Service Facilities: Extending Abstract Factories to Decouple Advanced Dependencies

Nigamanth Sridhar, Bruce W. Weide, and Paolo Bucci

Computer and Information Science, The Ohio State University,
2015 Neil Ave., Columbus OH 43210, USA
{nsridhar,weide,bucci}@cis.ohio-state.edu

Abstract. It is widely agreed that component interactions should be based on the import and export of interface information only, not on knowledge of implementation-specific details. This can be achieved in many cases either by explicit parameterization using templates (in languages that have them) or by using some variant of the abstract factory pattern. We introduce an alternative: the use of *service facilities*. This technique is similar both to the use of templates and to the use of factories, but it is preferable to both in several important ways. Service facilities can be used to decouple design-time concrete-to-concrete component dependencies in any reasonable programming language and with any component infrastructure that is based on design-by-contract principles.

1 Introduction

Direct design-time dependencies between *concrete components* are widely recognized as undesirable because they complicate software maintenance activities. Any change in the design of a single concrete component may well entail major changes in the rest of any system that uses it – a phenomenon termed "the hairball effect" by Clemens Szyperski. The reason is that a change might affect the interaction of the component being modified with the other components that were designed to depend on it, and changes there might affect the interactions with the components that were designed to depend on them, and so on [9].

It is, therefore, commonly recommended that design-time component dependencies generally should be limited to those between a concrete component and the *abstract components* that it implements and uses (or otherwise communicates with) [5]. In fact, all popular commercial component technologies are now based on this principle – variously called "design by contract", "programming by contract", "design to interfaces", "decoupling", etc. Modern programming languages such as Java and C# support the idea by giving *interfaces* (abstract components) the same linguistic status as *classes* (concrete components). A simple rule that supports good component design in these languages is that design-time coupling should be from classes to interfaces, not from classes to classes. Common advice for easing maintenance is that new interfaces may be introduced, but existing ones should remain fixed once they are deployed in a setting

C. Gacek (Ed.): ICSR-7, LNCS 2319, pp. 309–326, 2002.

where *reuse* is expected. The internal details of classes that implement those interfaces may change even after deployment, so long as they continue to implement the same interfaces. Just as importantly, however, new implementation classes may be added for existing interfaces and thereby become available as new implementation options for clients of those interfaces.

Eventually, of course – at the latest just before code is executed – someone must select some concrete component to implement each abstract component that is used in building a larger component or final system. This means that it is helpful when discussing component-based systems to distinguish between *component design time* and *component integration (composition) time*. By the former we mean the time at which a component is considered fully designed and is entered into a component library, i.e., where it still exists out of the context of any larger component or final system in which it might be (re)used. (Design time is when concrete-to-concrete dependencies should be avoided.) By the latter, we mean the time at which the component is selected for use from the library and assembled into a larger component or final system. Integration time with respect to a library component might occur at design time, or at compile time, link time, or run time, with respect to the larger component or system in which it is used.

An important question in component-based software engineering concerns how to reconcile the desire to decouple concrete-to-concrete dependencies at component design time with the need for easy assembly of concrete components at integration time. Two basic techniques have been suggested for this. Parameterization of components using a template mechanism, a.k.a. generics, is one decoupling approach [1,8]. As a template, a concrete component can be designed so that it depends on one or more abstract components, implementations of which are technically parameters that can be selected and bound at integration time as opposed to being fixed at design time. In languages with template support, integration time (the binding of concrete components to the abstract components they implement) means compile time because integration is achieved through template instantiation. This is relatively early integration-time binding but still much better from the maintenance standpoint than forcing such implementation commitments to occur at design time.

In modern distributed computing environments, compile time sometimes is not late enough for component integration to occur. Some information about concrete component availability or suitability simply may not be known until run time. Moreover, even where compile-time binding is appropriate, template mechanisms are not available in widely-used languages such as Java and C#, and they are not part of COM IDL, CORBA IDL, WSDL, .NET, etc., which are becoming widely used as the basis for component-based software today. So, explicit parameterization as a decoupling mechanism is effectively limited to compile-time use in C++ [7,8].

Consequently, several design patterns have been proposed to address the decoupling problem for use in languages that do not offer linguistic support for templates. In some cases, e.g., when developing COM components, such patterns must be used even when coding in languages such as C++ that do support

templates. The *abstract factory pattern* [4] is the canonical representative of these. When taken to the logical conclusion suggested by the metaphor on which it is based, the abstract factory pattern can be used rather effectively to decouple concrete-to-concrete dependencies.

But, as we will see, there remain drawbacks to both the above methods. In this paper, we introduce an alternative approach to decoupling concrete-to-concrete dependencies that is logically tantamount to using templates. This facilitates sound and modular reasoning about software behavior and has other advantages over the use of informal design patterns. However, like the abstract factory pattern, it can be employed in languages and that do not support templates but that permit component integration as late as at run time. We call the key elements of this approach **service facilities**. The name is intended to suggest a parallel to factories, because the idea is most easily explained using a service-oriented rather than a manufacturing-oriented metaphor.

Following introductions to abstract factories and service facilities (Sections 2 and 3), we present an example to illustrate how service facilities can be coded in Java (Section 4), discuss some points of comparison between service facilities and abstract factories (Section 5), and finally summarize our contributions and conclude (Section 6).

The problem of decoupling design-time concrete-to-concrete dependencies is inherent in system design (in fact, not just for software systems [9]), and it is common to all practical programming models today. The solutions presented in this paper are intended to be general enough to address the problem in distributed software systems built using component technologies such as COM, CORBA, and .NET. However, for the sake of simplicity of presentation, we use Java to illustrate service facilities. All the concepts map directly to programming constructs in commercial distributed component infrastructures.

2 The Abstract Factory Pattern

Readers who are familiar with the abstract factory pattern may wish to skim this section and proceed to Section 3. Unfortunately, the idea of a service facility is unlikely to make much sense to anyone who is not familiar with both the rationale for and some technical details of the abstract factory pattern, and the comparison between service facilities and abstract factories in Section 5 is certain to be difficult to follow without this background.

The abstract factory pattern is an approach that can dramatically reduce – but not quite eliminate – dependencies between classes (i.e., concrete components). If it is adopted uniformly, then every class has a corresponding **factory** class whose objects (class instances) can manufacture/construct/create objects of the original class, which is called the **product** class. The client program depends *almost* entirely on the interfaces (i.e., abstract components) implemented by the factory class and by the product class.

The binding of a reference to the factory object, which pins down the product's implementation class, technically happens at run time and hence can be

based on information that is not known until run time. However, in most languages the set of possible implementation choices must be known at compile time. For example, Java (like most other object-oriented languages) effectively dooms any approach to decoupling concrete-to-concrete dependencies so that the strongest possible conclusion is that it "almost" works. The reason is that everywhere a constructor is invoked, Java expects to see the name of the class the object is to be an instance of – not the name of an interface. Identifying constructor names with class names is known to introduce other problems as well [6]. But the goal of design patterns is not to suggest how to change the language deficiencies we are stuck with but to record the best ways people have found to work around them [2]. The objective of the abstract factory pattern is, therefore, not to remove this language restriction but to allow us to live with it.

The result of using the abstract factory pattern is that it is possible to *localize* each concrete-to-concrete dependency to a single line of code where the factory implementation class finally *must* be chosen if the client code is to compile. Now all objects of the product class are constructed not by invoking the product class's constructor but by invoking a non-constructor method of the factory object. The lines of code that ask factory objects to construct product objects do not introduce concrete-to-concrete dependencies, and need not change when the factory and product implementation classes are replaced by different ones that implement the same interfaces (functional behavior) with different performance or other non-functional properties.

2.1 Example: A Sequence Component

This section introduces a running example that has been chosen to illustrate what we are talking about, not because it is so complex that it compellingly demonstrates either the rationale for or the advantages of any particular approach to decoupling.

Figure 1 shows the design structure of a system that uses a Sequence product interface along with a SequenceF factory interface for this product. Only one sequence implementation "R1" (for "realization (implementation) number 1"; the name is unimportant) is shown in the figure. The two implementation classes for this implementation are SequenceF_R1 and Sequence_R1. Of course, an important reason for using factories is that it is expected that there are or eventually might be other implementations of sequences with the same two interfaces that could be selected for use in the client program. For example, the Sequence interface might include methods to add, remove, and update sequence entries by position. The "R1" implementation might take best-case constant and worst-case linear time time for each of these methods, and another implementation might always take log time for each of them. Supporting easy substitution of one such implementation for another, based on the client's performance needs, is one major reason for decoupling concrete-to-concrete dependencies.

Without the abstract factory pattern, the client program would have to construct sequences as follows, spreading the name of the class throughout the code:

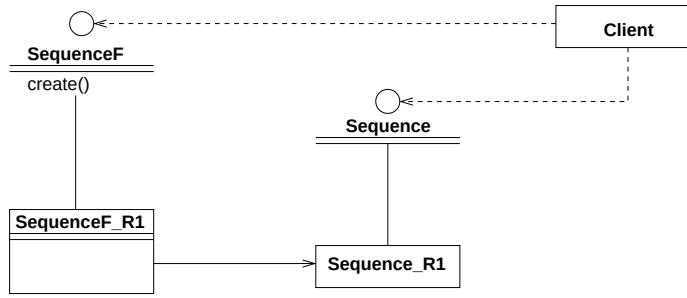


Fig. 1. Design based on the abstract factory pattern

```
Sequence s1 = new Sequence_R1 ();
...
Sequence s2 = new Sequence_R1 ();
...
s1.add (x, 3);
y = s2.remove (i);
...
```

The following code snippet shows how the client takes advantage of the abstract factory pattern:

```
SequenceF seqF = new SequenceF_R1 (); // Select implementation
...
Sequence s1 = seqF.create ();
...
Sequence s2 = seqF.create ();
...
s1.add (x, 3);
y = s2.remove (i);
...
```

This client can now easily switch to a different implementation for Sequence, the only code change being a change to the factory constructor, which appears in just one place in the code. A change from "_R1" to "_R2", for example, does not lead to a change in the client code except in this one inherently (in Java) unavoidable place. Note that the client declares both factory and product objects to have interface types.

2.2 Outsourcing

We continue by elaborating the above example a bit. Suppose the Sequence_R1 representation of sequences is built not just by using Java's built-in types, but by using other components, each of which might also have several alternative implementations. How can we design Sequence_R1 so it is possible to delay until

integration time the binding of implementation classes for these components used in the sequence representation?

To be specific, suppose the representation of a sequence in Sequence_R1 consists of two stacks (call them beforeStack and afterStack) positioned so that the tops of the two stacks "face each other" in the interior of the sequence they represent. That is, the top entries of the two stacks hold consecutive elements in the sequence. All additions to and removals from a sequence can be performed by shifting entries between the two stacks until the break between them is at the appropriate position in the sequence, followed by a push or a pop on afterStack. This code depends only on the Stack interface, not on any particular implementation of it, as illustrated in Figure 2.

```
void add (Object x, int pos)
{
   this.setLengthOfBeforeStack (pos); // Shift entries between stacks
   this.afterStack.push (x); // Push x onto afterStack
}
```

Fig. 2. Sequence_R1.add (Object x, int pos)

How do factories come into play here if they are not needed in the method bodies? In a real-life factory, a factory produces a product. The typical modern factory does not manufacture its product from scratch, however. Various parts of the product are built by other factories through *outsource manufacturing*, and various parts of their products are built by still others, and so on. An automobile factory assembles engines, doors, headlights, etc., but most of these pieces are not built in the automobile factory itself but outsourced from suppliers.

Returning from physical factories to the abstract factories used in software component design, the obvious software parallel of outsourcing is that factories (or perhaps products; the metaphor is not terribly revealing here) should know about the factories that create their constituent parts. Product creation is the only place where these factories are apparent; other methods such as add above typically do not involve factories of any kind. So a reasonable approach is to have a SequenceF_R1 factory object hold a reference to a second factory object that creates stacks, which the former can use when asked to create a new Sequence_R1 object.

Outsourcing entails adding either a new method or a constructor with parameters to the SequenceF_R1 class. This permits a client program, when components are assembled, to create a factory for stacks and then give the sequence factory a reference to it. The simple integration-time code from before:

```
SequenceF seqF = new SequenceF_R1 (); // Select implementation
```

is now slightly more complex:

```
StackF stkF = new StackF_R4 ();          // Select stack implementation
SequenceF seqF = new SequenceF_R1 (); // Select sequence implementation
seqF.setStackF (stkF);                   // Set stack outsource factory
```

The advantage of the more complex code is that we have decoupled the chosen implementation of sequences from the implementation of the stacks used to represent them. Binding implementation classes together is now done at integration time, not at design time, of Sequence_R1.

In the remainder of this paper, we do not address the question of whether or how a client programmer should have any knowledge of the representation of Sequence objects, let alone enough to determine which Stack implementation should produce the best performance profile for a particular client situation. Suffice to say that there is a straightforward way to develop new implementations of the Sequence interface from a fully decoupled one, so that some or all implementation class selection decisions can be made by the component implementers or by "middlemen" with access to enough performance-related information to select reasonable subcomponent implementations for ultimate use by clients [8]. Such *partially instantiated* concrete components are no longer decoupled from other concrete components; the trade-off is between ease of implementation substitution and ease of integration for the client. Here we have simply opened up this aspect of the design to the client in order to illustrate the idea of outsourcing as directly as possible.

3 Service Facilities

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?

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.

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 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.

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 (or, in terms of the usual OOP metaphor, to ask the product to change itself). 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. That is, neither the data object nor the client can manipulate a data object's value unless the client explicit requests participation by the bank. We call the software analogue of the bank a **service facility object** because it must be asked to help perform all services on, i.e., manipulations of, the data objects for which it is responsible.

Here is a summary of the software design differences between service facilities and factories:

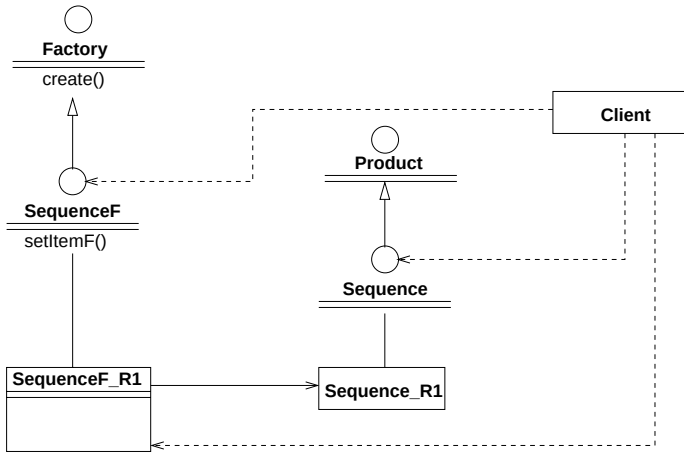
Service Facilities – There are two kinds of objects at run time: service facility objects and data objects. Each service facility object is responsible for creating certain data objects and then "keeping track of" and "protecting" all those data objects. All methods for manipulating the data objects are supplied by the service facility objects that created them. Data objects have no methods of their own.

Factories – There are two kinds of objects at run time: factory objects and product objects. Each factory object is responsible only for creating certain product objects. The client is responsible for "keeping track of" and "protecting" all the product objects that the factory objects have created for it. All methods for manipulating the product objects after their creation are supplied by the product objects themselves. Factory objects have no methods except for product creation.

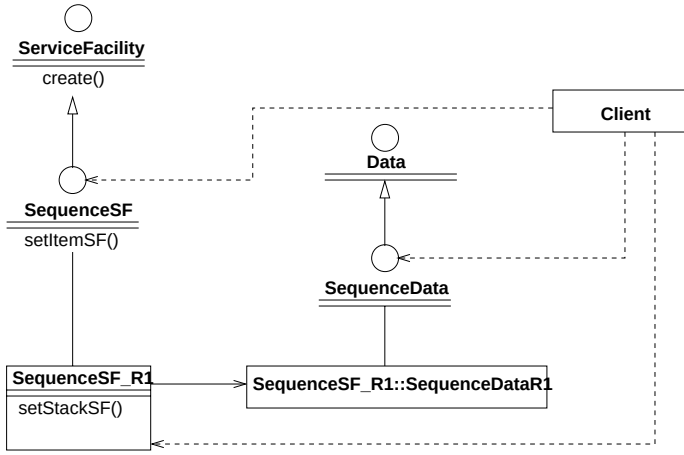
3.1 A Simple Example: A Sequence Component

Figure 3(b) shows the design structure of a system that uses a SequenceSF service facility interface and a SequenceData data interface. Again, only one implementation "R1" is shown. It has two implementation classes, SequenceSF_R1 and SequenceSF_R1::SequenceData_R1. The latter is, as we code service facilities in Java, an inner class that defines the data representation used by the sequence manipulation methods that are implemented in SequenceSF_R1. Other techniques that achieve the same visibility conditions might be more appropriate in other languages. The important condition is that the data representation in SequenceSF_R1.SequenceData_R1 must be hidden from clients but visible within SequenceSF_R1.

Figure 3(a) shows the abstract factory pattern for the same situation, slightly elaborated from Figure 1 to include explicit interfaces Factory and Product. These counterparts of ServiceFacility and Data in the service facility design help make clear the similarities and differences between the two designs. They also are



(a) Elaborated abstract factory design for sequences



(b) Service facility design for sequences

Fig. 3. Comparison of abstract factories and service facilities

important in allowing the abstract factory solution to deal with the fact that the sequences we have in mind are intended to be homogeneous, i.e., to contain the same kinds of entries in all positions. We discuss this point further in the next subsection.

The following code snippet shows how the client takes advantage of service facilities:

```
SequenceSF seqSF = new SequenceSF_R1 (); // Select implementation
...
Sequence s1 = seqSF.create ();
...
Sequence s2 = seqSF.create ();
...
seqSF.add (s1, x, 3);
y = seqSF.remove (s2, i);
...
```

As a reminder, here again is the same code with a design based on the abstract factory pattern:

```
SequenceF seqF = new SequenceF_R1 (); // Select implementation
...
Sequence s1 = seqF.create ();
...
Sequence s2 = seqF.create ();
...
s1.add (x, 3);
y = s2.remove (i);
...
```

There is little difference in the code the client writes with these two approaches. With service facilities, the receiver object in the add and remove calls is a service facility object, so the corresponding data object is now an explicit parameter. With factories, the receiver object is a product object and the factory object is not involved. It should be noted that the same similarity is observed if outsourcing is used.

Which raises the question: What is the counterpart to outsourcing in the bank metaphor? A client with a safe deposit box may decide to store in that box the key to another safe deposit box at another bank. In other words, with service facilities, it might make metaphorical sense for the data objects to hold references to the service facilities that are responsible for their data members. This is not necessary, though, because the only code that can manipulate these data members is in the service facility that is responsible for those data objects. It is therefore more economical for service facility objects to hold references to other service facility objects (as well as to their own data objects), and for data objects to hold *only* references to other data objects. This is how we have coded service facilities in the Java example detailed in the next section.

This client can now switch to a different implementation for Sequence, the only code change being a change to the service facility constructor, which appears in just one line. A change from "_R1" to "_R2", for example, does not lead to a change in the client code except in this one unavoidable place. Note that the client declares both service facility and data objects to have interface types.

3.2 Homogeneous Containers

Since Java has no template mechanism, container or collection types are normally handled by taking advantage of the fact that Object is at the root of the class hierarchy. If we simply make sequences of Objects then everything seems fine – except for some nasty issues that are typically swept under the rug. The source of the problem is manifest *underdefinition* of the entry type. The compile-time type of entries is so "weak" that the compiler support a client programmer normally expects in finding type errors is effectively lost. One consequence is that it now becomes possible for a client program to have a container object whose entries are a heterogeneous collection of objects of different types. This complicates reasoning about client program behavior, and it usually is not what the component designer or the client had in mind in any case, but it is not caught at compile time. Another problem is that upon removal from a container, the type of an entry can be unknown to the client program and must be cast to the type the client programmer thinks (hopes) it was when it was inserted. Since inserting into a container and removal from it might take place in two distant locations in a large client program, this can be a significant problem. (We has an unpleasant first-hand experience with this situation while using a java.util.Map implementation.) Java's reflection feature can help here in principle, but the client code is an ugly mess.

Moreover, because of underdefinition of the entry type, the implementation classes for Sequence cannot do anything to entries that cannot be done to Objects. A clear symptom of this problem involves *cloning*, i.e., making a "deep copy" of a container object. The popular java.util package, for example, defines many container classes. Each has a clone method which, it might be hoped, would make a deep copy of this. It is a surprise to most potential clients of this package (at least to those who read the documentation or encounter mysterious bugs) that clone does *not* make a deep copy for these objects; it makes what the documentation calls a "shallow copy". The reason? If the declared entry type for the method that inserts an entry into a Sequence object is Object, then the compiler does not know that the actual entry type has a clone method of its own. It might seem that this problem could be avoided merely by enforcing the condition that the entries inserted must implement Java's Cloneable interface. But this does not work because Cloneable is an empty "marker" interface that merely suggests by wishful naming that any implementing class should have a clone method. When the compiler is presented with code like y = x.clone() in the clone method for a Sequence implementation class, it is unhappy because x is not known to have a clone method. There are many other possible workarounds, none of which is at all satisfactory.

The use of abstract factories reveals another symptom of the underdefinition problem. Specifically, if an implementation class for Sequence ever needs to construct a new object of the entry type by invoking its factory's create method, then it cannot do so (for essentially the same reason it cannot invoke the entry's clone method). To address this problem as well as some of the others noted above, SequenceF can demand that each implementation class have a method,

say setItemF, that can be used to give each SequenceF object a reference to a factory object that can create its entry items. The new interface Factory in Figure 3(a) is now required as the type of the formal parameter to setItemF; but including this interface is a good idea in any case in order to document the details of the abstract factory pattern.

With service facilities, there is a corresponding method setItemSF in the interface SequenceSF. But the service facility for sequence items not only creates new entry objects, it is used to manipulate them. So it is possible to require that the argument to setItemSF be an implementation of some extension of ServiceFacility, e.g., one that includes a method to make a replica (clone) of its Data objects. For brevity and in order to concentrate on other points of comparison with abstract factories, we do not pursue the details of this suggestion here.

4 Example: Implementing Service Facilities in Java

In this section, we present the most important aspects of implementing the sequence example presented in Section 3.1.

4.1 The ServiceFacility and Data interfaces

The ServiceFacility interface is the root of all service facilities. This interface has the signature of one method, create(), that a client of a service facility calls to request a new object to be created. Figure 4 shows this interface.

4.2 The Data Interface

The return type of the create() method, Data, is also an interface. This interface serves as the root of all data objects that are used by a client in this model. Data is an empty interface, and serves only the purpose of enforcing some type safety. This interface is shown in Figure 5.

4.3 The SequenceSF Interface

The abstract component SequenceSF defines a generic homogeneous sequence component that could be specialized to contain items of any particular type. The SequenceSF interface therefore provides methods that a client would use to

```
package SF;

public interface ServiceFacility
{
    public Data create();
}
```

Fig. 4. The ServiceFacility interface

```
package SF;

public interface Data
{

}
```

Fig. 5. The Data interface

specialize the service facility to contain items of a certain type (in this case, just one: setItemSF). It also contains the signatures of the methods to manipulate sequences (say, for simplicity, just add, remove and length). The interface is presented in Figure 6. The SequenceData interface (not shown) extends Data but is otherwise empty.

```
package SF.Sequence;
import SF.*;

public interface SequenceSF extends ServiceFacility
{
    public void setItemSF (ServiceFacility sf);

    public void add (SequenceData s, int pos, Data x);
    public Data remove (SequenceData s, int pos);
    public int length (SequenceData s);
}
```

Fig. 6. The SequenceSF interface

4.4 SequenceSF_R1: An Implementation of SequenceSF

Figure 7 shows some parts of the SequenceSF_R1 class – an implementation of the Sequence. SequenceSF_R1 uses the two-stack representation as described in Section 2.2. For want of space, only one of the operations (add) is implemented completely here. There are similarly trivial bodies for the other operations.

The setStackSF method is used by the client to specify which implementation of the stack component is to be used in the sequence representation. This method is specific to the "_R1" implementation of sequence, which is why it is in the implementation class only, not in the SequenceSF interface. The private setLengthOfBefore operation simplifies the public method bodies. It is responsible for positioning the two stacks such that the item at position pos of the sequence is at the top of afterStack.

The structure of the SequenceSF_R1 class is the general structure of any service facility implementation in Java, as we code them. Note that the representation of the sequence, SequenceData_R1, is implemented as an inner class. The reason for this is that the service facility class needs access to the data members of that class. In C++, the same idea can be implemented using friend classes – SequenceSF_R1 must be a *friend* of SequenceData_R1.

5 Comparison to Abstract Factories

The design structures presented in Figure 3 suggest that abstract factories are very similar to service facilities. However, as we have observed earlier, there are some subtle differences between the two approaches that lead to some interesting consequences, which are described below.

```
public class SequenceSF_R1 implements SequenceSF
{
    private class SequenceData_R1 implements SequenceData
    {
        StackData beforeStack, afterStack;
        public SequenceData_R1 ()
        {
            beforeStack = (StackData) stkSF.create ();
            afterStack = (StackData) stkSF.create ();
        }
    }

    private ServiceFacility itemSF;
    private StackSF stkSF;

    public void setItemSF (ServiceFacility sf)
    { this.itemSF = sf; }
    public void setStackSF (StackSF sf)
    { this.stkSF = sf; }

    private void setLengthOfBefore (StackData before, StackData after, int len)
    {
        while (this.stkSF.length (before) < len)
        { this.stkSF.push (beforeStack, this.stkSF.pop (afterStack)); }
    while (this.stkSF.length (beforeStack) > len)
        { this.stkSF.push (afterStack, this.stkSF.pop (beforeStack)); }
    }

    public Data create ()
    { return new SequenceData_R1 (); }

    public void add (SequenceData s, int pos, Data x)
    {
        this.setLengthOfBefore (s.beforeStack, s.afterStack, pos);
        this.stkSF.push (s.beforeStack, x);
    }

    public Data remove (SequenceData s, int pos) { ... }
    public int length (SequenceData s) { ... }
}
```

Fig. 7. The SequenceSF_R1 class

5.1 Binary Operations

Consider a Point class shown in the code fragment in Figure 8. There is something different about the isEqualTo method, which tests the equality of two Point objects. Logically, equality checking is a binary operator, but as it is written in

```
class Point
{
    private int xVal;
    private int yVal;

    public int x () { ... }
    public int y () { ... }
    public bool isEqualTo (Point p)
    { return ((this.xVal == p.x ()) && (this.yVal == p.y ())); }
}
```

Fig. 8. The Point class

this class, the method takes only one parameter. The other parameter is the object on which the method is invoked; i.e., binary methods are asymmetric in standard object-oriented style. This causes several known problems with binary methods [3].

In the abstract factory approach, binary methods remain a problem, since the factory object only performs the task of creating a product object and all other operations are still associated with those product objects. With service facilities, however, the situation is different. The service facility class contains the methods used to operate on data objects, and each method takes as parameters all the data objects it has to operate upon. So, for the Point example, with a PointSF class that provides all the methods, equality checking no longer looks special with respect to the other methods. All methods in the PointSF class (shown in Figure 9) take at least one Point object as parameter. The areEqual method takes two parameters of the same type, and therefore actually looks like a binary operator. Gone with the funky asymmetric syntax are the usual problems associated with OO binary methods.

```
class PointSF_R1
{
    :
    :
    public int x (Point p) { ... }
    public int y (Point p) { ... }
    public bool areEqual (Point p, Point q)
    { return ((x (p) == x (q)) && (y (p) == y (q))); }
}
```

Fig. 9. The PointSF class

5.2 Layered Operations

Suppose that we want to implement an extension to the sequence component, to permit sorting of sequences. If designed with the abstract factory pattern, this extension is a derived class that inherits from the Sequence_R1 class. Suddenly, we have lost some of the decoupling we have gained from using abstract factories. This is because this extension only works for this particular sequence implementation. Now if the client wants to switch implementations for the underlying sequence component, the extension is no longer valid.

Another approach, one that respects decoupling, is to extend the interface (not the class) and to pass the sequence object to be sorted as an explicit parameter to the Sort operation. Such a class does not have any dependencies on other classes. Now, even if the client changes the implementation of sequences, no new implementation for the sort operation is required. This is a *layered* extension.

Layered extensions are natural to service facilities. If a sortable sequence is needed, we implement a new service facility class that holds a reference to some sequence service facility. This new class, SequenceSortSF_Ext, has the Sort method, which takes as a parameter the sequence data object to be sorted, and uses the underlying sequence service facility's methods to rearrange that sequence into sorted order.

5.3 Dynamic Relocation of Data Objects

As the safe deposit metaphor suggests, the service facility has full control of *where* a particular data object is physically located. In certain situations such as load balancing, fault recovery, etc., a service facility might decide to move the data objects it "controls" and "protects" to a different physical location. This kind of movement can be completely transparent to the client if the contents of the data objects are not altered in any way.

Such a behind-the-scenes optimization is difficult with the abstract factory pattern, because after a product object has been created, the factory no longer has any control over it. The client deals directly with the product object, and is the only party that "knows" about it, including where it is physically located. There are other ways to deal with this, but the abstract factory pattern itself does not help.

5.4 Dynamic Substitution of Implementations

It is common now for many systems – especially those for highly-available or reactive applications – to be non-terminating. They are required to run continuously, so upgrades and corrective maintenance must be done "on the fly". By way of analogy, consider one of the most important highly-available, reactive systems: humans. Replacing biological components, such as a kidney transplant, must be accomplished without killing the host! So too it should be with software: we should be able to "hot swap" components without killing the system.

We briefly allude to two kinds of scenarios where such implementation substitutions apply. One is the need to change only the algorithm used to compute something, while retaining the same data representation. This kind of substitution is extremely easy with service facilities. All we need to do is rebind the current service facility to a different service facility that provides the new algorithm. Nothing else in the client code changes if both the old and the new service facilities implement the same interface and rely on the same data representations. An example is changing the algorithm that should be used in the implementation of Sort described in Section 5.2.

On the other hand, it is often the case that different algorithms for implementing existing interfaces require different or novel data representations. For example, hashing and binary search tree algorithms for implementing a map component operate on dramatically different data structures. In this scenario, the representation of existing data needs to be converted to suit the new implementation. This is not free; but it can be done by a "representation converter". These kinds of substitutions are not supported by abstract factories, but they are readily handled with no additional complication (beyond what is obviously required in any solution) by service facilities.

6 Conclusions

The most common specific use of the abstract factory pattern is to decouple object declarations from object constructors, so that an object may be declared independently of any design-time commitment to its implementation. In general, abstract factories can be used to localize other concrete-to-concrete dependencies native to object-oriented systems. Despite their popular success in this area, though, abstract factories are insufficient for separating advanced component dependencies that arise in layered, hierarchically composed, and/or dynamically reconfigurable software.

We therefore propose the use of service facilities as a new technique that is powerful enough to address such higher-order considerations that arise in reusable component-based software. Service facilities leverage practitioner familiarity with abstract factories to address these concerns; and they solve a superset of the problems addressed by abstract factories. By uniformly replacing concrete-to-concrete dependencies with concrete-to-abstract dependencies, service facilities permit clients to assemble components independently of any design-time commitments to the implementations of their subcomponents. The commitments are deferred until integration time. Service facilities also allow familiar composition mechanisms such as parameterization to inductively propagate the benefits of abstract factories (and more) through all levels of a hierarchically composed system. Thus the decoupling problem can be solved once for each component, and the solution can be reused at all levels of a hierarchy.

We note that service facilities are offered as a design approach rather than a language mechanism. A language-neutral technique makes the benefits of using service facilities available in any OO-style implementation language or component infrastructure.

Finally, we readily admit that we do not yet have significant experience using service facilities in the design of large software systems. We offer the idea here in order to stimulate much-needed discussion of various techniques that can help decouple dependencies in such systems. Some of our current work involves building a programming environment for component-based software, including compilation of RESOLVE code [8] into Java (as the target language). This compiler generates Java code that uses service facilities as described here. It is an open question whether "real Java programmers" or "real .NET programmers" can be persuaded to embrace service facilities without significant promotion by their industrial sponsors. Our hope is that the clear similarities to abstract factories that we have emphasized will make the learning curve small for those who dare try them, encouraging others to experiment with service facilities and to assess and report on their practical effectiveness.

Acknowlegments

We gratefully acknowledge financial support from the National Science Foundation under grant CCR-0081596, and from Lucent Technologies. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the views of the National Science Foundation or Lucent.

References

1. D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The GenVoca model of software-system generators. *IEEE Software*, 11(5):89–94, September 1994.
2. G. Baumgartner, K. Lufer, and V. Russo. On the interaction of objectoriented design patterns and programming languages. Technical Report CSD-TR-96-020, Department of Computer Science, Purdue University, 1996.
3. K. Bruce, L. Cardelli, G. Castagna, T. H. O. Group, G. T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object-Oriented systems*, 1(3):221–242, 1995.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison Wesley, 1995.
5. B. Meyer. *Design by contract*, chapter 1. Prentice Hall, 1992.
6. B. Meyer. Overloading vs. object technology. *Journal of Object Oriented Programming*, October 2001.
7. D. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison-Wesley, 1996.
8. M. Sitaraman and B. W. Weide. Special feature: Component-based software using RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):21–67, 1994.
9. B. W. Weide. Component-based systems. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering.* John Wiley and Sons, 2002.