INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Specialization Patterns

Ulrik Pagh Schultz, Julia L. Lawall, and Charles Consel

N°3853

Janvier 1999

THÈME 2

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--3853--FR+ENG

Specialization Patterns

Ulrik Pagh Schultz, Julia L. Lawall, and Charles Consel

Thème 2 — Génie logiciel
et calcul symbolique
Projet COMPOSE

Rapport de recherche n°3853 — Janvier 1999 — 28 pages

Abstract: Design patterns offer numerous advantages for software development, but can introduce inefficiency into the finished program. Program specialization can eliminate such overheads, but is most effective when targeted by the user to specific bottlenecks. Consequently, we propose to consider program specialization and design patterns as complementary concepts. On the one hand, program specialization can optimize object-oriented programs written using design patterns. On the other hand, design patterns provide information about the program structure that can guide specialization. Concretely, we propose *specialization patterns*, which describe how to apply program specialization to optimize uses of design patterns.

In this paper, we analyze the specialization opportunities provided by specific uses of design patterns. Based on the analysis of each design pattern, we define the associated specialization pattern. These specialization opportunities can be declared straightforwardly using the specialization classes framework, developed in earlier work. Our experiments show that such specialization leads to significant performance improvements.

Key-words: design patterns, program specialization, object-oriented programming, Java, optimization

(Résumé : tsvp)

Supported in part by BULL, and in part by Alcatel under the Reutel 2000 contract.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Schémas de spécialisation

Résumé : Les schémas de conception sont reconnus pour faciliter le développement de programmes à objets. Cependant, ils introduisent souvent une certaine inefficacité au sein du logiciel final. La spécialisation de programmes est une technique particulièrement adaptée à l'optimisation des programmes à objets. Toutefois, l'utilisation efficace de la spécialisation sur des logiciels de grande taille requiert une connaissance détaillée des goulots d'étranglement de la part du programmeur.

Dans cet article, nous proposons d'associer spécialisation de programmes et schémas de conception. Notre propos est double. D'une part, la spécialisation permet de supprimer les inefficacités introduites par les schémas de conception. D'autre part, les schémas de conception fournissent des informations sur la structure du programme qui permettent de guider le processus de spécialisation. Plus précisément, nous introduisons la notion de *schémas de spécialisation* afin de mémoriser comment la spécialisation de programmes peut optimiser l'utilisation d'un schéma de conception. Nous démontrons l'applicabilité des schémas de spécialisation, en caractérisant les opportunités de spécialisation présentes dans plusieurs schémas de conception. Nous introduisons notre approche en présentant des exemples de programmes spécialisés automatiquement au moyen des schémas de spécialisation.

Mots-clé : schémas de conception, spécialization de programmes, programmation orientée object, Java, optimisation

1 Introduction

Design patterns, as presented by Gamma *et al.* [16], describe well-tested program structures that enhance modularity and code reuse. A program written using design patterns is structured into independent units that interact through generic interfaces, and that evolve over time in response to changing conditions. Because design patterns are well-documented, their use simplifies the understanding of programs constructed from many independent units. The flexibility inherent in this use of generic interfaces, however, intrinsically blocks optimization across objects, and thus can carry a significant performance penalty. This issue remains largely unaddressed in the design pattern community.

Many applications do not fully exploit the flexibility offered by design patterns. For a simple example, consider a typical use of the *iterator* design pattern [16], which separates traversal of a data structure from its representation. Using the iterator pattern, an implementation of a `Set` data structure might define the `member` method as follows:

```
public class Set {
  MinimalCollection coll;              // underlying collection
  public boolean member( Object o ) {
    Iterator e = coll.iterator();      // obtain iterator
    while( e.hasNext() ) {             // while iterator has elements
      Object x = e.next();             // obtain next iterator element
      if( x.equals( o ) ) return true;
    }
    return false;
  }
  ...
}
```

This definition of the `member` method can be used with any underlying `MinimalCollection` implementation, letting the programmer freely choose the most appropriate concrete representation for the task at hand. Nevertheless, our experiments show that the use of the iterator pattern blocks compiler optimization of the element retrieval operations. When the `member` method is used repeatedly to search `MinimalCollection` objects that have the same representation, the flexibility provided by accessing the data through an abstract interface is not needed. In this case, the `member` method can be optimized, by replacing the generic uses of the iterator pattern (underlined in the method definition) by direct accesses to the underlying data structure. This transformation gives a speedup ranging from 20% to 80%.[1] These measurements suggest that the optimizations performed by state-of-the-art compilers do not completely compensate for the genericity introduced by design patterns.

When the data representation is invariant over a period of time, specializing the program to this representation before execution improves efficiency. However, manual specialization is error-prone, and introduces excessive program-maintenance overhead. Recently, automatic program specialization has been shown to be effective in the context of object-oriented

[1] Experiments done with JDK 1.2.1 JIT and HotSpot compilers on SPARC architecture, with array and linked list representations of the underlying `MinimalCollection` data structure.

languages, specifically Java [22, 28]. Automatic program specialization systematically eliminates both algorithmic and structural overheads, and consequently can significantly improve performance. For example, program specialization has been shown to be effective for eliminating overheads introduced by software architectures [24].

Nevertheless, specialization is not always beneficial; for example specializing with respect to too many different representations can cause code explosion. Therefore, the user must explicitly target the specializer toward particular invariants and regions of code. The *specialization classes* framework proposed by Volanschi *et al.* [32] provides a simple and declarative notation for expressing such specialization opportunities. Nevertheless, the problem of identifying specialization opportunities remains. Profiling can help, however, it may not reveal systematic structural overheads that block optimization throughout the program. A systematic approach taking into account the program design is needed.

This paper

We observe that the use of design patterns in a program gives rise to patterns of structural properties, which in turn give rise to patterns of overheads. These patterns of overheads form patterns of opportunities for specialization. We propose the use of *specialization patterns* as a complement to design patterns, to describe when, how, and why a program structured using design patterns can benefit from specialization. This approach retains the program structuring advantages of design patterns, while relying on an automated transformation to map generic code into an efficient implementation. The contributions of this paper are as follows:

- We address the much overlooked problem of performance of programs written using design patterns, by analyzing the overheads systematically introduced by the use of design patterns.
- We describe how to systematically apply program specialization to eliminate these overheads, automatically mapping well-structured generic implementations into monolithic efficient ones.
- We define specialization patterns for three well-known design patterns: the builder pattern, the bridge pattern, and the strategy pattern.
- We provide several examples of how specialization can optimize uses of design patterns, and provide benchmarks showing the effect of specialization on realistic versions of these examples.

Earlier work has addressed the declaration of *what* to specialize in the form of specialization classes [32] and *how* to specialize in the form of a prototype Java specializer [28]. Here, we address the key issue of selecting *where* to specialize.

First, Section 2 describes our perspective on design patterns, followed by Section 3, which explains program specialization. Section 4 describes specialization of design patterns by means of specialization patterns. Afterwards, Section 5 provides a substantial example, combining several design patterns. Then, Section 6 assesses the application of program specialization to uses of design patterns. Finally, Section 7 presents related work, Section 8 discusses future work, and Section 9 concludes.

Conventions: Throughout this paper, we consider only the design patterns described by Gamma *et al.* [16]. Also, to make it easier to display example programs, we have omitted the Java visibility modifiers for fields and methods.

2 Design Patterns

When a software system evolves frequently, reusable and reconfigurable program components are needed. The notions of reusability and reconfigurability are central to object-oriented programming: reuse is provided by inheritance, while reconfigurability is achieved by organizing the program using objects, which serve as composable building blocks. Nevertheless, these features tend to distribute functionality across the entire program. As a result, programs that heavily exploit inheritance and object composition can be difficult for programmers to understand and for compilers to optimize.

2.1 The structure of adaptable programs

The degree of adaptability in a program is determined by the ease with which the implementation can be modified as needs change, and the ability of the program to react to changing conditions at run time. These issues are affected by how classes are defined.

A simple approach to adapting a program is to define a new class by using inheritance to extend the state and behavior of an existing class. The use of inheritance eliminates the need to reimplement generic functionality. Nevertheless, because the inheritance relationship is declared explicitly in the source program, the program must be rewritten to adapt to changing conditions.

Rather than defining a class in terms of an existing class, a new class can be defined using object composition, by referring to the methods and local state of other objects. Like inheritance, object composition eliminates the need to reimplement existing functionality. This approach, however, also allows adaptability at run time, as the local state of the referenced objects changes, or as these objects are replaced by other objects respecting the same interface. Defining objects in terms of abstract interfaces allows an object to be replaced by another object that has an unrelated implementation.

2.2 The role of design patterns

The use of inheritance and object composition to enhance adaptability is non-trivial and can make it difficult to understand how program components fit together. Design patterns address these issues. Each design pattern focuses on a single problem often encountered in developing an adaptable program, and proposes a widely applicable solution. By following a design pattern the programmer takes advantage of a well-tested program structure that is open for future extensions. Adaptable programs can be described in terms of the design patterns they implement, which provides a guide as to how the functionality of the program is likely to be distributed among the class definitions.

Furthermore, to simplify program development and facilitate communication, programs may be explicitly organized according to well-known design patterns, even when the full flexibility provided by the chosen design patterns is not needed. Particularly in this case, effective optimization techniques for the kinds of programs that result from the use of design patterns are critically needed.

2.3 Overheads introduced by design patterns

In practice, the separation of classes using abstract interfaces implies that the method definition associated with a given method invocation is often not obvious to the compiler. When this is the case, the method invocation must be implemented as a virtual call. The use of virtual calls divides a program into separate blocks that must be individually optimized, effectively erecting optimization barriers throughout the program. Virtual calls both defeat branch prediction (and thereby instruction pipelining), and inhibit inlining, blocking subsequent traditional intra-procedural compiler optimizations [6, 14].

Many compilers go to great lengths to eliminate virtual calls [1, 12, 13, 27]; some even make use of constrained specialization techniques [10, 20], such as customization [7]. Standard compiler optimizations rely on static analysis; when the method referenced by a method call can be determined statically, the method call can be implemented using a direct call. Calls to inherited methods can often be implemented as direct calls, because the inheritance hierarchy is explicit at compile time. When a method of a referenced object is invoked, however, the call can only be implemented as a direct call if it can be statically determined that at run time the reference always refers to an object of the same type. This property rarely holds in large programs with complex class hierarchies that are written using design patterns, where the potential of the program to adapt to changing conditions is reflected in a multitude of possible implementations at each adaptable program point.

When the number of possible callees is limited, some compilers replace a virtual call by a conditional that selects at call-time which callee to invoke with a direct call. This transformation can be directed using automatically gathered profile information, and enables further optimizations through inlining or customization [17, 20]. However, the cost of a runtime decision remains, and the control flow is only somewhat simplified.

These observations are illustrated by the the benchmarks reported in Section 6. Using state-of-the-art Java compiler technology, we found that programs written using design patterns that operate through abstract interfaces run at about half the speed of programs that explicitly use direct calls.

3 Program Specialization

Program specialization is the optimization of a program (or a program fragment) based on information about the context in which it is used, thus generating a dedicated implementation. In our framework, specialized methods are added to existing classes; no changes are made to the class hierarchy.

One approach to automatic program specialization is *partial evaluation*, which performs aggressive inter-procedural constant propagation of all data types, and performs constant folding and control-flow simplifications based on this information. Partial evaluation has been extensively investigated for functional [4, 8], logic [23], and imperative [2, 3, 9] languages. This technique has been recently extended to Java, by Schultz *et al.* [28], using C as an intermediate language. The implementation combines the Harissa bytecode to C compiler [26] with the Tempo program specializer for the C language [9]. For this paper, we have extended this approach with an automatic translation of specialized programs back to Java.

3.1 Optimization by specialization

In the context of design patterns, we are primarily interested in using specialization to eliminate virtual calls. Concretely, we would like to specialize a program written using design patterns to the types of the objects it manipulates, as well as to (some of) the values these objects contain. By specializing the program with respect to a fixed object structure, we safely bypass the abstract interfaces that isolate program components, possibly triggering other optimizations, either during specialization or at compile time, and produce a monolithic block of optimized code.

A partial evaluator can rarely deduce specialization invariants from a large program as precisely as a human programmer can, and specialization of non-critical parts of a program may cause unwanted overheads. Thus, specialization is most effective when directed by the user towards a limited part of the program where specialization is believed to be beneficial. For this reason, the Tempo specializer has been designed to operate on a program slice, which can be re-inserted into the program after specialization. Such a program slice and the invariants for which it is to be specialized can be concisely described using specialization classes. Specialization classes insert guards into the specialized program that ensure that the specialized code is used only when the invariants are satisfied [32]. In the context of design patterns, specialization classes allow the programmer to specialize for local invariants that only hold for the objects that play a role in the use of a design pattern.

3.2 Limitations of specialization

The benefits of specialization are limited by the degree to which the specializer can propagate the user-supplied invariants through the program, and by the utility of the transformations triggered by these invariants. Specialization by partial evaluation does not propagate information that in some way depends on values not known to the specializer. Consequently, this form of specialization is most effective when there is a clean separation between the terms that depend only on explicit constants or user-supplied invariants, and the unknown terms of the program. Specialization classes can be used to provide the specializer with extra information, but because of the need to react at run time when specialization invariants are invalidated, the use of specialization classes does add some inefficiency. Excessive propagation of invariants can also be detrimental. In particular, specialization can cause code explosion either by too much loop unrolling, or by generating a very large number of specialized methods.

3.3 Specialization example

As an example of automatic program specialization, let us revisit the example of the Iterator, described in Section 1.

```
public class Set {
  MinimalCollection coll;                  // underlying collection
  public boolean member( Object o ) {
    Iterator e = coll.iterator();          // obtain iterator
    while( e.hasNext() ) {                 // while iterator has elements
      Object x = e.next();                 // obtain next iterator element
      if( x.equals( o ) ) return true;
    }
    return false;
  }
  ...
}
```

We specialize the use of the iterator pattern in the `member` method to the specific type of the iterator object, thus reducing the number of virtual calls. Suppose that the `MinimalCollection` object referenced through the `coll` field is known to be an object of a specific implementation class named `Array`, presented in the appendix. The `Array` data structure always uses the `ArrayIterator` iterator (also found in the appendix), so it is advantageous to specialize the `member` method for the `coll` field being of `Array` type.

The specialization invariant can be declared using a specialization class[2] as follows:

```
specclass Member_Array specializes Set {
  Array coll;                    // coll field is of type Array
  void member( Object o );       // specialize the member method
}
```

Specializing according to `Member_Array` unfolds the references to the methods of the enumerator, yielding the following method:

[2] To improve expressiveness, we have slightly generalized the syntax of specialization classes presented by Volanschi *et al.*[32]. For example, we allow invariants over formal parameters, and invariants that declare an object to have a specific type. These extensions are straightforward to implement, and do not significantly changes the specialization class framework.

```
public boolean member_Array( Object o ) {
  ArrayIterator e = new ArrayIterator( (Array)coll );
  while( e.current < e.max ) {
    Object x = e.array.elements[e.current++];
    if( x.equals( o ) ) return true;
  }
  return false;
}
```

The specialized code explicitly allocates a new `ArrayIterator`, which is local to this method. It is also now explicit that the array elements are accessed sequentially within the loop. Both of these features can be exploited by a compiler performing intra-procedural optimizations. The automatically specialized definition is between 20% and 80% faster than the original definition.

Specialization with respect to one invariant can often trigger other specialization opportunities. Here, if the length of the `Array` object is known, the specializer can unroll the loop, so that only the code needed to compare the unspecified data contained in the array remains. Similarly, if the type of the elements contained in the `Array` object is known, the specializer can select the corresponding definition of the `equals` method, allowing the call to be inlined into the loop. Thus, specialization of design patterns can trigger further optimization: eliminating the indirection provided by the design pattern makes it possible to exploit other invariants.

4 Specialization Patterns

Design patterns facilitate communication of design ideas by encapsulating a characterization of a common problem, together with a programming strategy that solves the problem, as well as examples and documentation, into a single logical unit. Specialization patterns complement design patterns, by documenting a specialization process that results in an efficient implementation.

4.1 Specialization patterns: definition and use

A specialization pattern describes the overheads intrinsic in using a particular design pattern, documents how to use specialization to eliminate these overheads, and provides an example that clearly illustrates how to specialize a use of the design pattern. In addition, a specialization pattern can refer to other specialization patterns, to describe how multiple design patterns can be specialized together. Specialization patterns not only guide specialization after a program has been written, but can also help the programmer structure the program so that subsequent specialization will be beneficial.

In the spirit of design patterns, specialization patterns are based on the template of Figure 1, ensuring a consistent format. The template includes sections that relate the specialization pattern to the design pattern, criteria for judging when it is worthwhile to

Name: The name of the associated design pattern.

Description: A short description of the design pattern.

Extent: The program slice that is relevant when optimizing a use of the design pattern.

Overhead: Possible overheads associated with use of the design pattern.

Compiler: Analysis of when these overheads are eliminated by standard compilers.

Approach: Specialization strategies that eliminate the identified overheads.

Condition: The conditions under which the specialization strategies can be effectively exploited.

Specialization class: Guidelines for how to write the needed specialization classes, and how to most effectively apply them.

Applicability: A rating of the overall applicability of specialization to a use of the design pattern, using the other information categories as criteria.

Example: An example of the use of specialization to eliminate the identified overhead; the example may include specialization classes or textual descriptions.

Figure 1: Specialization pattern template

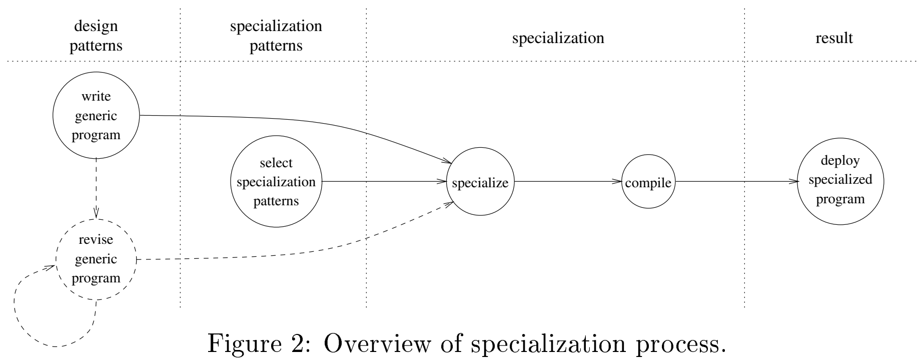


Figure 2: Overview of specialization process.

specialize a use of the design pattern, detailed instructions for specializing, and finally a specialization example. Sample specialization patterns are illustrated in Figures 3, 5, and 7.

Just as a design pattern takes into account different programming languages, a specialization pattern should take into account different program specializers. At the time of writing, the only existing specializer for a realistic object-oriented language that the authors are aware of is the specializer described by Schultz *et al.* [28]. In the context of this paper, that specializer will thus serve to define the minimal expected functionality. As other specializers are developed, the specialization patterns can be refined accordingly.

Figure 2 shows how specialization patterns fit into the software development process. Once the program is written, specialization opportunities are identified as indicated by the specialization patterns corresponding to the design patterns used in the program. Specialization classes are then written as suggested by the selected specialization patterns. The program and the specialization classes are then provided to the specializer, which generates a specialized program.

We now identify the specialization opportunities provided by *creational*, *structural*, and *behavioral* design patterns, and present examples of specialization patterns.

4.2 Creational Patterns

A creational design pattern abstracts the construction of objects, known as the *products*, delegating parts of the instantiation process to auxiliary classes. The use of a creational pattern separates the operations on an object from the underlying representation, allowing the representation to be changed transparently, without affecting the rest of the program. Nevertheless, this abstraction barrier implies that the product objects must be accessed using virtual calls, which block optimization.

Memory allocation and object initialization are dominating factors during object creation, so simply eliminating the virtual calls associated with the creation process is unlikely to significantly optimize a program. Thus, specialization should also be applied to the parts of the program where the products are used, by specializing the uses to the concrete type of each product. Such specialization permits direct access to the product objects, enabling ordinary intra-procedural optimizations. However, such a specialization strategy is *only* worthwhile when the specializer can determine how the products are manipulated after they have been created. Thus, a specialization pattern for a creational pattern can only give very limited information on when it is worthwhile to specialize. On the positive side, the product is known to the specializer when its creation can be traced to a specific use of a creational pattern, so creational patterns do not obfuscate the specialization process.

Example: builder pattern

Figure 3 defines the specialization pattern for the builder pattern. As an example of applying the builder specialization pattern, Figure 4a shows the `ListBuilder` interface for creating `AbstractList` lists using the builder pattern. An implementation must provide the methods `start`, which initializes the list, `add`, which extends the list, and `getProduct`, which finishes the production sequence by returning the head of the list. Also defined is the class `Main` with a method `f`, which uses the `ListBuilder` interface to construct a list, and then accesses the head of the list just produced. Figure 4b shows the concrete builder implementation `LinkedListBuilder`, which produces linked lists of type `LList` (the definitions of `AbstractList` and `LList` are trivial, and not shown here).

The specialization class of Figure 4c specifies that the method `f` of the class `Main` should be specialized with respect to a specific builder, namely the `LinkedListBuilder` implementation.

Name: Builder pattern

Description: The builder pattern allows a complex structure to be created by invoking a sequence of methods defined in a generic builder interface. The use of this pattern separates the construction process from the underlying representation.

Extent: Specialization is applied to a collection of classes implementing the concrete representation of a structure, a class implementing the builder interface, and a client, which builds a structure using the generic operations provided by the builder interface. Also any subsequent use of the product structure.

Overhead: Separation of the type of the product from the client means that product objects must be accessed using virtual calls. For the same reason, little is known about the initial state of the product objects.

Compiler: When there is either just a single kind of builder or a single kind of product, a compiler can usually generate direct calls for accessing the methods of the product object. Nevertheless, a compiler typically does not make use of initialization information.

Approach: Specializing the client with respect to a particular implementation of the builder makes the objects comprising the structure directly accessible to the client. Accesses to the components of the structure can then be implemented using direct calls to the methods of these objects. Information about the current state of these objects can be used for further optimizations.

Condition: The type of the builder must be known to the specializer (possibly as a specialization class invariant). To guarantee specialization of the builder, the sequence of building actions must be fixed within the program. Furthermore, to guarantee direct use of the products and that information about their state is exploited by the specializer, they must be used in a fixed way.

Specialization class: The specialization class should fix the type of the builder, and specify specialization of a method that both uses the builder and the resulting product objects.

Applicability: High when the specialization class can be placed properly and the product objects are used often. Low to none otherwise.

Example: See Figure 4 and explanation in text.

Figure 3: Builder specialization pattern

In the specialized program (Figure 4d), virtual calls have been replaced by direct data-structure manipulations.[3] Specialization first replaces the virtual calls through the `ListBuilder` interface by direct calls, and then inlines the method definitions into the caller, eliminating temporary variables when appropriate.

Specialization to a single concrete implementation permits the product objects to be accessed directly as long as they are not manipulated in a dynamic way. In the example,

[3] As is the case for all of the examples shown in this paper, the specialized code has been resugared for readability.

```
interface ListBuilder {
 void start();
 void add( Object o );
 AbstractList getProduct();
}
class Main {
 ListBuilder b;
 void f() {
  b.start();
  b.add("x");
  b.add(new Vector());
  AbstractList p = b.getProduct();
  something( p.getElm() );
 }
}
```

(a) Use of builder through interface

```
class LinkedListBuilder
      implements ListBuilder {
 LList head, c;
 void start() {
  head = new LList(null);
  c = head;
 }
 void add( Object x ) {
  c.next = new LList(x);
  c = c.next;
 }
 AbstractList getProduct() {
  return head.next;
 }
}
```

(b) Concrete builder for linked lists

```
specclass Main_LL
          specializes Main {
 LinkedListBuilder b;
 f();
}
```

(c) Declaration of specialization to the `LinkedListBuilder` builder

```
void f_LinkedListBuilder() {
 LinkedListBuilder b;
 b.head = new LList(null);
 b.c = b.head;
 b.c.next = new LList("x");
 b.c = b.c.next;
 b.c.next =
   new LList(new Vector());
 b.c = b.c.next;
 AbstractList p = b.head.next;
 something( p.elm );
}
```

(d) Result of specialization

Figure 4: Specializing a use of the builder pattern.

the product is used in a fixed way, and the virtual call to `getElm` has been replaced by its concrete definition in the `LList` class. If desired, the method `something` can also be specialized, adapting it to the concrete value stored in the first element of the `LList` object. Had the product been manipulated in a dynamic way, the benefits of specialization would have been negligible.

Other creational patterns

In addition to the builder pattern, the abstract factory and prototype patterns also hide the types of the objects that they produce; thus, uses of these patterns are good targets for specialization. But as is the case for all creational patterns, whether the program will benefit from specialization depends on how the products are manipulated. The factory and singleton patterns are much simpler, and the types of the objects that they produce is usually evident. Uses of these patterns are thus easily handled by an optimizing compiler, but can of course be specialized as well.

4.3 Structural Patterns

Structural design patterns organize relations between objects, enhancing reuse and replacability. They allow the programmer to combine individual objects respecting a common interface into compound objects that behave in new ways. By separating the objects using interfaces, structural patterns allow the object structure to be transparently extended, and new classes implementing the interface to be added. This flexibility implies, however, that the components must interact using virtual calls, obscuring the flow of control through the object structure.

A program that builds and traverses an object structure can be specialized to a specific layout of this structure. Specialization permits the objects of the structure to interact directly, and all of the basic operations on the structure to be collected in a single place. If the structure is not modified after its creation, the methods that manipulate it can be directly specialized to its layout. More generally, specialization classes can be used to describe layouts that are of interest. As always, specialization classes introduce overheads, so the latter approach might not be beneficial if the structure changes too often, or if the amount of computation within the structure is too limited. Specialization of a structural pattern can generate code having size proportional to the size of the object structure. Thus, when a structure may be prohibitively large, specialization should be applied with caution to avoid code explosion.

Example: bridge pattern

Figure 5 defines the specialization pattern for the bridge pattern. As an example of applying the bridge specialization pattern, Figure 6a shows a use of the bridge pattern. The class `Complex` represents complex numbers, with the specific implementation deferred to a `ComplexImpl` object. The `multiply` operation is delegated to the concrete implementation, whereas the `square` operation is defined in the interface object. The function `f` of the class `SquareFn` simply computes the square of a complex number. Two concrete implementations of `ComplexImpl` are given in Figure 6b: the `RectComp` implementation uses rectangular coordinates to represent complex numbers, whereas the `PolarComp` implementation uses polar coordinates.

The specialization class `SquareFn_RectComplex` (Figure 6c) specifies that `f` should be specialized to complex numbers that fulfill the invariants given in the `Rectangular` specialization class. This specialization class specifies that the implementation object has type `RectComp`. The result of specialization is an implementation of `f` where the complex numbers are accessed directly, allowing the mathematical operations to be applied directly to the object fields. Specialization first replaces virtual calls from `f` to the bridge interface object by direct calls, and similarly replaces virtual calls from the interface object to the implementation

Name: Bridge pattern

Description: The bridge pattern separates an object into a generic interface object and an implementation object. The client accesses the object via the generic interface object, which provides access to a specific implementation object using delegation. The bridge pattern decouples an abstraction from its implementation, so that the two can vary independently.

Extent: Specialization is applied to an interface object which has operations that are defined in terms of generic operations on an implementation object.

Overhead: To allow the implementation to vary independently from the generic interface object, method invocations from the generic interface object to the implementation object are implemented using virtual calls. Furthermore, the client typically requires access to the methods of the implementation object, which implies two levels of method invocation (first a call to the interface object, then the subsequent call to the implementation object).

Compiler: Since there normally is a hierarchy of implementation classes, a compiler can rarely eliminate the virtual call from the interface object to the implementation object.

Approach: When the components connected by the bridge are known, the bridge can often be eliminated. This allows direct access from the interface object to the implementation object.

Condition: If the coupling between the interface object and the implementation object never changes or only changes in a fixed way, then it can be specialized to remove one level of indirection. If the interface object is used by the program in a fixed way, then the interface indirection can be specialized away as well.

Specialization class: The specialization class should at least fix the type of the implementation object, and should ideally specify specialization of a method the uses the interface object several times. Alternatively, when a single method in the interface object calls is implemented using several methods in the implementation object, the specialization class can specify specialization of the interface object.

Applicability: Medium when the specialization class can be placed properly or when there are many calls from the interface to the implementation object. Low otherwise.

Example: See Figure 6 and explanation in text.

Figure 5: Bridge specialization pattern

object by direct calls. These direct calls are inlined, eliminating temporary variables when appropriate, to produce the specialized implementation of `f` shown in Figure 6d.

Other structural patterns

As is the case for the bridge pattern, the adapter, composite, decorator, facade, and proxy structural patterns allow the building of structures from objects hidden behind generic interfaces, so uses of these patterns are good targets for specialization, and specialization is guaranteed to simplify the program when the structure does not change or when it can be

```
interface ComplexImpl {
 void mult( Complex c );
 double r();
 double i();
}
class Complex {
 ComplexImpl imp;
 void multiply( Complex c ) {
  imp.mult( c );
 }
 void square() {
  imp.mult( this );
 }
 double r() { return imp.r(); }
 double i() { return imp.i(); }
}
class SquareFn {
 void f( Complex x ) {
  x.square();
 }
}
```

(a) Generic number interface and use

```
class RectComp
      implements ComplexImpl {
 double r, i;
 void mult( Complex c ) {
  double cr = c.r();
  double ci = c.i();
  double nr = r*cr - i*ci;
  double ni = r*ci + i*cr;
  r = nr; i = ci;
 }
 double r() { return r; }
 double i() { return i; }
}
class PolarComp
      implements ComplexImpl {
  ...polar coordinates...
}
```

(b) Specific number implementations

```
speclass SquareFn_RectComplex
         specializes SquareFn {
 void f( Complex x ),
         Rectangular x;
}
specclass Rectangular
          specializes Complex {
 RectComp imp;
}
```

(c) Declaration of specialization to the `NormalNum` implementation

```
void f_RectComplex( Complex x ) {
 RectComp tmp = x.imp;
 double cr = x.imp.r;
 double ci = x.imp.i;
 double nr = tmp.r*cr - tmp.i*ci;
 double ni = tmp.r*ci + tmp.i*cr;
 tmp.r = nr; tmp.i = ni;
}
```

(d) Result of specialization

Figure 6: Specializing a use of the bridge pattern.

encapsulated using specialization classes. The flyweight pattern optimizes memory usage by sharing objects, and cannot be specialized in any obvious way.

4.4 Behavioral Patterns

Behavioral patterns abstract over the control flow, providing generic ways of parameterizing behavior. They offer a clean separation between different aspects of an overall behavior, making it possible to construct new behaviors by composing individual objects or classes.

The overall behavior is distributed among cooperating objects, and can be modified by changing the way objects are composed together. Every time the collaborating objects that implement a behavior are used for a specific function, they must interact with each other using virtual calls.

A program using a behavioral pattern can be specialized to a specific behavior, by specifying the values and objects that control the behavioral pattern. Specialization transforms the complete description of the behavior into a single unit. Nevertheless, the behavioral design patterns are so diverse that it is only for specific patterns that we can guarantee benefits from specialization. Depending on the specific pattern in question, specialization can be done by specializing the pattern use to the object structure that it processes, and possibly to any values that control how it processes the object structure. In any case, if the objects that make up the use of the pattern are cannot be determined by the specializer, the behavioral pattern cannot in general be specialized.

Example: strategy pattern

Figure 7 defines the specialization pattern for the strategy pattern. As an example of its application, Figure 8a shows a use of the strategy pattern. The `Image` class represents an image using pixels defined by the `RGB` class. The `process` method of an `Image` object applies the pixelwise processing strategy stored in the field `op` to each the pixel of the image. Figure 8b defines two such single-pixel operations: `Scale`, which scales a pixel (thereby changing its brightness), and `RedOnly`, which discards all but the red component.

The specialization class `ScaleByTwoProcess` (Figure 8c) declares that the operation is a `Scale` operation, and that the scaling value is 2.0. We thus specialize the `Image` class to a strategy that is specified not only in terms of its type, but also in terms of its internal state. Specialization merges the effect of the strategy object into the original `process` method (Figure 8d), by eliminating the virtual call to the strategy method, inlining the call, and propagating the known scaling value.

Other behavioral patterns

As is the case for the strategy pattern, precise specialization patterns can be given to the chain of responsibility, interpreter, mediator, observer, state, and visitor patterns. For the interpreter and visitor patterns, specialization is beneficial when the use of the pattern can be specialized with respect to the structure processed by the pattern, in which case the use of the pattern can be completely eliminated. The command and iterator design patterns represent opportunities for specialization, but it is difficult to precisely specify when this is the case, except for the most basic case where the behavior is obtains genericity through inheritance, and can easily be handled by an optimizing compiler. The memento pattern externalizes the state of an object, and cannot be specialized in any obvious way.

Name: Strategy pattern

Description: The strategy pattern allows clients to transparently replace one algorithm by another. This design pattern allows clients to choose among whole families of algorithms rather than just a single algorithm.

Extent: Specialization is applied to a family of algorithms all implementing the same abstract interface, and a client that uses such algorithms through this abstract interface.

Overhead: All operations provided by the algorithm must be accessed through the abstract interface. The less computation is done by the algorithm, the more this overhead is becomes noticeable.

Compiler: Unless the strategy is chosen explicitly before it is used, a compiler is unlikely to bypass the abstract interface.

Approach: By specializing the client to the concrete algorithm, the abstract interface can be bypassed, and the algorithm can be inlined into the client. This opens opportunities for further specialization and optimization of the algorithm to the context in which it is being used.

Condition: If the coupling between the client and the concrete strategy being used never changes, then the client can be specialized to this strategy. If the coupling does not change during the invocation of a method in the client, the specialization classes can introduce a local invariant, allowing this method to be specialized to the strategy. If changes in the coupling are completely fixed then the client is guaranteed to be specializable to the strategy.

Specialization class: The specialization class should fix the type of the strategy, and specify specialization of a method that applies the strategy.

Applicability: High when the strategy is used inside a loop, medium when used a few times, low when used only once.

Example: See Figure 8 and explanation in text.

Figure 7: Strategy specialization pattern

5 A Complete Example

To illustrate the combined effects of specialization of several design patterns, we provide a complete example: a graphical application. We first describe the example, focusing on the toolkit used to write the application, and then characterize the overheads present in the application and explain how they can be eliminated.

5.1 Description

Our example is a graphical text editor application written using an abstract windowing toolkit, styled after the Java JDK 1.1 AWT (abstract windowing toolkit). Graphical win-

```
interface RGBOP {
 void handle( RGB pixel );
}
class RGB { double r, g, b; }
class Image {
 RGB [][]img; int w, h;
 RGBOP op;
 void process() {
  for(int i=0; i<w; i++)
   for(int j=0; j<h; j++)
    op.handle(img[i][j]);
 }
 ...
}
```

(a) RGB Image which uses operator

```
class Scale implements RGBOP {
 double s;
 void handle( RGB p ) {
  p.r*=s;p.g*=s;p.b*=s;
 }
}
class RedOnly implements RGBOP {
 void handle( RGB p ) {
  p.g=0;p.b=0;
 }
}
```

(b) RGB pixel operations

```
specclass ScaleByTwoProcess
          specializes Image {
 ScaleByTwo op;
 void process();
}
specclass ScaleByTwo
          specializes Scale {
 s == 2.0;
}
```

(c) Declaration of specialization to the `Scale` operation

```
void process_ScaleByTwo() {
 for(int i=0; i<w; i++ )
  for(int j=0; j<h; j++ ) {
   RGB p = img[i][j];
   p.r*=2.0;p.g*=2.0;p.b*=2.0;
  }
}
```

(d) Result of specialization

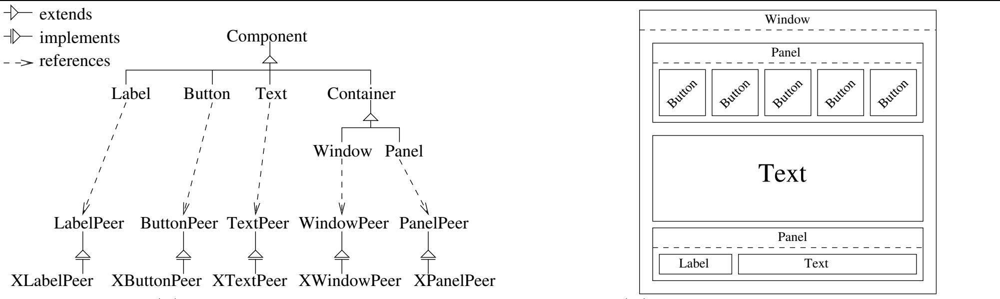
Figure 8: Specializing a use of the strategy pattern.

dowing toolkits often contain many opportunities for specialization, and the JDK 1.1 AWT is no exception.[4]

We focus on the following uses of design patterns in the toolkit:

- The structural pattern *composite* is central to most graphical toolkits, allowing graphical widgets and widget containers to be freely combined. To use the composite pattern, each graphical widget and container extends an abstract class, `Component` (the name used in JDK 1.1).
- To allow our toolkit to function with any concrete windowing environment, we separate each component into its general representation and its system-specific *peer*, using the bridge pattern (Section 4.3). The peer objects are created using the *abstract factory* creational pattern, which defines a general interface for creating peers. To use the

[4]The JFC Swing library contains even more opportunities for specialization, but the standard JDK 1.1 AWT is sufficient for this example.



(a) Class hierarchy (b) Structure of the application

Figure 9: Overview of graphical application.

abstract factory, we let a central class (named `Toolkit` in JDK 1.1) function as an interface for instantiating peers.

- To simplify event handling, we use the *observer* behavioral pattern (for which there are standard interfaces in JDK 1.1). Clients subscribe to events generated by specific widget objects, such as buttons, and are notified when these events occur (using standard Java method calls).

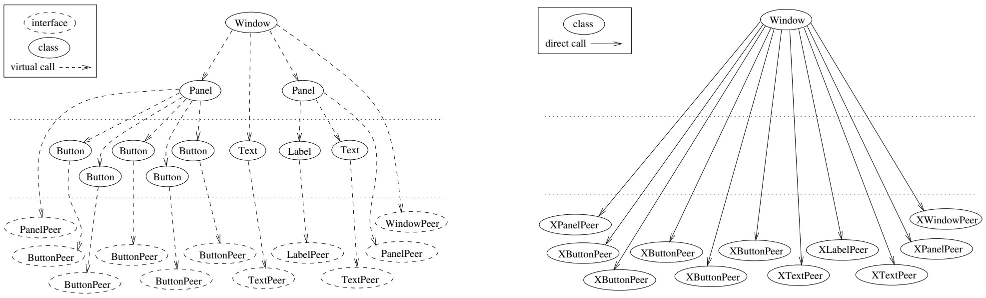
The class hierarchy of the basic graphical objects of the toolkit is shown in Figure 9a.

When launched, the text editor initializes itself, arranging its graphical appearance (illustrated in Figure 9b), and then waits for input events to be generated by the user, each event triggering a specific action.

5.2 Overheads

Each use of a design pattern in the graphical toolkit gives rise to a specific overhead. The use of the composite pattern makes it possible to change the graphical appearance of the program at run time. A virtual method such as `repaint`, that is implemented by all the `Component` objects, must use virtual calls to traverse the composite structure. Furthermore, the use of the abstract factory (together with the bridge) hides the types of the peer objects, in principle allowing new peers to be introduced at any point; all calls from a `Component` object to its native implementation are virtual. Finally, every time an event is generated, a corresponding event object is created and passed to the observers currently subscribed to this event. Each observer inspects the event and acts accordingly.

Figure 10a shows the call graph of invoking the `repaint` method on the top-level window object. The object structure is traversed, using virtual calls to propagate the `repaint` operation to all the peer objects.



(a) Before specialization (b) After specialization

Figure 10: Call graph of `repaint()`, before and after specialization.

5.3 Specialization

First, the application is specialized to the way the composite objects are composed. The `repaint` method of the root `Window` object can thus refresh the entire application at once, without traversing the widget structure. Next, the application is specialized to the concrete peer objects being used. This transformation allows composite objects to directly manipulate their peer objects. Finally, the application is specialized to the concrete observer/observee relations of the application. An event now results in the actions that it implies being directly performed throughout the application.

Figure 10b shows the call graph of invoking the `repaint` method after specializing for the widget structure and a specific set of peers. Calls are made directly to the peer objects, without traversing the object structure.

A more complex application might manipulate its graphical appearance while running. However, such an application would often be divided into several independent units, that are configured individually. Specialization can be applied separately to each such unit.

6 Assessment

Program specialization is applicable when design patterns are not used in their full generality. Design patterns allow the same program parts to be used to implement different functionalities. Specialization can optimize such program parts when the exact functionality is fixed, either because it is explicit in the program or because it is made so using specialization classes.

On the contrary, those uses of design patterns whose features are completely exploited are not suitable for specialization. Highly dynamic programs that often reconfigure themselves are easy to write using design patterns, but are difficult to specialize. Specialization classes are only useful here when a fixed implementation can be selected outside of the critical regions of the program, since there is an overhead associated with switching between spe-

	JDK 1.2 JIT			HotSpot		
Benchmark	Normal	Spec.	Speedup	Normal	Spec.	Speedup
Bridge (mandelbrot)	2.582	2.583	1.00	6.458	5.682	1.14
Builder (matrix)	4.654	3.778	1.23	4.988	2.613	1.91
Strategy (image)	3.298	1.626	2.03	2.283	0.768	2.97
Iterator (member)	6.132	5.114	1.20	6.227	3.409	1.83

Table 1: Benchmark results (real time, in seconds)

cialized implementations. Knowledge of the degree of adaptability associated with each use of a design pattern is thus essential for using specialization patterns to optimize a program.

To illustrate the performance benefits of eliminating uses of design patterns by specialization, we consider a few benchmarks, based on the examples of Section 4. In practice, however, the improvement due to specialization can vary widely, depending on the number of specialization opportunities introduced by eliminating the abstraction barriers created by the use of design patterns.

For benchmarks, we use the builder design pattern to build matrices with sparse and dense underlying representations, the bridge design pattern to compute the Mandelbrot set using complex number arithmetic, and finally the strategy pattern to perform a number of different image processing tasks. In addition, the iterator example from Sections 1 and 3.3 is used to implement various set operations. The benchmarks have been done using Sun's JDK 1.2.1 JIT and HotSpot compilers on a 300MHz UltraSparc, ignoring the first iteration of each benchmark to minimize cache effects and ensure that all dynamic optimization is complete. The results are shown in Table 1.

The speedup due to specialization varies with the complexity of the adaptation taking place in the benchmark. The bridge benchmark only has a few, simple points of adaptation and is dominated by numerical computation, so the benefit due to simply specializing away the bridge ranges from non-existent to minor. The iterator and builder benchmarks have more points of adaptation, and so they benefit more from specialization. Last, the strategy benchmark has a single but critical point of adaptation, that can be completely eliminated using specialization, which greatly simplifies the program control flow.

7 Related Work

Program rewriting techniques can be used in place of program specialization to map uses of design patterns into efficient implementations, as shown by Turwé and De Meuter [30]. Here, a program transformation engine based on Prolog rewriting rules is used to perform architectural transformations before compilation. While their optimization technique is very different from partial evaluation, their overall approach can be unified with specialization patterns: for each design pattern, a specialization pattern can describe what rewriting rules give the best optimizations.

Templates in C++ allow the programmer to express static information about types and simple values, thus providing more information to the compiler. The information can be used to fix types in the program, and even to perform simple partial evaluation of integral values [31]. For example, rather than implementing the strategy pattern with a virtual call, the choice of strategy can by statically fixed using templates [16]. However, templates specialize on a class-by-class bases, and cannot specialize for the way objects are composed together, except when this is done statically in the program. In addition, explicit syntax is needed to express properties using templates, and source code must be manually duplicated to retain the generic behavior.

Many compilation systems implement generally applicable optimizations similar to those performed performed by program specialization, but without requiring user guidance. To reduce the complexity of performing analysis, simplified type inference algorithms such as Class Hierarchy Analysis are used [12], combined with profile information that guides speculative optimizations such as receiver-prediction [18, 20]. Since techniques such as inlining and specialization for types (customization [7] and method argument specialization[10]) can cause code explosion, the same profiling information is used to focus these optimizations on the critical parts of the program [11, 20]. The optimizations offered by such systems depend on the accuracy of the analyses and profiling system. As a result, the level of optimization is difficult to predict, and structural overheads are not easily detected. By contrast, specialization is parameterized by information provided by the programmer, and can produce source code that can be manually inspected for remaining inefficiencies before being shipped as a complete product.

Software architectures offer a global approach to organizing programs, by working within a framework [29]. A program written using a software architecture uses a generic infrastructure, which supports mechanisms to offer extensibility and modularity. However, there is a fundamental difference between design patterns and software architectures. A design pattern describes a design idea without being limited to a single, concrete implementation, whereas a software architecture gives a concrete implementation of an infrastructure without specifying the components that can be integrated to form a complete application. Marlet *et al.* have shown program specialization to be an effective tool for eliminating the architectural overheads of software architectures, automatically transforming a program written using a software architecture into an efficient implementation [24]. Since there is a concrete implementation for a given software architecture, specialization can be applied to a program written using this software architecture, without requiring guidance from the user.

8 Future Work

In this paper we have shown that a given design pattern provides enough structure to a program to systematically enable its optimization using program specialization. However, intertwining many design patterns may affect the specialization opportunities of the resulting program. To address this issue we are studying how the composition of design patterns impacts specialization opportunities, and are characterizing specialization patterns resulting from design pattern compositions.

The Java Beans framework is defined using standard Java constructs under certain constraints. Just like a given framework systematically introduces specific overheads, the Java Beans component architecture also introduces overheads into programs. Specialization can be automatically applied to optimize away these overheads. For example, the standard Java Beans event model can be specialized with an overall effect similar to the specialization of the strategy design pattern (Section 4.4). Concretely, we aim to completely automate the specialization process for the specific case of Java Beans, by automatically generating specialization classes.

With a more formal definition of design patterns, it is possible that user guidance of the specialization process could be greatly simplified. For example, if the source language had support for design patterns [5, 19, 21] or if the program were developed using a CASE tool that supports design patterns [25], specialization classes could be automatically generated for each use of a design pattern. These specialization classes would then precisely define the specialization capabilities of the resulting program.

9 Conclusion

Design patterns focus on how programs should be structured to offer features such as modularity and extensibility. However, this structuring is directly mapped into an implementation; features are directly implemented in terms of mechanisms that cause overheads at run time. Still, these overheads are predictable since they are inherent to each design pattern.

This paper introduces specialization patterns: an approach aimed at optimizing patterns of overheads identified in design patterns. This optimization process, based on program specialization, removes abstraction layers by exploiting information about object composition.

We have demonstrated the applicability of our approach to several kinds of design patterns (creational, structural, and behavioral). For each kind of design pattern, we have characterized specialization opportunities. Examples have been used to concretely show the effectiveness of program specialization in removing the overheads inherent to design patterns.

In effect, we have shown that program specialization can be used systematically to map programs developed using design patterns into efficient implementations. This mapping is guided by information provided by design patterns. As a result, we have extended the scope of design patterns: not only do they guide program development, but they also enable systematic optimization of the resulting programs.

Acknowledgements

We would like to thank Phillipe Boinot, Aino Cornils, and Gilles Muller for their helpful comments on this paper. We would also like to thank Miguel A. de Miguel and Peter Chang for timely completion of a prototype implementation of the C to Java translator

```
class Array implements MinimalCollection {
  Object []elements;
  int size;
  Array(int size) {
    this.size = size;
    this.elements = new Object[size];
  }
  public Object get(int n) { return elements[n]; }
  public int getSize() { return size; }
  Iterator iterator() {
    return new ArrayIterator( this );
  }
  ...other methods for implementing the MinimalCollection interface...
}
class ArrayIterator implements Iterator {
  Array array;
  int current, max;
  ArrayIterator( Array a ) {
    this.array = a;
    this.current = 0;
    this.max = a.getSize();
  }
  boolean hasNext() { return current<max; }
  Object next() { return array.get(current++); }
}
```

Figure 11: Relevant parts of `Array` and its iterator `ArrayIterator`.

A Iterator Example Implementation

Figure 11 shows those parts of the `Array` and `ArrayIterator` classes that are relevant to the iterator example shown in the introduction.

References

[1] G. Aigner and U. Hölzle. Eliminating virtual function calls in C++ programs. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–166, Lisbon, Portugal, June 1996. Springer.

[2] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.

[3] R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Orlando, FL, USA, June 1994. Technical Report 94/9, University of Melbourne, Australia.

[4] A. Bondorf. *Self-Applicable Partial Evaluation*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1990. Revised version: DIKU Report 90/17.

[5] J. Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, November 1996.

[6] B. Calder and D. Grunwald. Reducing indirect function call overhead in C++ programs. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 397–408. ACM Press, 1994.

[7] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, A dynamically-typed object-oriented programming language. In Bruce Knobe, editor, *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation (SIGPLAN '89)*, pages 146–160, Portland, OR, USA, June 1989. ACM Press.

[8] C. Consel. A tour of Schism: a partial evaluation system for higher-order applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 66–77, Copenhagen, Denmark, June 1993. ACM Press.

[9] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996.

[10] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. *ACM SIGPLAN Notices*, 30(6):93–102, June 1995.

[11] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: an optimizing compiler for object-oriented languages. In *OOPSLA' 96 Conference*, pages 93–100, San Jose (CA), October 1996.

[12] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of ECOOP'95*, Aarhus, Denmark, August 1995. Springer-Verlag.

[13] D. Detlefs and O. Agesen. Inlining of virtual methods. In ECOOP'99 [15], pages 258–278.

[14] K. Driesen and U. Hölzle. The direct cost of virtual function calls in C++. In *OOPSLA'96 Conference Proceedings*, volume 31, 10 of *ACM SIGPLAN Notices*, pages 306–323, New York, October6–10 1996. ACM Press.

[15] *Proceedings of the European Conference on Object-oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, Lisbon, Portugal, June 1999.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[17] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In *OOPSLA'95 Conference Proceedings*, volume 30, 10, pages 108–123, New York, October 1995. ACM Press.

[18] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In *Proceedings of OOPSLA '95*, pages 108–123, Austin, TX, October 1995.

[19] G. Hedin. Language support for design patterns using attribute extension. In J. Bosch and S. Mitchell, editors, *ECOOP'97 Workshop Reader*, volume 1357 of *Lecture Notes in Computer Science*, pages 137–140. Springer-Verlag, June 1998.

[20] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 326–336, New York, NY, USA, June 1994. ACM Press.

[21] S. Krishnamurthi, Y. Erlich, and M. Felleisen. Expressing structural properties as language constructs. In S.D. Swierstra, editor, *Programming Languages and Systems, 8th European Symposium on Programming (ESOP'99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 258–272. Springer-Verlag, 1999.

[22] J.L. Lawall and G. Muller. Efficient incremental checkpointing of Java programs. Research Report 3810, INRIA, Rennes, France, November 1999.

[23] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.

[24] R. Marlet, S. Thibault, and C. Consel. Mapping software architectures to efficient implementations via partial evaluation. In *Conference on Automated Software Engineering*, pages 183–192, Lake Tahoe, Nevada, November 1997. IEEE Computer Society.

[25] T.D. Meijler, S. Demeyer, and R. Engel. Making design patterns explicit in FACE, a framework adaptive composition environment. In M. Jazayeri and H. Schauer, editors, *Proceedings ESEC/FSE '97*, volume 1301 of *Lecture Notes in Computer Science*, pages 94–110. Springer-Verlag, September 1997.

[26] G. Muller and U. Schultz. Harissa: A hybrid approach to Java execution. *IEEE Software*, pages 44–51, March 1999.

[27] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA '94 Conference Proceedings*, volume 29:10 of *SIGPLAN Notices*, pages 324–324. ACM, October 1994.

[28] U. Schultz, J. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In ECOOP'99 [15], pages 367–390.

[29] M. Shaw and D. Garlan. *Software Architecture.* Prentice Hall, 1996.

[30] T. Tourwe and W. De Meuter. Optimizing object-oriented languages through architectural transformations. *Lecture Notes in Computer Science*, 1575:244–258, 1999.

[31] T. L. Veldhuizen. C++ templates as partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 13–18, San Antonio, TX, USA, January 1999. ACM Press.

[32] E.N. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In *OOPSLA'97 Conference Proceedings*, pages 286–300, Atlanta, USA, October 1997. ACM Press.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399