DYNAMICALLY RECONFIGURABLE PARAMETERIZED COMPONENTS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Nigamanth Sridhar, M.Sc.(Tech.), M.S.

* * * * *

The Ohio State University

2004

Dissertation Committee:

Paolo A.G. Sivilotti, Co-Adviser

Bruce W. Weide, Co-Adviser

Neelam Soundarajan

Approved by

Co-Adviser

Co-Adviser

Department of Computer and Information Science

ABSTRACT

With the size and complexity of software systems growing at a very fast pace, one of the concerns that we have to address is that of tractable reasoning. We need to make sure that the software we build does not exceed the limits of our understanding. Since we only know how to reason about software of limited sizes, we need to find a way of always keeping the logical size of the programs we write small. Software should thus be built from small, well-understood components put together in predictable ways.

Parameterization is a technique that can greatly help in building scalable, flexible, and robust software systems. An important consideration with parameterized components is the time of binding parameters to the component; whether a commitment to parameters can be changed or not depends on the binding time. In order to achieve maximum flexibility, parameters should be bound as late as possible. Postponing parameter binding to execution time allows for the selection of parameters to be most effective, because it is at run time that details of a system are most complete. Further, dynamically bound parameterized components also present the possibility for dynamic reconfiguration.

In this dissertation, we present a methodology for building dynamically reconfigurable parameterized components. The methodology is presented as a design pattern, with minimal assumptions on the target programming language or environment. The

Service Facility design pattern offers a model of parameterized components that supports dynamic binding of parameters. Further, the model supports a mode of dynamic reconfiguration called dynamic module replacement, which involves re-binding of some or all of the parameters to a parameterized component during run time. The model also includes safety conditions to ensure that the dynamic parameter bindings (and re-bindings) are correct with respect to type-safety.

The techniques and ideas presented in this dissertation are in the context of well-known and widely-used technologies (such as Java, .NET, and XML) so as to enable these ideas to be inducted into practice quickly. The solutions presented here are generic and incrementally deployable.

To my parents...

ACKNOWLEDGMENTS

This dissertation has my name on it, but I would never have been able to get all of this research done, and written this thesis if not for a long list of people. I would like to first thank my advisors Paul Sivilotti and Bruce Weide. If it wasn't for Bruce's support even before I came to OSU, I would not be where I am today. They have both provided me with ample encouragement, technical advice, freedom to do my research, and finally, but importantly financial support through graduate school. Every time I have walked into their offices with a problem (technical or otherwise), I have always come out feeling much better. I also have to thanks Neelam Soundarajan for spending time with me whenever I had questions, and for all the help in finishing this dissertation.

The two most important people who have stayed with me through the years I have spent in graduate school, and have helped me in numerous occasions — Jason Hallstrom and Scott Pike. If not for the "team support" that they gave me, I would have had a much harder time getting through graduate school. Not to mention the numerous hours of research we got done together, while playing with the football and the aerobie. Those long hours I spent in Dreese were so much fun! Some of the earliest research I did was along with Scott, and a lot of the ideas for this thesis became clear in my mind through the many hours of talking to Scott. And I have

to thank Jason for giving me rides from home to campus and back everyday through my last quarter when I was writing the dissertation.

Thanks are also due to Hilary for all the support and stimulating research discussions we have had, and the whole Software Engineering lab — Chris Bohn, Scott Kagan, Mike Gibas and Prakash Krishnamurthy. All of them are great people and have helped me along the way many times over the last five years. The Distributed Systems group is another bunch of people that I have to thanks for all the great 888 discussions, and the numerous practice talks (including my thesis defense) they have listened to and given comments on — Murat, Vinod, Sandip, Greg, Vinayak.

This list of acknowledgments will not be complete without the Reusable Software Research Group — Bill Ogden, Tim Long, Paolo Bucci, Wayne Heym, Ben Tyler, Joan Krone. They gave me all the feedback and more on my research. They hardened me for all the conference talks I gave.

Last but most important, I would like to thank my parents for having put me through school, and having always been there to support me. I can never thank them enough for all the sacrifices they have had to make so I could get a great education, and achieve all my goals. My sister has been a great source of love, affection, and fun during times when I felt down.

This work has been supported by the National Science Foundation (NSF) under grant CCR-0081596, and by gifts from Lucent Technologies and Microsoft Research.

VITA

October 13, 1976	Born – Madras, India
June 1997	M.Sc.(Tech.) Information Systems, Birla Inst. of Technology & Science, Pilani, India
July 1997 – December 1998	Member of Technical Staff, Lucent Technologies
June 2000	M.S. Computer & Information Science, The Ohio State University
January 1999 – present	Graduate Teaching Associate, The Ohio State University
January 1999 – present	Graduate Research Associate, The Ohio State University

PUBLICATIONS

Research Publications

Paolo A.G. Sivilotti, Scott M. Pike, and Nigamanth Sridhar. A new distributed resource-allocation algorithm with optimal failure locality. In *Proceedings of the 12th IASTED International Conference on Parallel and Distributed Computing and Systems*, volume 2, pages 524–529. IASTED/ACTA Press, November 2000.

Nigamanth Sridhar, Bruce W. Weide, and Paolo Bucci. Service facilities: Extending abstract factories to decouple advanced dependencies. In *Proceedings of the 7th International Conference on Software Reuse*, pages 309–326, April 2002.

Scott M. Pike and Nigamanth Sridhar. Early reply components: Concurrent execution with sequential reasoning. In *Proceedings of the 7th International Conference on Software Reuse*, pages 46–61, April 2002.

Jason O. Hallstrom, Scott M. Pike, and Nigamanth Sridhar. Iterators reconsidered. In *Proceedings of the Fifth Workshop on Component-Based Software Engineering*, Orlando, FL, May 2002.

Nigamanth Sridhar and Paolo A.G. Sivilotti. Lazy snapshots. In S.G. Akl and T.Gonzalez, editors, *Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 96–101, Cambridge, MA, November 2002. IASTED, ACTA Press.

Nigamanth Sridhar and Jason O. Hallstrom. Generating configurable containers for component-based software. In *Proceedings of the Sixth Workshop on Component-Based Software Engineering*, Portland OR, May 2003.

Nigamanth Sridhar, Scott M. Pike, and Bruce W. Weide. Dynamic module replacement in distributed protocols. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 620–627, May 2003.

Nigamanth Sridhar and Bruce W. Weide. Reasoning about parameterized components with dynamic binding. In *Proceedings of the Workshop on Specification and Verification of Component-Based Systems at ESEC/FSE 2003*, pages 92–95, Helsinki, Finland, September 2003.

Jason O. Hallstrom, Nigamanth Sridhar, Anish Arora, Paolo A.G. Sivilotti, and Willam M. Leal. A container-based approach to object-oriented product lines. *Journal of Object Technology*, April 2004. (to appear).

FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in:

Software Methodology	Prof. Bruce W. Weide
Software Systems	Prof. Paolo A.G. Sivilotti
Computer Graphics	Prof. Roger Crawfis

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure **Page**

LIST OF LISTINGS

CHAPTER 1

INTRODUCTION

God's Law: *"Human capacity for understanding is limited."*

— William Buxton, 2001.

1.1 The Problem

One of the most important problems facing software engineering today is the scalability of software. Advancement in computer technology governed by Moore's Law [Moore 2000] has tempted software engineers to scale their systems at the same pace. Along with the size of these software systems, their complexity, and hence the effort required to understand them, is also growing at an alarming pace. However, all software engineers are human, after all, and human understanding is limited, as expressed by William Buxton's "God's Law" [Buxton 2001]. How can we build larger systems, and continue to be able to understand them?

We appear to know how to reason about small systems, but can we apply the same techniques to large systems? Dijkstra pointed out in his Notes on Structured Programming [Dijkstra 1972]:

> *Apparently we are too much trained to disregard differences in scale, to treat them as "gradual differences that are not essential". We tell ourselves that what we can do once, we can also do twice and by induction we fool ourselves into believing that we can do it as many times as needed, but this is just not true! A factor of a thousand is already far beyond our powers of imagination!*

Put otherwise, the techniques that allow reasoning about a program with ten lines of code cannot be applied directly to a program with ten thousand lines of code. The complexity of large-scale software simply outstrips the capacity of our unaided intellectual abilities. This limit on our understanding underscores the importance of **modular components and composition techniques that support reasoning reuse about system behavior and correctness**. Existing software components should be reused to build newer larger systems.

Fortunately, reuse is fashionable these days among software engineers. Nearly every software company now has a software component "library" from which different development groups can pick components and use them in their own systems. These components are usually ones that have been used before in projects, but are general enough to be used in other scenarios as well. So when a project team picks up a component from this library and integrates it into their system, there should be enough that can be said about the component's behavior that it doesn't need to be reanalyzed from scratch. This should be a huge saving in time — the development team does not have to re-implement the functionality provided by this component, or the explanation of what it does.

What happens in reality is, however, quite different. There usually are some parts of the component that have not been generalized enough, or are incomplete, or in the worst case, not well documented. In such cases, the development team, in addition to building their own project, also has to spend time trying to tailor the component

from the library to their particular project. Such change is dangerous because once the component is changed in even the smallest manner, there is a chance that the original specification will not hold. So while a developer may expect a component to behave in a certain way based on its specification, the behavior of the component may turn out to be something completely (or worse, subtly) different! Ultimately this kind of reuse leads them into the tar pit mired by complexities of scale [Brooks 1975].

What we really need is code reuse combined with reasoning reuse. Thirty years of research on information hiding [Parnas 1972, Parnas 1979] dictates that reuse should be maintained at a level that respects the rules of modularity and abstraction. It is this kind of composition that will lead to systems that can still fit into the limited capacity of human understanding, because at any point in the reasoning process, the composition that is being considered is still small enough for our well-known techniques to apply.

Along with the complexities of scale, another big problem we are faced with is the requirement that successful software systems must *evolve* with time. As human needs for the systems change, we should be able to effect changes in our software without having to interrupt the services that the software provides. Software should be flexible enough to accommodate situations that were not directly envisioned at the time the system was designed. The need for *dynamic reconfiguration* in software is real [Kramer and Magee 1985, Hicks 2001].

Our work addresses problems that *inevitably* must be overcome to pave the way for future progress in software engineering theory and practice. In any realistic scenario we can envision for the future, software systems will only get bigger and more complex. It is, therefore, imperative we find ways to make such large systems tractable. We can

understand and reason about large systems only if they are built from components whose behavior can be reasoned about modularly. This requires that our reasoning about components in isolation must be composable to support reasoning about the same components under composition. That is, we must be able to compose our components and reasoning hand in hand. The effort required for reasoning should not climb exponentially, which is where we are headed if we continue along the path of code reuse that is the current industry norm. **Code reuse must be complemented with reasoning reuse**. This is possible only if the units of composition strictly adhere to the properties of abstraction, encapsulation, and information hiding.

1.2 The Proposed Approach

The research community knows how to build software components that respect encapsulation, abstraction and information hiding. Further, strong theoretical bases have been developed to demonstrate how such components can be designed and then composed to construct large software systems. Yet, the current culture of reuse has not been influenced enough for modular component-based methods to inform best practices. This gap between the state of knowledge and state of current practice is cause for concern. We view this as a research problem in itself that must be addressed.

Our approach, therefore, is to make modular, component-based system development more attractive to practitioners. To transfer ideas from the research knowledge base at large, we are motivated to recast the core findings of that research in a form that is adoptable by practitioners. Desiderata for such recasting include the following:

1. The research results must be applicable in the domain of real development environments. The results must not only be research prototypes, but also incorporated into mainstream programming languages and/or development environments.

2. The use of these ideas should not require overarching changes in the methodology of software development, nor should it require global changes to current system infrastructures. As a litmus test, modular components should be incrementally deployable into existing and developing systems.

In keeping with the above desiderata, the approach here is to express a fundamentally sound model of component-based software by using design patterns [Gamma, Helm, Johnson and Vlissides 1995], which are codified solutions to commonly occurring situations in software. Design patterns are language-independent, save some explicitly stated restrictions on language features. The most important characteristic of design patterns with respect to incremental deployment is that they require no changes to the programming language or the platform. (This is not to say that programming language changes would not help [Baumgartner, Läufer and Russo 1996], only that they are not really necessary.)

We use a kind of *parameterized programming* [Goguen 1984] as the basis for our solution. Parameterization has been shown to be a strong technique for building scalable, reliable, and flexible software. Further, the time at which a parameterized component is instantiated (*i.e.,* its parameters are fixed) is a very important consideration. Since, in general, we have more information about a software system and its environment during run-time, our approach to parameterization supports binding parameters during this phase of the software system, too.

Dynamic binding of parameters has many advantages over static binding, since the system can (in principle) then respond to changes in the environment during execution. Dynamically-bound parameterized components support *dynamic module replacement* — a mode of dynamic reconfiguration where the granularity of reconfiguration is at the level of modules. Our methodology includes support for safe replacement of modules.

Dynamic binding also brings with it a set of challenges. Ensuring the type-correctness of parameter bindings is more difficult than in the case of static binding. In the latter case, the binding is done at compile time, and so the compiler has full control over the type-safety. We provide proof obligations, and checking components to monitor type-correctness, while allowing for the flexibility of dynamic binding.

Finally, we believe that the *science* of design of software systems should generally advance in lock-step with the *engineering* of such designs. There seem to be more radical changes in the science now than there are in engineering. The average amount of time an idea takes to be translated into practice, after it has been developed by the research community, is estimated to be eighteen years [Gibbs 1994]. Our approach is therefore also geared to tackle the problem of quick dissemination of research results to practitioners.

1.3 The Thesis

This dissertation defends the following thesis:

1. Parameterized programming provides a strong basis for building reusable software components, and delaying the binding of parameters to a component until run time increases the flexibility of these components.

2. A methodology for dynamic parameterization can be built as a design pattern that places minimal restrictions on the target programming language.

3. Software systems built from dynamically bound parameterized components can adapt to changes in the environment during system execution, and need not require re-starting the system.

1.4 Contributions

This dissertation makes the following key contributions:

1. We provide a methodology for implementing parameterized components that supports dynamic binding of parameters. This model is presented as a design pattern and is applicable in a variety of common programming languages and platforms. We explain how this design pattern, called the Service Facility pattern, can be viewed as a composition of several other well-known design patterns.

2. We show that not only can parameters of parameterized components be bound at run-time, they can also be *re-bound* — components can respond to changes in run-time situations by way of dynamic reconfiguration. We specify the conditions under which it is safe to perform dynamic reconfiguration, and outline how such reconfiguration can be implemented using our design pattern model.

3. With static parameter binding, the binding of parameters to components is done (and checked) at compile-time. The compiler can thus provide strong guarantees for the correctness of the templates, since it has access to the full type system. When parameters are bound at run-time, however, we can no longer use the

type system to check for correctness of component instantiation. We provide a set of proof obligations, which guarantee the correctness of parameter bindings at run-time, along with a method for dynamically checking them.

1.5 Organization of the Thesis

In Chapter 2, we review background material on parameterized components. We outline the different modes of parameterization that we are interested in studying, along with a comparison among the different modes. We point out some of the advantages and challenges of each of these parameterization modes.

In Chapter 3, we describe the Service Facility design pattern. We explain why this pattern is important, and point out connections with other well-known design patterns. We introduce a resource allocation component as an example to explain the design pattern, and use this example in later chapters as well.

In Chapter 4, we present some of the challenges that arise from using dynamic binding for parameterized components. We describe proof obligations that have to be satisfied in order for the binding of parameters to be type-correct. We also describe a style of checking components that enforce type-safe parameter bindings at run time.

Chapter 5 describes how the Service Facility design pattern can be used for dynamic reconfiguration, which amounts to "component transplant on a live system". The particular mode of dynamic reconfiguration that we study here is dynamic module replacement. We point out the importance of this technique, and outline the conditions that are necessary for effecting dynamic module replacement in component-based, distributed systems. We explain this technique in the context of the distributed resource allocation example introduced in Chapter 3.

We present relevant related research in Chapter 6, and point out some of the major connections between our work and different bodies of work in the literature. Finally, we present our conclusions and some of the directions for future research in this area in Chapter 7.

CHAPTER 2

PARAMETERIZED COMPONENTS

2.1 Introduction

Traditionally, parameterization 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 proposals for generics in Java [Bracha, Odersky, Stoutamire and Wadler 1998] and the .NET common language runtime [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.

[1]We use the word *instantiation* to mean the setting of all parameters of a template. We refer to what is often called instantiation in the OO literature — creation of a new object of a given class — as *object creation* in order to avoid confusion.

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, such 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.

To illustrate this point, consider a payroll system that uses some routine to sort a list of items. Without loss of generality, let us assume that the particular implementation used is QuickSort. Suppose that the maintenance team for this piece of software discovers, after the system has been released and deployed, that QuickSort is not the most efficient sorting algorithm for this application domain. How much effort would it take for this routine to be replaced with a new one that implements, say, MergeSort?

At first glance, it does not seem like much work. After all, sorting algorithms are part of most every system running today. But let us draw up a list of things that might need consideration:

- What is the kind of item being sorted?
- What is the data structure used to store the items?
- How is the ordering of items determined?

- How are items input to the sort routine?
- How are results output from the sort routine?
- When is the sorting actually done?

In the worst case, all of the above may have been hard-coded into the sort routine in the existing system, making it a non-trivial effort to write the sort routine with a different algorithm, and then integrate it with the rest of the system.

If, on the other hand, the problem of sorting is recast as a component instead of a sort routine, we can construct a parameterized SortingMachine component, that is inspired by the design presented in [Weide, Ogden and Sitaraman 1994, Sitaraman, Weide, Long and Ogden 2000]. Such a component is parameterized by the kind of item being sorted, ordering of items, input/output schemes and the actual algorithm used. For instance, in the payroll example, the sorting machine could have been parameterized by PayrollRecord as the item, a PayrollRecordComparator component that would decide if two payroll records are in the right order, input/output as "one item at a time", and QuickSort as the algorithm. Further, the SortingMachine component could be parameterized by the desired ordering.

In this case, switching from QuickSort to MergeSort is reduced to merely changing one line of code in the component integration module. The client code that needs to sort requires *no* change. In fact, in an environment that supports dynamic linking, the payroll system need not even be taken down — if we design the components and associated infrastructure carefully. The whole change can be effected by hot-swapping the component implementations at run time.

In general, we recognize four kinds of template parameters [Edwards, Heym, Long, Sitaraman and Weide 1994] — types, components, constants, and math definitions.

Listing 2.1: Stack Template

```
template <class Item>
class Stack {
public:
    void push(Item& x);
    void pop(Item& x);
    int length();
};
```

A *constant* parameter lets the client specialize the template by a particular value. For example, if the Stack component was bounded, this size limit would be a constant parameter to the stack template. A *type* parameter lets the client specialize the template by supplying a specific type, PayrollRecord in our current example. A *component* parameter is used to allow the client to set up a relationship between components. The client can provide realizations of specific interfaces that the template would use. Examples of this are the PayrollRecordComparator and QuickSort parameters to the SortingMachine component. Finally, *math definitions* that are needed in the specification can be passed in as parameters. Template parameters can also be *restricted* — the actual parameter could be required to implement certain functionality in order to be valid.

2.2 Static Parameterization

In the case of C++ templates (as with Ada generics, and similar language-supplied parameterized programming support constructs), integration is done at compile time. So actual template parameters are bound to the formals at compile time. We call this mode of parameterization *static parameterization*, since the binding of parameters to

Listing 2.2: Instantiating the Stack Template

```
1 class PayrollRec { /* ... */ };
2 typedef Stack<PayrollRec> StackOfPayrollRec;
3 StackOfPayrollRec s1;
```

a template remains static for its lifetime. Each such instantiated template defines a new type that joins the existing types of the program.

As an example, consider the C++ definition of a Stack template presented in Listing 2.1. This template is parameterized by the type of Item that the client supplies. The Stack component exports methods that the client can use to push and pop Item elements, as well as to query the length of the stack.

Consider a client who wants to construct a stack of payroll records. In order to do this, the client first defines a PayrollRecord class, and then instantiates the Stack template with PayrollRecord. This is shown in Listing 2.2.

As can be seen from this code segment, at the time of instantiating the Stack template, the client program defines a new type. This new type, StackOfPayrollRecord is added to the set of types for this compilation unit. From now on, all variables declared of this type are governed by the strong typing rules of the language.

Further, before a template can be used, it *must* be instantiated. Without the instantiation step, a program that tries to use a template will not compile. The compiler, when it starts checking a particular program, expects all the templates that the program uses to be properly instantiated with actual parameters.

2.3 Dynamic Parameterization

While static parameterization offers a way for component designers to delay the decision of which particular actual parameters to bind to each formal template parameter, the decision cannot be postponed beyond compile time. Once the binding has been done, and the client program has been compiled into object code, no changes are possible to the binding without recompiling the client program.

In some situations, however, the decision of which parameter to bind to a particular template is not apparent until run time. Further, such decisions may have to be changed *during* execution of the system. With statically-bound templates, such delay in commitment [Thimbleby 1988] is impossible. What we need in such cases is to be able to postpone the binding of parameters until run time. We call this mode of parameterization *dynamic parameterization*.

Consider, for example, a network in which the nodes all compete for a single resource. The nodes in the network need not know about where the resource is physically located, nor do they need to know which conflict resolution algorithm must be executed to gain access to the resource. We can design a ResourceManager component that manages all accesses to the resource. Now, whenever a particular client wants to use the resource, a request can be sent to the ResourceManager which, based on the conflict resolution algorithm, allocates the resource to the various clients. The choice of the particular conflict resolution algorithm may be influenced by other aspects of the system — the size of the network, the frequency of requests from each client, etc. In different network situations, we would like to be able to use different conflict resolution protocols. In other words, we would like the ResourceManager component to be parameterized by a conflict resolution protocol.

Moreover, if this parameterized component supports dynamic binding of parameters, the vision is that we should be able to supply a suitable protocol as a parameter during system execution. In fact, with dynamic parameterization, the choice of the algorithm should be changeable after the system has been deployed. Let us suppose that the system was initialized to use a token ring algorithm to resolve conflicts. If the number of clients in the network becomes too high, this algorithm is no longer efficient. At such a time, the system should be reconfigureable to use a more efficient algorithm, maybe one that organizes the clients in a tree [Raymond 1989]. This change should be made *without* stopping the system.

Current language mechanisms for building parameterized components do not support dynamic binding. The methodology that we define in this dissertation provides a way of implementing components that support this mode of parameterization.

2.4 Dependent Parameterization

The flavor of parameterization that we have seen so far is simple — the component does not place any restriction on the parameter(s). This, however, is not always true. There are cases where the parameters have to satisfy some restrictions in order to be valid. For example, consider a Sorter component that sorts elements in a list of items. Further, let this component be parameterized by the type of item. For the sort method to be able to sort the list of elements, there has to be a way to compare two elements of this type. We call parameterization of this kind *dependent parameterization.*

In Listing 2.3, if the class PayrollRecord does not implement the ">" operation, the compiler would produce an error message at line 9 saying that the operator is not supported. The compiler thus checks that the parameters supplied by the client

Listing 2.3: Sorter Template

```
template <class Item>
class Sorter
{
public:
    /* ... */
    void sort()
    {
        /* ... */
        if (x > y)
        /* ... */
    }
};

class PayrollRec { /* ... */ };

typedef Sorter<PayrollRec> PayrollRecSorter;
```

produce type-correct bindings. However, there is no syntactic way to specify this restriction in the programming language so that the client can be informed of such a restriction, and hence can avoid the illegal parameter binding. In fact, none of the widely-used languages that support generics, support syntactic specification of such restrictions[2].

Fortunately, some specification languages do allow such restricted parameters. One such language is AsmL [Barnett and Schulte 2001], an executable specification language designed for .NET. For example, the AsmL specification in Listing 2.4 for the ExternalSorter component requires the parameter T to implement two interfaces — ISerializable and IComparable. If the parameter supplied does not implement both the specified interfaces, the AsmL compiler produces an error message and stops.

[2]Ada95 does support certain kinds of restrictions, but is not widely used for building commercial software. The C# 2.0 language (scheduled for release in 2005) will also have generics that support some restrictions

Listing 2.4: Sorter component specified in AsmL

```
interface ExternalSorter
of
    <T implements ISerializable
         and IComparable>
    Sort(xs as Sequence of T)
```

Such a specification informs the client programmer of the correct kind of parameters to supply to the template.

Unfortunately, the richness in specification stops at the specification language level, and is not extended to the programming language level at this time. Proposed C# generics, for example, offer no way to require that T implement both ISerializable and IComparable. One solution would be to create a new interface (called say, ISerializableAndComparable) that extends both ISerializable and IComparable, and require that T implement this new interface ISerializableAndComparable.

The best that can be done is for the component programmer to include code in the component methods to check the type of the parameter. These checks can, however, only be made at run time. Listing 2.5 shows an example of the kinds of runtime checks that the programmer would write in the component code.

Suppose that the parameter that is bound to the template is indeed an incorrect parameter. A run time check such as the one in Listing 2.5 will catch the error, but only after the system has been deployed. In the case of static dependent parameterization, this is too late to catch the error since the binding has already taken place. In order to change the parameter to supply a new, correct parameter, the system will have to be stopped, and recompiled. So given the language mechanisms for generics that are currently available, there is no way to effectively specify restrictions on

Listing 2.5: Runtime checks in C# code to check correctness of template parameters

```
class C : ExternalSorter
{
    void sort(object[] xs)
    {
        if ( !(xs is ISerializable &&
             xs is IComparable) )
        throw new InvalidArgumentException(
            ''Incorrect type: Argument xs!'');
        /* ... */
    }
}
```

template parameters, and strictly enforce such restrictions. In the case of dynamic dependent parametrization, however, the binding of the parameter happens only at run time. So the check can, in fact, catch the error *at the time it happens*. The client can then provide a different parameter that actually satisfies the restrictions.

2.5 Chapter Summary

In this chapter, we have provided an overview of parameterization, and defined the different modes of parameterized programming that we are interested in studying. We extend the view of parameterized programming to include not only abstract data types, but a more general approach to building reusable software. We have presented the key differences between static and dynamic parameterization, as well as some of the main challenges that we encounter in each of these modes.

We use the term *static parameterization* to refer to the mode of parameterization where the parameters to a component are bound statically, *i.e.,* no further change of parameters is possible once the binding has been completed. This is the mode of parameterization supported by all languages that provide linguistic constructs for

implementing parameterized components. The downside of this mode of parameterization is the inability to make any modifications to a software system after it has been deployed.

Such modification is possible, in principle, in the case of *dynamic parameterization*. In this mode, parameters to a component are bound only during run time. Since we have more information about a software system's behavior and its environment during execution than before, the choice of parameters can be more effective. Further, dynamically-bound parameters can be *re-bound* if situations change during execution. The downside with dynamic parameterization over the static version is that ensuring correctness of instantiation becomes a challenge, since the compiler can no longer ensure such correct bindings.

Finally, we use the term *dependent parameterization* to refer to the mode of parameterization in which the choice of one parameter to a particular component may depend on the choice of other parameters. The challenges that arise when dealing with dependent parameterization involve the specification of these restrictions, and ways to strictly enforce them.

CHAPTER 3

THE SERVICE FACILITY DESIGN PATTERN

3.1 Introduction

In this chapter, we introduce and describe the Service Facility design pattern. The Service Facility pattern enables the construction of parameterized components that support dynamic binding. This design pattern combines elements of three well-known design patterns — Abstract Factory, Proxy, and Strategy [Gamma et al. 1995] — to provide a methodology for building flexible parameterized components. This methodology is particularly useful in languages that do not provide a mechanism for parameterized programming (such as templates or generics). Even in languages that do support such a mechanism, the pattern offers the advantages of dynamic parameterization.

3.2 Dependencies in Software Systems

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 [Weide 2002].

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) [Meyer 1992]. 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 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. It is helpful, therefore, 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 more likely at compile time, link time, or run time, in the context of 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. Parameterization of components using a template mechanism is one decoupling approach [Batory, Singhal, Thomas, Dasari, Geraci and Sirkin 1994, Sitaraman and Weide 1994]. As a template, a concrete component can be designed so that it depends on one or more abstract components, implementations of which are 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.

With the advent of modern distributed computing environments, the length of the integration phase is more loosely defined; component integration may only occur

on the first use of that particular component, which itself could happen a long time after the system has been deployed. Especially in software systems in which different parts are implemented at different times, it is unreasonable to expect component integration to be confined to one point in time. In the context of our discussion so far, what this lengthening of integration time means is that the decoupling mechanism that we consider should be flexible enough to carry any such decoupling throughout the lifetime of the software system.

3.3 Design Patterns

Several key problems in software design are commonly recurring — nearly every large-scale software project encounters some set of problems that every member on the team has already dealt with in a different project. However, the solutions to these problems are usually so wrapped in with the particular project in question, that these solutions could hardly be re-used.

Around the late 1980s and early 1990s, software engineering researchers borrowed some ideas from Christopher Alexander's work on identifying *patterns* in architectural design [Alexander 1979]. Alexander reduced his architectural design activity into a step involving identifying previously-known problems, and another step involving the application of known solutions to these problems. Further, he invented a language in which to express these patterns in a way that was immediately accessible to other architects as well [Alexander 1977]. Alexander defines a pattern as:

> "a three-part rule, which expresses a relation between a certain context, a problem, and a solution."

This work on architectural patterns has driven a number of software engineers to find and document such commonly recurring problems (and reusable solutions to

such problems). The first such effort dates back to 1987, with a pattern language for designing user interfaces with Smalltalk [Beck and Cunningham 1987]. Since then there have been a number of efforts at identifying various kinds of patterns, in different stages of the software development process.

Design patterns are codified solutions to commonly recurring problems at the design phase [Gamma et al. 1995]. Each design pattern defines the context in which it is to be applied, the structure of the solution, and a list of potential connections to other design patterns or other parts of the software system. The "Gang of Four", who are credited with the first set of cataloged design patterns, define a classification for design patterns. *Creational patterns* are those that decouple dependencies introduced at the time of object creation. *Structural patterns* define ways in which the composition of several objects and classes can be accomplished without concrete dependencies among them. *Behavioral patterns* decouple dependencies that are introduced during the execution of a software system, and that alter the behavior of the system.

In the rest of this section, we describe three design patterns from the Gang of Four catalog. The descriptions we present include all the major features of the patterns required to understand the rest of our own methodology. For a complete treatment of the patterns themselves, including usage pitfalls, we refer the reader to [Gamma et al. 1995].

3.3.1 The Abstract Factory Pattern

The Abstract Factory pattern is an approach that can dramatically reduce — but not quite eliminate — dependencies between classes (i.e., concrete components) at the

time of object creation. 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 and C# (like most other object-oriented languages) effectively doom 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/C# 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 [Meyer 2001]. 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 [Baumgartner et al. 1996]. 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.

Example: A Sequence Component

Figure 3.1 shows the UML[3] [Booch, Rumbaugh and Jacobson 1998] 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, or 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 includes methods to add, remove, and update sequence entries by position. The "R1" implementation might take best-case constant and worst-case linear 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.

[3]For the benefit of readers who do not have experience with UML, we present a short tutorial on the notation in Appendix A.

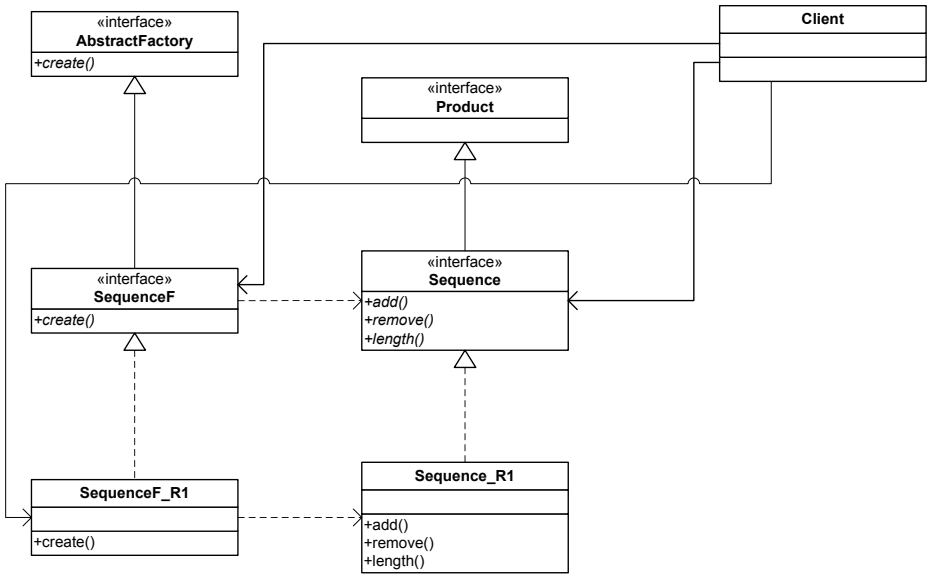

Figure 3.1: Sequence design based on the abstract factory pattern

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

```
1 Sequence s1 = new Sequence_R1 ();
2 /* ... */
3 Sequence s2 = new Sequence_R1 ();
4 /* ... */
5 s1.add (x, 3);
6 y = s2.remove (i);
7 /* ... */
```

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

```
1 SequenceF seqF = new SequenceF_R1 (); // Select implementation
2 /* ... */
3 Sequence s1 = seqF.create ();
4 /* ... */
```

```
5 Sequence s2 = seqF.create ();
6 /* ... */
7 s1.add (x, 3);
8 y = s2.remove (i);
9 /* ... */
```

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/C#) unavoidable place. Note that the client declares both factory and product objects to have interface types.

3.3.2 The Proxy Pattern

Consider a client/server application in which the client first obtains an object reference to the server, and then starts interacting with it. Further, suppose that the decision of whether the client and server run on the same machine, or on physically distributed locations, is not known at the time of programming the client application. How can the client program be designed to interact with the server without this knowledge?

One possible solution to this situation is that at every instance where the client application wants to send a message to the server, it checks whether the server is a local or a remote object. Based on this information, the client could either make a simple local method invocation, or send a remote method invocation. The following pseudocode shows a client program accessing a Subject server si. These lines of code (3 through 10) have to be repeated at every place in the client program where an interaction with si is needed. This solution, however, lacks efficiency (a check has

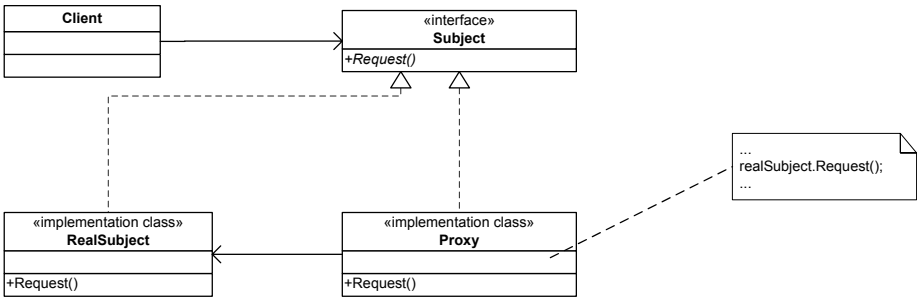

Figure 3.2: UML design structure for the Proxy pattern

to be made every time a method invocation is made) and elegance (the client code includes numerous such checks, which are distracting).

```
1  Subject si = subjectFactory.create();
2  /* ... */
3  if (si is local instance)
4  {
5      si.Request();
6  }
7  else
8  {
9      send Request() as remote invocation to si.
10 }  /* ... */
```

A cleaner solution to this problem is to use the Proxy pattern. Under the Proxy pattern, the aforementioned client application would be programmed to interact with a *proxy*, which would in turn know how to actually access the server instance. This proxy object is a local object on the client side. The client just makes simple method invocations on the proxy, and does not care about the actual physical location of the server. Figure 3.2 shows the UML design structure of a client that uses the Proxy design pattern to access a server instance. Using the Proxy pattern, the client code

now looks like the following:

```
1 Subject si = subjectFactory.create();
2 /* ... */
3 si.Request();
4 /* ... */
```

As can be seen in the UML diagram, the Proxy implements the same interface as the subject that the client wants to interact with. Because of this, the client program can be written without the knowledge that it is interacting with a proxy object. As far as the client program is concerned, it obtained a reference to an object that implements the Subject interface, and it can make method invocations on this object just like any other.

Apart from the use of the Proxy pattern shown above, there are other advantages to using this pattern. Since the client no longer has a direct reference to the object instance, the object instance can be modified without the client having to be notified of such change. Examples of such changes include dynamic load balancing, where the physical location of the object is changed — which is at least expensive if not impossible if the client holds physical references to the object instance.

While the Abstract Factory pattern decouples the dependency between a client program and a concrete object class, once an object instance has been created by the factory, there is a concrete dependency introduced in the system. The client instance has a direct reference to a concrete object instance. The Proxy pattern extends this decoupling through the lifetime of the object. The dependency between the concrete client *instance* and the concrete object *instance* is broken using the Proxy pattern.

The client depends *only* on the interface, and not on any particular implementation, *throughout the lifetime of the system.*

3.3.3 The Strategy Pattern

The Strategy design pattern is another technique that can be used to decouple behavioral dependencies between concrete components. Portions of a component's behavior that may change during the lifetime of the system are abstracted out and placed in a separate component, with interface-level dependencies between the components.

In order to explain this pattern better, we go back to the payroll sorter example from Chapter 2. The example shows a parameterized SortingMachine component with the sorting algorithm as one of its parameters. At system integration time, an implementation of the sorting algorithm is provided to the SortingMachine. In this case, we observed that the particular sorting algorithm is independent of the SortingMachine component, and moreover, that the algorithm could change during the lifetime of the component. Therefore, the sorting algorithm was abstracted into a separate component. This is, in essence, what the Strategy pattern does.

Figure 3.3 shows the UML design of the SortingMachine component using the Strategy pattern. In this pattern, the SortingMachine plays the role of the *context*, and SortingAlgorithm is called the *abstract strategy*. The implementations of SortingAlgorithm — QuickSort and MergeSort — play the role of *concrete strategy*.

Under the Strategy pattern, the context is decoupled from the strategy. There is an abstract dependency from the context to the abstract strategy (an interface). At integration time, a concrete implementation of the strategy is supplied to the context.

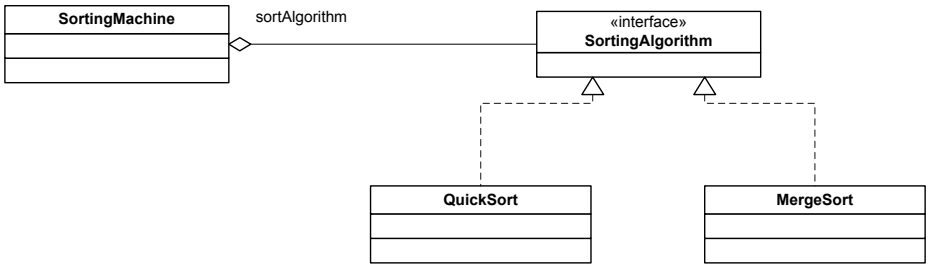

Figure 3.3: UML design diagram of a system using the Strategy design pattern.

The behavior of the context can thus be varied by changing the particular concrete strategy that is supplied.

The context holds a reference to a strategy. This is a run-time relationship, which has to be set up during execution. The context class includes a data member in which to store the object reference to the strategy instance, and an overloaded constructor that takes a strategy instance as a parameter. A common way of instantiating a context object instance with a particular strategy is to pass an instance of the strategy as a parameter to the context's constructor. The constructor can initialize the strategy data member with this object reference.

```
1  public class SortingMachine
2  {
3      private SortingAlgorithm sortAlgorithm;
4      /* ... */
5
6      public SortingMachine(SortingAlgorithm sorter)
7      {
8          /* ... */
9          this.sortAlgorithm = sorter;
10         /* ... */
11     }
12     /* ... */
13 }
```

Invoking the constructor with different strategy instances produces context instances with appropriate behavior.

```
1 SortingMachine quickSorter = new SortingMachine(new QuickSort());
2 SortingMachine mergeSorter = new SortingMachine(new MergeSort());
```

Using the overloaded constructor as the only way to set the strategy in the context class limits the flexibility that can be afforded by the Strategy pattern, since the strategy cannot be changed once the context object has been constructed. In order to address this concern, in addition to the overloaded constructor (or as an alternative to it), a separate method is defined to initialize the strategy data member. The context class exposes a public method[4] to allow the strategy to be modified. The method can be called by the client itself, or by an administrative component, and passed an instance of a new concrete strategy in order to change the strategy being used.

```
1 SortingMachine sorter = new SortingMachine();
2 sorter.setSortingAlgorithm(new QuickSort());
3 /* ... */
4 sorter.setSortingAlgorithm(new MergeSort());
```

The Strategy pattern thus breaks dependencies among various *behavioral* aspects of a software system. Component boundaries are defined around behavioral entities rather than data types. This is in accordance with Parnas's observation that systems should be decomposed into modules based on behavior, rather than subroutines tied to data structures [Parnas 1972].

[4]This could also be a property in .NET languages.

3.4 The Service Facility Pattern

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.

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 aspect of the service-industry metaphor also offers a solution to the decoupling problem during an object's lifetime (Section 3.3.2).

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.

This composition of the Abstract Factory and Proxy pattern lies at the core of the **Service Facility** pattern. 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** (or Serf, for short) because it must be asked to help perform all services on, i.e., manipulations of, the data objects for which it is responsible.

Like to 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. Figure 3.4 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.

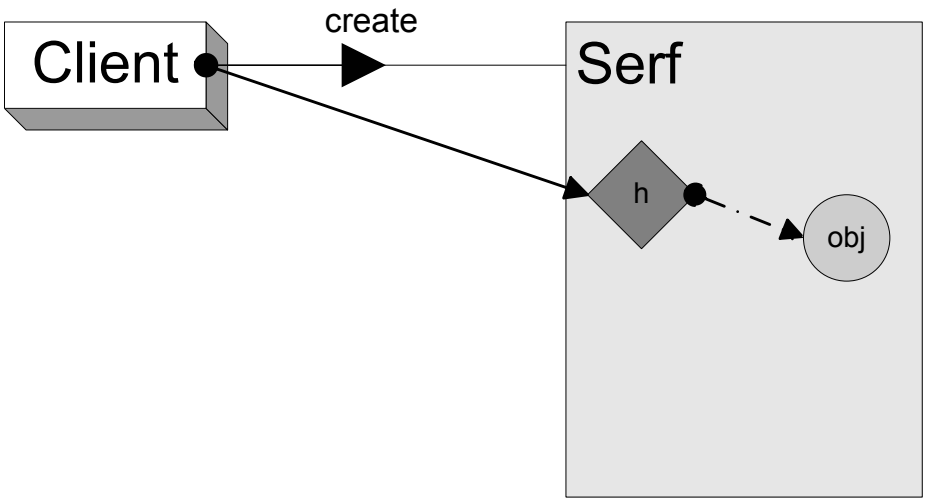

Figure 3.4: Object Creation in Serfs

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 (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. Figure 3.5 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.

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.*

Apart from decoupling dependencies between the client instance and object instances, the Service Facility pattern also supports decoupling of dependencies between

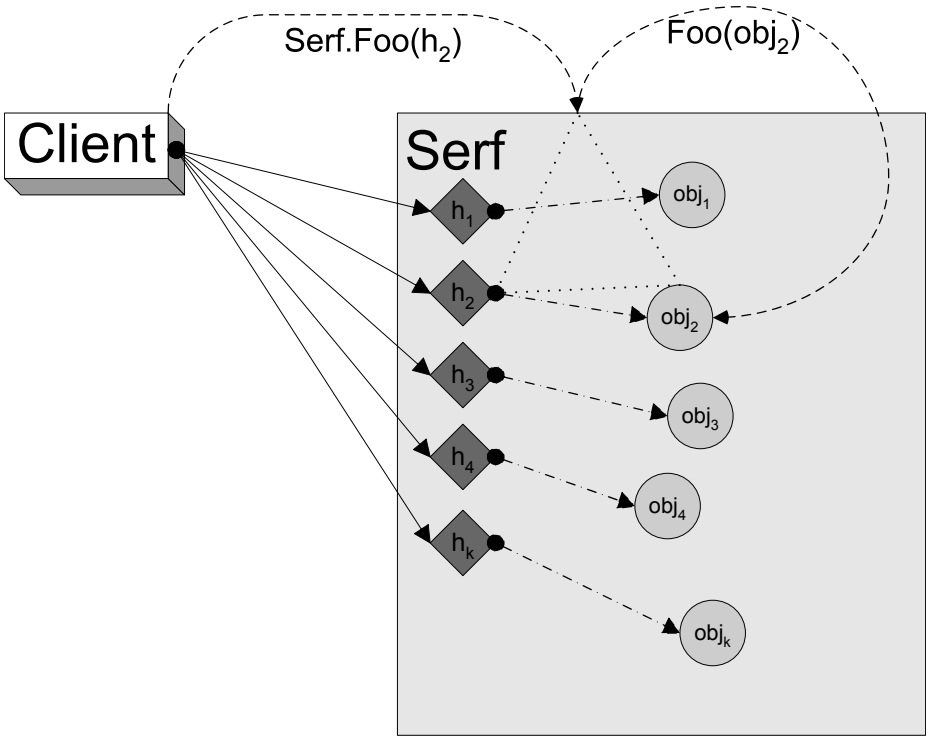

Figure 3.5: Method redirection in Serfs.

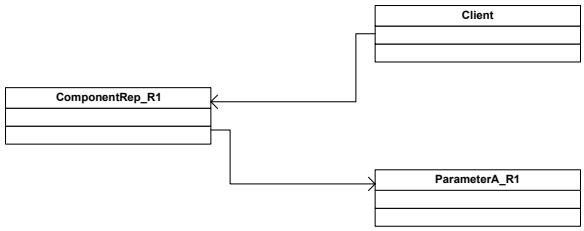

Figure 3.6: UML design of a system with concrete dependencies

different behavioral aspects of a component. The pattern allows the construction of dynamically-bound parameterized components. Each Serf is a template, whose parameters are set as *strategies* (Strategy design pattern).

3.4.1 Implementing Service Facilities

Figure 3.6 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 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.

Figure 3.7 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. Every Serf implements the ServiceFacility interface, and the objects that a Serf creates

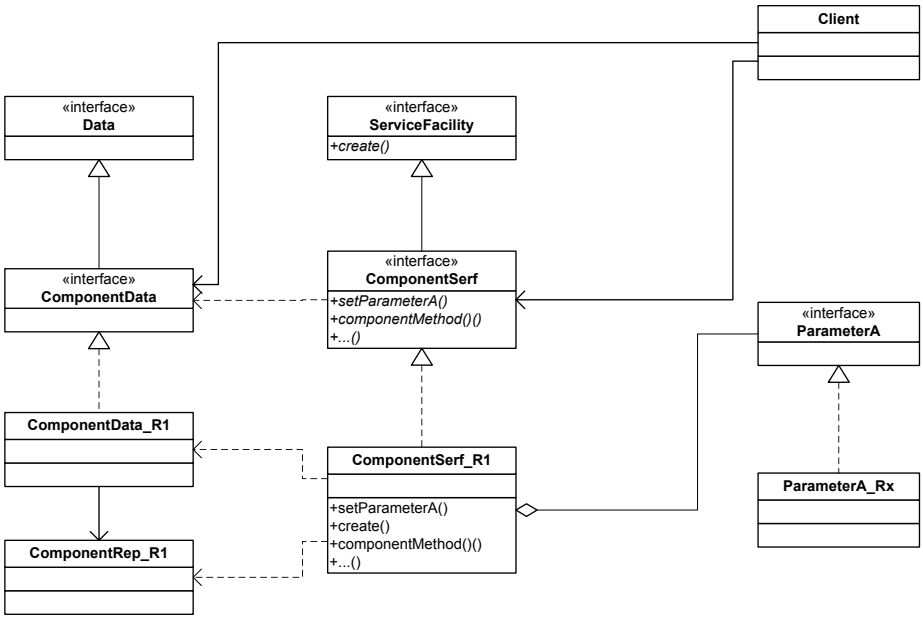

Figure 3.7: UML design structure of the Service Facility pattern

all implement the Data interface. Listing 3.1 shows these two interfaces written in C#. 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.

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 in turn holds a reference to an instance of

Listing 3.1: The Data and ServiceFacility interfaces

```
1  namespace ServiceFacility
2  {
3      public interface Data
4      {  }
5
6      public interface ServiceFacility
7      {
8          Data create();
9      }
10 }
```

ComponentRep_R1. So Client uses the ComponentRep_R1 instance through the handle it holds (the ComponentData_R1 instance).

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.

```
ComponentSerf cSerf = new ComponentSerf_R1();
/* ... */
ComponentData c = (ComponentData) cSerf.create();
```

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 Figure 3.6 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:

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

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

```
cSerf.componentMethod(c);
```

This view departs from the traditional object-oriented style where an object includes the data, as well as the operations defined on it. We outline some of the reasons for taking this view in Section 3.4.4.

The other dependency in Figure 3.6 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. Details of such change during execution are presented in Chapter 5.

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.

Fortunately, though, we have already seen a way of ensuring correctly instantiated templates (Chapter 2). 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. A detailed treatment of these type-safety checks is presented in Chapter 4.

Implementer View

Listing 3.2 shows the skeletal structure of a service facility class ComponentSerf_R1 in C#[5]. There are five major sections in every service facility component, including this one:

Type Definition (Lines 3 — 21) A service facility exports some types — typically one, but sometimes zerp, sometimes more than one. A type exported by a service facility is expressed as an *inner class*. 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.

Template Parameter Definition (Lines 23 — 36) 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.

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

The rep Method (Lines 41 — 46) 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 48 — 57) 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 can perform administrative maintenance if needed. The importance of this will be seen in Chapter 5 when we discuss dynamic reconfiguration.

Component Method(s) (Lines 59 — 65) 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).

Listing 3.2: Structure of a service facility

```
1 public class ComponentSerf_R1 : ComponentSerf
2 {
3     // type definition
4     internal class ComponentData_R1 : ComponentData
5     {
6                                                    Continued
```

Listing 3.2 continued

```
        internal ComponentRep_R1 rep;

        internal ComponentData_R1()
        {
            rep = new ComponentRep_R1();
            /* Initialize representation fields */
        }
    }

    internal class ComponentRep_R1
    {
        /* Data members */
    }
    // end type definition

    // template parameter
    private ParameterA pASerf;
    public ParameterA PASerf
    {
        get
        {
            return pASerf;
        }
        set
        {
            pASerf = value;
        }
    }
    // end template parameter

    // list of objects created
    ArrayList listOfObjects = new ArrayList();

    // rep method to extract real object instance internally
    private ComponentRep_R1 rep(ComponentData cData)
    {
        return ((ComponentData_R1) cData).rep;
    }
    // end Rep method

    // create method from ServiceFacility
    public Data create()
    {
```

Continued

Listing 3.2 continued

```
        ComponentData_R1 newObj = new ComponentData_R1();
        listOfObjects.Add(newObj);
        return newObj;
    }
    // end create method

    // component methods
    public void componentMethod(ComponentData c)
    {
        ComponentRep_R1 cRep = rep(c);
        /* ... */
    }
    // end component methods
}
```

3.4.2 Performance Considerations

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. The extra computation is what extracts the representation instance that corresponds to the data handle that the client program holds. 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.

Weak References

The service facility is 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 3.3.

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

Listing 3.3: create() method with weak references to data objects

```
public Data create()
{
    ComponentData_R1 newObj = new ComponentData_R1();
    WeakReference wrObj = new WeakReference(newObj, false);
    listOfObjects.Add(wrObj);
    return newObj;
}
```

Listing 3.4: rep() method using lazy initialization

```
private ComponentRep_R1 Rep(ComponentData c)
{
    if (((ComponentData_R1) c).rep == null)
    {
        ((ComponentData_R1) c).rep = new ComponentRep_R1();
        /* Initialize representation fields */
    }
    return ((ComponentData_R1) c).rep;
}
```

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 3.4). Again, while lazy initialization is not a central feature of the Service Facility pattern, it is a useful optimization.

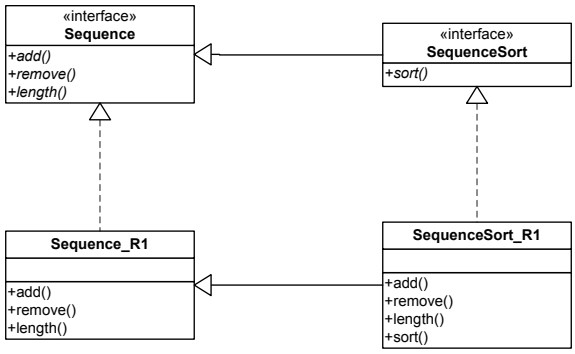

Figure 3.8: Extending Sequence_R1 with sort using inheritance

3.4.3 Enhancements

The standard object-oriented way of adding functionality to an existing component is to use *inheritance* [Meyer 1988]. The new functionality is placed in a separate component (*derived class*) that inherits functionality from the component that is being extended (*base class*). When a class C' inherits from another class C, the derived class C' acquires all the data members and methods of the base class C. The derived class could, however, elect to *override* some or all of the methods in the base class. Moreover, the derived class C' includes new methods that provide the enhanced functionality.

Constructing enhancements using inheritance has a few problems. First of all, inheritance creates a concrete dependency between the base class and the derived class — the derived class *borrows* data and methods from the base class. Further, the enhancement in the form of the derived class is tied to exactly one implementation of the abstract component that is being extended.

Consider an abstract Sequence component with the operations add, remove, and length to add an item, remove an item, and query the length, respectively. Consider an implementation of the Sequence component, Sequence_R1. If we wanted to extend the functionality of the Sequence component to include a Sort operation to sort the sequence in some order, the standard OO way is to create a derived class of Sequence_R1 called SequenceSort_R1, that has not only the Sequence operations (inherited from Sequence_R1), but also the sort operation. This scenario is show in Figure 3.8. The sort operation is contained in the SequenceSort interface, which extends the Sequence interface. The SequenceSort_R1 class implements the SequenceSort interface, meaning that this class has to provide the operations that both the interfaces Sequence and SequenceSort provide. This class inherits the Sequence operations from Sequence_R1, and implements sort on its own.

Now let us suppose that we create a new implementation of Sequence, say Sequence_R2. This implementation cannot use the Sort extension, since SequenceSort_R1 is a derived class of Sequence_R1, and is therefore tightly coupled to that implementation. If we want to be able to sort instances of Sequence_R2, we would have to create a new derived class, this time inheriting from Sequence_R2 to add this functionality. This is not a scalable way of software development, since every new implementation of a component has to be accompanied by all the enhancements to the component as well. It also results in proliferation of nearly-identical code (here, Sort body) and hence the loss of single point of control over change.

The more scalable approach is to build a single enhancement component, SequenceSort_Enh, that can be used with any implementation of Sequence. This is the scenario shown in Figure 3.9. In this case, the enhancement is built using a

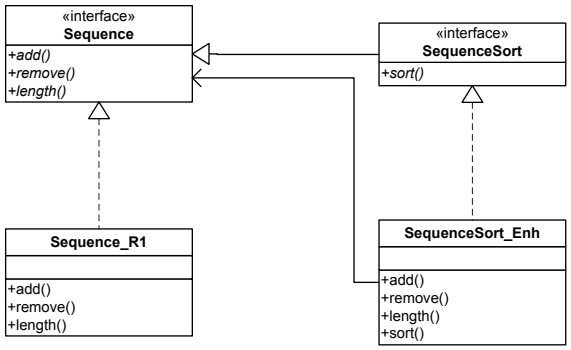

Figure 3.9: Extending Sequence with sort using delegation

technique known as *delegation*, or *object composition* [Gamma et al. 1995]. The SequenceSort_Enh provides the same set of methods as SequenceSort_R1 — add, remove, length, and sort; but instead of *inheriting* the Sequence methods from *a particular* implementation of Sequence, it now *delegates* these methods to *any* implementation of Sequence. This is denoted in the figure via a "uses" arrow from SequenceSort_Enh to the Sequence interface; meaning SequenceSort_Enh "uses some implementation of" the interface.

The standard way of implementing enhancements in the Service Facility pattern is to use delegation, rather than using inheritance. Further, the implementation of the base component is provided to the enhancement as a parameter. The enhancement does not refer to any of the state in the implementations; it does not have access to the implementation class's state. The enhancement interacts with the base implementation as a client of the base implementation, invoking operations on the base implementation. Such enhancements are called *layered* enhancements, since the new operations are implemented as layers over the base component's operations.

3.4.4 Separating Code from Data

As we have seen in Section 3.4.1, 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 (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 Clemens 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.

As opposed to traditional object-oriented systems, where a class defines a single type and 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. 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. Apart from the

cosmetic asymmetry that this notation leads to, there are also technical reasons that the component-oriented approach is favored over the object-oriented one.

Binary Operations

Let us consider a (regular, object-oriented) Point class which looks like the code segment in Listing 3.5[6]. The class exports three public methods (functions), two to access the values of xVal and yVal, and the third to check if a second Point instance is equal to the receiving Point instance. There seems to be something different about the equal method, which tests the equality of two Point objects. Equality is a binary operator, but as it is written in the class, the method takes only one parameter. The other parameter is the object instance on which the method is invoked. As may be seen from this, binary methods, when expressed in the object-oriented notation, are asymmetric.

The ColorPoint class extends the Point class using inheritance, and adds state (cVal). Moreover, the equal() method from Point is overridden in ColorPoint to mean something different. The method breakit() in Listing 3.6 exposes the problem that results from this inheritance relationship and the asymmetric notation for expressing the binary method equal(). The method works perfectly fine if invoked with an instance of Point as the parameter. But if the method is invoked as follows:

```
1 ColorPoint cp = new ColorPoint(3, 4, "white");
2 breakit(cp);
```

the program will not work, since at the time the equal() method is invoked in the breakit() method (Line 4), the invocation is sent to the equal() method defined in the

[6]This example is borrowed from [Bruce, Cardelli, Castagna, Group, Leavens and Pierce 1995].

Listing 3.5: The Point and ColorPoint classes

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

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

class ColorPoint : Point
{
    private string cVal;

    public string c() { return cVal; }
    public bool equal(ColorPoint p)
    {
        return ((this.xVal == p.x()) &&
                (this.yVal == p.y()) &&
                (this.cVal == p.c()));
    }
}
```

Listing 3.6: Method exposing the binary method problem

```
public void breakit(Point p)
{
    Point newPoint = new Point(3, 4);
    if (p.equal(newPoint))
    /* ... */
}
```

ColorPoint class. This method, however, assumes that the parameter it receives is also a ColorPoint, and tries to invoke the c() method on it, which will fail in this particular example, since the parameter to the equal() method (newPoint) is an instance of Point.

Note that the compiler will accept this program as correct. The reason that this problem shows up is that according to object-orientation, inheritance is a mechanism for achieving subtyping [Cardelli 1984]. Since ColorPoint inherits from Point, the compiler considers it a subtype of Point. However, as shown by the example above, ColorPoint is not a behavioral subtype of Point, since the equal() methods in the two classes do not follow the contravariant rule for subtyping [Castagna 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. Moreover, since the Service Facility pattern does not use inheritance to extend functionality, the problem shown above does not arise either.

3.4.5 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. Please note that the rest of the dissertation will not follow the formulation presented in this section; we will use the component view

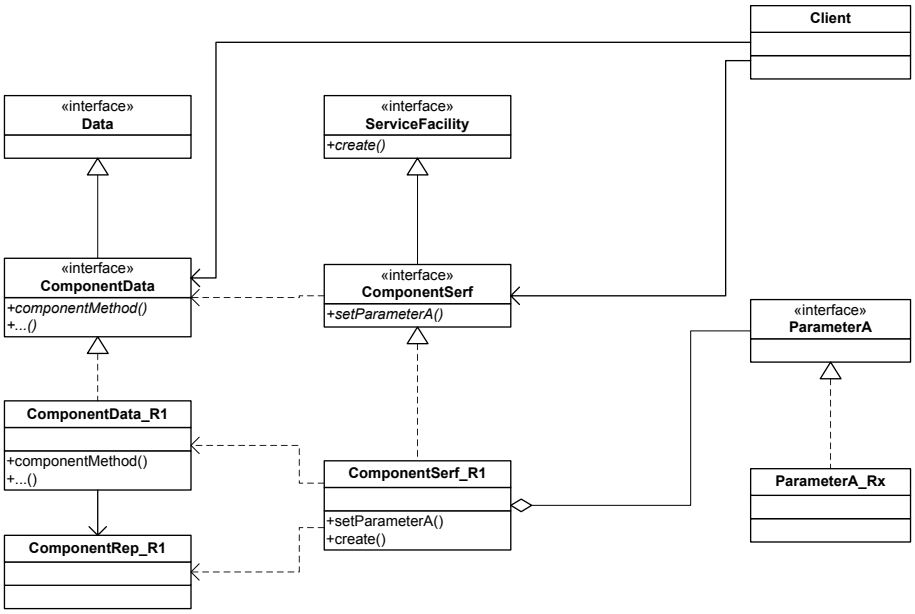

Figure 3.10: UML design showing Serflets in the Service Facility pattern

presented so far. However, modifying the concepts presented in later chapters to the object-oriented view should not be difficult, and we leave this to the reader who does not want to give up the obj.Meth 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, Figure 3.7) to the data class (Component-Data_R1). The new UML design is presented in Figure 3.10. Although we move the methods into ComponentData_R1, this is not the same as the *product* object from the Abstract Factory pattern (Section 3.3.1). 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();
```

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 data 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.

3.4.6 Mediation in Service Facility Wrappers

The Service Facility pattern also offers a solution path for problems of *separation of concerns* [Tarr, Ossher, Harrison and Sutton 1999]. Apart from the functional properties, software components typically also include behavior that is not central to the functionality of the component. Examples of such behavior include message logging, transaction management, etc. Several solutions to these problems have been proposed — aspect-oriented programming [Kiczales, Lamping, Menhdhekar, Maeda, Lopes, Loingtier and Irwin 1997], subject-oriented programming [Harrison and Ossher 1993], containers [SunMicrosystems 2001], interceptors [Hallstrom, Leal and Arora

2003]. All of these solutions include some way of tracking the flow of messages to and from the component instance.

Since the service facility is essentially a *wrapper* [Büchi and Weck 2000], it offers a way of observing all messages flowing into and out of the data objects it manages. Not only can the Serf observe the messages flowing across, it can perform tasks based on the nature and content of these messages. For instance, the Serf is a natural place to log all method invocations going to the data objects it maintains. The Serf can also act as a checking wrapper for the data objects that can ensure that the client and the component both respect design by contract [Edwards, Shakir, Sitaraman, Weide and Hollingsworth 1998]. Method invocations can be intercepted on the way into the component for pre-condition checks, and responses can be intercepted to make sure that the post-condition holds.

In fact, these non-functional services can themselves be modularized, and provided to the Serf as parameters. The Serf can then be built in such a way that for every method invocation, all the non-functional services are performed before the method is actually executed. For instance, Figure 3.11 shows a service facility wrapper that includes three services — message logging, transaction management, and contract checking. The Serf performs all these tasks, and then executes the method.

3.5 Bringing It All Together: Resource Allocation Example

The problem of *mutual exclusion* involves the synchronization of the activities of (logically) concurrent processes. The synchronization can be viewed as a way to coordinate access to a shared resource, or a privilege to execute a particular section of code. The problem of mutual exclusion has been well studied and numerous solutions

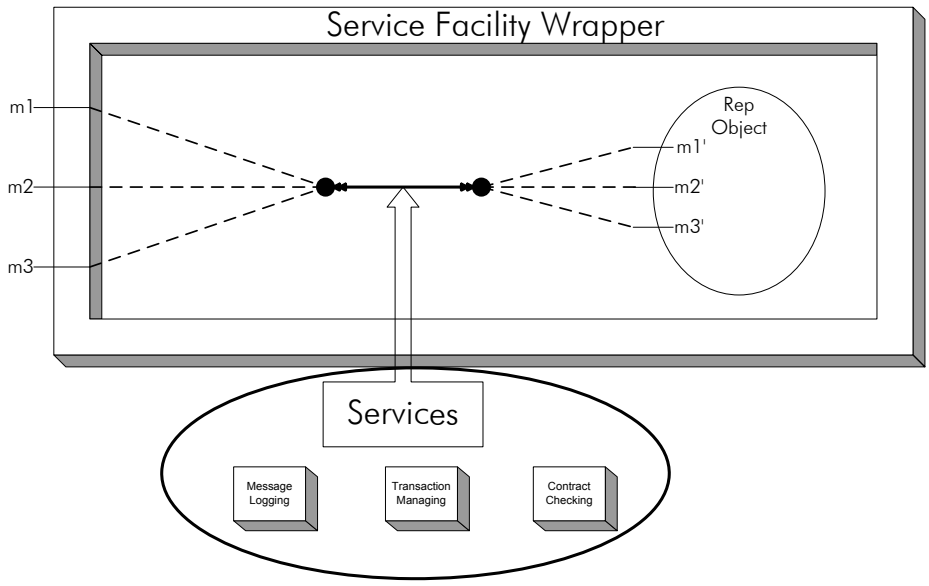

Figure 3.11: A Serf wrapper that uses multiple services to perform checks on intercepted messages

are available in the literature. For a comprehensive presentation of mutual exclusion and applications see [Lamport and Lynch 1990] and [Lynch 1996]. For a list of such solutions, we refer the reader to [Neilsen and Mizuno 1991].

One common drawback of all these solutions is that the processes in the network are tied to a particular conflict resolution protocol. Once the network application has been deployed, it is not possible to change the conflict resolution policy without stopping the processes, or using some specialized configuration languages not suitable for enterprise development environments. This is undesirable because the conflict resolution policy for mutual exclusion is a design decision that is completely independent from the problem domain. In the ideal case, we should be able to change the conflict resolution policy even while the system application layer is running on top of it.

Listing 3.7: The ResourceSerf as a C# Serf

```
interface ResourceSerf : ServiceFacility
{
    public ProtocolSerf protSerf
    {
        get; set;
    }

    Resource joinNetwork();
    void leaveNetwork(Resource r);
    bool isRequested(Resource r);
    bool isAvailable(Resource r);
    void request(Resource r);
    void release(Resource r);
}
```

In order to be able to make such changes at execution time, the component must be designed for such change [Parnas 1979]. Listing 3.7 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

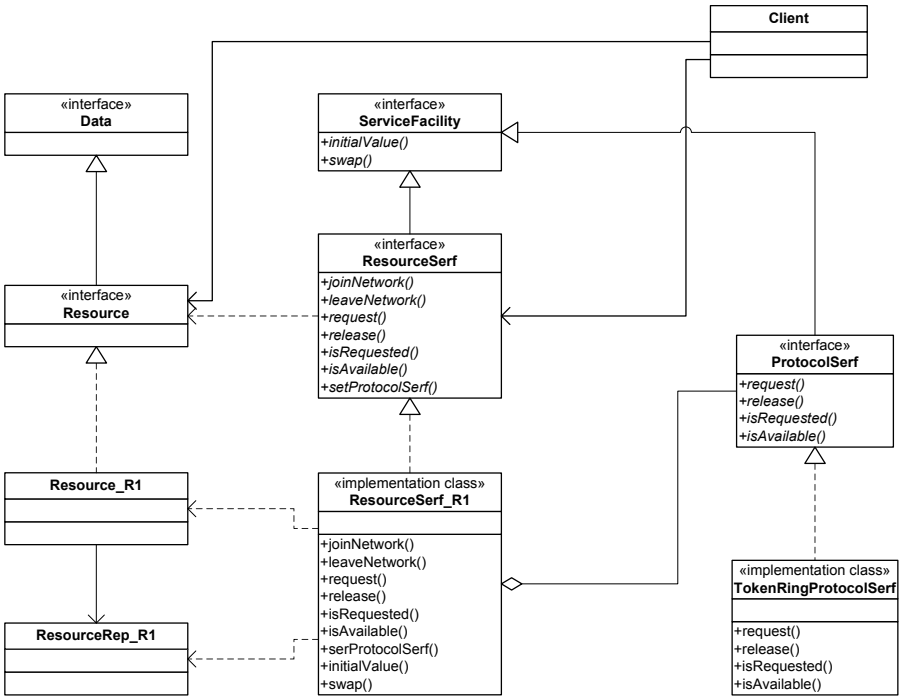

Figure 3.12: UML Design for the resource allocation example using the Service Facility pattern.

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.

Figure 3.12 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

Listing 3.8: create() and request() methods from ResourceSerf

```
internal class ResourceRep_R1
{
    bool requested;
    bool available;
    ProtocolProxy protocolProxy;

    public ResourceRep_R1()
    {
        protocolProxy = (ProtocolProxy) protSerf.create();
        requested = false;
        available = false;
    }
}

public Data create()
{
    Resource_R1 newResource = new Resource_R1();
    WeakReference wrRes = new WeakReference(newResource, false);
    clientList.Add(wrRes);
    return newResource;
}

public void request(ref Resource r)
{
    ResourceRep_R1 rRep = Rep(r);
    rRep.requested = true;
    ProtocolProxy pp = rRep.protocolProxy;
    protSerf.request(ref pp);
}
```

the client wants to use. The actual conflict resolution is done 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 3.8 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 should not have to be 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[7].

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.

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 this case, however, the client only holds a reference to a logical handle, and not to the actual representation object.

[7]The client processes may experience lower performance levels during reconfiguration, but the client does not have to *do* anything for the reconfiguration to occur.

3.6 Chapter Summary

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 chapter, we presented three kinds of concrete dependencies that occur in different phases of a software system's life cycle; and design pattern solutions to each of these different kinds of dependencies. The Service Facility pattern unifies these independent solutions to a common model for developing flexible software components that can be specialized for different usage scenarios through parameterization. We presented details of the Service Facility pattern as well as some possible extensions and optimizations for the pattern.

A preliminary version of the work described in this chapter was presented at the International Conference on Software Reuse in Austin, TX in April 2002 [Sridhar, Weide and Bucci 2002].

CHAPTER 4

ENSURING TYPE-CORRECT DYNAMIC PARAMETER BINDING

4.1 Introduction

In Chapter 2, we outlined some of the challenges that we face when building dynamically bound parameterized components. We noted there that we need ways of checking at run time whether the parameters supplied to a component meet all of the restrictions imposed on them. In Chapter 3, we presented the Service Facility pattern as a way of implementing dynamically bound parameterized components.

In this chapter, we present in detail the kinds of run time checks that are needed to ensure type-correct bindings of parameters to templates during execution time. In addition, we show how these checks can be generated automatically from XML descriptions of the components.

4.2 Specifying Parameterized Components

In order to specify parameterized components, we use the RESOLVE [Edwards et al. 1994] notation. As an example, we present StackContract (borrowed from [Edwards et al. 1994]) specified using the RESOLVE notation in Listing 4.1. This module defines one type (Stack) and its interface exports three operations on this type — push,

pop, and length. The type definition describes a mathematical model (string of Item in this case), as well as the set of legal values that a new instance of this type can assume upon initialization (empty string in the case of Stack). Each module can export zero or more types. In the case of a module that exports more that one type, the types are identified using a *type identifier*. For the sake of simplicity, in this discussion we only deal with components that export exactly one type, and so we will no longer refer to the type identifier.

The *global context* of this contract introduces other modules or facilities that this component *uses*. In this particular example, StackContract makes use of an Integer component, and therefore *imports* the standard realization of that component (StandardIntegerFacility). In programming language terms, the global context serves the same purpose as import statements in Java, using statements in C#, or #include statements in C++.

The parameters to this template are specified in its *parametric context*. In the particular example, StackContract is parameterized by the type of item that is contained in a stack. In general, parameters can be of four different kinds: constants, types, facilities, and math definitions. A *constant* parameter lets the client specialize the template by a particular value. For example, if the stack component was bounded, this size limit would be a constant parameter to the stack template. A *type* parameter lets the client specialize the template by supplying a specific type, as is the case in our current example; the client program provides the type of items that the Stack would hold. A *facility* is an instance of some template. Thus, a facility parameter is used to allow the client set up an integration-time relationship between components. The

Listing 4.1: The contract for StackContract specified using RESOLVE

```
1  contract StackContract
2      context
3          global context
4              facility StandardIntegerFacility
5          parametric context
6              type Item
7
8      interface
9          type Stack is modeled by string of Item
10             exemplar s
11             initialization
12                 ensures
13                     |s| = 0
14
15         operation push (
16                 alters s: Stack,
17                 consumes x: Item
18             )
19             ensures
20                 s = <#x> * #s
21
22         operation pop (
23                 alters s: Stack,
24                 produces x: Item
25             )
26             requires
27                 |s| > 0
28             ensures
29                 #s = <x> * s
30
31         operation length (
32                 preserves s: Stack
33             ) : Integer
34             ensures
35                 length = |s|
36 end StackContract
```

client can provide realizations of specific contracts that the template can use. Finally, *math definitions* that are needed in the specification can be passed in as parameters.

Template parameters can also be *restricted* — the actual parameter could be required to implement certain functionality in a valid binding (dependent parameterization, Section 2.4). An example of such a restriction would be the requirement that the kind of items that can be put in the Sorter template from Listing 2.3 appropriately implement the comparison operator (>).

4.2.1 Specifying Serfs in RESOLVE

The contract presented in Listing 4.1 specifies that it requires, as part of its global context, the standard integer facility. Recall that a facility is an instance of a template, all of whose formal parameters have been bound to actuals. In order to accommodate run-time binding of parameters, we introduce new notation to the RESOLVE language. A *service facility* is an instance of a template that is bound to its parameters dynamically, rather than statically.

Further, we unify all the different kinds of parameters that can be part of the parametric context of a RESOLVE template to be service facilities. In the case of type parameters, we specify in the parametric context a service facility that *defines* the required type; a constant parameter is expressed as a service facility that provides the constant value; and math definitions are expressed as service facilities that provide program functions that correspond to the math definition.

Substituting service facilities for facilities, we can translate the RESOLVE StackContract (Listing 4.1) into StackSerfContract (Listing 4.2). It is easy to see that the

Stack and StackSerf C# interfaces (Listing 4.3) can be generated from StackSerf-Contract. All the information needed to generate these interfaces is available in the contract. In general, the type(s) exported by a SerfContract is used to generate the type interface(s) (Stack in the example), and the interface part of the contract, along with the parametric context is used to generate the Serf interface.

4.2.2 Realizing RESOLVE Contracts as Serfs

Abstract RESOLVE components are expressed in the Service Facility pattern as interfaces; concrete components are expressed as classes. In the languages that we consider (Java and .NET languages[8]), interfaces are first-class constructs in the language. Interfaces in these languages are comprised of *method signatures*. There is a direct mapping from the interface in the RESOLVE specification to the programming language interface. Further, the template parameters listed in the concept's parametric context are also represented by methods in the interface. Each facility parameter in the concept corresponds to two methods — one *setter*, and one *getter*. In C#, each template parameter can be represented by a single *property* [Archer 2001].

For example, Listing 4.3 shows the C# interface for a StackSerf component. There is a one-to-one correspondence between the operations in the RESOLVE contract in Listing 4.2 and the methods in this interface. Further, the parameter in the parametric context of StackSerfContract corresponds to the *property* ItemSerf. This property will translate to a setter and a getter method for ItemSerf in realizations of this interface.

[8]All languages that respect the Common Type System of the Microsoft .NET Common Language Runtime have the same set of features [Microsoft 2002a]. Henceforth, whenever we want to refer to .NET languages, we will use C# as the representative.

Listing 4.2: The contract for StackSerf

```
1  contract StackSerfContract
2      context
3          global context
4              service facility StandardIntegerSerf
5          parametric context
6              service facility ItemSerf
7                  defining type Item
8
9      interface
10         type Stack is modeled by string of Item
11             exemplar s
12             initialization
13                 ensures
14                     |s| = 0
15
16         operation push (
17                 alters s: Stack,
18                 consumes x: Item
19             )
20             ensures
21                 s = <#x> * #s
22
23         operation pop (
24                 alters s: Stack,
25                 produces x: Item
26             )
27             requires
28                 |s| > 0
29             ensures
30                 #s = <x> * s
31
32         operation length (
33                 preserves s: Stack
34             ) : Integer
35             ensures
36                 length = |s|
37 end StackSerfContract
```

Listing 4.3: C# Stack and StackSerf interfaces

```
1  public interface Stack : Data
2  { }
3
4  public interface StackSerf : ServiceFacility
5  {
6      void push(Stack s, Data x);
7      void pop(Stack s, Data x);
8      int length(Stack s);
9
10     // Template parameter(s)
11     ServiceFacility ItemSerf
12     {
13         get;
14         set;
15     }
16 }
```

4.3 Specifying Dynamically-Bound Parameterized Components

When using static parameterization, the compiler enures that the template bindings are all legitimate from the point of view of type-correctness of parameter bindings. With dynamically-bound Serf templates, we have to ensure that we perform the same set of checks to ensure legitimacy of the bindings. Before certifying a particular template as valid, a template-aware compiler performs the following checks:

R1. Enforcing Instantiation. Before it is used in a client program, a template must be fully instantiated — all parameters to the template must have been supplied, and

R2. Enforcing Restrictions. The actual template parameters result in type-correct bodies for the template's methods.

Before we describe how the Serf pattern can be augmented with additional proof obligations to ensure correct bindings, we take a short detour to convince the reader that compiler checks are not sufficient to enforce the above requirements when dealing with dynamic binding.

Since Serfs are really just objects at the linguistic level, as soon as the constructor has been invoked by the client and an object has been allocated, the compiler considers the Serf object ready for use. However, under the semantics of the Serf design pattern, this Serf may not be ready for use yet. The reason is that there may be template parameters that need to be supplied to the Serf. So until these parameters have been set, the client should not be allowed to invoke any methods on the Serf, including create(). Since the template parameters are really just properties (or data members) of a run-time object, the compiler currently does not check for these properties being set. We could instrument the compiler to perform additional static analysis to ensure that every path leading to a method call on the Serf already has lines of code that set the template parameters. This would require that we had special syntax to distinguish data members that represent template parameter from regular data members. However, we would like to be able to perform these checks without making changes to the language or the compiler, so that the solution is immediately deployable.

In Section 2.4, we have shown how the syntactic constructs provided by programming languages are not sufficient to completely specify templates in the presence of dependent parameters. As another example of such dependent parameters, consider a Sequence component. A Sequence is modeled by a mathematical string, in which items can be added to and deleted from any arbitrary position (pos). Further, the

component also provides an operation to query the length of the sequence. Consider a realization of this component that is layered on top of a Stack component. To be specific, suppose the representation consists of two stacks (call them beforeStack and afterStack) positioned so that the top elements of the two stacks "face each other" in the interior of the sequence they represent. The current value of pos is now the length of beforeStack. Additions (deletions) can now be performed by first adjusting the elements in the two stacks such that the length of beforeStack is equal to the desired pos value, and then pushing (popping) an element onto (from) beforeStack. Listing 4.4 shows the add method from this realization of SequenceSerf.

Clearly, this realization of SequenceSerf does not care which particular implementation of StackSerf is used. So we can design the SequenceSerf component to be a template that takes two parameters — a type parameter that stands for the kind of Item that the sequence holds, and a facility parameter that provides some implementation of StackSerf. In fact, the two stacks (beforeStack and afterStack) could come from separate Stack realizations, in which case, the Sequence would be parameterized by the item type, and two realizations of Stack. We deal with the simpler case here, since the problem we are trying to illustrate shows up even then. Under the Serf pattern, a client program will then create and set the template parameters of (*i.e.,* instantiate) a StackSerf object, and then pass that object as the facility parameter to the SequenceSerf instance. Listing 4.5 shows a snippet from a client program instantiating a SequenceSerf object.

There is an indirect restriction that this client has to follow — the Item parameters to the stack and sequence templates have to be the same. This restriction cannot be encoded by any syntactic means allowed by C# (or Java).

Listing 4.4: C# implementation of add for SequenceSerf realization layered on top of StackSerf

```
public void add(Sequence s, Data x, int pos)
{
    SequenceRep_R1 sRep = Rep(s);
    if (stackSerf.length(sRep.beforeStack) < pos)
    {
        while (stackSerf.length(sRep.beforeStack) < pos)
        {
            Data tempX = itemSerf.create();
            stackSerf.pop(sRep.afterStack, tempX);
            stackSerf.push(sRep.beforeStack, tempX);
        }
    }
    else
    {
        while (stackSerf.length(sRep.beforeStack) > pos)
        {
            Data tempX = itemSerf.create();
            stackSerf.pop(sRep.beforeStack, tempX);
            stackSerf.push(sRep.afterStack, tempX);
        }
    }
    stackSerf.push(sRep.afterStack, x);
}
```

Listing 4.5: A client instantiating a SequenceSerf

```
/* ... */
IntegerSerf intSerf = new IntegerSerf_Std();

StackSerf stackSerf = new StackSerf_R1();
stackSerf.ItemSerf = intSerf;

SequenceSerf seqSerf = new SequenceSerf_R1();
seqSerf.StackSerf = stackSerf;
seqSerf.ItemSerf = intSerf;
/* ... */
```

Now let us see how we can augment Serf interfaces with contract checking in order to enforce the proper use of Serfs as parameterized components. We will handle the two requirements, R1 and R2, separately. We enrich the specification of each component with additional pre- and post-conditions. There is a wide variety of specification languages we can choose from, and any language that supports parameterization constructs can be used. We use the eXtensible Markup Language (XML) [Harold and Means 2] to encode these specifications, so that we can emphasize their language-independent nature. Further, [Hallstrom and Soundarajan 2002] describes a tool that can use such embedded XML specifications to create program documentation. Listing 4.6 shows a partial specification of the StackSerf contract written in XML. We describe this specification in detail in the rest of this section.

Listing 4.6: Partial XML specification for StackSerf

```
<contract>
    <name>StackSerfContract</name>
        <parameter>
            <name>ItemSerf</name>
            <type>ServiceFacility</type>
            <defines>type Item</defines>
        </parameter>

        <interface>
            <operation>
                <name>create</name>
                <returntype>Data</returntype>

                <precondition>
                    ItemSerf != null
                </precondition>

                <postcondition>
                    create = EmptyStack
                </postcondition>
            </operation>


```

Continued

Listing 4.6 continued

```
        <operation>
            <name>push</name>

            <parameter>
                <name>s</name>
                <type>Stack</type>
                <mode>alters</mode>
            </parameter>

            <parameter>
                <name>x</name>
                <type>Data</type>
                <mode>consumes</mode>
            </parameter>

            <precondition>
                ItemSerf != null
            </precondition>

            <postcondition>
                ensures s = (#x) * s
            </postcondition>
        </operation>

        <operation>
            <name>pop</name>

            <parameter>
                <name>s</name>
                <type>Stack</type>
                <mode>alters</mode>
            </parameter>

            <parameter>
                <name>x</name>
                <type>Data</type>
                <mode>produces</mode>
            </parameter>

            <precondition>
                ItemSerf != null and
                | s | > 0
            </precondition>
```

Continued

Listing 4.6 continued

```
                <postcondition>
                    ensures #s = (x) * s
                </postcondition>
            </operation>

            <operation>
                <name>length</name>
                <returntype>int</returntype>

                <parameter>
                    <name>s</name>
                    <type>Stack</type>
                    <mode>preserves</mode>
                </parameter>

                <precondition>
                    ItemSerf != null
                </precondition>

                <postcondition>
                    ensures length = | s |
                </postcondition>
            </operation>
        </interface>
</contract>
```

R1. Enforcing Instantiation Before the Serf can be used to create data objects, we require that the Serf has been properly instantiated, *i.e.,* all the template parameters have been set. For each template parameter, the property that corresponds to that parameter must have been set to a value other than its initial value. In this case, we will just use the initial value conventions of Java/C# — for example, Object type variables are initialized to be null, and int variables are initialized to 0.

In order to make sure that by the time we use a Serf it is properly instantiated, we include a check to make sure that all the parameters have actually been set in the

pre-condition of each method, including the create() method (lines 15, 42, 66, and 88 in Listing 4.6). So, in accordance with design by contract, clients that want to use a Serf object have to first instantiate it by supplying appropriate actual parameters.

Currently we have built a mapping between the XML specification and assertion checks in C#. In the case of Java Serf components, the XML specification can be used to generate JML [JML n.d., Cheon 2003] annotations to specify the pre-conditions. We can then use the assertion generation and checking tools that are part of JML to generate and check the pre-conditions at run-time.

R2. Enforcing Restrictions In the foregoing discussion, we have presented one way of making sure that a Serf object is actually instantiated before it is used. However, how do we make sure that the parameters that have been supplied are appropriate from the type-correctness standpoint?

The solution we use is again to rely on design by contract. We embed the restrictions on parameters in the specification of the component. From these specifications, run-time checks can be generated that interrogate the incoming parameter to make sure that the restrictions on the parameter have in fact been met. Again, in the case of Java Serfs, we can use the JML Run time Assertion Checker (RAC) [Cheon 2003] to generate and check the assertions.

Each of the parameters to the Serf may be annotated with restrictions. For instance, in the StackExternalSorter example, we impose a restriction on the ItemSerf parameter that it implement the ISerializable and IComparable interfaces (Listing 4.7). This requirement, however, is stated in the XML contract, but not in the corresponding C# interface for reasons cited earlier in this section. Instead, the requirement is

Listing 4.7: ExternalSorter interface in C# with XML annotations

```
1  public interface StackExternalSorter : ServiceFacility
2  {
3      /// <contract>
4      ///     <parameter>
5      ///         <name> s </name>
6      ///         <type> Stack </type>
7      ///         <mode> alters </mode>
8      ///     </parameter>
9      ///
10     ///     <precondition>
11     ///         StackBase != null and
12     ///         ItemSerf != null
13     ///     </precondition>
14     ///
15     ///     <postcondition>
16     ///         s is permutation of #s  and
17     ///         IS_ORDERED (s)
18     ///     </postcondition>
19     /// </contract>
20     void Sort(Stack s);
21
22     // Template parameters
23
24     /// <restriction>
25     ///     implements ISerializable
26     /// </restriction>
27     StackSerf StackBase
28     {
29         get;
30         set;
31     }
32
33     /// <restriction>
34     ///     implements ISerializable
35     /// </restriction>
36     ///
37     /// <restriction>
38     ///     implements IComparable
39     /// </restriction>
40     ServiceFacility ItemSerf
41     {
42         get;
43         set;
44     }
45 }
```

Listing 4.8: The setItemSerf method in StackExternalSorter

```
1  private StackSerf stackBase;
2  public StackSerf StackBase
3  {
4      get
5      {
6          return stackBase;
7      }
8      set
9      {
10         if ( !(value is ISerializable &&
11                value is IComparable) )
12             throw new InvalidArgumentException(
13                 ''Argument is of incorrect type!'');
14         else
15             stackBase = value;
16     }
17 }
```

encoded as the aforementioned run-time checks. The setItemSerf method in StackExternalSorter will now have a precondition that the parameter it gets passed implements both the required interfaces.

This precondition can be checked during execution using the Reflection API in C# and Java. The setItemSerf method uses reflection to query the parameter it gets passed to see the list of interfaces that parameter object implements (Listing 4.8). If ISerializable and IComparable are not part of this list, the precondition check fails, and the instantiation does not complete successfully. This failure in instantiation is viewed as a failure to meet the contract, and an exception is thrown. The key gain here is that the error (in this case, the attempt to bind an incorrect parameter to a template) is actually caught at the time it occurs, and the client can be given a chance to correct it, by the exception handler routine.

Listing 4.9: Enforcing stability of parameters

```
1 /// <precondition>
2 ///     Parameter = null
3 ///     /* Remaining precondition */
4 /// </precondition>
5 void setParameter(...)
6 {
7     /* ... */
8 }
```

R3. Enforcing Stability of Parameters In addition to these two kinds of checks, there is one more check that is necessary in the case of dynamically bound parameterized components. This check ensures that the component satisfies the requirement that once a parameter has been set, it cannot be unset, or changed. This condition can be enforced by requiring that the set*Parameter* method for any Parameter can only be invoked once. We do this by including another clause in the pre-condition of every set*Parameter* method to check if the Parameter is null (Line 2 in Listing 4.9).

4.4 Instantiation-Checking Components

One drawback of the inline checks described in the end of the previous section is that they have to be executed every time the method is invoked. Even if the correctness of the Serf template, and its parameters has been established, these checks will be executed on every future method call. Such inline checks could soon become a performance burden on the entire system. If we could have some way by which the checks can be "turned off" once the correctness has been established, then this performance burden can be avoided.

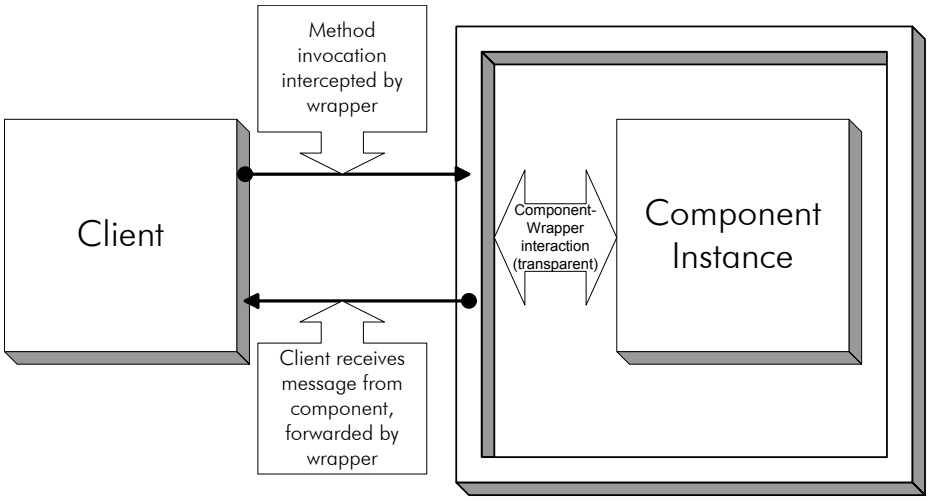

Figure 4.1: Instantiation-checking wrapper

The instantiation-checking code is completely a separate dimension of concern [Tarr et al. 1999] from the rest of the component code. We can separate these checks into a wrapper for the component. When a client program now makes a method invocation on the component, the wrapper first intercepts the call, performs all the checks, and if they all pass, forwards the method call to the component (Figure 4.1). This approach is the same as that used in [Edwards et al. 1998], except that the kind of assertion-checking done here is much simpler. Still, the performance burden remains as every method invocation has to traverse an extra level of indirection.

We do have the ability, however, to turn off the checking code. The Service Facility pattern supports dynamic module replacement (Chapter 5), and hence, once the client has passed the parameter binding phase, the instantiation-checking wrappers can be removed, thus removing the performance overhead.

4.5 Chapter Summary

Dynamic binding of parameters to components greatly increases the flexibility of software systems. But the flexibility does come at a cost — care must be taken to ensure that the bindings that take place at run time are indeed correct with respect to type safety. In this chapter, we have presented the various kinds of checks that need to be performed in order to ensure type-correct parameter bindings. We also presented a specification framework in which service facility components can be specified.

The specification framework uses the RESOLVE specification language as its basis, and extends the language to include the notion of dynamically bound parameterized components. We have shown a transformation of the modified RESOLVE specification into code written in Java/C#. The transformation first goes from the RESOLVE specification to a specification written in XML, and finally to Java/C# code. We have also shown how the checks to ensure instantiation can be abstracted out into separate instantiation-checking wrappers.

A preliminary version of the work described in this chapter was presented at the Workshop on Specification and Verification of Component-Based Systems at ESEC/FSE in Helsinki, Finland in September 2003 [Sridhar and Weide 2003].

CHAPTER 5

DYNAMIC RECONFIGURATION USING THE SERVICE FACILITY PATTERN

5.1 Introduction

In Chapter 4, we presented a specification framework and rules for using the Service Facility patter as a way of constructing type-safe, dynamically bound parameterized components. One of the conditions that we presented there was that each parameter to a component can only be set once, and after that has happened for the first time, the parameter cannot be unset, or changed (R3, Section 4.2.1). In this chapter, we relax this condition, and allow (some or all) parameters to components to be changed, if the execution environment so requires.

This relaxation is made in order to allow for dynamic reconfiguration of components. If a particular component supports reconfiguration, then the precondition for the set*Parameter* method is weakened to remove the clause Parameter = null. This is done so that the set*Parameter* can, in fact, be called more than once. The reconfiguration is done inside the set*Parameter* method. In this chapter, we present details of how a component can be made to support reconfiguration, and how such reconfiguration can be effected using the Service Facility design pattern.

5.2 Dynamic Reconfiguration

Dynamic reconfiguration refers to changing, updating, or otherwise modifying a system during execution. Dynamic reconfiguration is essential for supporting the operation and evolution of long-lived and highly-available systems for which it is either not possible or not economic to terminate execution in order to carry out invasive maintenance activities. Many naturally occurring systems exhibit some degree of flexibility with respect to change and adaptation, particularly when the consequences of *not* accommodating change are grave. An obvious example is the human body itself. When faced with a failing component, such as a kidney, it's a highly-desirable property that surgeons can replace the component without killing the host, as it were. Such is not the case with most software systems, where component substitution typically involves killing and re-deploying the entire application.

Dynamic reconfiguration is a material factor for any system that is subject to runtime *evolutionary* changes; that is, circumstances that cannot be anticipated or statically accommodated prior to system deployment [Kramer and Magee 1990]. For many online systems, such as banking applications, it is economically prohibitive to compromise availability for either planned or unplanned downtime. For distributed systems, like telecommunication switching networks, it may not be possible to coordinate downtime because the application spans over multiple administrative domains. In either case, robust systems must be designed to support modes of reconfiguration which enable applications to be changed on-the-fly.

5.3 Dynamic Module Replacement

The particular mode of dynamic reconfiguration that we are concerned about is known as *module replacement*. When applications are designed to depend on an object's interface rather than on its implementation, module replacement enables the implementation to be "hot swapped" at run-time without breaking the client code or side-affecting adjacent modules. Several approaches to achieving module replacement have been proposed in the literature, but most depend on non-enterprise research languages [Kramer and Magee 1985, Liskov 1988, Bloom and Day 1993, Hofmeister 1993], or make restrictive assumptions involving special-purpose middleware [Ben-Shaul, Holder and Lavva 2001, Malabarba, Pandey, Gragg, Barr and Barnes 2000]. These limiting factors on software deployment and portability render most existing approaches inapplicable to real software development.

Further, almost all the work in the area of dynamic module replacement has been concentrated on the domain of data container components [Bloom and Day 1993]. We extend the idea of module replacement into distributed protocols, and other algorithms recast as component modules. We show that the power of module replacement is only limited by our ability to effectively modularize our software.

Dynamic module replacement, however, is not the only way of effecting dynamic reconfiguration. This mode is best suited when the system being reconfigured can be properly modularized, and moreover, when the granularity of the desired reconfiguration can be at the module level. In the case of finer-grained reconfiguration, other modes of reconfiguration have been proposed, such as *interception* [Hallstrom et al. 2003]. Dynamic Link Libraries (DLLs) also provide a way of reconfiguration, where replacing a DLL with a different one that provides implementations for the

same set of interfaces provided by the old one allows for the new implementation to be used. In this chapter, however, we do not consider these other modes of dynamic reconfiguration. We only concern ourselves with module replacement.

The Service Facility pattern can be viewed as the next-generation of the well-known Abstract Factory pattern [Gamma et al. 1995], along with elements of the Proxy and Strategy patterns. Like abstract factories, Serfs are responsible for creating *product* objects for clients. Unlike abstract factories, however, Serfs only return a product reference to the client; the implementation of the product actually resides within the Serf itself, which then "services" the subsequent method invocations of the client. As such, a Serf can be viewed as a *wrapper* which introduces a level of indirection between the client and the product object. This enables the Serf to act as an *interceptor* capable of decoupling the abstract product interface (used by the client) from the actual product implementation (residing within the Serf). As a wrapper and an interceptor, a Serf can insulate clients from module replacement by delaying intercepted method invocations while reconfiguring the product class encapsulated within the wrapper. Thus, Serfs provide a clean **synchronization mechanism** that can be used during updates and configuration changes during system execution.

5.3.1 Conditions for Dynamic Module Replacement

In a running system, what is required to permit substitution of the implementation of a module without stopping system execution? We present here the conditions required for realizing dynamic reconfiguration. Each condition presented here translates to a step in the corresponding operational outline of the reconfiguration process.

1. **Initiation:** Module replacement must be initiated, either internally by the module itself, or externally by a third-party.

2. **Module Integrity:** The consistency of modules undergoing replacement must be preserved. Typically this can be achieved by restricting or regulating how client invocations can (or cannot) be interleaved with module replacement activities.

3. **Module Rebinding** The new (target) replacement module must be dynamically loaded and linked into the runtime environment, so that new object instances can be created to replace their old counterparts.

4. **State Migration:** The abstract state of each old object instance must be transmitted to the new counterpart object instance.

5. **Instance Rebinding:** Each client-side object handle must be redirected to its new object instance counterpart. This involves rebinding old object handles to new instances of the target replacement module. Old object instances should be finalized.

Previous research has documented other considerations which must be addressed for dynamic reconfiguration. Examples include protection, security, and the consistency of module substitutions. Such issues — pertaining to the safety and semantic correctness of module replacement — have been explored elsewhere in the literature [Kramer and Magee 1985, Hofmeister and Purtilo 1993, Endler 1992]. Any reliable reconfiguration strategy must address these questions, but ultimately a sufficient infrastructure for achieving module replacement is required. The focus of this

discussion is to supply such an infrastructure for accomplishing reconfiguration in enterprise-scale languages.

5.4 Dynamic Module Replacement using Service Facilities

Now let us see how Serfs actually satisfy the five conditions presented in Section 5.3.1. We note that languages supporting reflection provide clean solutions to the *Initiation* and *Module rebinding* steps. *State migration* can be realized indirectly by the Serf, or directly by the original and target modules themselves. The material support offered by Serfs pertains primarily to managing *Module integrity* and *Instance rebinding*.

Initiation. In the case of internal initiation, this is simple — the component triggers reconfiguration in response to a condition in its environment, such as a fault, performance degradation, etc. Internal initiation is essentially *planned* change, and so has limited scope; it is generally not possible to predict all changes that might occur in the environment.

As an example of internal initiation, consider a Matrix component that uses another component MatrixMultiplier to perform matrix multiplication. Suppose that there are two implementations of MatrixMultiplier — SparseMatrixMultiplier, to perform multiplication on sparse matrices, and DenseMatrixMultiplier to perform multiplication on dense matrices. The Matrix component may be initialized with the SparseMatrixMultiplier, and as it becomes denser, the component may decide to reconfigure itself to use the DenseMatrixMultiplier. Note that this may require a change in the data representation as well. In this case, the entire reconfiguration is fully planned at the time of system design. This is what we mean by internal initiation.

Responding to *unplanned* changes requires support for external initiation by a third party. The system being reconfigured has not support built into it to make decisions for when reconfiguration should take place. The initiation is usually done by an external process which detects the need for reconfiguration, or by a human administrator.

In environments that support reflection, such as Java and .NET, initiation involves an outside agent being able to invoke a reconfiguration method on the Serf. For example, rebinding a template parameter can be initiated by invoking set*Parameter* and specifying an appropriate replacement module. In the case of environments that do not support reflection, external initiation can be achieved using DLLs, socket communications, or even the file I/O system in primitive circumstances.

In order to further clarify this distinction between internal and external initiation, consider the *Windows Update* service in the Microsoft Windows XP operating system. This service can be set up to automatically find updates to the operating system, download and install them to reconfigure parts of the operating system. What kind of reconfiguration initiation is this? By our definition of internal and external initiation, the *Windows Update* system performs external initiation of reconfiguration. The system being reconfigured here is the part of the operating system that needs an upgrade; this component is not designed for reconfiguration. The *Windows Update* program is the third party that detects that this component can be upgraded, and then proceeds to reconfigure the component.

Module Integrity. Access control is required when interleaved client invocations can disrupt module replacement or compromise the integrity of the object instances themselves. Since the Serf wraps the module implementation, it serves as a "gateway"

which can choose to defer method calls while effecting module substitution. This creates a local synchronization mechanism that may delay servicing a client invocation during reconfiguration. After completing the reconfiguration, deferred invocations are delegated to the new target object instances.

Not all method invocations need to be deferred, however. Only those method invocations that will be affected by the reconfiguration process need to be deferred. More importantly, the method invocations that may have a critical impact on the reconfiguration must not be deferred. Since the Serf can observe *all* messages going through it, the decision of which messages to allow, and which to defer can be made in the Serf. This decision is dependent on the particular kind of system that is reconfigured, and must be specified by the system designer.

Module Rebinding. The initiation step provides the Serf with information about the new module implementation. In the case of environments that support dynamic class loading, such as Java and .NET, the initiating agent can supply the name of the new implementation directly. The Serf can then use reflection to locate and load the new class, and then create instances of the new module type. In the case of .NET applications, the initiator can simply point the Serf to the source code of the new implementation, which the Serf can then compile, load, and create instances. We note that reflection simplifies this step for the application programmer. As with the initiation step, however, languages without reflection or dynamic class loading can resort to cruder alternatives as substitutes.

State Migration. Migration amounts to recording each object's abstract state, and then reconstructing a "next generation" object with an equivalent abstract value from the target module. This can be achieved using two complementary operations:

one for externalizing an object's concrete state into the abstract value it represents, and one for internalizing an abstract value into an object by constructing a concrete state that represents it. Object instances can support their own state migration by implementing such value-transmission methods directly [Herlihy and Liskov 1982]. Alternatively, these methods can be realized indirectly by the Serf itself, provided that the original and target modules support observability and controllability of their abstract states [Weide, Edwards, Heym and Long 1994].

Instance Rebinding. This is the fundamental step in getting dynamic module replacement to work. The primary task is to decouple the dependency between the module interface (known to the client), and its runtime objects (which realize the interface). Parameterization mechanisms such as C++ templates can decouple this dependency at design-time, but the problem of module replacement requires decoupling this concrete dependency *at run-time*. Essentially, each client-side object handle held by the client must be rebound to the new object instance. As an interceptor, the Serf can simply redirect method invocations to the appropriate instance of the new module. Binding changes happen *within* the Serf, and are thus invisible to the client. This can be achieved by maintaining a map from logical client handles to object instances. Once the rebinding has been effected, all future calls to the module will be delegated to the new module instance.

Figures 5.1 and 5.2 illustrate instance rebinding in a service facility. Each of the client handles maps to a particular object that resides in the Serf. When module replacement is done, the client handles are modified to point to a different location — the next-generation objects. This is done *within* the Serf and so the client has no knowledge that such a change is being made.

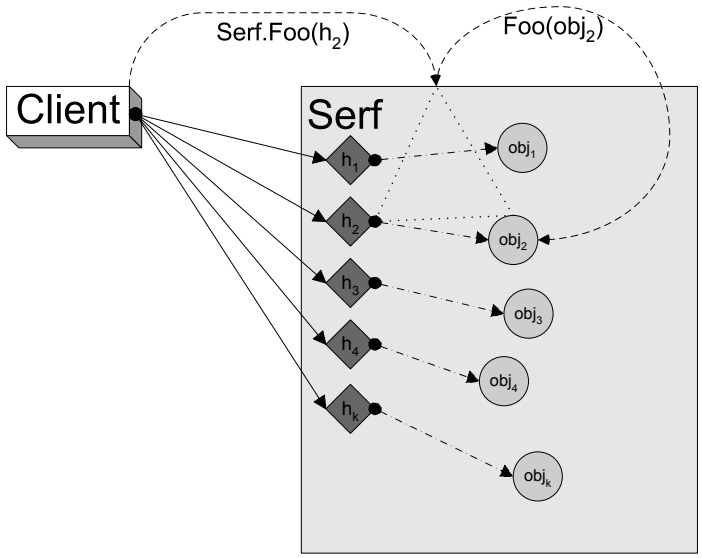

Figure 5.1: Serf redirecting call to client handle. The client makes the invocation to the handle h_1, the Serf dispatches the call to the object o_1.

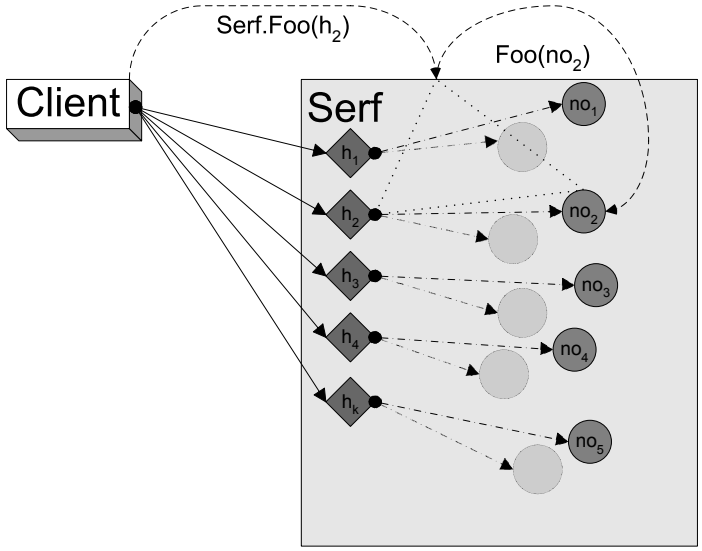

Figure 5.2: After instance rebinding, the old objects are destroyed, and the client handles now point to the new object instances. A client invocation on h_1 is now dispatched to the new object instance no_1.

Discussion. Both Java and C# both support reflection, thus enabling dynamic module replacement using the Service Facility pattern. Further, since all .NET languages compile to the same intermediate language (MSIL), all features available in C# are available in all .NET languages [Microsoft 2002b], thus making the Serf approach uniform across all .NET languages. We have outlined how Serfs provide a flexible, language-neutral infrastructure that can be used in mainstream, production platforms that support reflection to provide an easy approach to dynamic module replacement.

5.4.1 Allowing Reconfiguration

The component that is being reconfigured should also have some way of notifying the environment that it does support reconfiguration. We can do this by creating an empty interface called IReconfigurable, and requiring that every component that supports reconfiguration to implement this interface. At runtime then, the administrator (or whichever third party is initiating the reconfiguration) checks using reflection if the component implements IReconfigurable or not. The initiator then proceeds to perform the reconfiguration if the component supports it. Listing 5.1 shows the Java source code required to make this check in the initiator.

This information (that a component supports reconfiguration) does not really belong in the interface of the component. In fact, the interface (the set of operations) that the component exports is exactly the same whether or not it supports reconfiguration. In particular, the set*Parameter* methods look exactly the same syntactically; the only difference is in their behavioral precondition. This information belongs in

Listing 5.1: Java source in the initiator to check if a component supports dynamic reconfiguration

```
/* ... */
Class cls = componentSerf.getType();
Class interfaces[] = cls.getInterfaces();
bool supportsReconfig = false;
for (int i = 0; i < interfaces.length; i++)
{
    if (interfaces[i].getName() == ''IReconfigurable'')
        supportsReconfig = true;
}
if (supportsReconfig)
{
    componentSerf.setParameter(...);
}
/* ... */
```

the *meta-data* — data *about* the component. C# (and other .NET languages) provides a nice construct to include such meta-data in a component, without introducing spurious inheritance hierarchies.

A C# class (or interface) can be tagged with *attributes*. These attributes make up the meta-data of the class. Further, programmers can define new attributes that carry meaning along with the class that the client can use. In this case, we can define an attribute called *Reconfigurable*. We then tag every class that supports reconfiguration with this attribute. Again, the initiator of reconfiguration checks to see if the Serf that needs to be reconfigured is tagged with the Reconfigurable attribute, and proceeds only if the class is tagged. Listing 5.2 shows the Reconfigurable attribute, and the ComponentSerf_R1 class being tagged with the attribute. The class is tagged simply by putting the name of the attribute in brackets on the line before the class definition. Notice that although in the definition of the attribute we call it ReconfigurableAttribute

Listing 5.2: The Reconfigurable attribute and the ComponentSerf_R1 class tagged with the attribute

```
1 public class ReconfigurableAttribute : Attribute
2 { }
3
4 [Reconfigurable]
5 public class ComponentSerf_R1 : ComponentSerf
6 {
7     /* ... */
8 }
```

we can simply refer to it as Reconfigurable; C# recognizes this shorthand through type inference [Archer 2001]. Listing 5.3 shows the C# code in the initiator that performs this check and starts the reconfiguration.

5.5 Case Study: Mutual Exclusion

In this section, we trace through the steps required for dynamic module replacement outlined in 5.3.1. We use the resource allocation example from Chapter 3. The example illustrates the replacement of the actual conflict resolution protocol (algorithm) used to determine which client in the network currently has access to the resource.

Initiation. As we have seen earlier, in an environment that supports reflection, external initiation can be done quite elegantly. A console program can prepare a message and pass it to ResourceSerf to initiate the reconfiguration. In this particular case of protocol replacement, the message that needs to be prepared and sent is an invocation to the setProtocolSerf method (Listing 5.4). As a parameter to this method call, the initiator will pass a ProtocolSerf that implements the required conflict resolution protocol. ResourceSerf can then reconfigure itself to use this new protocol

Listing 5.3: C# code to check if the Serf supports reconfiguration

```
/* ... */
Type type = typeof(componentSerf);
boolean supportsReconfig = false;
foreach (Attribute attr in
    type.GetCustomAttributes(true)
{
    ReconfigurableAttribute reconAttr =
        attr as ReconfigurableAttribute;
    if (reconAttr != null)
    {
        supportsReconfig = true;
    }
}
if (supportsReconfig)
{
    componentSerf.setParameter(...);
}
/* ... */
```

instead of the old one. The entire reconfiguration is handled inside this method, as detailed in the subsequent steps below.

Module Integrity. What happens if the request for reconfiguration arrives at a time when the resource is held by one of the clients in the network? What happens to the messages, if any, that are in transit at the current time? When is it safe to perform the reconfiguration, while ensuring that there are no interference effects?

In order to ensure safety, the ResourceSerf needs to make sure that the resource is not being held by any of the clients in the network. To achieve this, the Serf makes itself one of the clients of the resource. We require that the ProtocolSerf always guarantees that the ResourceSerf can, in fact, become one of the clients of the resource. For example, if the ProtocolSerf implementation has a limit n on the number of client allowed, then it always reserves one slot for the ResourceSerf and

allows only $n - 1$ clients. ResourceSerf thus gets issued its own proxy, and requests access to the resource on its own. When the ResourceSerf does get access to the resource, no other client process can be holding the resource. This is guaranteed by the safety specification of the conflict resolution protocol. Therefore, changing the protocol at this time would not affect any of the clients. In fact, the clients can be completely unaware of this change with respect to the correctness of the ResourceSerf in eventually allocating the resource to it.

Lines 10 through 14 in Listing 5.4 show the Serf creating proxies for itself in the old and new ProtocolSerf modules. Lines 17 through 24 show the Serf acquiring exclusive access to the resource under both the old and the new protocols. The ResourceSerf acquires the resource in the old protocol in order to bring the old protocol to a state of quiescence. It does the same with the new protocol to delay the new ProtocolSerf to start accepting requests until after all the client proxies have been transferred over to the new protocol.

Figures 5.3 and 5.4 show ResourceSerf participating in the conflict resolution. Originally the conflict graph consists of the proxies of each client in the network. Each edge in the conflict graph represents a contention for the resource. For each edge in the conflict graph, only one of the nodes it is incident upon can have access to the resource. Since this is a fully connected conflict graph, only one node in the entire graph can have access to the resource, thus satisfying the safety requirement of mutual exclusion. The Serf's proxy is added to the conflict graph, and this new node shares an edge with each of the other proxies, thus maintaining the fully-connected property of the conflict graph.

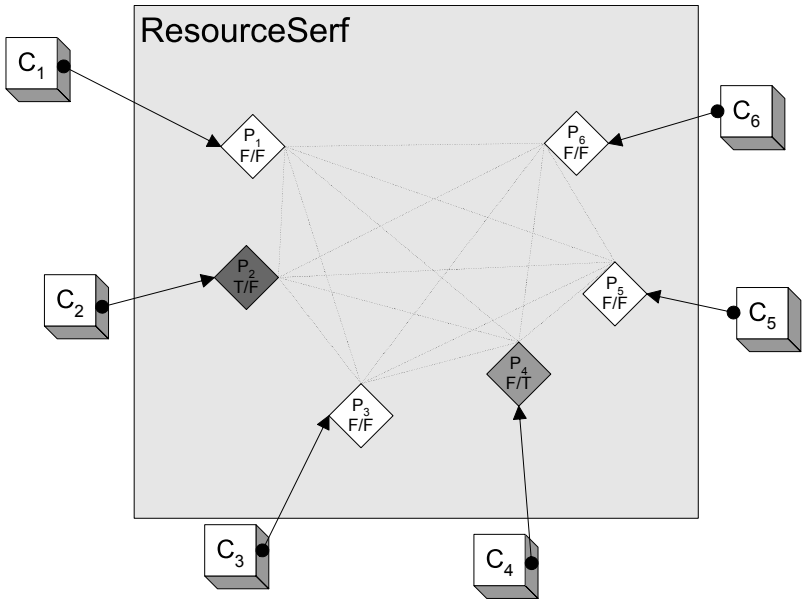

Figure 5.3: Conflict graph before reconfiguration

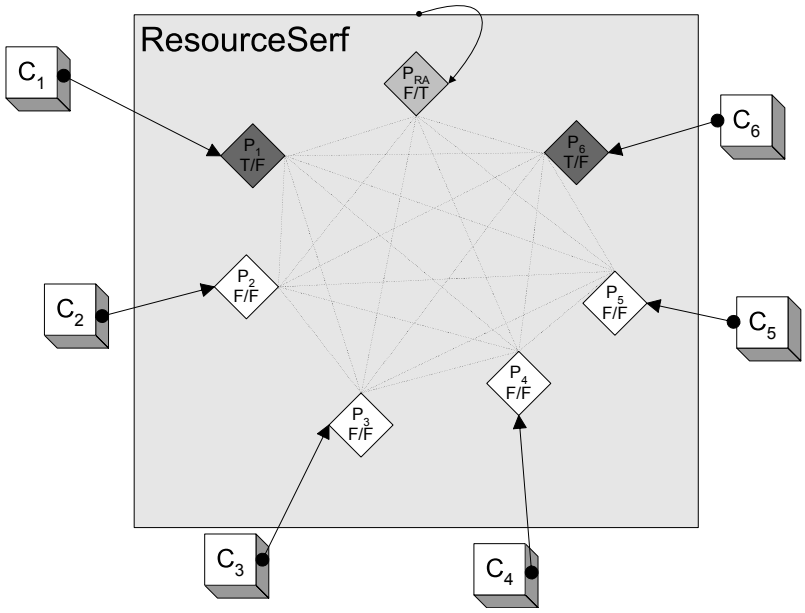

Figure 5.4: Conflict graph during reconfiguration — includes the proxy representing the Serf.

Module Rebinding. The module that implements the new conflict resolution protocol is created as part of the initiation step by the external initiator program. This new module is passed into the Serf through the method invocation. This step involves binding the new ProtocolSerf module to the ResourceSerf. One of the following two scenarios could be true when the setProtocolSerF (Listing 5.4) method is invoked:

1. ResourceSerf is a service facility that is being instantiated for the first time. This case is trivial, and is handled in lines 3 through 6 in the method.

2. ResourceSerf is being reconfigured, and there are live proxies in the network. This is the interesting case. Before this new module can be bound in line 45, the state migration and instance rebinding steps must be completed. Once they have been completed, then the rebinding itself can be done.

State Migration. State migration in this example consists of taking a checkpoint of the abstract states of each of the proxies under the old protocol module, and "rolling forward" the new protocol to this checkpoint. In this example, the ResourceSerf directly takes this checkpoint, by individually querying each proxy instance for its abstract state, and setting a new proxy instance to the same state. The abstract state of each proxy is a pair of booleans, one of which (available) is the same for all proxies — **false**. This is so because the resource is currently with the ResourceSerf. The ResourceSerf, thus transfers state by simply calling isRequested() on each of the current-generation proxy instances, and for each proxy for which requested is true, the ResourceSerf calls request() on the corresponding next-generation proxy. The abstract state of the entire conflict resolution layer has been migrated to the next-generation protocol. Lines 37 through 40 in setProtocolSerf provide this behavior.

Instance Rebinding. Acting as an interceptor between the client and its proxy in the mutual exclusion layer, ResourceSerf has a chance to observe, and if necessary, hold method invocations that the client makes on its proxy. How exactly this is achieved is as follows. The client only holds a *logical* reference to the proxy. The actual reference is stored inside this handle, accessible only by the ResourceSerf. In addition, the ResourceSerf maintains a map from the logical references to actual proxy instances. So there is a decoupling of the dependency between the logical handle that the client holds, and the actual protocol implementation. To complete the reconfiguration, ResourceSerf simply updates each client's resource proxy such that the handles now point to the ProtocolProxy instances instead of the old ones. This is done in lines 26 through 41 in Listing 5.4. From this point on, there is no further action necessary to ensure that method invocations arrive at the new conflict resolution module.

Listing 5.4: The setProtocolSerf method in ResourcerSerf

```
1  public void setProtocolSerf(ProtocolSerf psf)
2  {
3      if (protSerf == null) // First time
4      {
5          protSerf = psf;
6      }
7
8      else  // reconfiguration
9      {
10         ProtocolSerF newProtSerf = psf;
11         ProtocolProxy serfProxy =
12             (ProtocolProxy) protSerf.create();
13         ProtocolProxy newSerfProxy =
14             (ProtocolProxy) newProtSerf.create();
15
16         // Acquire resource under old protocol
17         protSerf.request(ref serfProxy);
18         /* Wait on separate thread until resource is acquired */
19
20         // Acquire resource under new protocol
21         newProtSerf.request(ref newSerfProxy);
22                                                    Continued
```

Listing 5.4 continued

```
        /* Wait on separate thread until resource is acquired */

        foreach (Resource res in clientList)
        {
            ResourceRep_R1 resRep = Rep(res);
            Resource tempR = (Resource) initialValue();
            ProtocolProxy tempPP =
                (ProtocolProxy) protSerf.create();
            ProtocolProxy newTempPP =
                (ProtocolProxy) newProtSerf.create();

            tempPP = resRep.protocolProxy;
            resRep.protocolProxy = newTempPP;
            if (protSerf.isRequested(tempPP))
            {
                newProtSerf.request(ref newTempPP);
            }
        }

        // Now that all the proxies have been transferred,
        // replace the protocol serf.
        protSerf = newProtSerf;

        // Release the resource under the new protocol
        protSerf.release(ref newSerfProxy);
    }
}
```

5.5.1 Performance Considerations

The Module integrity step of the above reconfiguration guarantees that the safety specification of resource allocation is not violated. The client may, however, experience higher response time during the period when the reconfiguration is taking place. The slower performance is due to the fact that while the reconfiguration is going on, the ResourceSerf is holding on to the resource. The liveness specification of resource allocation is not violated, however, since the ResourceSerf will release the resource once the reconfiguration is over.

There may be a further degradation is the response time due to multiple reconfigurations happening in a particular time period. Suppose that a phase of reconfiguration is started from generation G_1 to G_2, and m clients were waiting for the resource in G_1. Corresponding requests are made on behalf of these m clients in G_2. But let us suppose that before all of these m clients have been serviced in G_2, a new reconfiguration request arrives at the ResourceSerf to migrate to a new generation G_3. The clients who were transferred from G_1 to G_2, but did not get serviced before the reconfiguration started may again get transferred. If such reconfiguration requests keep arriving at a rate faster than all clients in a generation have been serviced, then the response time from the point of view of these clients may be very long, and there is even a possibility of the liveness specification getting violated.

This situation can be avoided by adopting one of two strategies. The first is that in the event that a second reconfiguration request arrives at the ResourceSerf while there are still clients who were transferred from the previous generation, the new reconfiguration request is deferred until all of these clients have been serviced. This way, no client will be made to wait across more than one reconfiguration process. However, such postponement of reconfiguration may not always be possible. In such cases, the second strategy can be adopted: the clients that have already been through one reconfiguration could first be transferred into the new generation, and the protocol started. Then, once all of these clients have been serviced (by the new ProtocolSerf module), the remaining clients are transferred over. This way again, clients will only need to wait through one reconfiguration at the most before they get access to the resource.

5.6 Distributed Service Facilities

In the discussion so far, we have extended the idea of dynamic module replacement outside the domain of data collection components using the example of resource allocation in distributed systems. We consider two different families of protocols for solving mutual exclusion. A representative for the first family is a simple centralized queue-based protocol, where client requests are queued in a simple queue, and the client at the head of the queue gets access to the resource. In this case, we are dealing with a centralized solution to a distributed problem. All clients deal with a single ResourceSerf. This model is presented in Figure 5.5. This solution has its obvious drawbacks in performance penalties owing to no concurrency at all. A representative for the second family of protocols employs a distributed token ring [Lynch 1996] for resolving conflicts. This is a distributed solution that allows changes in topology at run-time. However, in the case of its service facility implementation, the module that controls client access to this protocol is still centralized (Figure 5.6).

Both the above approaches leave more to be desired. Even though the second case encapsulates a distributed protocol implementation, the fact that it is behind a centralized gateway makes it prone to problems — single point of failure, performance bottleneck, etc. The truly distributed solution, where a set of distributed clients are represented by a set of distributed proxies in the protocol, and moreover, the gateway itself is distributed, is clearly the desired solution. Such a solution can tolerate faults, and does not involve a single bottleneck. A sketch of this solution is illustrated in Figure 5.7.

Serfs clearly have two separate functions — object creation, and object maintenance. In order for the distributed implementation to be possible, we separate these

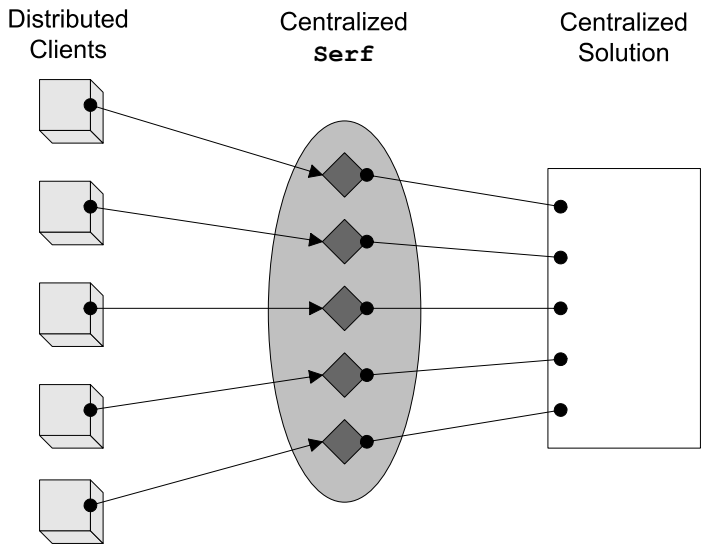

Figure 5.5: Distributed clients using a centralized solution

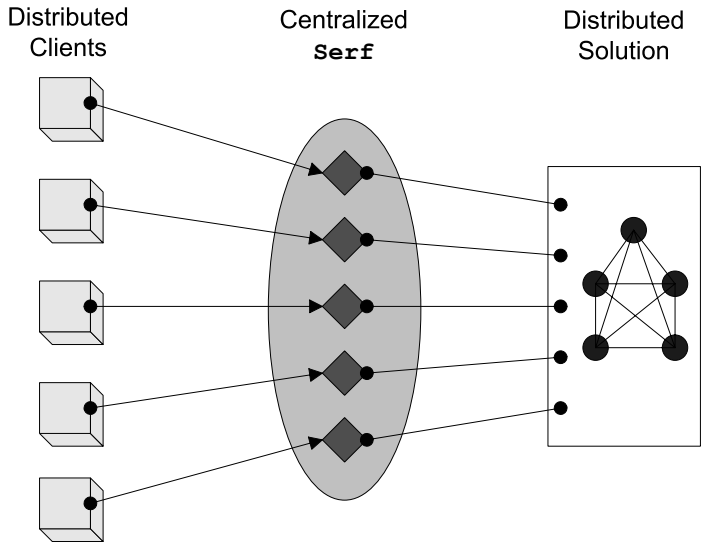

Figure 5.6: Distributed clients using a distributed solution but through a centralized gateway

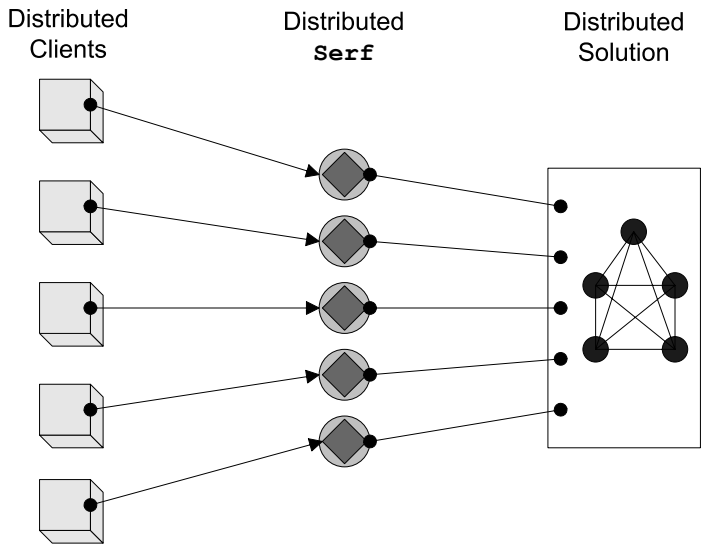

Figure 5.7: Distributed clients using a distributed solution

two functions. The main Serf still handles object creation; and rather than creating a logical handle that does not have any functionality on its own, creates a Serflet wrapper (Section 3.4.5) that encapsulates the actual product instance. The object maintenance function is now downloaded into the Serflet. In addition, the Serf manages the Serflets in a connected topology thereby enabling coordination among the Serflets, for instance, in the event of dynamic reconfiguration.

5.7 Chapter Summary

In this chapter, we have presented the use of the Service Facility design pattern for building systems that support dynamic reconfiguration. The particular mode of dynamic reconfiguration that we consider is known as *dynamic module replacement*. When using this mode of reconfiguration, it is required that the system be designed

as a collection of modules that encapsulate different aspects of functionality. Reconfiguration in such a system then can be effected by way of replacing the particular module that corresponds to the part of the system that needs change.

We outlined the conditions that must be satisfied for dynamic module replacement to be effected, as well as an operational view of the steps in the reconfiguration process that satisfy these conditions. These steps involved in dynamic module replacement are: Initiation, Module Integrity, Module Rebinding, State Migration, and Instance Rebinding. We also showed how the Service Facility pattern provides support to this dynamic module replacement process. We also demonstrated the use of the Service Facility pattern for dynamic reconfiguration by way of an example in which a resource allocation component is reconfigured to use different conflict resolution algorithms.

The work described in this chapter was presented at the 23^{rd} International Conference on Distributed Computing Systems in May 2003 [Sridhar, Pike and Weide 2003].

CHAPTER 6

RELATED WORK

In this chapter, we outline some of the research in the literature that is related to the problems that this dissertation addresses. We examine related work in the two main areas that the contributions of this dissertation lie in — parameterized programming, and dynamic reconfiguration. Since the solutions presented in this dissertation are based in design patterns, we include a treatment of other design pattern research in each of these two areas, along with a comparison with our work.

6.1 Parameterized Programming

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, Winkler, Meseguer, Futatsugi and Jouannaud 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 language for specifying parameterized components, which includes a formal semantics for these components as well as support for restricted parameters.

6.1.1 Language Support

Several programming languages have included support for parameterized programming over the years. We briefly outline the most popular among these.

Ada Generics

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 1996, Booch 1987, Musser and Stepanov 1989].

Genericity in Ada can be introduced at various levels:

- A whole package can be parameterized by the types it uses. An example is a generic Set package that exports a generic Set abstract data type, and associated

operations. The package is parameterized by the kind of items that the Set holds.

- A procedure or a function can be defined to be a generic, where the type(s) of its parameter(s) (and return type, in the case of functions) are left open to be fixed later during instantiation.

Parameters to a generic unit in Ada can be specified to be restricted. For instance, if we were designing a generic function FindMax that is parameterized by a type ValueType, and in the function we need to compare two items of ValueType, the parameter would have a restriction on it, saying that it has to support comparison.

```
1 GENERIC
2     TYPE ValueType IS PRIVATE;
3     WITH FUNCTION Compare(L, R: ValueType) RETURN Boolean;
4
5 FUNCTION FindMax(L, R: ValueType) RETURN Boolean;
```

At the time of instantiation, the client, along with a type for ValueType also supplies an operation to match Compare.

```
1 FUNCTION IntegerFindMax IS
2     NEW FindMax (ValueType=>Integer, Compare=">");
```

The operation that we supply to the generic to match the Compare operation is only checked for structural consistency. The behavior of the operation itself cannot be tested at compile time. This instantiation creates a new function definition that takes two Integer variables and returns the bigger one.

C++ Templates

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.

```
1  template <class T>
2  class Stack
3  {
4  private:
5      int size;
6      T* stackPtr ;
7  public:
8      int push(T&);
9      int pop(T&);
10     int length();
11 }
12
13 template <class T>
14 void Swap(T& left, T& right);
```

Templates can be used in conjunction with inheritance to provide an alternative to using pure inheritance as a way of extending functionality. Consider the following example, where we construct a Print extension to Stack.

```
1  template <class Stack_Base>
2  class Stack_Print : Stack_Base
3  {
4  public:
5      void print();
6  }
```

In this case, the Stack_Print template can be instantiated with any implementation of Stack. The Print operation is implemented as a *layered extension*, by making calls to the primary methods (push, pop, and length) in the Stack_Base class. Note that even though the Stack_Print template inherits from the Stack_Base class, it may not use the data members directly since they may be declared private in the Stack_Base class.

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. For example, in the Stack_Print example, if the print method makes invocations to push, pop, and length on the template parameter, but the client tried to instantiate it with a class that does not provide these methods, the instantiation will not succeed.

C# Generics

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 2003, Richter 2002] will include support for generics [Hejlsberg 2003, Microsoft 2003, Kennedy and Syme 2001]. 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. For instance, the following listing shows a Dictionary class that is parameterized

by a key type K and a value type V, and there is a constraint K that it implements the IComparable interface.

```
class Dictionary<K, V> where K: IComparable
{
    public void Add(K key, V value)
    {
        /* ... */
        if (key.CompareTo(x) == 0) { /* ... */ }
        /* ... */
    }
}
```

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.

6.1.2 OO Frameworks

Object-oriented frameworks [Bosch, Molin, Mattsson and Bengtsson 2000] are another approach that has been used to build parameterized software. Frameworks are a technique for object-oriented code structuring. An object-oriented framework [Fayad and Schmidt 1997] provides a set of classes that collaborate in a precise manner to provide a common architectural framework on which a family of similar products can be built. The common collaborative structure is captured in the form of key methods, referred to as *template methods* in the design patterns literature [Gamma et al. 1995], that direct the ow-of-control, and call the appropriate *hook methods* of various classes. Hook method implementations are deferred to derived-class designers, who provide implementations appropriate to the individual applications.

Since frameworks include abstract classes, they are not executable by themselves. A framework is specialized to create an executable framework instance by providing implementations for hooks, in concrete subclasses of the abstract classes. Frameworks therefore provide a clean approach to building parameterized components in languages that do not provide constructs to support them. The commonalities are implemented in the framework, while the variabilities are left open as hooks for the individual instantiations to fill in. One problem with using OO frameworks to build parameterized components, however, is with respect to reasoning about them. Reasoning about enriched behavior in OO frameworks is inherently difficult, since at the time of developing the framework, there is no way to predict the different ways in which the hooks can be specialized in the individual framework instances [Soundarajan 2000].

6.2 Dynamic Reconfiguration

Dynamic reconfiguration has been well studied, and a variety of solutions to the problem of updating software systems dynamically have been proposed in the literature. A majority of these solutions are either housed in specialized languages or architectures that were made with the intent of dynamically reconfigurable systems; or come with specialized modifications to existing languages.

6.2.1 Specialized Languages and Architectures

DYMOS: Dynamic Modification System

The Dynamic Modification System (DYMOS) proposed by Lee is one of the first system that supported dynamic software update [Lee 1983]. DYMOS is designed for *Mod (pronounced "StarMod") [Cook 1980], a multi-threaded variant of Modula [Wirth 1980]. DYMOS permits changes to module definitions at various levels

— type, function, and data definitions. Loop bodies can be modified at run time as well, providing a way of infinite loops to be updated so they will terminate.

The DYMOS system performs updates coded in a separate language that can be used to specify which modules are to be updated, along with conditions of when these updates are to be performed. These commands are given to a command interpreter that is housed in the runtime environment; the system also includes a source code management system and a *Mod compiler. The runtime environment then takes input from the command interpreter, makes the necessary changes to the source code, runs the new modules through the compiler, and injects the new modules into the system. The DYMOS system has complete knowledge of the entire application, as well as any updates that are performed. This knowledge is exploited in ensuring that all updates that are applied to the application are, in fact, correct.

Dynamic Reconfiguration in Conic

The Conic system [Kramer and Magee 1985, Magee, Kramer and Sloman 1989, Kramer and Magee 1990] provides a programming language which is based on Pascal, a configuration language and a distributed operating system for the construction of dynamically configurable distributed systems. Each process in a distributed program in Conic runs on a physically separate node; the configuration of the system is specified separately from the application code. Reconfiguration in Conic is effected by changing the configuration program associated with the application, and feeding this to the operating system. The operating system then can perform the necessary changes. Conic does not, however, allow the code inside of an application program to be modified during execution.

Dynamic Module Replacement in Argus

The Argus language [Liskov 1988] supports building distributed programs that can be dynamically reconfigured. The Argus language is based on the CLU programming language [Liskov 1981, Liskov 1993, Liskov, Snyder, Atkinson and Schaffert 1990]. The system uses the concept of *guardians*, which are like UNIX processes that can communicate among themselves using a RPC-like interface. Guardians are also capable of being restarted if they crash. Each guardian contains within it data objects that store its state. These data objects, in accordance with the information hiding principle, are only accessible through invocations of operations (called *handlers* on the guardian. Invocations on handlers are passed around as messages between processes.

Guardians are *resilient* to failures. A part of the guardian's state is marked as *stable*, and this part of the state is saved across crashes. The rest of the state is volatile, and will be lost as a result of a crash. The guardian's state can, however, be restored through a special recovery process that initializes the volatile objects using the data in the stable objects. The design of the guardian should be in such a way that all the volatile objects can be derived from the stable objects. A distributed program in Argus consists of a number of guardians, residing in distributed locations, or nodes. The connections between the guardians in a system can be rerouted dynamically providing for a system that allows changes during execution.

Bloom proposed a system that supports dynamic module replacement in Argus [Bloom 1983, Bloom and Day 1993]. The guardian's code can be replaced even while it is running. A collection of guardians, called a *subsystem*, can be replaced simultaneously. This allows for flexibility and type-safety, since a guardian G may be used by others, and replacing G may also involve replacing the others that depend on

G. Before a guardian (or guardians in a subsystem) can be replaced, it is forced into a *quiescent* state so that no new transactions occur while the replacement is taking place. Once the reconfiguration is complete, the lookup calls are exchanged among the guardians, so that the new guardians can be recognized.

Day proposed another system of reconfiguration using the crash recovery features of Argus [Day 1987]. To effect the replacement of a guardian, the old state of the guardian is encoded and stored. The old guardian is then crashed. A new guardian is then started, decodes the old guardian's state, and restores itself to a consistent state in the system. The new guardians are discovered through new lookup calls, similar to Bloom's system.

The state transmission methods [Herlihy and Liskov 1982] that Argus and its associated dynamic reconfiguration systems use are the biggest contribution that influenced our work on dynamic module replacement using service facilities. These state transmission methods do come at a performance burden, but guarantee consistency in system state.

The PolyLith Software Bus

PolyLith [Hofmeister 1993, Purtilo 1990] is a distributed programming system that supports reconfiguration. The system provides for structural, geometric and modular reconfiguration in systems. Similar to Conic and Argus, distributed systems built on PolyLith run on a collection of processes, and all communication between processes is defined by special libraries using the PolyLith bus.

Since the PolyLith bus controls all communication in the application that is being reconfigured, it can manage the exact time of reconfiguration by controlling the order and timing of the delivery of messages between processes. The bus can

also work with the application module in capturing the abstract state of the module across a reconfiguration step. The methodology used here is similar to Argus, in that the system uses the value transmission method over abstract data types proposed by [Herlihy and Liskov 1982]. The PolyLith system, like our system, expects the module that is being replaced to provide operations to externalize its state to an abstract value, and to internalize the abstract value to construct its new state.

The system also allows structural changes to the application. Processes can be added or removed during execution. This is done in a fashion similar to the one used in Conic — the configuration changes are written as events in a configuration language, and the PolyLith bus uses these events to effect the change.

Dynamic Software Updating using Popcorn and TAL

Hicks presents a system of updating functionality in a running system [Hicks 2001]. Hicks' system uses Popcorn [Morrisett, Crary, Glew, Grossman, Samuels, Smith, Walker, Weirich and Zdancewic 1999], a C-like language that includes some enhancements such as more flexible variable declarations, and a C++-like namespace system. The language also improves on C by disallowing pointer arithmetic and pointer casts. Popcorn also includes exceptions and parametric polymorphism.

Dynamic configuration changes are implemented as *patches* in Typed Assembly Language (TAL) [Morrisett, Walker, Crary and Glew 1999]. TAL extends assembly language with typing rules and annotations, and memory management primitives. The dynamic patches that Hicks' system uses is based on the idea of *verifiable native code* [Necula 1997]. The use of verifiable code increases the robustness of both the running program that is being updated, as well as the dynamic patches.

DRSS: Dynamic Reconfiguration Subsystem

The Dynamic Reconfiguration Subsystem (DRSS, pronounced 'Doris') [Hallstrom et al. 2003] is an open container architecture (i.e. extensible hosting environment) similar to those used to host Enterprise Java Beans [SunMicrosystems 2001]. The DRSS container provides dynamic reconfiguration services to its hosted instances, without the components having been explicitly designed to support dynamic reconfiguration. The runtime reconfiguration services provided by the platform include the ability to dynamically add, remove, and replace component implementations (similar to the work presented in [Rodrigues, Castro and Liskov 2001, Sridhar et al. 2003]), as well as the ability to dynamically deploy *cross-cutting* modifications, while managing the scope of their effect.

The DRSS architecture is loosely based on the *Interceptor* design pattern [Schmidt, Stal, Rohnert and Buschmann 2000]. At the core of the DRSS architecture lies a set of *interceptor* chains that monitor message flow in and out of the component that needs to be reconfigured. Every message flowing into the component goes through the *receive chain* of interceptors. Each of these interceptors could modify the message, or may perform additional tasks. Once the message has been handled by the component, outgoing messages flow through the *send chain* of interceptors. The behavior of the component is altered at run time by choosing the set of interceptors to apply in the send and receive chains. The interceptor chains support dynamic addition/deletion, allowing components hosted in DRSS to be modified on the fly.

6.2.2 Language/Architecture Enhancements

HADAS

The HADAS system [Ben-Shaul et al. 2001] allows the construction of software components in Java that can be reconfigured at runtime. Components in HADAS are composed of multiple parts that encapsulate different aspects of the component such as memory management, operations, data, etc. Reconfiguring a component would involve changing one of these parts that make up the entire component. This approach is, however, tied intimately with the HADAS runtime environment and is not programming-language neutral.

Dynamic C++ Classes

Dynamic C++ classes [Hjálmtýsson and Gray 1998] extend C++ using a library approach where C++ classes can be altered at runtime. They allow for new functionality to be introduced into an executing program without sacrificing type safety or performance. This work also uses an extra level of indirection to allow for replacement. The client program holds a reference to a proxy (which is an instance of a new class called Dynamic introduced in the system), which holds references to multiple versions of the same class. Constructor calls from the client are always sent to the most recent version, while other method invocations are sent to the class that created the object.

This is similar to our approach, except that existing instances of a class that has been replaced cannot be changed. The model requires holding on to the old class as long as the last instance from that class is still alive. This may not be possible doing some useful program restructuring activities.

Dynamic Java

Dynamic Java [Andersson, Comstedt and Ritzau 1998] is an extension to Java that has been proposed to allow for dynamic updates to be performed on Java systems. The system, like with the dynamic C++ classes, allows multiple versions of the same class to coexist. The system modifies the class java.lang.Class in the Java class library, and adds a new data field substitute to this class. The substitute field contains a reference to the next available version of the class, if any. Unlike the case with dynamic C++ classes, existing instances of the old version of the class are converted to instances of the new class. However, all the instances are not transferred over at the time the new class is loaded. Instead, the cost of transfer is amortized over the instances by adopting a lazy approach to the transfer. After the new class has been loaded, the first time an instance is accessed through a method invocation, it is first transferred and then the method is executed. This system also depends on the class (the old and new versions) to provide the methods to transmit state.

Dynamic Java Virtual Machine

Malabarba et al. have proposed an extension to the Java virtual machine to support dynamic reconfiguration [Malabarba et al. 2000]. In their system, they extend the virtual machine by creating a new class loader to replace Java's default class loader [Liang and Bracha 1998, Qian, Goldberg and Coglio 2000]. The system includes a new class loader called DynamicClassLoader which extends the regular ClassLoader class that is included in Java. Rather than using the Java ClassLoader to load and resolve references to classes in Java programs, the authors use the new DynamicClassLoader. Any class loaded by this class loader is automatically a *dynamic class* — it

can be modified at run time. The DynamicClassLoader class includes methods called reloadClass and replaceClass that can be used by an administrator to perform dynamic reconfiguration.

In this work, Malabarba et al. also include details about type-safety issues of when a class can be updated, and replaced with a new one. The system also handles the different object update models that the Dynamic Java system from [Andersson et al. 1998] considers — version barrier, global update, passive partitioning, and active partitioning.

CHAPTER 7

CONCLUSIONS

7.1 Summary of Contributions

This dissertation set out to defend the following thesis:

1. Parameterized programming provides a strong basis for building reusable software components, and delaying the binding of parameters to a component until run time increases the flexibility of these components.

2. A methodology for dynamic parameterization can be built as a design pattern that places minimal restrictions on the target programming language.

3. Software systems built from dynamically bound parameterized components can adapt to changes in the environment during system execution, and need not require re-starting the system.

In defense of parts (1) and (2), we presented the Service Facility pattern in Chapter 3 as a way of building dynamically bound parameterized components. In Chapter 4, we presented ways to ensure that the dynamic binding of parameters to parameterized components built using the Service Facility pattern is type-correct. In defense

of part (3), we showed in Chapter 5 how the components built using the Service Facility pattern can be dynamically reconfigured through module replacement.

An important point that we laid out in the beginning of this dissertation was that the ideas presented here should be immediately accessible to the software development community. We have stayed true to this goal throughout this research program. All of the contributions are housed in technologies that have been widely accepted, and used by developers. We have not made any specialized modifications to programming languages or environments. Instead, we have describe a stylized, disciplined way of programming using available technologies, such as Java, .NET, and XML, that can be used by any developer who works with these technologies.

7.2 Future Work

7.2.1 Component Containers

Component containers [SunMicrosystems 2001] provide a model of component-based software engineering where the component designer deals only with the specific functionality that the component must export. Once the functional code has been implemented, the component is dropped inside the container, which encapsulates a variety of orthogonal, non-functional services depending on the environment, and the requirements of the component. In this context, a container is an extensible environment that provides runtime support to the objects it hosts. To be clear, the traditional object-oriented programming literature [Meyer 1988] uses the term container to refer to classes that serve as collections of objects (*e.g.,* lists, stacks, queues). We refer to these classes as collection classes, reserving the term container to refer to an extensible object hosting environment. Similar to an operating system

hosting processes, a container hosts objects, transparently providing services to the objects it hosts. That is, hosted objects are imbued with additional services just by virtue of executing within the container, with little or no additional programming effort on the part of the class designer. Containers provide a model of object-oriented software development that supports a clean separation of concerns between core object functionality, and system-level peripheral services.

As such, containers are a nice way of imposing architecture for the components, in the form of how the components need to be built and how they need to be composed. The abstraction of the container in which a particular component resides is the abstract view of the component when considering composition. So when reasoning about compositional behavior, it is important to consider each component in the system in conjunction with the container it is placed in. When considering composition analysis of a component-based system built using a container technology, two kinds of container–component interaction need to be considered:

1. Container functionality is seen as added functionality (*e.g.,* exporting the component as a web service)

2. Each operation to the component is augmented with container services (*e.g.,* logging, pre-condition checking, etc.)

The problem with this view of containers comes when we try to reason about composition. The regular theories of composition do not account for the changes in the interface of the component that we are trying to reason about — and this is what happens when using containers. Under various circumstances, we may not be able

to predict how the components in the system interact with one another, or how a particular container influences the behavior of a component inside it.

Containers as Parameterized Components

In our model, we do not make a distinction between components and containers. Instead, we view all components in the system alike. Depending on the particular part of the system that a particular component is in, it could provide different kinds of services in addition to the functionality that the component designer writes. These extra services could, in fact, be written by the component designer himself, but that would go against the whole philosophy of component-based software engineering, which is to increase reuse.

One observation to be made is that the kinds of services that we are dealing with here are concerns that are potentially (in fact, usually) completely orthogonal to the actual functionality of the component. For instance, a WebSearch component may also want to make use of a logging service. Such a logging service can be developed completely independent of the specific component it is being used in. In fact, the logging code looks exactly the same regardless of what the functionality of the component is.

These are the kinds of services that can be weaved in as *aspects* [Kiczales et al. 1997]. While aspect-oriented programming can be used for a variety of situations, and for providing a spectrum of services, there are limitations. For instance, the code weaving is done on source code, before deployment. This has several problems:

1. Source code instrumentation generally makes it much harder to prove correctness of programs;

2. No changes to these services can be made once the program has started executing.

3. Often, a developer does not have access to every component's source code.

Another approach to solving this same problem is to use parameterized components, where each of these services, such as logging, load balancing, etc., is implemented in its own module, and these modules are bound as parameters to the component that we are building. The component can take a varying number of template parameters(using the Service Facility pattern), and thus provide a varying number and kinds of services in different instantiations. One very important consideration with such parameterized components that could drastically affect their usefulness is the time of binding parameters to the template. If the implementation mechanism used is compile-time binding as with C++ templates, we are faced with a similar problem as with AOP — the services are statically bound, and no changes are possible after instantiation. The other problem that we saw with AOP, however, goes away — since reasoning about the behavior of parameterized components does not involve dealing with source code instrumentation.

The Service Facility pattern is aimed at building scalable, yet verifiable component-based systems. The pattern also provides a clean way to construct extensible parameterized components. We plan to explore the construction of extensible containers using the Service Facility pattern. This work is ongoing research, and some preliminary results are presented in [Sridhar and Hallstrom 2003].

7.2.2 Towards a Component Model

Modularity [Baldwin and Clark 2000] and changeability [Parnas 1979] in software systems have been studied by many researchers in academia over the years. Further, reusable software components have been recognized as an effective way to realize these principles in software systems.

Several different definitions and accompanying theories of software components have been proposed. While these definitions may disagree on specifics, most of them agree on a few general characteristics components should possess. The most important, and widely accepted, of these characteristics are:

- Components are reusable, and support local certification (modular verification) of component properties;
- Components support predictable assembly — properties of a system composed of smaller components can be derived (predicted) from the properties of the constituent components;
- Components are units of change, and hence support reconfiguration

Although several such theories exist [Sitaraman and Weide 1994, Szyperski 1997, Szyperski 2003], there is no model of development that the industry can use to develop components with all these characteristics. The purported software component industry of today only comes close to some of these characteristics; it is constrained by the development model into which these theories are shoe-horned — object-oriented languages such as Java, C++, and C#.

In order to be able to realize the full potential of component-based software, we need to move beyond these object-oriented languages. We need to build new languages

in which true component-based systems can be engineered. Researchers must focus on bringing the languages that they use as research vehicles into the mainstream so developers can actually use them in building real, enterprise-scale systems.

However, just coming up with a new language is not going to be enough. The development community today is too entrenched in using languages such as Java to readily accept such a change. While the majority of the research energy should be invested in developing the "science of design" for good software components, a strong emphasis also should be placed on easing the transition into the new paradigm. In the absence of such a comfortable transition, these new tools may never see the light of day in the software development industry.

In order to accommodate the development community's reluctance to move away from object-oriented languages, the aforementioned transition must be housed in these very languages. The Service Facility pattern offers a way for components built in these languages to have the properties listed above. One of the future directions for work on the Service Facility pattern is to develop it into a component model that developers can use as a crutch to shift incrementally from the object-oriented programming paradigm into the component-oriented programming paradigm.

APPENDIX A

UNIFIED MODELING LANGUAGE NOTATION

A.1 Introduction

The Unified Modeling Language (UML) [Booch et al. 1998, Jacobson, Booch and Rumbaugh 1999] is a standardized language used for specifying, visualizing, constructing, and documenting artifacts of software systems. UML is used at various phases of the software development life cycle to develop and communicate models of the software system. UML static structure diagrams are used early on in the system design phase to establish the different components of the system, as well as the static relationships between them. The UML also includes other model languages for the other phases of the development life cycle, but these other languages are outside of the scope of this document. We only present the static structure notation that is useful in understanding the UML diagrams that we use to describe design patterns in this dissertation.

A.2 Static Structure Notation

A.2.1 Interface

An *interface* (abstract component) is denoted by using a rectangular box separated into two rows (Figure A.1). The top row contains the name of the interface, preceded by the stereotype <<interface>>. The bottom row contains the list of methods that this interface exports. The names of the methods are written in *italic* font to denote that these are just method signatures, and that the interface does not contain any implementations for the methods.

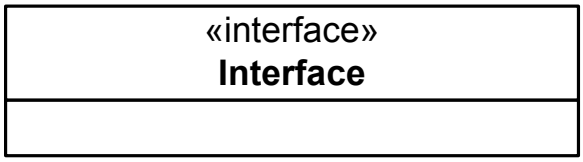

Figure A.1: A UML interface

A.2.2 Class

A *class* (concrete component) is denoted using a rectangular box (similar to the interface box) with three rows (Figure A.2). The top row, like in the interface box contains the name of the class. The middle row contains the list of data members that the class contains. The bottom row contains the list of methods included in this class.

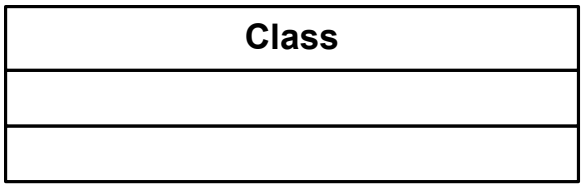

Figure A.2: A UML class

A.3 Relationships

A.3.1 Uses

The *uses* relationship is used whenever one class depends on another class in some way. The relationship is denoted by drawing a solid line with an open arrow at the end (Figure A.3).

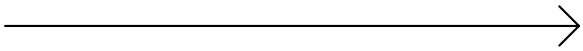

Figure A.3: The *uses* arrow

When drawn between two classes, it means that ClassA makes a reference to ClassB in its code, and the dependency is a concrete-to-concrete (class-to-class) dependency (Figure A.4). The name of ClassB actually occurs in the code of ClassA.

The uses relation can also be established between a class ClassA and an interface Interface (Figure A.5). When used in this manner, ClassA's code does not name any particular implementation of Interface, but can use any implementation of Interface. This is a concrete-to-abstract dependency.

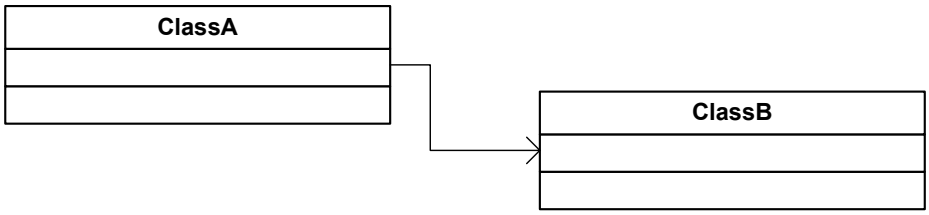

Figure A.4: The *uses* relation

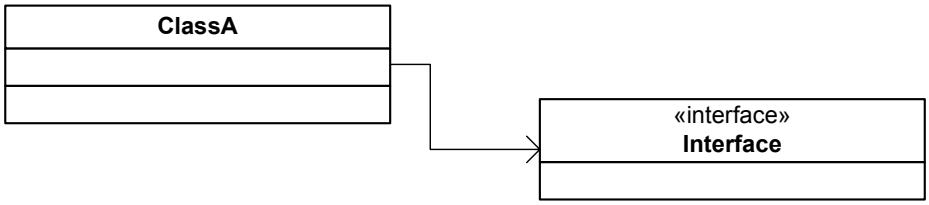

Figure A.5: The *uses* interface between a class and an interface

A.3.2 Extends

The *extends* relationship in UML is used to denote inheritance. *Extends* is a binary relationship that exists between two entities of the same kind (both classes, or both interfaces). The relationship is shown by drawing a solid line with a closed, but unshaded arrowhead (Figure A.6).

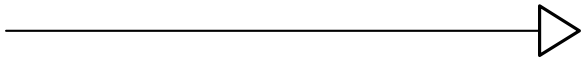

Figure A.6: The *extends* arrow

When a class ClassB *extends* another class ClassA, the *derived class* ClassB inherits all the data members, as well as the methods that are included in the *base class* ClassA (Figure A.7). Interface inheritance works the same way, except that only the signatures of the methods are inherited, since interfaces do not contain implementations for the methods.

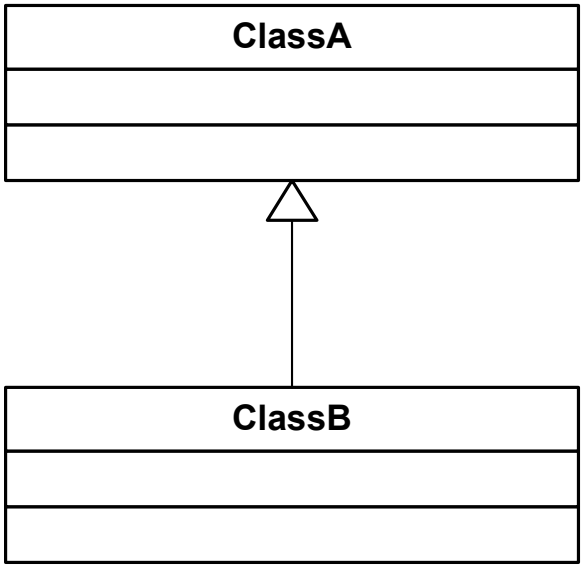

Figure A.7: The *extends* relation

A.3.3 Implements

A class *implements* an interface when it provides a realization for all the method signatures in the interface. The relationship is shown by drawing a dashed line with a closed, but unshaded arrowhead (Figure A.8).

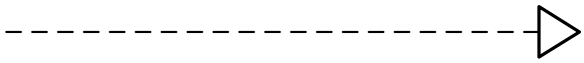

Figure A.8: The *implements* arrow

For a class Class to *implement* an interface Interface, the Class must provide implementations for all the methods in the Interface (Figure **??**). If any methods are left out, this is not a legal relationship.

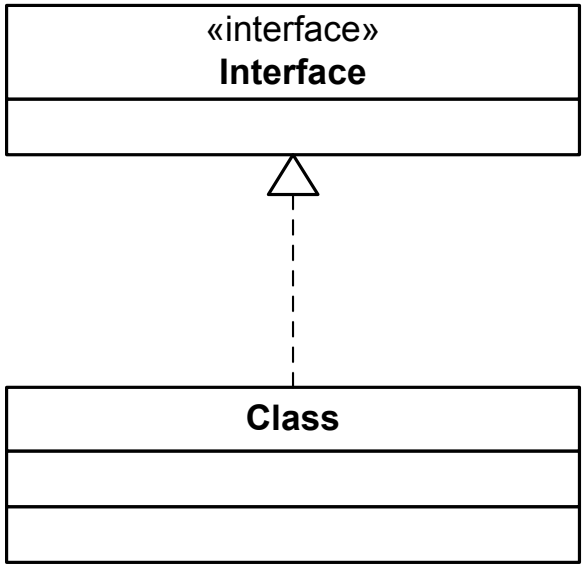

Figure A.9: The *implements* relation

A.3.4 Instantiates

The *instantiates* relation is used when objects of one class create instances of another class. The relationship is shown by drawing a dashed line with an open arrowhead (Figure A.10).

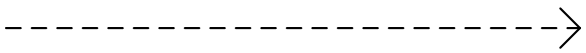

Figure A.10: The *instantiates* arrow

When this relation exists between two classes ClassA and ClassB, the code in ClassA includes a call to the constructor of ClassB (Figure A.11). Since in most object-oriented languages the name of the constructor is the same as the name of the class, this is usually a concrete-to-concrete relationship.

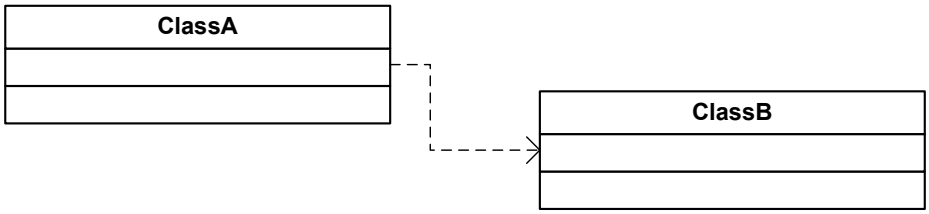

Figure A.11: The *instantiates* relation

A.3.5 Composition

The *composition* relationship is used to denote that one class includes as part of its state an instance of another class. The relation is shown by drawing a solid

line between the two classes, with a diamond on the side of the class that owns the instance (Figure A.12).

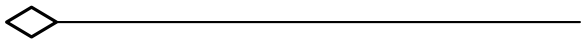

Figure A.12: The *composition* arrow

One of ClassA's data members is an instance of ClassB. When drawn between two classes, this relationship is a concrete-to-concrete relationship because again, the name of ClassB occurs in ClassA's code (Figure A.13).

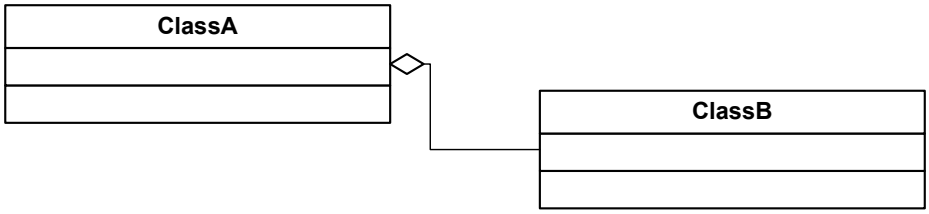

Figure A.13: The *composition* relation

The relationship can also be shown between a class ClassA and an interface Interface; this shows a concrete-to-abstract relation (Figure A.14). Only the name of the Interface occurs in the code of ClassA. An instance of any implementation of Interface is handed to the ClassA instance.

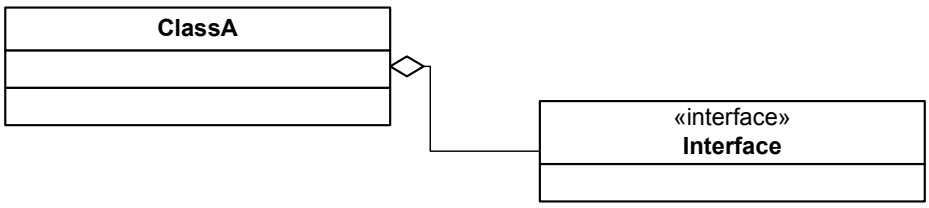

Figure A.14: The *composition* relation between a class and an interface

BIBLIOGRAPHY

Alexander, C.: 1977, *A Pattern Language*, Oxford University Press, New York, NY.

Alexander, C.: 1979, *The Timeless Way of Building*, Vol. 1 of *Center for Environmental Structure Series*, Oxford University Press, New York, NY.

Andersson, J., Comstedt, M. and Ritzau, T.: 1998, Run-time support for dynamic java architectures, *Proceedings of the ECOOP'98 Workshop on Object-Oriented Software Architectures*, Brussels. Available as Technical report 13/98 University of Karlskrona/Ronneby.

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

Baldwin, C. and Clark, K.: 2000, *Design Rules: The Power of Modularity*, Vol. 1, MIT Press, Cambridge, MA.

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

Barnett, M. and Schulte, W.: 2001, The ABCs of specification: AsmL, Behavior, and Components, *Informatica* **25**(4).

Batory, D., Singhal, V., Thomas, J., Dasari, S., Geraci, B. and Sirkin, M.: 1994, The GenVoca model of software-system generators, *IEEE Software* **11**(5), 89–94.

Baumgartner, G., Läufer, K. and Russo, V.: 1996, On the interaction of object-oriented design patterns and programming languages, *Technical Report CSD-TR-96-020, Department of Computer Science, Purdue University* .

Beck, K. and Cunningham, W.: 1987, Using pattern languages for object-oriented programs, *OOPSLA'87: Object-Oriented Programming, Systems, Languages, and Applications Conference, Proceedings*.

Ben-Shaul, I., Holder, O. and Lavva, B.: 2001, Dynamic adaptation and deployment of distributed components in HADAS, *IEEE Transactions on Software Engineering* **27**(9), 769–787.

Bloom, T.: 1983, *Dynamic Module Replacement in a Distributed Programming System*, PhD thesis, Laboratory for Computer Science, The Massachussets Institute of Technology, Cambridge MA.

Bloom, T. and Day, M.: 1993, Reconfiguration and module replacement in argus: Theory and practice, *IEE Software Engineering Journal* **8**(2), 102–108.

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

Booch, G., Rumbaugh, J. and Jacobson, I.: 1998, *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA.

Bosch, J., Molin, P., Mattsson, M. and Bengtsson, P.: 2000, Object-oriented framework-based software development: problems and experiences, *ACM Computing Surveys* **32**(1es), 3.

Bracha, G., Odersky, M., Stoutamire, D. and Wadler, P.: 1998, Making the future safe for the past: Adding genericity to the java programming language, *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, ACM Press, pp. 183–200.

Brooks, F. P.: 1975, *The Mythical Man-Month*, Addison-Wesley Publishing Co., Reading, Mass.

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

Büchi, M. and Weck, W.: 2000, Generic wrappers, *Proceedings of the European Conference on Object-Oriented Programming 2000*, number 1850 in *Lecture Notes in Computer Science*, Springer-Verlag, pp. 201–225.

Buxton, W.: 2001, Less is more (more or less), *in* P. J. Denning (ed.), *The Invisible Future: the seamless integration of technology into everyday life*, McGraw Hill, pp. 145–179.

Cardelli, L.: 1984, A semantics of multiple inheritance, *in* G. Kahn, D. MacQueen and G. Plotkin (eds), *Semantics of Data Types, International Symposium, Sophia-Antipolis, France*, Springer-Verlag, pp. 51–67. LNCS 173.

Castagna, G.: 1995, Covariance and contravariance: Conflict without a cause, *ACM Transactions on Programming Languages and Systems* **17**(3), 431–447.

Cheon, Y.: 2003, A runtime assertion checker for the java modeling language, *Technical Report TR #03-10*, Department of Computer Science, Iowa State University.

Cook, R. P.: 1980, *Mod — a language for distributed computing, *IEEE Transactions on Software Engineering* **6**(6), 563–571.

Day, M.: 1987, *Replication and reconfiguration in a distributed mail repository*, Master's thesis, Laboratory for Computer Science, The Massachussets Institute of Technology, Cambridge MA.

Dijkstra, E. W.: 1972, Notes on structured programming, *in* O. Dahl, E. Dijkstra and C. Hoare (eds), *Structured Programming*, number 8 in *A.P.I.C. Studies in Data Processing*, Academic Press, chapter 1, pp. 1–82.

Edwards, S. H., Heym, W. D., Long, T. J., Sitaraman, M. and Weide, B. W.: 1994, Specifying Components in RESOLVE, *Software Engineering Notes* **19**(4), 29–39.

Edwards, S. H., Shakir, G., Sitaraman, M., Weide, B. W. and Hollingsworth, J.: 1998, A framework for detecting interface violations in component-based software, *Proceedings: Fifth International Conference on Software Reuse*, pp. 46–55.

Endler, M.: 1992, Support for consistency-preserving dynamic reconfigurations in distributed systems, *Proceedings of the 3rd Workshop on Future Trends of Distributed Computing Systems*, IEEE Computer Society Press, Los Alimitos, California, pp. 185–91.

Fayad, M. and Schmidt, D.: 1997, Object-oriented application frameworks, *Communications of the ACM, Special Issue on Object-Oriented Application Frameworks* **40**(10).

Feldman, M. B.: 1996, *Software Construction and Data Structures with ADA 95*, Addison-Wesley Longman Publishing Co., Inc.

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

Gibbs, W. W.: 1994, Software's chronic crisis, *Scientific American (International Edition)* pp. 72–81.

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

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

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

Hallstrom, J. O., Leal, W. M. and Arora, A.: 2003, Scalable evolution of highly-available systems, *IEICE/IEEE Joint Special Issue on Assurance Systems and Networks* . (to appear).

Hallstrom, J. O. and Soundarajan, N.: 2002, Incremental development using object-oriented frameworks: A case study, *Journal of Object Technology* **1**(3).

Harold, E. R. and Means, W. S.: 2, *XML in a Nutshell*, 2nd edn, O'Reilly & Associates.

Harrison, W. and Ossher, H.: 1993, Subject-oriented programming: a critique of pure objects, *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, ACM Press, pp. 411–428.

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

Herlihy, M. P. and Liskov, B.: 1982, A value transmission method for abstract data types, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **4**(4), 527–551.

Hicks, M.: 2001, *Dynamic Software Updating*, PhD thesis, Department of Computer and Information Science, University of Pennsylvania.

Hjálmtýsson, G. and Gray, R.: 1998, Dynamic C++ classes, *Proceedings of the USENIX 1998 Annual Technical Conference*, USENIX Association, Berkeley, USA, pp. 65–76.

Hofmeister, C. and Purtilo, J. M.: 1993, Dynamic reconfiguration in distributed systems: Adapting software modules for replacement, *International Conference on Distributed Computing Systems*, pp. 101–110.

Hofmeister, C. R.: 1993, *Dynamic Reconfiguration of Distributed Applications*, PhD thesis, Department of Computer Science, University of Maryland.

Jacobson, I., Booch, G. and Rumbaugh, J.: 1999, *The Unified Software Development Process*, Addison-Wesley.

JML: n.d., The java modeling language home page, URL:http://www.jmlspecs.org.

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

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: 1997, Aspect-oriented programming, *in* M. Akşit and S. Matsuoka (eds), *Proceedings European Conference on Object-Oriented Programming*, Vol. 1241, Springer-Verlag, Berlin, Heidelberg, and New York, pp. 220–242.

Kramer, J. and Magee, J.: 1985, Dynamic configuration for distributed systems, *IEEE Transactions on Software Engineering* **SE-11**(4), 424–436.

Kramer, J. and Magee, J.: 1990, The evolving philosophers problem: Dynamic change management, *IEEE Transactions on Software Engineering* **16**(11), 1293–1306.

Lamport, L. and Lynch, N.: 1990, Distributed computing: Models and methods, *in* J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Vol. B, Elsevier Science Publishers, chapter 18, pp. 1157–1199.

Lee, I.: 1983, *DYMOS: A Dynamic Modification System*, PhD thesis, Department of Computer Science, University of Wisconsin, Madison, WI.

Liang, S. and Bracha, G.: 1998, Dynamic class loading in the java virtual machine, *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM Press, pp. 36–44.

Liskov, B.: 1981, *CLU reference manual*, Vol. 114 of *Lecture Notes in Computer Science*, Springer-Verlag Inc., New York, NY, USA.

Liskov, B.: 1988, Distributed programming in argus, *CACM* **31**(3), 300–312.

Liskov, B.: 1993, A history of CLU, *ACM SIGPLAN Notices* **28**(3), 133–147.

Liskov, B., Snyder, A., Atkinson, R. and Schaffert, C.: 1990, Abstraction mechanisms in CLU, *in* S. B. Zdonik and D. Maier (eds), *Readings in Object-Oriented Database Systems*, Kaufmann, San Mateo, CA, pp. 47–58.

Lynch, N.: 1996, *Distributed Algorithms*, Morgan Kaufmann Publishers, San Francisco, California.

Magee, J., Kramer, J. and Sloman, M.: 1989, Constructing distributed systems in conic, *IEEE Transactions on Software Engineering* **15**(6), 663–675.

Malabarba, S., Pandey, R., Gragg, J., Barr, E. and Barnes, J. F.: 2000, Runtime support for type-safe dynamic java classes, *Proceedings of the 14th European Conference on Object-Oriented Programming*, pp. 337–361.

Meyer, B.: 1988, *Object-Oriented Software Construction*, Prentice-Hall.

Meyer, B.: 1992, *Design by contract*, Prentice Hall, chapter 1.

Meyer, B.: 2001, Overloading vs. object technology, *Journal of Object Oriented Programming* .

Microsoft: 2002a, *Microsoft Visual C# .NET Language Reference*, Microsoft Press, Redmond, Washington.

Microsoft: 2002b, Vs live conference.

Microsoft: 2003, Professional developers conference, Microsoft Corporation, Los Angeles, CA.

Moore, G. E.: 2000, Cramming more components onto integrated circuits, *Readings in computer architecture*, Morgan Kaufmann Publishers Inc., pp. 56–59.

Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S. and Zdancewic, S.: 1999, TALx86: A realistic typed assembly language, *Second Workshop on Compiler Support for System Software*, Atlanta, GA.

Morrisett, G., Walker, D., Crary, K. and Glew, N.: 1999, From system F to typed assembly language, *ACM Transactions on Programming Languages and Systems* **21**(3), 527–568.

Musser, D. R. and Stepanov, A. A.: 1989, *The Ada Generic Library linear list processing packages*, Springer-Verlag New York, Inc.

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

Necula, G. C.: 1997, Proof-carrying code, *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM Press, pp. 106–119.

Neilsen, M. L. and Mizuno, M.: 1991, A dag-based algorithm for distributed mutual exclusion, *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS)*, IEEE Computer Society, Washington, DC, pp. 354–360.

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

Parnas, D. L.: 1979, Designing software for ease of extension and contraction, *IEEE Transactions on Software Engineering* **SE-5**(2), 128–138.

Platt, D. S.: 2003, *Introducing Microsoft .NET*, 3 edn, Microsoft Press.

Purtilo, J. M.: 1990, The PolyLith software bus, *Technical Report University of Maryland Institute for Advanced Computer Studies Report No. UMIACS-TR-90-65*, University of Maryland at College Park.

Qian, Z., Goldberg, A. and Coglio, A.: 2000, A formal specification of java class loading, *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM Press, pp. 325–336.

Raymond, K.: 1989, A tree-based algorithm for distributed mutual exclusion, *ACM TOCS* **7**(1), 61–77.

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

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

Rodrigues, R., Castro, M. and Liskov, B.: 2001, BASE: Using abstraction to improve fault tolerance, *Proceedings of the 18th ACM Symposium on Operating System Principles*, Banff, Canada, pp. 15–28.

Schmidt, D., Stal, M., Rohnert, H. and Buschmann, F.: 2000, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, Vol. 2 of *Software Design Patterns*, John Wiley & Sons Ltd", West Sussex, England.

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

Sitaraman, M. and Weide, B. W.: 1994, Special feature: Component-based software using RESOLVE, *ACM SIGSOFT Software Engineering Notes* **19**(4), 21–67.

Sitaraman, M., Weide, B. W., Long, T. J. and Ogden, W. F.: 2000, A data abstraction alternative to data structure/algorithm modularization, *in* M. Jazayeri, R. Loos and D. Musser (eds), *Generic Programming*, Vol. 1766 of *LNCS*, Springer-Verlag, pp. 102–113.

Soundarajan, N.: 2000, Documenting framework behavior, *ACM Computing Surveys (CSUR)* **32**(1es), 14.

Sridhar, N. and Hallstrom, J. O.: 2003, Generating configurable containers for component-based software, *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering*.

Sridhar, N., Pike, S. M. and Weide, B. W.: 2003, Dynamic module replacement in distributed protocols, *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pp. 620–627.

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

Sridhar, N., Weide, B. W. and Bucci, P.: 2002, Service facilities: Extending abstract factories to decouple advanced dependencies, *Proceedings of the 7th International Conference on Software Reuse*, pp. 309–326.

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

SunMicrosystems: 2001, J2ee 1.3 specification, http://java.sun.com/j2ee/download.html.

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

Szyperski, C.: 2003, Component technology: what, where, and how?, *Proceedings of the 25th International Conference on Software Engineering*, pp. 684–693.

Szyperski, C. A.: 1997, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley.

Tardo, J. J.: 1981, *The design, specification, and implementation of OBJ-T: A language for writing and testing abstract algebraic program specifications*, PhD thesis, Department of Computer Science, University of California Los Angeles.

Tarr, P., Ossher, H., Harrison, W. and Sutton, S. M.: 1999, N degrees of separation: multi-dimensional separation of concerns, *Proceedings of the 21st international conference on Software engineering*, IEEE Computer Society Press, pp. 107–119.

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

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

Weide, B., Edwards, S., Heym, W. and Long, T.: 1994, Characterizing observability and controllability of software components, *Proceedings of the 4th International Conference on Software Reuse*, pp. 62–71.

Weide, B. W.: 2002, Component-based systems, *in* J. J. Marciniak (ed.), *Encyclopedia of Software Engineering*, John Wiley and Sons.

Weide, B. W., Ogden, W. F. and Sitaraman, M.: 1994, Recasting algorithms to encourage reuse, *IEEE Software* **11**(5), 80–88.

Wirth, N.: 1980, Modula-2, *Technical Report Report 36*, Institut für Informatik, ETH, 8092 Zürich, Switzerland.