Specialization Patterns

Ulrik P. Schultz, Julia L. Lawall*, and Charles Consel†

Compose Group, IRISA / INRIA, University of Rennes I
Campus Universitaire de Beaulieu, F-35042 Rennes Cedex, France
E-mail: {ups,jll,consel}@irisa.fr

Abstract

Design patterns offer many advantages for software development, but can introduce inefficiency into the final program. Program specialization can eliminate such overheads, but is most effective when targeted by the user to specific bottlenecks. Consequently, we propose that these concepts are complementary. Program specialization can optimize programs written using design patterns, and 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 using the specialization classes framework, developed previously. In our experiments, such specialization significantly improves performance.

1. Introduction

Design patterns, as presented by Gamma *et al.* [17], 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 can evolve over time. Because design patterns are well-documented, their use simplifies the understanding of programs constructed from many independent units. The 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. Consider a typical use of the *iterator* design pattern [17], which separates the 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 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. 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. Replacing the generic uses of the iterator pattern (underlined in the method definition) by direct accesses to the underlying data structure 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, specializing the program to this representation before execution improves efficiency. Manual specialization is error-prone, and introduces excessive program-maintenance overhead. Recently, automatic program specialization has

*Author's current address: DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark.

†Author's current address: LaBRI / ENSERB, 351 cours de la Libération, F-33405 Talence Cedex, France.

[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.

been shown to be effective in the context of 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 eliminate 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. Profiling can help, but 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 that form patterns of opportunities for specialization. We propose the use of *specialization patterns* as a complement to design patterns, to describe when, how, and where 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 analyze the overheads systematically introduced by the use of design patterns.
- We describe how to systematically apply program specialization to eliminate these overheads, automatically mapping a modular, generic implementation into a monolithic, efficient one.
- We define specialization patterns for two well-known design patterns: the builder pattern and the strategy pattern.
- We provide several examples of how specialization can optimize uses of design patterns, and show 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* [31] and *how* to specialize in the form of a prototype Java specializer [28]. Here, we address the key issue of selecting *where* to specialize.

The rest of this paper is organized as follows. First, Section 2 describes our perspective on design patterns. Section 3 then explains program specialization. Section 4 describes specialization of design patterns by means of specialization patterns. Then, Section 5 assesses the application of program specialization to uses of design patterns. Finally, Section 6 presents related work, Section 7 discusses future work, and Section 8 concludes.

2. Design Patterns

Inheritance and delegation are fundamental to the structure of adaptable object-oriented systems. Inheritance allows a new class to extend or override the behavior of an existing class. Delegation defines a new class in terms of references through abstract interfaces to existing objects. In this section, we describe how general-purpose design patterns in the style of Gamma *et al.* [17] organize the use of these basic adaptation mechanisms, and identify the overheads that the use of design patterns can introduce.

2.1. The role of design patterns

The use of inheritance and delegation obscures the relationship between program components. Design patterns address this issue. 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.

2.2. Overheads introduced by design patterns

Objects interact using method invocations, which can be implemented either as direct calls or virtual calls. Virtual calls defeat branch prediction (and thereby instruction pipelining) and inhibit inlining, blocking subsequent traditional intra-procedural compiler optimizations [6, 15]. Thus, many compilers go to great lengths to replace virtual calls by direct calls [1, 13, 14, 26]; some even using constrained specialization techniques [11, 20], such as customization [7]. Even so, virtual calls can only be completely eliminated when static analysis can safely determine that the class of the receiver object never changes.

Most design patterns distribute functionality among objects that cooperate through abstract interfaces, simplifying software adaptation into a reorganization of the objects that constitute the program. This software structure and the potential to reorganize the program objects at any moment implies that objects generally interact using virtual calls. However, design patterns may often provide more adaptability than is needed within a specific phase or run of a program. Thus, propagating extra information about the

identity of objects throughout the program may allow replacement of virtual calls by direct calls, beyond what can be expected from an optimizing compiler.

These observations are illustrated by the the benchmarks reported in Section 5. 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 optimizes a program fragment based on information about the context in which it is used, thus generating a dedicated implementation. 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 adapts a program to known (*static*) information about its execution context, leaving behind only the program parts controlled by unknown (*dynamic*) data. Partial evaluation has been extensively investigated for functional [4, 8], logic [23], and imperative [2, 3, 9] languages, and has been recently extended to Java, by Schultz *et al.* [28], using C as an intermediate language. Since then, we have extended this approach to automatically produce specialized Java source programs, thus implementing a complete Java-to-Java specializer[2].

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.

Partial evaluation relies on a human programmer to detect specialization invariants and to direct specialization towards critical parts of the program. A program part 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 to ensure that the specialized code is used only when the invariants are satisfied [31]. 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. However, because of the need to react at run time when specialization invariants are invalidated, the use of specialization classes does add some inefficiency, and applying a specialization class in the wrong place can eliminate all benefits due to partial evaluation. Thus, specialization classes are only useful here when a fixed implementation can be selected outside of the critical regions of the program.

[2]See the JSpec homepage `http://www.irisa.fr/compose/jspec` for more details on the implementation and its availability.

Specialization example

As an example, let us revisit the example of the Iterator, described in Section 1. We can 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` object 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, as follows:

```
specclass Member_Array specializes Set {
  Array coll;                 // field has type Array
  boolean member( Object o ); // specialize member
}
```

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

```
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, depending on the choice of Java compiler (see Section 5 for details).

Specialization with respect to one invariant can often trigger other specialization opportunities. For example, 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. Nevertheless, not all specialization invariants are beneficial. For example, unrolling the loop might lead to code explosion. Specializing with respect to the type of elements of the `Array` object might cause generation of too many specialized

Name: The name of the associated design pattern.
Description: A short description of the design pattern.
Extent: The minimal program slice that is relevant when optimizing a use of the design pattern.
Overhead: Possible overheads associated with a 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

variants, if the member operation is applied to sets having too many different kinds of object values. These issues suggest that the use of specialization must be controlled, and requires some insight into the overall program structure. This insight can be derived from knowledge of the program's use of design patterns.

4. Specialization Patterns

Design patterns facilitate communication of design ideas by encapsulating a characterization of a common problem and its solution. Specialization patterns complement 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 to using a particular design pattern, and documents how to use specialization to eliminate these overheads. 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 specialization will be beneficial.

Specialization patterns are based on the template of Figure 1. The template includes sections that relate the specialization pattern to the design pattern, criteria for judging when it is worthwhile to specialize a use of the design pattern, detailed instructions for performing specialization, and a specialization example. To illustrate the problems that can be addressed by specialization patterns, we now identify the specialization opportunities provided by *creational*, *structural*, and *behavioral* design patterns [17], 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. Nevertheless, this abstraction barrier implies that the products must be accessed using virtual calls, which blocks optimization.

Memory allocation and object initialization dominate the cost of object creation, so specializing only to eliminate virtual calls associated with the creation process is unlikely to significantly optimize a program. Thus, the parts of the program where the products are used should also be specialized with respect to the concrete type of each product. Such specialization permits direct access to the products, enabling ordinary intra-procedural optimizations. However, such specialization is only effective when the specializer can determine how the products are manipulated after they have been created. This is outside the part of the program covered by the creational pattern, so a specialization pattern can only give limited information on when it is worthwhile to specialize.

Example: builder pattern

Figure 2a 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 returns 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 `i`'th element of the list just produced. Figure 2b shows the concrete builder implementation `LinkedListBuilder`, which produces linked lists of type `LList`. The definitions of `AbstractList` and `LList` are given in Figure 3.

Figure 4 defines the specialization pattern for the builder pattern. The specialization pattern suggests to specialize the program fragment with regards to a concrete builder implementation. Accordingly, the specialization class of Figure 2c specifies that the method `f` of the class `Main` should be specialized with respect to the `LinkedListBuilder` implementation, and in addition for a specific list index. In the specialized program (Figure 2d), virtual calls

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

(a) Use of builder through interface

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

(b) Concrete builder for linked lists

```
specclass Main_LL specializes Main {
 LinkedListBuilder b;
 void f(int i), i==1;
}
```

(c) Declaration of specialization to the `LinkedList-Builder` builder

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

(d) Result of specialization

Figure 2. Specializing a use of the builder pattern.

```
interface AbstractList {
  Object lookup( int index );
  ... other methods defining AbstractList ...
}
class LList implements AbstractList {
  Object elm; LList next;
  LList( Object e ) { this.elm=e; }
  Object lookup( int i ) {
    return i==0?elm:next.lookup(i-1);
  }
  ... other methods implementing AbstractList ...
}
```

Figure 3. List data structure.

have been replaced by direct data-structure manipulations.[3] Specialization replaces the virtual calls through the `List-Builder` interface by direct calls, which are inlined during post-processing.

Specialization to a single concrete implementation permits the products to be accessed directly as long as they are not manipulated in a dynamic way. In the example, the product is used in a fixed way, and the virtual call to `lookup` has been replaced by a specialized version of its concrete definition in the `LList` class. If desired, the method `X.something` can also be specialized, adapting it to the concrete value stored as the second element of the `LList` object. Had the product been manipulated under the control of dynamic data, the benefits of specialization would have been negligible.

[3] In all of the examples shown in this paper, the specialized program has been produced automatically, and then resugared for readability.

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 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, allowing the programmer to combine individual objects that respect a common interface into a compound object that behaves in a new way. 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.

A program that builds and traverses an object structure can be specialized to a specific layout of this structure. Specialization permits the objects to interact directly, and combines all of the basic operations on the structure into a single method, facilitating optimization. If the structure is not modified after its creation, the methods that traverse it can be directly specialized to its layout. When the structure is modifiable, specialization classes can be used to describe layouts that are of interest. As always, specialization classes introduce overheads, so the latter approach

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, thus separating 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. Specialization can also be applied to any subsequent use of the product structure.

Overhead: Separation of the type of the product from the client means that product must be accessed using virtual calls.

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. 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.

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

Example: See Figure 2 and explanation in text.

Figure 4. Builder specialization pattern

might not be beneficial if the structure changes too often. Because specialization of a structural pattern can generate code having size proportional to the size of the object structure, specialization should be applied with caution to avoid code explosion.

The strucural patterns bridge, adapter, composite, decorator, facade, and proxy all build structures from objects hidden behind generic interfaces, so uses of these patterns are good targets for specialization. Specialization is guaranteed to simplify the program when the structure does not change or when it can be encapsulated using specialization classes. The flyweight pattern optimizes memory usage by sharing objects, and cannot be specialized in any obvious way.

For the lack of space, we do not include a specialization pattern example for structural patterns. We refer the interested reader to the technical report [27] for details.

4.4. Behavioral Patterns

Behavioral patterns abstract over the control flow, providing generic ways of parameterizing behavior. They separate different aspects of an overall behavior, making it possible to construct new behaviors by composing individual objects or classes. Every time the collaborating objects 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 we can guarantee benefits from specialization only for specific patterns. 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 cannot be determined by the specializer, the behavioral pattern cannot in general be specialized.

Example: strategy pattern

Figure 5a 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 pixel of the image. Figure 5b defines two such single-pixel operations: `Scale`, which scales a pixel (thereby changing its brightness), and `RedOnly`, which discards all but the red component.

Figure 6 defines the specialization pattern for the strategy pattern. The specialization pattern suggests to specialize for a specific algorithm. The specialization class `ScaleByTwoProcess` (Figure 5c) 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 5d), by eliminating the virtual call to the strategy method, inlining the call, and propagating the known scal-

```
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 {
 ...keep only red component...
}
```

(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 5. Specializing a use of the strategy pattern.

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 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 (as in the example), allowing this method to be specialized 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 repeatedly, medium when used a few times, low when used only once.

Example: See Figure 5 and explanation in text.

Figure 6. Strategy specialization pattern

ing value.

Other behavioral patterns

As is the case for the strategy pattern, precise specialization patterns can be given for 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 completely fixed. The template method pattern 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 general way.

5. Assessment

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

		JDK 1.2 JIT			HotSpot		
Benchmark	LOC	Normal	Spec.	Speedup	Normal	Spec.	Speedup
Builder (matrix)	628	4.654	3.778	1.23	4.988	2.613	1.91
Bridge (mandelbrot)	225	2.582	2.583	1.00	6.458	5.682	1.14
Strategy (image)	392	3.298	1.626	2.03	2.283	0.768	2.97
Iterator (member)	467	6.132	5.114	1.20	6.227	3.409	1.83

Table 1. Benchmark results (time in seconds)

abstraction barriers created by the use of design patterns.

For benchmarks, we use the builder pattern to build matrices with sparse and dense underlying representations, the bridge pattern to compute the Mandelbrot set using complex arithmetic, and the strategy pattern to perform a number of different image processing tasks. In addition, the iterator example from Sections 1 and 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, including the number of lines of code (LOC) of each benchmark.

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 is negligable. 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.

6. Related Work

Turwé and De Meuter use program rewriting techniques to develop a program transformation engine based on Prolog that performs architectural transformations before compilation [30]. While their optimization technique is very different from partial evaluation, their 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. For example, rather than implementing the strategy pattern with a virtual call, the choice of strategy can be statically fixed using templates [17]. However, templates specialize on a class-by-class basis, and cannot specialize for the way objects are composed together, except when the object composition is fixed in the program. In addition, explicit program syntax is needed to express specialization using templates, and source code must be manually duplicated to retain the generic behavior.

Many compilers implement generally applicable optimizations similar to those 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 [13], 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 [11]) can cause code explosion, the same profiling information is used to focus these optimizations on the critical parts of the program [12, 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.

Where a design pattern can be said to describe a micro-architecture that is implemented specifically for the program being developed, a software architecture defines a program-wide recurring code organization [29]. Marlet *et al.* have shown that program specialization can automatically eliminate the flexibility overhead of software architectures and generate an efficient implementation [24]. Specialization of a collection of programs written according to a software architecture is often simpler than specializing a collection of programs written using the same design patterns, since all programs based on the same software architecture implementation can be specialized using the same specialization strategy.

7. 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. 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 component architecture is defined using standard Java constructs under certain constraints. Just as a framework can systematically introduce specific overheads, the Java Beans component architecture also introduces overheads into programs. Specialization can be automatically applied to optimize away these overheads. 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, when the source language has support for design patterns [5, 19, 21] or when the program is developed using a CASE tool that supports design patterns [10, 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.

8. 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 because 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 delegation.

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.

```
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 MinimalCollection methods ...
}
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 7. `Array` **and** `ArrayIterator`**.**

Acknowledgements

We would like to thank Philippe Boinot, Aino Cornils, Renaud Marlet, and Gilles Muller for their helpful comments on this paper. We would also like to thank Miguel A. de Miguel and Peter Chang for their help in the timely completion of our complete Java specializer.

A. Example Implementation Details

Figure 7 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. 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*, Nov. 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 B. 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. 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, Feb. 1996.

[10] A. Cornils and G. Hedin. Statically checked documentation with design patterns. In R. Mitchell, J. Jezequel, J. Bosch, B. Meyer, A. C. Wills, and M. Woodman, editors, *Proceedings of TOOLS Europe 33*, pages 419–431, Mt. St. Michel, France, June 2000. IEEE Computer Society Press.

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

[12] 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), Oct. 1996.

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

[14] D. Detlefs and O. Agesen. Inlining of virtual methods. In ECOOP'99 [16], pages 258–278.

[15] 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, NY, USA, Oct. 1996. ACM Press.

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

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

[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, Oct. 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. 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. Lawall and G. Muller. Efficient incremental checkpointing of Java programs. In *Proceedings of the International Conference on Dependable Systems and Networks*, New York, NY, USA, June 2000. IEEE. To appear.

[23] J. Lloyd and J. 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, NV, USA, Nov. 1997. IEEE Computer Society.

[25] T. 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, Sept. 1997.

[26] 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, Oct. 1994.

[27] U. Schultz, J. Lawall, and C. Consel. Specialization patterns. Research Report 3853, INRIA, Rennes, France, Jan. 2000. Extended version.

[28] U. Schultz, J. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In ECOOP'99 [16], 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. In S. Jähnichen, editor, *Compiler Construction - 8th International Conference, CC '99*, volume 1575 of *Lecture Notes in Computer Science*, pages 244–258, Amsterdam, The Netherlands, Mar. 1999. Springer-Verlag.

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