Cleveland State University
EngagedScholarship@CSU

Electrical Engineering and Computer Science Faculty Publications

Electrical and Computer Engineering Department

5-2007

Serfs: Dynamically-Bound Parameterized Components

Nigamanth Sridhar
Clevealand State University, n.sridhar1@csuohio.edu

Follow this and additional works at: https://engagedscholarship.csuohio.edu/enece_facpub

Part of the Computer Sciences Commons

How does access to this work benefit you? Let us know!

Original Citation

Sridhar, N. (2007). Serfs: Dynamically-bound parameterized components. The Journal of Systems & Software, 80(5), 736-749. doi:10.1016/j.jss.2006.07.024

Repository Citation

Sridhar, Nigamanth, "Serfs: Dynamically-Bound Parameterized Components" (2007). *Electrical Engineering and Computer Science Faculty Publications*. 51.
https://engagedscholarship.csuohio.edu/enece_facpub/51

This Article is brought to you for free and open access by the Electrical and Computer Engineering Department at EngagedScholarship@CSU. It has been accepted for inclusion in Electrical Engineering and Computer Science Faculty Publications by an authorized administrator of EngagedScholarship@CSU. For more information, please contact library.es@csuohio.edu.

Serfs: Dynamically-bound parameterized components

Nigamanth Sridhar *

Electrical and Computer Engineering, Cleveland State University, 334 Stilwell Hall, 2121 Euclid Avenue, Cleveland, OH 44115, United States

1. Introduction

Traditionally, *parameterization* (Goguen, 1984) has been used as a technique for building generic data types, especially popular in the domain of data collections, such as stacks, queues, lists, etc. This view has been further popularized by template class libraries such as the C++ Standard Template Library (STL) (Musser and Saini, 1996). The recent addition of generics to Java (Bracha et al., 1998) and the .NET common language run-time (Kennedy and Syme, 2001) also endorse this view of parameterized components.

The essential idea with generic data collection components is that the component designer simply designs the collection while leaving some parts of the component (generally, the kind of items in the collection) unspecified. Later, when a client program wants to use such a generic component, it *instantiates*[1] the template by "filling in" the incomplete parts of the component definition, thereby resulting in a complete definition for the component.

Our view of parameterized components, however, is much broader. We view parameterization as a mechanism for hierarchical composition of (arbitrarily complex) components. This view is rooted in the idea that systems are composed of a number of (preferably independent) design decisions (Parnas, 1972). Each of these design decisions is encapsulated in its own module. Dependencies between these modules are kept at the abstract (interface) level, so that a change to one module does not ripple through the rest of the system – *change is localized*. Components are designed as templates, and are specialized by client programs that pick the appropriate design decisions at integration time by instantiating the template with parameters. Each parameter *defines* a design decision to be made; binding a particular parameter indicates *committing* to a particular decision.

As an illustration of this point, consider a distributed system in which different nodes in a network need mutually exclusive access to a given resource. It is common for such a network to use a mutual exclusion (mutex) layer in order to guarantee safe access to the resource. Clients in the network could then simply request access to the resource via the mutex layer. This design decouples network nodes from resource allocation. Drilling down further, the actual conflict resolution protocol[2] is a separate design decision. Depending on considerations such as network topology, density, etc., the choice of which conflict resolution protocol to use may vary. In our model of parameterization, the mutex layer would be designed as a parameterized component that can be specialized by a specific conflict resolution protocol based on deployment considerations.

An implementation of this mutex layer in C++, for example, would be a template class that takes a conflict resolution protocol object as a template parameter. Therefore, at design time, the software does not depend on the particular protocol that will be used, but just the fact that *some* protocol will be used. At compile time, the template is instantiated with an actual parameter, thus delaying commitment until a time where the choice is better informed (Thimbleby, 1988). We call this mode of parameterization **static parameterization**.

However, compile time may not be late enough to make this choice. Consider the case where the network is dynamic – nodes may be added or removed during execution. In such a system, the choice of conflict resolution protocol will also need to change. Static parameterization does not allow such dynamic change. We need a way of dynamically supplying parameters to a template – we need to be able to support **dynamic parameterization**. This is the primary contribution of this paper. We define a composition of design patterns called the **service facility** pattern, which is a design methodology that allows the construction of dynamically-bound parameterized components.

An important consideration in the design of this methodology is applicability. Being a composition of design patterns (Gamma et al., 1995), the service facility methodology is language-neutral (with a few restrictions), thus affording developers flexibility in their choice of implementation language. The main requirement that the implementation language must satisfy is that it should support *introspection*. That is, each object must be able to query data (such as its dynamic type) about itself. A common way for languages (such as Java, C#, etc.) to implement introspection is by way of a reflection library.

The rest of this paper is organized as follows. In Section 2, we describe the service facility pattern, and explain how the composition of abstract factory, proxy, and strategy patterns can be effectively achieved. Section 3 provides details of how service facility components can be implemented in C# (and Java), and Section 4 outlines some of the performance considerations to be accounted for in the implementation. In Section 5, we provide an alternate version of the design pattern that is consistent with the OO point of view. After presenting some related work in Section 7, we conclude with a summary of our contributions in Section 8.

2. The service facility pattern

The service facility pattern is a composition of three design patterns – strategy, abstract factory, and proxy. Each of these patterns independently solve different object decoupling problems: strategy deals with component assembly; abstract factory decouples object creation; and proxy deals with object maintenance. In this section, we show how these three patterns are composed to yield the service facility pattern.

The service facility pattern makes a distinction between two kinds of objects – service facility objects and data objects. **Service facility objects** (Serfs) are conceptually stateless, and provide functionality – they export zero, one, or more type(s) and operations defined on the type(s). The Serf can create data objects of the types it defines (these objects are stateful), as well as operate on them. **Data objects**, on the other hand, contain state, and have unique identities. There are multiple instances of these data objects, and all of these instances are maintained by the Serf that created them. This distinction is similar to the distinction that Szyperski makes between components and objects (Szyperski, 1999). Serfs are similar to what Szyperski refers to as components, and data objects are similar to what he refers to as objects.

2.1. Component assembly

The strategy pattern (Gamma et al., 1995) helps encapsulate algorithms as components, and allows a client object to pick from among different implementations of such algorithms at run-time. In the service facility methodology, the strategy pattern forms the basis for component assembly. Each Serf component in the system is a set of design decisions. Each of these design decisions is encapsulated in its own component, and delivered as *strategies*. At component design time, all the Serf sees are the interfaces of these strategy components (AbstractStrategy in the strategy pattern).

[2] For a comprehensive presentation of mutual exclusion and applications see (Lamport and Lynch, 1990; Lynch, 1996). For a list of such solutions, we refer the reader to Neilsen and Mizuno (1991).

The Serf, therefore, can be viewed as analogous to a template class in C++. Each of the abstract strategies is a parameter that needs to be supplied to the Serf in order to instantiate it. These parameters are supplied at component assembly time. In the case of the service facility pattern, component assembly time is delayed until the system has been deployed. Parameters are supplied as strategies by invoking methods on the Serf object. More details of how this is implemented is presented in Section 3.

For instance, in the mutex example presented in Section 1, the mutex layer component is designed as a Serf that takes as a parameter a conflict resolution protocol. This protocol is itself packaged in its own Serf. At design time, the mutex Serf is only dependent on the *interface* of the protocol Serf. The actual protocol is provided to the mutex Serf at assembly time.

2.2. Object creation

The abstract factory pattern (Gamma et al., 1995) solves the problem of object creation by designating a third party as a *factory* that creates objects. The client now requests the factory object for an instance of a *product* class, rather than calling the constructor of the product class. The abstract factory pattern is based on a manufacturing-industry metaphor. What happens if we use a service-industry metaphor to address the decoupling problem during object creation?

Consider a safe deposit box that can be rented from a bank. The client initially needs to ask the bank for one. The bank continues to hold the box; the client merely gets a key for it. However, any change to the contents of the box can be made only at the client's behest. The bank cannot add anything to or remove anything from the box on its own. In fact, the bank cannot even open the box (except possibly under extreme legal circumstances) because it needs the other key from the client. Similarly, the client cannot change the contents of the box on his own – he needs the bank's key to open it. In short, any change to the contents of the box is initiated by the client, and the client and the bank cooperate in opening the box and changing its contents.

This situation is different from the factory metaphor is several ways. The most important is that a factory's role is limited to product creation. After that, the factory is out of the picture and the client is on his own to change the product. The bank's role is significant throughout the lifetime of the safe deposit box because without the bank the client can do *nothing* to the box. The bank also controls the location of the box and is responsible for securing it so that only the client can access its contents.

We call the software analogue of a safe deposit box a **data object** because it holds information for a client (but cannot manipulate that data on its own). We call the software analogue of the bank a **service facility object** (or Serf, for short) – it creates data objects, as well as performs operations on those objects (at the client's behest).

Like the abstract factory pattern, the service facility pattern provides an abstract interface to clients of a component that can be used for creating instances of the type(s) the component exports. However, while the abstract factory pattern is based on a manufacturing-industry metaphor (whereby the factory is out of the picture once the object is created), the service facility pattern is based on a service-industry metaphor (the Serf always remains as an intermediary between the client and the object instances). When a client calls create( ) on a Serf to ask for a data object, the Serf creates not just the object, but also a handle to the object. Fig. 1 shows how objects are created in the service facility pattern: the client asks the Serf for an object; the Serf creates it, and a handle. The handle is returned to the client, and the object is hidden from the client behind the handle. From this point on, the client interacts with the object instance by giving this handle to the Serf.

2.3. Object maintenance

Continuing a little further along with the metaphor, we could say that the client receives a *proxy* to the safe deposit box. As is the case with the proxy pattern (Gamma et al., 1995), the client can access the box through this proxy. Notice that the bank can, if it is deemed necessary or desirable, change the physical location of the safe deposit box, as long as its

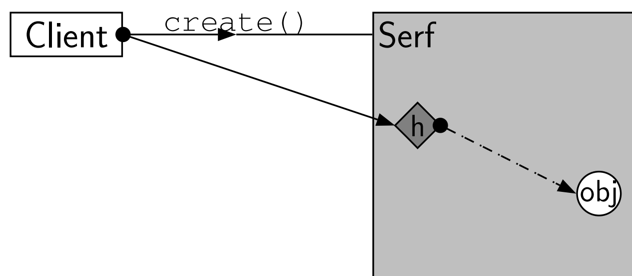

Fig. 1. Object creation in Serfs.

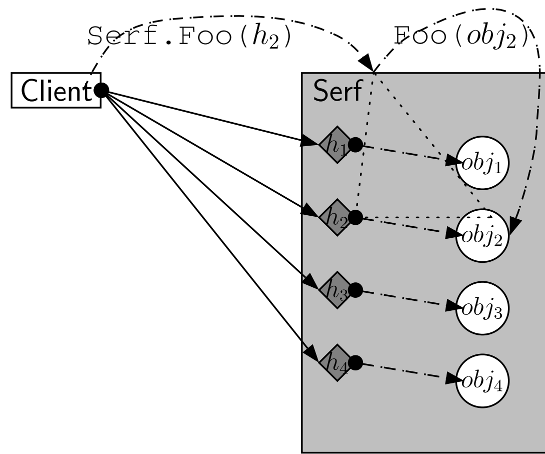

Fig. 2. Method redirection in Serfs.

contents are left unchanged. This does not affect the client's logical view of the box. The client is not concerned about where his safe deposit box is physically located, as long as he has access to it and he alone can control the contents of the box.

Through the lifetime of the program, when the client wants to modify data objects, rather than invoking the method on the object directly, it invokes the method through the Serf using the handle it received (which acts as a proxy for the data object). The Serf then finds the object that the handle refers to, and routes the method invocation to that object. Fig. 2 shows how method invocations are redirected in the service facility pattern. The client makes the invocation on its handle; the Serf redirects the call to the corresponding object. Conceptually, one can see all the objects created by a Serf as being held "inside" the Serf, with the Serf intercepting every method invocation going to each object. This, however, need not be the case in an actual implementation. The Serf and data objects can reside anywhere in memory independent of each other, even on different machines in the case of distributed systems.

The service facility pattern introduces an extra level of indirection between the client and the objects it uses. The key point here is that this level of indirection is *maintained through the entire lifetime of each object*. This extra level of indirection is the enabling factor for dynamic reconfiguration.

2.4. Separating code from data

As opposed to traditional object-oriented systems, where a class defines a single type and each instance of the class exports the operations associated with this type, under the service facility pattern, the operations on the data objects are defined in the Serf. The data objects are simply containers for state and do not export any operations. All operations on the data object are defined in the Serf. When the client wants an operation performed on a data object, it invokes the operation on the Serf object, and passes the data object as a parameter. Taking this (component-oriented) view as opposed to an object-oriented view does yield some advantages. Consider an object obj, which has a method defined on it called Meth that takes one parameter. An invocation to this method is of the form obj.Meth(param). In this case, the receiver of this method (obj) is a *distinguished parameter* to the operation – the method has access to the private state of this object alone, and not of the other parameters to the method, even if obj and param are of the same type. Apart from the cosmetic asymmetry that this notation leads to in the case of such *binary methods*, there are also technical reasons that the component-oriented approach is favored over the object-oriented one (Bruce et al., 1995).

In the case of the service facility pattern, since the methods are part of the Serf, the cosmetic asymmetry of binary operations (or for that matter, any polyadic operation) is not a problem. All arguments to the operation are expressed as parameters to the method, since the receiver of the method is the Serf object, and not one of the arguments to the operation (*e.g.*, serf.Meth(obj, param)). Moreover, the service facility pattern does not use inheritance to extend functionality, the problem shown in Bruce et al. (1995) does not arise either. Component functionality in Serfs is extended by using delegation and layering (Gamma et al., 1995; Sitaraman and Weide, 1994).

3. Implementing service facilities

Fig. 3 shows the UML design of a system with a client program that uses a concrete class named ComponentRep_R1. Further, ComponentRep_R1 depends on ParameterA_R1 to provide some of its behavior. This system is not flexible, since

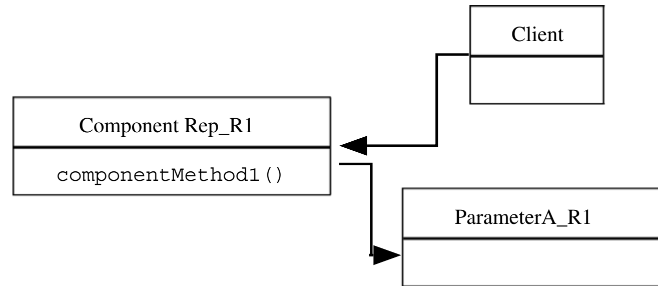

Fig. 3. UML design of a system with concrete dependencies.

any change in either ComponentRep_R1 or ParameterA_R1 will result in a change in the remaining classes in the system. In the rest of this section, we show how we can use the service facility pattern in order to decouple such dependencies, and convert the system so that Client depends on a component that provides the same interface as ComponentRep_R1, and is parameterized by the functionality that ParameterA_R1 provides through its own (separate) interface.

Fig. 4 shows the UML description of the same system using the service facility pattern in order to decouple the dependencies present. The ServiceFacility and Data interfaces are at the root of the service facility object and data object hierarchies, respectively. Every Serf implements the ServiceFacility interface, and the objects that a Serf creates all implement the Data interface. Listing 1 shows these two interfaces written in C#.[3] As can be observed from this listing, the Data interface is simply an empty interface. This interface is only used to provide a uniform structure across the service facility pattern, and to establish a syntactic difference between service facility objects, and data objects. The ServiceFacility interface has one method – create( ), which returns a Data object. Every service facility class must implement the ServiceFacility interface.

Listing 1: The Data and ServiceFacility interfaces

```
1 namespace ServiceFacility {
2   public interface Data {}
3
4   public interface ServiceFacility {Data create( ); }
5 }
```

Note that the create method of ServiceFacility returns an instance of Data. But ComponentRep_R1, which is what the Client really wants to use, does *not* implement this interface. However, recall that the client only gets a handle to the object instance, and not the object instance itself. In this particular case, the Client receives an instance of ComponentData_R1 (which does implement the Data interface). This object in turn holds a reference to an instance of ComponentRep_R1. So Client uses the ComponentRep_R1 instance through the handle it holds (the ComponentData_R1 instance).

3.1. Client view

The Client now depends on the ComponentSerf interface (dependency on an abstract component), rather than ComponentRep_R1 (dependency on a concrete component). The ComponentSerf implementations act as factories to create objects of type ComponentData. The Client therefore no longer needs to name the particular concrete class ComponentRep_R1 in order to create object instances. In fact, the only concrete component it needs to name in the new design is the specific implementation of ComponentSerf that it wants to use. Once the Client has picked the implementation of ComponentSerf, all future uses of this component can go through the interface reference. Switching to a new implementation of ComponentSerf will simply involve changing this one line of code. We now have a *single point of control* in the Client over which implementation is in use.

```
1 ComponentSerf cSerf = new ComponentSerf_R1( );
2 /* ··· */
3 ComponentData c = (ComponentData) cSerf.create( );
4 /* ··· */
5 ComponentData d = (ComponentData) cSerf.create( );
```

[3] Although the code samples presented are all in C#, corresponding constructs are available in Java as well.

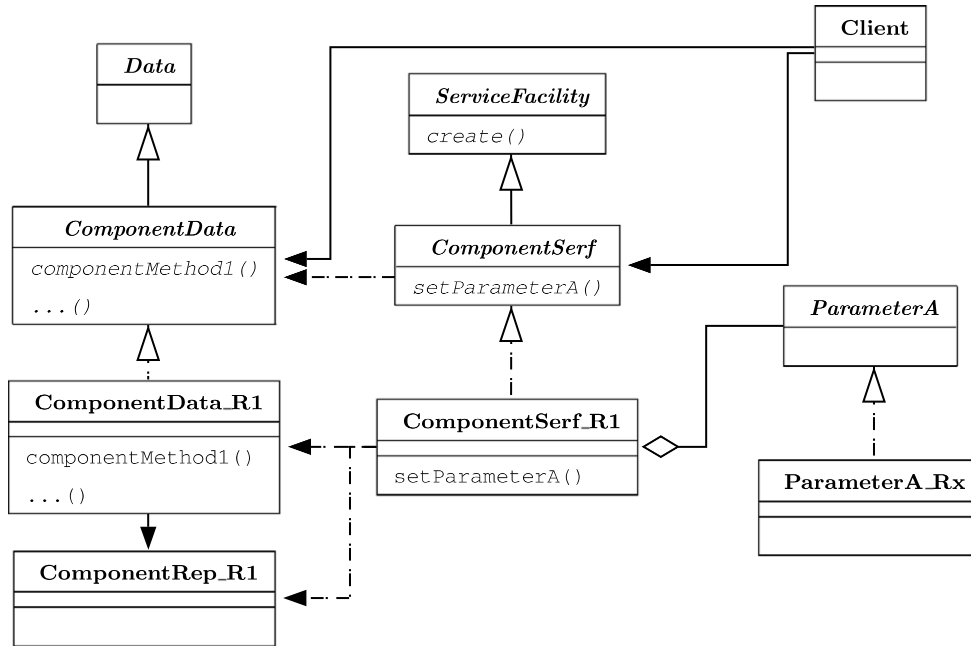

Fig. 4. UML design structure of the service facility pattern.

The service facility pattern offers another level of decoupling – separating the code that operates on a piece of data from the data itself. An important difference to note going from the design in Fig. 3 to the design using the service facility pattern is that the method componentMethod( ) that used to be in ComponentRep_R1 is now in ComponentSerf_R1. ComponentRep_R1 now only holds the data members, and does not include any of the interface methods. The ComponentSerf has these methods instead. The Client, instead of invoking componentMethod( ) on the object instance directly:

```
1 ComponentRep_R1 crep = new ComponentRep_R1( );
2 crep.componentMethod( );
```

now invokes the method on the ComponentSerf with the particular object instance handle as a parameter to the method:

```
1 ComponentData c = (ComponentData) cSerf.create( );
2 cSerf.componentMethod(c);
```

The other dependency in Fig. 3 is the dependency between ComponentRep_R1 and ParameterA_R1. In the design using the service facility pattern, this dependency is eliminated by making the ComponentSerf (which creates and manages ComponentData objects) a parameterized component, with ParameterA as a parameter to this component. During system execution, the Client invokes the setParameterA( ) method, passing it an instance of ParameterA:

```
1 ParameterA parA = new ParameterA_R1( );
2 cSerf.setParameterA(parA);
```

Now if the Client at some point wants to change the implementation of ParameterA that ComponentSerf should use, it can invoke the setParameterA( ) method again with an instance of the new implementation. This change in ParameterA automatically affects all instances of ComponentData, even the ones that have already been created. This is made possible by the extra level of indirection that is maintained between the client and the data objects that it uses.

There are some important safety considerations to be taken into account while effecting such re-binding. While a full treatment of these proof obligations is outside the scope of this paper, the most important of these are the following:

- the new parameter instance should be of the same (or a behavioral subtype) of the old parameter instance;
- component properties with respect to safety and liveness must be preserved.

Full details of such change during execution and correctness considerations are presented in Sridhar et al. (2003).

Since the template parameters are set at run-time, there is no way for the compiler to ensure that they are set, let alone in a type-safe way (*i.e.*, with actuals that would have allowed compile-time type-checks to succeed). At the point that the ComponentSerf object is declared and constructed, the compiler ‘‘believes’’ that the object is ready for use. However, under the semantics of Serfs, this object has not been fully instantiated and is therefore not ready for use. The client, therefore, has proof obligations to satisfy – that the Serf has been instantiated with appropriate parameters.

Since the parameters are supplied to the Serf template only at run-time, we have full knowledge of the actual *dynamic* type of each parameter, and the actual value of every data member and property in the Serf. The Serf uses this type information to perform run-time type-checking in order to ensure legal bindings.

Some examples of such run-time checks include:

- any method invocation to a Serf before it is fully instantiated is disallowed;
- each setParameter method checks the dynamic type of the parameter instance to ensure that the binding is legal.

A more detailed treatment of these type-safety checks is presented in Sridhar and Weide (2003). To make the task of programming easier, these type-safety checks can be enforced by dynamic instantiation-checking components. The DynInstaCheck tool described in Sridhar (2006) uses an aspect-oriented approach to generating such checking components.

3.2. Implementer view

Listing 2 shows the skeletal structure of a service facility class ComponentSerf_R1 in C#.

Listing 2: Structure of a service facility

```
1 public class ComponentSerf_R1 : ComponentSerf {
2   // typedefinition
3   internal class ComponentData_R1 : ComponentData {
4     internal ComponentRep_R1 rep;
5
6     internal ComponentData_R1( ) {
7       rep = new ComponentRep_R1( );
8       /* Initialize representation fields */
9     }
10  }
11
12  internal class ComponentRep_R1 {/* Data members */ }
13
14  // template parameter
15  private ParameterA pASerf;
16  public ParameterA PASerf {
17    get { return pASerf; }
18    set {pASerf = value; }
19 }
20
21  // list of objects created
22  ArrayList listOfObjects = new ArrayList( );
23
24  // rep method to extract real object instance internally
25  private ComponentRep_R1 rep(ComponentData cData) {
26    return ((ComponentData_R1) cData).rep;
27  }
28
29  // create method from ServiceFacility
30  public Data create( ) {
31    ComponentData_R1 newObj = new ComponentData_R1( );
32    listOfObjects.Add(newObj);
33    return newObj;
34  }
35
```

```
36   // componentmethods
37   public void componentMethod(ComponentData c) {
38     ComponentRep_R1 cRep = rep(c);
39     /*... */
40   }
41 }
```

There are five major sections in every service facility component:

Type definition (lines 2–12). A service facility exports some types – typically one, but sometimes zero, sometimes more than one. A type exported by a service facility is expressed as an *inner class*.[4] Moreover, this class is declared as *internal*, meaning that the class is not visible outside of the service facility component. The more important implication of this is that *only* the service facility can create instances of this class.
There are two inner classes in the code listing. The first one, ComponentData_R1, defines the handle that the client will be given in response to the create( ) method. The second class, ComponentRep_R1, defines the actual representation (data members) of the type. This class is invisible outside the service facility (even as an interface).
Template parameter definition (lines 14–19). The parameters to the service facility are specified here. Each parameter is represented by a private data member, and a public property that a client can use to control its value. Each property gets automatically translated to two methods – a getter and a setter. In the case of Java service facilities, each parameter to the service facility is represented by a private data member, and two public methods in the style of a JavaBean getter and setter.
The rep Method (lines 24–27). The client holds only a handle to the actual object instance. This handle is an instance of ComponentData_R1. Every method invocation to the service facility includes this handle as one of the parameters. For the service facility to perform the operation on the actual object instance (the ComponentRep_R1 instance that the handle corresponds to), it has to extract the representation instance from inside the handle. The rep method does this extraction, and returns the representation instance (the actual object) that the operation can then modify.
The create Method (lines 29–34). This method performs the factory function of the service facility. An instance of the Data object is created and returned to the client. The service facility also holds a reference to the newly created data object. This way, it can perform administrative maintenance (*e.g.*, dynamic reconfiguration) if needed.
Component method(s) (lines 36–40). All methods that used to be in the interface of Component_R1 are now in ComponentSerf_R1. Each method first invokes the rep (private) method on the ComponentData_R1 instance that is passed into the method. The operation is then performed on the representation instance (ComponentRep_R1).

4. Performance considerations

4.1. Extra level of indirection

The service facility pattern introduces an extra level of indirection between a client program and all data objects it uses. This means that, for every method invocation the client program makes on a data object, an extra level of indirection has to be traversed before the method is executed. This is a constant-time overhead that is seen on every method invocation, and is linear with the total number of invocations in a given application (Fig. 5).

In the normal object-oriented case, when a method invocation obj.m( ) is made, there is one reference to be navigated before the method can be executed. In the service facility model, method invocation is of the form serf.m(obj). In this case, the method invocation is made on the serf (at the same cost as a regular OO method call). Rather than executing the method itself, the proxy obj has to be first dereferenced to extract the representation object. Finally the actual method is executed to operate on this representation object. This is the extra cost introduced by the service facility model. However, this extra performance burden is a trade-off between the speed of execution and the amount of flexibility that is afforded by the programming model.

If, at the time of system design, no change is ever expected in a part of the system, then that part can be programmed without this extra indirection layer. That would definitely increase system performance. But if such a guarantee cannot be made at design time, and changes are expected, then this performance degradation (which is a constant-time degradation) may be acceptable, since it increases the flexibility of the resulting software system.

[4] In a language that does not support inner classes (such as C++), *friend* classes can be used to achieve a similar effect.

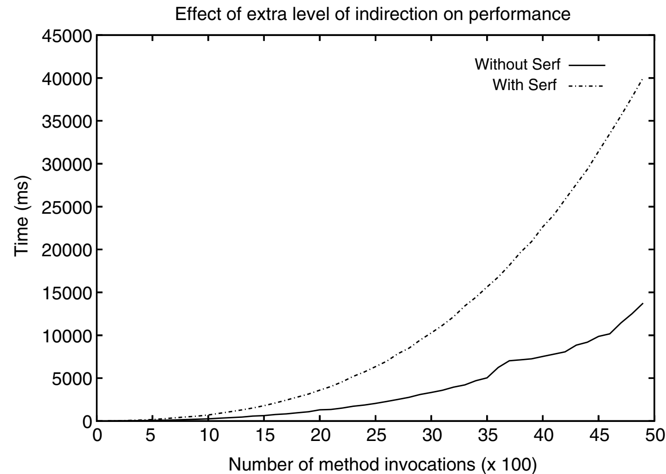

Fig. 5. Effect of the extra level of indirection on performance.

4.2. Weak references

The service facility is typically a long-running component, and may potentially never go out of scope. Since it holds a reference to every single object that it creates, these objects will also not go out of scope as long as the service facility is around, even if these data objects go out of scope in the client, or the client that uses the data objects goes out of scope. This could potentially result in large memory leaks, which could degrade the performance of the entire system.

In order to avoid this situation, the reference that the service facility holds on the data objects it creates is a *weak reference* (van der Linden, 1998; Robinson, 2002). A weak reference is a reference that lets the garbage collector delete it even when some object that refers to it is still alive. As long as there is at least one *strong* reference to an object, the garbage collector will not collect it. But as soon as the last strong reference has been destroyed, the object is marked for collection, and in the next iteration of the garbage collector, the object will be finalized. So if the client instance that requested a particular data object goes out of scope, this data instance will be finalized in spite of the fact that the service facility still holds a reference to the data object. With the service facility using weak references, the create( ) method will look like Listing 3.

While the use of weak references is a great way to ease the life of the programmer, weak reference support in the implementation language is not integral to the service facility methodology. If the implementation language does not support weak references, the Serf will have to do some extra work. Specifically, it will need to maintain reference counts for the data objects it creates, and will need to properly dispose of data object references when such objects are no longer being used by clients.

Listing 3: create( ) method with weak references to data objects

```
1 public Data create( ) {
2   ComponentData_R1 newObj = new ComponentData_R1( );
3   WeakReference wrObj = new WeakReference(newObj, false);
4   listOfObjects.Add(wrObj);
5   return newObj;
6 }
```

Listing 4: rep( ) method using lazy initialization

```
1 private ComponentRep_R1 Rep(ComponentData c) {
2   if (((ComponentData_R1) c).rep = null) {
3     ((ComponentData_R1) c).rep = new ComponentRep_R1( );
4     /* Initialize representation fields */
5   }
6   return ((ComponentData_R1) c).rep;
7 }
```

4.3. Lazy initialization

A client may sometimes want to declare and create an instance of a data object simply for use as a "catalyst" in some operation – the object is declared for the sole purpose of internal computation within the operation and is invisible outside. In such cases, why should the object be fully initialized? We can improve the performance of the create() operation by postponing the initialization to the time when the object first gets used. The service facility only creates the data object (the handle), and does not create the actual representation object instance. Then when an operation is invoked on a particular data object for the first time, the representation object instance is created at that time. This check, and creation of the object instance, is done in the rep method (Listing 4). Again, while lazy initialization is not a central feature of the service facility pattern, it is a useful optimization.

5. Serflets: Keeping code and data together

The service facility pattern provides a basis for programming component-oriented systems using object-oriented programming languages. However, the pattern does not require programmers to give up the object-oriented programming notation, and switch to the component view. While keeping the component view is certainly more advantageous in many ways, we present here an object-oriented version of the pattern that uses object-oriented notation.

The aspect of the service facility pattern that brings about this change in view from object-oriented to component-oriented is the fact that the methods defined on a type are housed inside the service facility as opposed to the data object itself. This is the aspect that we will relax in this section. We will move the methods from the service facility class (ComponentSerf_R1, Fig. 4) to the data class (ComponentData_R1). The new UML design is presented in Fig. 6. Although we move the methods into ComponentData_R1, this is not the same as the *product* object from the abstract factory pattern. We still maintain the extra level of indirection that the service facility pattern introduces.

The separation between the actual representation object and the logical handle that the client holds is still maintained. The only difference is that rather than the client having to invoke methods on the ComponentSerf object with the ComponentData object as a parameter, it invokes methods on the ComponentData object directly. We call this ComponentData object a *Serflet*.

```
1 ComponentSerf cSerf = new ComponentSerf_R1( );
2 ParameterA paramA = new ParameterA_R1( );
3 cSerf.ParamA = paramA;
4 ⋮
5 ComponentData c = cSerf.create( );
6 c.componentMethod( );
```

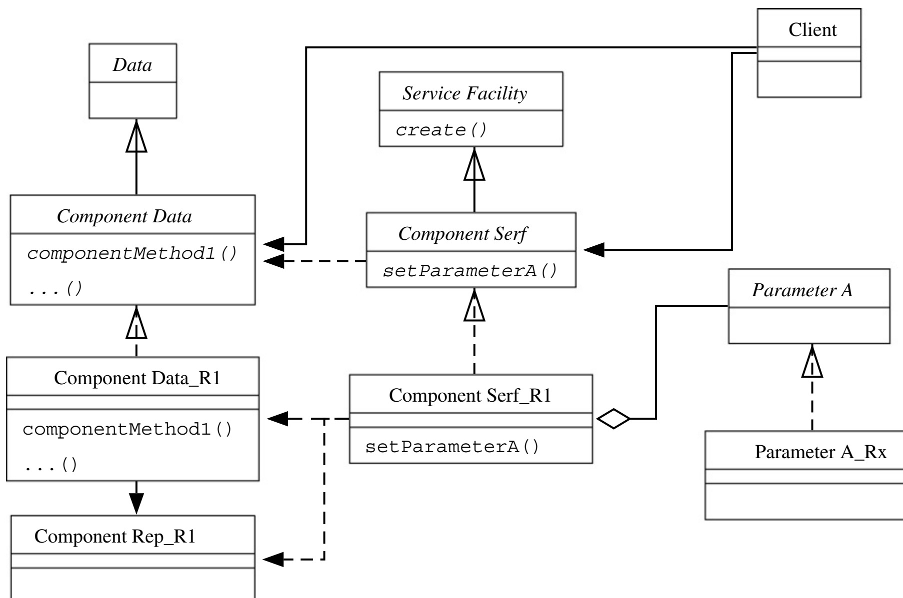

Fig. 6. UML design showing Serflets in the service facility pattern.

In every operation in the Serflet, the actual data rep instance is dereferenced, and the operation is performed. This is how the extra level of indirection is maintained.

The Serf still contains the template parameters to the component, and performs any administrative tasks, such as dynamic reconfiguration. The Serf also holds (weak) references to all the Serflet objects it creates. So although the Serflet objects (client proxies) cannot be modified without the client knowing about such change, the representation objects can be modified without the client having to be notified.

Listing 5: The ResourceSerf as a C# Serf

```
1 interface ResourceSerf : ServiceFacility {
2   public ProtocolSerf protSerf
3   {get; set;}
4
5   Resource joinNetwork( );
6   void leaveNetwork(Resource r);
7   bool isRequested(Resource r);
8   bool isAvailable(Resource r);
9   void request(Resource r);
10   void release(Resource r);
11 }
```

6. Bringing it all together: resource allocation example

Let us go back to the mutual exclusion example that we looked at in Section 1. We present here the design of a mutex component that supports dynamic change of conflict resolution protocol.

Listing 5 presents the ResourceSerf interface for a resource manager object. This is a parameterized component that can be specialized by the conflict resolution protocol (ProtocolSerf). These parameters are *restricted* parameters – the parameters must implement their specified interface specifications. The component exports one type (Resource). When a client process wants access to the resource that is managed by this component, it invokes the joinNetwork method, which returns an object of type Resource. From that point on, whenever it wants to use the resource, the client requests the resource by calling request on its Resource instance. In effect, the client treats this Resource handle as the actual shared resource itself.

In addition, there is a property to represent the component parameter (ProtocolSerf). An instance of Resource is modeled by two boolean variables – requested and available. When a client requests its resource r, r.requested is set to **true**. Then when the resource is available for use by the client, r.available becomes **true**. Only one of the resource proxies in the system can have both requested and available set to **true** at the same time.

Let us assume a polling implementation of the request( ) method in ResourceSerf. Thus, a client will make a request( ) invocation on its proxy, and then poll the isAvailable( ) method until the resource becomes available. At this point, the client uses the resource, and releases it by invoking release( ) when finished.

Fig. 7 shows the UML description of a client using ResourceSerf. The system uses the service facility pattern. ResourceSerf is the main component in this system. This component is parameterized by a ProtocolSerf, and manages a Resource that the client wants to use. The actual conflict resolution is performed by the ProtocolSerf. For every client, the Serf creates a representative in the ProtocolSerf. When a client makes a request on its logical handle, the ResourceSerf makes a request on behalf of this client on the ProtocolSerf. Listing 6 shows a parts of ResourceSerf – the constructor for ResourceRep_R1, and the create( ) and request( ) methods – to illustrate this interaction between ResourceSerf and ProtocolSerf.

The reader may have observed from this example that a change in the ProtocolSerf implementation is going to immediately affect every single client proxy that the ResourceSerf has created. However, the client processes are not involved in this change. The levels of decoupling that the service facility pattern introduces make way for the change in ProtocolSerf to be insulated from the clients.[5]

In this example, ResourceSerf plays the role of the *factory* under the abstract factory pattern. It creates objects of type Resource (the *product* from abstract factory) on behalf of the client. The client no longer needs to keep track of which particular concrete implementation of ResourceSerf is in use. Further, if the ResourceSerf is reconfigured to use a different ProtocolSerf, the client does not have to concern itself with such a change.

[5] The client processes may experience lower performance levels during reconfiguration, but the client does not have to *do* anything for the reconfiguration to occur, and performance returns to optimal levels once reconfiguration is complete.

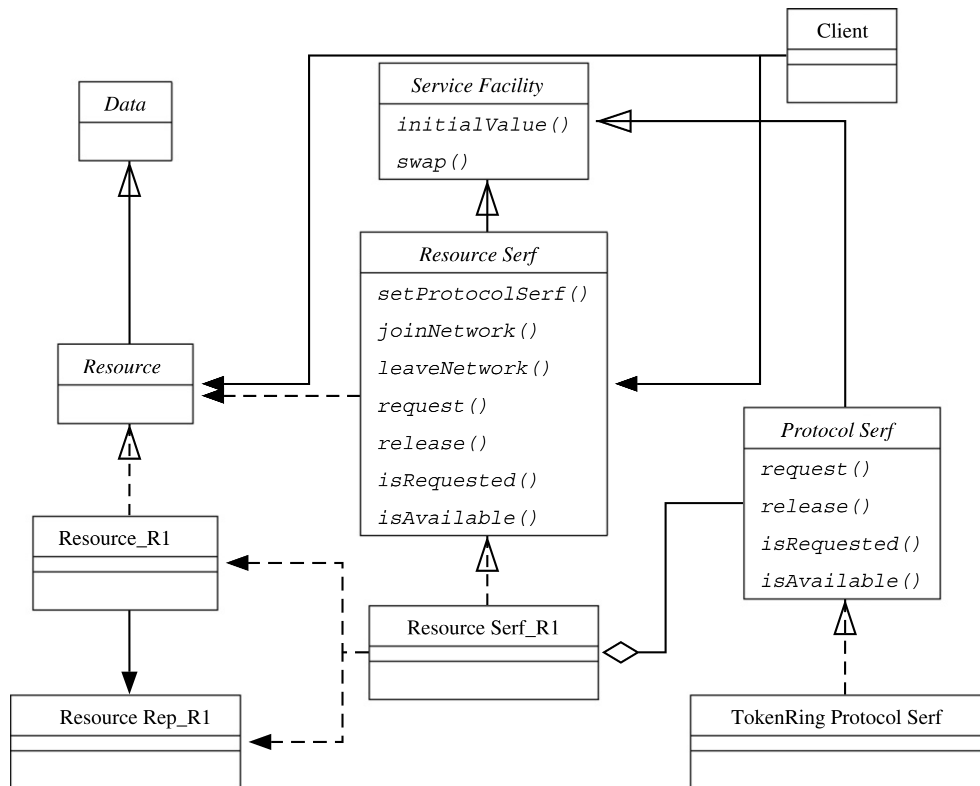

Fig. 7. UML design for the resource allocation example using service facilities.

Listing 6: create( ) and request( ) methods from ResourceSerf

```
1 internal class ResourceRep_R1 {
2   bool requested;
3   bool available;
4   ProtocolProxy protocolProxy;
5
6   public ResourceRep_R1( ) {
7     protocolProxy = (ProtocolProxy) protSerf.create( );
8     requested = false;
9     available = false;
10   }
11 }
12
13 public Data create( ) {
14   Resource_R1 newResource = new Resource_R1( );
15   WeakReference wrRes = new WeakReference(newResource, false);
16   clientList.Add(wrRes);
17   return newResource;
18 }
19
20 public void request(ref Resource r) {
21   ResourceRep_R1 rRep = Rep(r);
22   rRep.requested = true;
23   ProtocolProxy pp = rRep.protocolProxy;
24   protSerf.request(ref pp);
25 }
```

ResourceSerf also plays the role of the *context* in the strategy pattern, and uses an instance of ProtocolSerf as a *strategy* to resolve conflicts and provide mutually exclusive access to clients vying for the resource. The ResourceSerf is no longer coupled to a single conflict resolution algorithm.

Finally, ResourceSerf also acts as a *proxy* (proxy pattern) to the Resource (*subject*) that the client gets back from the create( ) method. Since a change in ProtocolSerf will affect all client proxies that have already been created by ResourceSerf, it is essential that the client not have direct references to the representation objects. In the case of the service facility model, however, this is not a problem. The client only holds a reference to a logical handle, and not to the actual representation object.

7. Related work

Parameterized programming has been recognized as a powerful technique for building reusable software. Goguen (1984) captures the essential basis behind the technique, and details the list of features that a programming system should support in order to allow for parameterized programming. In this work, Goguen describes parameterized programming using OBJ (Goguen and Malcolm, 1996; Goguen et al., 1993; Tardo, 1981). The primary requirements that Goguen lists include modularity, hierarchical structure, restricted parameters, information hiding, module modification, and an underlying formal semantics.

Our approach to building parameterized components using the service facility pattern does include support for all these requirements. Under the service facility pattern, design decisions are encapsulated in their own modules, and all interactions between these modules occur through the interface method invocations. Thus the modularity and information hiding requirements are respected. Parameters to a service facility can themselves be parameterized components, and it is possible to build a hierarchical system of parameterized service facilities. Finally, the service facility pattern is based on the RESOLVE (Sitaraman and Weide, 1994) language for specifying parameterized components, which includes a formal semantics for these components as well as support for restricted parameters.

Several programming languages have included support for parameterized programming over the years. We briefly outline the most popular among these. All of these languages, unfortunately, only support static parameterization.

The Ada programming language (Barnes, 1989) has support for generic units. A generic component in Ada is parameterized at the level of types it uses. To be used in a program, the generic component is then instantiated by supplying actual parameters to take the place of the formal parameters in the generic component definition (Feldman, 1997; Booch, 1987; Musser and Stepanov, 1989).

Templates in C++ (Stroustrup, 1991) are similar to Ada generics; the language allows definition of template classes and template functions. Template classes allow the definition of generic types, as in Ada. Similarly, template functions can be used to define methods that can be specialized by their parameter types. The general syntax for defining templates in C++ is to mark the class or function using the keyword template, and the list of parameters the template needs to be instantiated with. Moreover, templates can be used in conjunction with inheritance to provide an alternative to using pure inheritance as a way of extending functionality.

The Standard Template Library (STL) for C++ (Musser and Saini, 1996) provides a variety of data collection template classes that can be used in C++ programs. C++ does not provide a way to specify type restrictions on template parameters. However, at the time of instantiation, if the parameter does not provide all the methods that the template expects the parameter to provide, the instantiation will produce a compile-time error.

The next version of the C# language (Archer, 2001; Sharp and Jagger, 2002), scheduled for release of the next version of the .NET framework (Platt and Ballinger, 2001; Richter, 2002) will include support for generics (Kennedy and Syme, 2001; Hejlsberg, 2003; Microsoft, 2003). As with Ada and C++, generics in C# can be used to parameterize classes and methods. Further, they can also be used to parameterize the new constructs that C# introduces – interfaces and delegates. C#, like Ada, does provide constructs to specify constraints on template parameters. At the time of instantiation, the compiler checks to see if the parameter supplied to the generic actually satisfies the constraints and if not, throws a compile-time error. The .NET framework also comes packaged with a variety of generic data collection classes, similar to the C++ STL.

8. Conclusion

Robust, scalable software design involves limiting concrete dependencies among various parts of a software system to the bare minimum. In order for software to be flexible, it is important for dependencies among components to be strictly limited to the abstract (or interface) level as far as possible. At some point, however, these interfaces need to be replaced by actually executing code. Parameterization provides a nice way to postpone such commitment to concrete components.

In contrast to static parameterization (the kind that is afforded by language support), dynamic parameterization allows software designers to pick concrete bindings until after the software system has been deployed. This is good because decisions about parameters are better informed by the state of external considerations during execution than at compile time.

Design patterns are codified solutions to commonly recurring problems in software systems. They offer an elegant solution in that they are not tied to a particular programming language. Instead, they provide a set of assumptions that they make about the target programming language, and any language that satisfies these assumptions can be used to implement the solution.

In this paper, we presented the service facility pattern, which provides a design pattern-based methodology for designing and constructing software components that support dynamic parameterization. The paper also outlined some implementation considerations along with a case study example.

The component-oriented software design approach supports the construction of scalable and flexible software systems. The Serf components as presented in this paper provides an effective way to implement software components in object-oriented languages such as Java and C#. Serf components can be implemented in any OO language that supports introspection.

References

Archer, T., 2001. Inside C#. Microsoft Press.

Barnes, J.G.P., 1989. Programming ADA. Addison-Wesley Longman Publishing Co., Inc.

Booch, G., 1987. Software Components with ADA. Benjamin-Cummings Publishing Co., Inc.

Bracha, G., Odersky, M., Stoutamire, D., Wadler, P., 1998. Making the future safe for the past: adding genericity to the java programming language. In: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications. ACM Press, pp. 183–200.

Bruce, K., Cardelli, L., Castagna, G., Group, T.H.O., Leavens, G.T., Pierce, B., 1995. On binary methods. Theory and Practice of Object-Oriented systems 1 (3), 221–242.

Feldman, M.B., 1997. Software Construction and Data Structures with Ada 95. Addison-Wesley, Reading, MA.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Goguen, J., 1984. Parameterized programming. IEEE TSE SE 10 (5), 528–543.

Goguen, J., Malcolm, G., 1996. Algebraic Semantics of Imperative Programs. MIT Press.

Goguen, J., Winkler, T., Meseguer, J., Futatsugi, K., Jouannaud, J.-P., 1993. Introducing OBJ. In: Goguen, J. (Ed.), Applications of Algebraic Specification using OBJ. Cambridge.

Hejlsberg, A., 2003. Visual C# "Whidbey": language enhancements. Presentation at Microsoft Professional Developers Conference 2003, October, 2003.

Kennedy, A., Syme, D., 2001. Design and implementation of generics for the .NET common language runtime. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. ACM Press, pp. 1–12.

Lamport, L., Lynch, N., 1990. Distributed computing: models and methods. In: van Leeuwen, J. (Ed.), Handbook of Theoretical Computer Science, vol. B. Elsevier Science Publishers, pp. 1157–1199 (Chapter 18).

Lynch, N., 1996. Distributed Algorithms. Morgan Kaufman Publishers, San Francisco, CA.

Microsoft, 2003. Professional Developers Conference. Microsoft Corporation, Los Angeles, CA.

Musser, D., Saini, A., 1996. STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library. Addison-Wesley.

Musser, D.R., Stepanov, A.A., 1989. The Ada Generic Library Linear List Processing Packages. Springer-Verlag, New York, Inc.

Neilsen, M.L., Mizuno, M., 1991. A dag-based algorithm for distributed mutual exclusion. In: Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS). IEEE Computer Society, Washington, DC, pp. 354–360. Available from: <citeseer.nj.nec.com/neilsen91dagbased.html>.

Parnas, D.L., 1972. On the criteria to be used in decomposing systems into modules. Communications of the ACM 15 (12), 1053–1058.

Platt, D.S., Ballinger, K., 2001. Introducing Microsoft .NET, first ed. Microsoft Press.

Richter, J., 2002. Applied Microsoft .NET Programming. Microsoft Press.

Robinson, S., 2002. Advanced .NET Programming. Wrox Press.

Sharp, J., Jagger, J., 2002. Microsoft Visual C# .NET Step by Step. Microsoft Press.

Sitaraman, M., Weide, B., 1994. Component-based software using RESOLVE. Software Engineering Notes 19 (4), 21–22.

Sridhar, N., 2006. Dynamic instantiation-checking components. In: SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing. ACM Press, New York, NY, USA, pp. 1442–1446.

Sridhar, N., Weide, B.W., 2003. Reasoning about parameterized components with dynamic binding. In: Proceedings of the Workshop on Specification and Verification of Component-Based Systems at ESEC/FSE 2003, Helsinki, Finland, pp. 92–95.

Sridhar, N., Pike, S.M., Weide, B.W., 2003. Dynamic module replacement in distributed protocols. In: Proceedings of the 23rd International Conference on Distributed Computing Systems.

Stroustrup, B., 1991. The C++ Programming Language, second ed. Addison-Wesley, Reading, MA.

Szyperski, C., 1999. Components and objects together, Software Development 7 (5).

Tardo, J.J., 1981. The Design, Specification, and Implementation of OBJ-T: A Language for Writing and Testing Abstract Algebraic Program Specifications. Ph.D. thesis, Department of Computer Science, University of California, Los Angeles.

Thimbleby, H., 1988. Delaying commitment. IEEE Software 5 (3), 78–86.

van der Linden, P., 1998. Just Java 2. Prentice Hall.

Post-print standardized by MSL Academic Endeavors, the imprint of the Michael Schwartz Library at Cleveland State University, 2014