parsed once and converted to a list of Constraint structures. Once this is done, the constraint-checking methods of the exemplars can iterate over the Constraint list and need not perform expensive string comparisons to determine the type of constraint to check.

Note that class MdartsArray exports a public function called "make()" that invokes the Md_Base::ConstructServer() function. This is the function used (directly or indirectly) by applications to create customized server objects. A direct use of make() by an application would look like: MdartsArray<int> *array_ob = MdartsArray::make("size=80,range_checked");. If make() fails to create a valid MdartsArray server object (as could happen if the contract contains constraints not supported by any of the exemplars), it returns NULL. If this object construction syntax is undesirable, it is possible to encapsulate the server object pointer and the make() call in an envelope class that forwards server functions to the internal object.

The make() function in each abstract factory class passes ConstructServer() the contract string and the list of exemplars of derived classes. ConstructServer() parses the contract and converts it into a list of Constraint structs. It then submits the constraint list to each exemplar until one of them returns a clone (we use the simple "first contractor to accept the contract wins" bidding technique). The Md_Base pointer returned by ConstructServer() is cast to a pointer to an MdartsArray object. This is a type-safe operation since all exemplars on the list belong to classes derived from MdartsArray.

Each exemplar's stageConstraintCheck() function is invoked by the Md_Base::ConstructServer() function before the constraints are checked. StageConstraintCheck() is used to initialize the state of the exemplar to eliminate carryover from prior contracts. For example, if an exemplar derived from MdartsArray processes the constraint "size=1000", it sets its internal theSize variable to 1000. This size variable is used to determine how big the clone object's array will be. If a subsequent and completely different contract is processed that does not specify the MdartsArray size, we want theSize to default to some constant value rather than retaining the value from the previous contract.

It may be that state variables must be initialized at multiple points in the class hierarchy. Therefore, if derived classes implement stageCon-

straintCheck() to initialize any state specific to that subclass, before returning they should also invoke their base class(es) stageConstraintCheck() function(s). Once the exemplar state is initialized via stageConstraintCheck(), each constraint in the contract is evaluated by the exemplar's checkConstraint() function. CheckConstraint() returns a boolean result to indicate whether the constraint is acceptable.

```
class RangeCheckedArray: public MdartsArray {
typedef inherited MdartsArray;
protected:
   // constructors
   RangeCheckedArray(Exemplar) {
   registerExemplar(this); }
   RangeCheckedArray(int s) : MdartsArray(s) { }
   static RangeCheckedArray * TheExemplar;

   Md_Base *  clone()  { return new
   RangeCheckedArray(sdatap→theSize); }

   int checkConstraint(const Constraint& c) {
      switch (c.constraint_type) {
      case range_checked:
         return 1;
      default:  // defer other constraints to base
         return inherited::checkConstraint(c);
      }
   }
};
// definition of static (one per class) data members
RangeCheckedArray *
RangeCheckedArray::TheExemplar =
   new RangeCheckedArray(Exemplar());
```

Each service class need only recognize a subset of the constraints defined by the abstract factory class. Like stageConstraintCheck(), checkConstraint() chains up the inheritance hierarchy, deferring to its base class when unrecognized constraints are encountered (see the default clause in the switch statement of RangeCheckedArray's checkConstraint() function).

Figure 5 illustrates the object creation sequence in our real-time database (called MDARTS). The

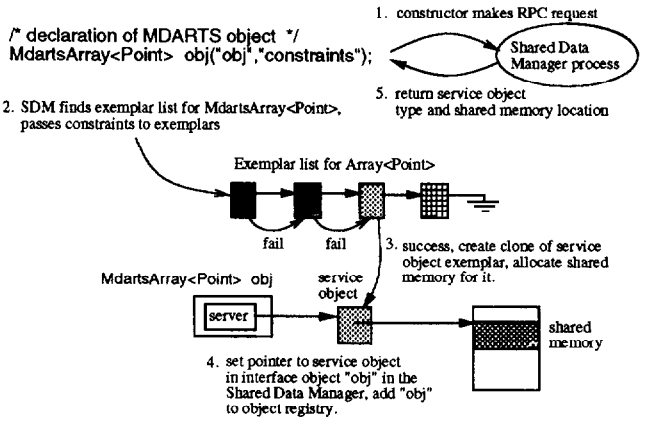


Figure 5: MDARTS Object Construction.

application declares an object, and the constructor for that object forwards the type information of its class and the object's name and contract string to an MDARTS database server. This server uses exemplar-based object construction to select and instantiate a server object that satisfies the contract.

5 Related Work

There has been considerable recent interest in allowing applications to influence implementation mapping decisions of services in the domains of system software and languages. Mach allows users to implement and replace some of the basic services of the operating system [14]. In [15], Krueger *et al.* present an approach to application-specific virtual memory management. Jones discusses tools for replacing system services by redirecting system calls to user code [16]. Anderson argues that operating systems should place as much functionality as possible under application control [17]. The trend toward application customization can also be seen in the domains of compilers (e.g., Open C++ [18], parallelizing compilers [19], and Traces in Scheme [20]), and window systems (Silica [21]). Kiczales and Lamping examine the trend toward service customization and identify the key themes of mapping dilemmas and meta-protocols [1].

The techniques we describe in this paper com-

bine meta-protocols with class hiding through abstract class factories. Gamma *et al.* [8, 9] discuss using abstract class factories to hide concrete classes from applications. We expand on this idea by showing how exemplar-based server construction allows applications to extend factories without modifying existing code. Furthermore, our declarative contract meta-protocol provides a flexible means of communicating application requirements and server characteristics that are not captured in the abstract service interface.

6 Conclusion

We have described a new approach to service customization and specification based on client-side contract strings and exemplar-based server construction. Our techniques permit the encapsulation of entire subtrees of classes in an object-oriented library. Instead of exposing the application developer to many similar customized classes, the library implementer can define a small set of abstract factory classes and hide the hierarchies of specialized concrete service classes from applications. Besides simplifying the application interface, this "class hiding" permits radical restructuring of the library implementation without breaking existing applications. Our techniques should prove very useful for the development and management of large class libraries.

Although our exemplar and client-side contract implementation requires class library implementers to follow certain protocols, the price is not high compared with the benefits of software contracts and class hiding. Much of the complexity of our technique is localized in the **Md_Base** class and in the abstract factory class of each service hierarchy. The additional support required in customized service classes is nominal. Nevertheless, in simple class libraries the additional complexity and overhead of our techniques may not be justified. Library designers can determine which service classes are sufficiently complex to benefit from our approach.

There are many ways our techniques could be extended. The runtime overheads associated with class factories could be eliminated if the server selection were performed at compile time. Furthermore, it would be interesting to develop more sophisticated techniques for organizing exemplars, conducting the bidding process, and negotiating constraints with exemplars if none of the exemplars were willing to bid on the original contract.

Acknowledgements

We would like to thank Gregor Kiczales and the other reviewers for their helpful feedback on an earlier draft of this paper.

References

[1] G. Kiczales and J. Lamping, "Operating systems: Why object-oriented?," in *Proc. of IWOOOS*, pp. 25–30, October 1993.

[2] V. B. Lortz, *An Object-Oriented Real-Time Database System for Multiprocessors*, PhD thesis, University of Michigan, March 1994.

[3] R. Wirfs-Brock and B. Wilkerson, "Object-oriented design: A responsibility-driven approach," in *Proc. of OOPSLA*, pp. 71–75, October 1989.

[4] B. Meyer, "Applying "design by contract"," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, October 1992.

[5] R. N. Chang and C. V. Ravishankar, "A service acquisition mechanism for the client/service model in cygnus," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 90–97, May 1991.

[6] K. Ravindran and K. K. Ramakrishnan, "A model for naming for fine-grained service specification in distributed systems," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 98–105, May 1991.

[7] R. Helm and Y. S. Maarek, "Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries," in *Proc. of OOPSLA*, pp. 47–61, October 1991.

[8] T. Eggenschwiler and E. Gamma, "ET++swapsmanager: Using object technology in the financial engineering domain," in *Proc. of OOPSLA*, pp. 166–177, 1992.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *Proc. of ECOOP*, pp. 406–431, 1993.

[10] K. Vidyasankar, "Concurrent reading while writing revisited," *Distributed Computing*, pp. 81–85, 1990.

[11] D. Ungar and R. B. Smith, "Self: The power of simplicity," in *Proc. of OOPSLA*, pp. 227–242, October 1987.

[12] W. R. LaLonde, D. A. Thomas, and J. R. Pugh, "An exemplar based smalltalk," in *Proc. of OOPSLA*, pp. 322–330, September 1986.

[13] J. O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison Wesley, 1992.

[14] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for UNIX development," in *Proc. Summer 1986 USENIX Technical Conference and Exhibition*, June 1986.

[15] K. Krueger, D. Loftesness, A. Vahdat, and T. Anderson, "Tools for the development of application-specific virtual memory management," in *Proc. of OOPSLA*, pp. 48–64, 1993.

[16] M. B. Jones, "Transparently interposing user code at the system interface," in *Workshop on Workstation Operating Systems*, pp. 98–103, April 1992.

[17] T. E. Anderson, "The case for application-specific operating systems," in *Workshop on Workstation Operating Systems*, pp. 92–94, April 1992.

[18] S. Chiba and T. Masuda, "Designing an extensible distributed language with a meta-level architecture," in *Proc. of ECOOP*, pp. 482–501, 1993.

[19] L. H. R. Jr., "A study on the viability of a production-quality metaobject protocol-based statically parallelizing compiler," in *Proc. of the Int'l Workshop on New Models for Software Architecture*, pp. 107–112, November 1992.

[20] G. Kiczales, "Traces (a cut at the "make isn't generic") problem," in *Proc. of the Int'l Symposium on Object Technologies for Advanced Software*, pp. 27–43, 1993.

[21] R. Rao, "Implementational reflection in silica," in *Proc. of ECOOP*, pp. 251–267, 1991.