A Comparative Study of Techniques to Write Customizable Libraries

Baris Aktemur
University of Illinois at Urbana-Champaign, USA
aktemur@illinois.edu

Sam Kamin
University of Illinois at Urbana-Champaign, USA
kamin@illinois.edu

ABSTRACT

Code libraries are characterized by feature-richness — and, consequently, high overhead. The *library specialization problem* is the problem of obtaining a low-overhead version of library code when the rich feature set is not needed. A version of that problem is this: Given a class with certain core functionality and some "optional" features, how can we offer the client a menu of features such that the specific class answering this request is unencumbered by fields or computation not needed for the requested features? This paper presents a comparative study of several approaches to this version of the library specialization problem. We evaluate object-oriented programming, feature-oriented programming, colored IDE, aspect-oriented programming, C-style preprocessor directives, and fragment-oriented program generation. We find that all of these techniques have shortcomings.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*; D.3.4 [**Programming Languages**]: Processors—*Code generation*

General Terms

Design, Languages

Keywords

library specialization, feature-oriented programming, program generation

1. LIBRARY SPECIALIZATION

Libraries come enriched with many features and thus carry high overhead. The *Library specialization problem* is the problem of obtaining a low-overhead version of library code when only a subset of the features is needed. To put it differently, the question is this: Given a class with certain core functionality, some "optional" features, and the user's selection of features, how can we make the class free of fields and computation incurred by unwanted features? This problem is important especially in resource-constrained settings.

In this paper we provide a comparative evaluation of several techniques that can address this problem: Object-Oriented Programming (OOP), Feature-Oriented Programming (FOP), Colored IDE (CIDE), Aspect-Oriented Programming (AOP), C-style preprocessor directives (PP), and Fragment-Oriented Program Generation (FOPG). We find that OOP, FOP and AOP have problems in handling *fine-grained* features that affect code in the middle of a method or that depend on local variables. The importance of such features is emphasized in [12]; see section 6. While the other techniques provide better support for fine-granularity, they have problems in other dimensions.

Our method is experimental. We posit a specific specialization problem and attempt to solve it with various techniques. The problem is inspired by the C# collection library C5 [14], which provides classes such as list, bag, and tree, enriched with advanced features (e.g. updatable&slidable views, snapshots, event listeners).

In C5, *every* collection class comes with all these features (or all that make sense for the particular type of collection), and pays the price in additional space and time overhead. There is no such thing as a "plain linked-list." That is what gives rise to the library specialization problem.

We base our study on a simplified example inspired by C5. This is a collection library with two basic implementations — linked-list and array-list — and two optional features:

View feature: The classes provide the ability to create views (i.e. sublists) of them. A view keeps a reference to its underlying list. A regular list keeps references of its views.

Counter feature: The classes contain a field, `counter`, to keep track of the number of list operations performed on the list, such as add, reverse, etc. A method call on a view causes both the view's and its underlying list's counters to be incremented.

The View feature comes directly from C5. The Counter feature we chose is the simplest feature we could think of that still illustrates some aspects of interest for this problem.

We first describe the two implementations of the list in Java-like syntax. The base versions with no optional features are given in Figure 1. We refer to these codes as the *core* functionality. The fully-loaded implementations are shown in Figures 2 and 3, respectively. In each, the lines that have feature-related logic are annotated with $\mathcal{V}$ and $\mathcal{C}$

SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.

```
1   interface List {
2     void add(Object item);
3     ...
4     void reverse();
5   }
6
7   class ArrayList implements List {
8     Object[] items;
9     int size;
10
11    void add(Object item) {
12      if (size == items.length)
13        expand();
14      items[size] = item;
15      size++;
16    }
17    ...
18    void reverse() {
19      int length=size/2;
20      int end=size-1;
21      for (int i=0; i < length; i++) {
22        Object swap = items[i];
23        items[i] = items[end-i];
24        items[end-i] = swap;
25      }
26    }
27  }
28
29  class LinkedList implements List {
30    Node first,last; // This is a doubly linked list
31    int size;
32
33    void add(Object item) {
34      Node newnode = new Node(item);
35      newnode.next = last;
36      newnode.prev = last.prev;
37      last.prev.next = newnode;
38      last.prev = newnode;
39      size++;
40    }
41    ...
42    void reverse() {
43      Node a = first.next, b = last.prev;
44      for(int i=0; i<size/2; i++) {
45        Object swap = a.item;
46        a.item = b.item; b.item = swap;
47        a = a.next; b = b.prev;
48      }
49    }
50  }
51  class Node {
52    Node prev, next;  Object item;
53    Node(Object i){ item = i; }
54  }
```

Figure 1: The List interface and the plain ArrayList and LinkedList classes.

markers. C5 has similar ArrayList and LinkedList classes, but extended with several advanced features.

In principle, the user might want any of eight possible classes: each implementation with each subset of the above two features. We have shown four of these. Our question is how to describe all eight (keeping in mind that the number of classes grows exponentially with the number of features).

We must state at the outset that we are interested in manual techniques, where the programmer explicitly writes code to create specialized libraries. Automated techniques, such as backwards slicing, to filter out the unused features, would be most convenient. But such techniques always have their limitations: can they handle aliasing, unknown or native methods, and other complex language features? Furthermore, they necessarily require a whole-program analysis, which is often impossible to provide in an object-oriented

```
1     class ArrayList implements List {
2       Object[] items;
3       int size;
4   C   int counter = 0;
5   V   ArrayList underlying;
6   V   ArrayList[] views;
7   V   int offset;
8
9       void add(Object item) {
10  C     counter++;
11  CV    if(underlying != null) underlying.counter++;
12  V     if ((underlying != null ? underlying : this).size
13              == items.length)
14          expand();
15  V     items[offset+size] = item;
16  V     (underlying != null ? underlying : this).size++;
17  V     fixViewsAfterInsertion();
18      }
19  V   void fixViewsAfterInsertion() { do the fixing }
20      ...
21      void reverse() {
22  C     counter++;
23  CV    if(underlying != null) underlying.counter++;
24        int length=size/2;
25  V     int end=offset+size-1;
26        for (int i=0; i < length; i++) {
27  V       Object swap = items[offset+i];
28  V       items[offset+i] = items[end-i];
29          items[end-i] = swap;
30        }
31  V     disposeOverlappingViews();
32      }
33  V   void disposeOverlappingViews() { ... }
34    }
```

Figure 2: The ArrayList class enriched with two features. $\mathcal{C}$ and $\mathcal{V}$ denote the lines added or modified by the Counter and View features, respectively.

setting. For this study we consider methods that can be applied by a library writer right now.

The most naive approach would be to provide all the possible implementations. But, as noted above, the number of implementations is exponential in the number of features, and a substantial amount of code would be duplicated; we clearly want to avoid this.

Organization of the paper: We first provide a terminology in Section 2 to help us in the evaluation. This is followed by introduction of the evaluation criteria in Section 3. Sections 4 through 9 apply and evaluate the six techniques listed above. We finally conclude and give future directions in Section 10.

Throughout this paper, we use a Java-like syntax. We ignore visibility modifiers (e.g. public, private) for brevity. We use sans serif font when referring to source code in text.

2. CLASSIFICATION OF FEATURE CODE

When a feature is included, it affects the core code in several places. Any such effect is called an *impact*. We provide the list below to classify an impact. The referenced line numbers are from Figure 3.

Kind:

Code adder: The impact brings in a new piece of code, without changing the original code. e.g. lines 4, 35.

Modifier: The impact modifies a part of the original code. e.g. line 17 (compare to line 39 in Figure 1).

Position:

Member: The code introduced by the impact is a field or a

```
1     class LinkedList implements List {
2       Node first,last;
3       int size = 0;
4   C   int counter = 0;
5   V   LinkedList underlying;
6   V   LinkedList[] views;
7   V   int offset;
8
9       void add(Object item) {
10  C     counter++;
11  CV    if(underlying != null) underlying.counter++;
12        Node newnode = new Node(item);
13        newnode.next = last;
14        newnode.prev = last.prev;
15        last.prev.next = newnode;
16        last.prev = newnode;
17  V     (underlying != null ? underlying : this).size++;
18  V     fixViewsAfterInsertion(newnode);
19      }
20  V   void fixViewsAfterInsertion(Node newnode) {
21  V     do the fixing    // uses offset
22  V   }
23      ...
24      void reverse() {
25  C     counter++;
26  CV    if(underlying != null) underlying.counter++;
27
28  V     foreach view in views
29  V       check and arrange the position of view for reversal
30
31        Node a = first.next, b = last.prev;
32        for(int i=0; i<size/2; i++) {
33          Object swap = a.item;
34          a.item = b.item; b.item = swap;
35  V       mirrorViewSentinels(a,b,i);
36          a = a.next; b = b.prev;
37        }
38  V     if(size%2 != 0) mirrorViewSentinels(a,b,size/2);
39      }
40  V   void mirrorViewSentinels(Node a, Node b, int i) {
41  V     do the mirroring
42  V   }
43    }
```

Figure 3: The LinkedList class enriched with two features. $\mathcal{C}$ and $\mathcal{V}$ denote the lines added or modified by the Counter and View features, respectively.

method. Therefore it can be placed anywhere at the class member level. e.g. line 4.

Prefix/Suffix: The code introduced by the impact is either before or after a method body. Hence, only newly introduced code can be a *prefix/suffix*. e.g. line 10.

Interleaved: The impact is located inside a method body. e.g. line 35.

Dependence:

Non-dependent: The free variables of the impact are only the fields of the core functionality or the fields introduced by other impacts of the same feature. e.g. line 17.

Local-dependent: The impact refers to variables that are locally defined in the surrounding method of the impact's location. e.g. lines 35 and 38.

Feature-dependent: The impact refers to variables that are defined by other features. e.g. line 11.

These impacts have varying degree of complexity. Moreover, not every combination is meaningful. For each approach we evaluate, we show which impacts cause difficulty.

3. EVALUATION CRITERIA

We would like a solution to the library specialization problem to have several properties. We will evaluate the approaches based on the following criteria:

Expressibility: Can any impact be expressed? Does handling an impact require extra fields or method calls?

Modularity: Can the features and the core code be separated to their own modules? How much are the features and the core code isolated from the rest of the library?

Reusability: Can a feature be reused on another core code, if appropriate? Note that modularity is necessary to have reusability.

Extensibility: How easy is it to add a new feature?

Binary operability: Many libraries are shipped to users as binaries for security reasons. Can the technique work at binary level, without requiring the existence of the source?

We will make a note when the technique we are evaluating has a problematic issue not covered by the list above.

We view expressibility as the primary criterion, because it asks whether the job can be done: can the method produce the correct (i.e. minimal) code for the given subset of features? It is not enough to say, the given method produces the same functional effect as what we would like — the version with all features already accomplishes that! Furthermore, we take it as given that if a method is unable to satisfy this goal on our example, then this shortcoming will become even more pronounced on larger examples. On the other hand, a method may be capable of producing *nearly* the correct code, and possess distinct advantages in the other criteria; that is, expressibility cannot be the *only* criterion.

4. OBJECT-ORIENTED PROGRAMMING

Object-oriented programming languages are designed for modularity, and inheritance is a way of adding or modifying functionality non-destructively. Yet OOP cannot satisfactorily solve the problem, for reasons that are widely known.

A natural attempt is to start with the plain ArrayList and LinkedList classes, as given in Figure 1, and add the features. Assuming a single-inheritance model, one can subclass the plain classes and add the Counter and View features, resulting in two new subclasses for each plain class.

A problem here is that features are "embedded" in the subclasses and hence are not reusable. Keeping the feature impacts in a separate module (i.e. a class) and using them via aggregation increases reusability. However, aggregation requires adding extra fields. This conflicts with the main motivation of the library specialization problem. Multiple inheritance gets around this problem by allowing extending both the plain class and the feature. Nonetheless, multiple inheritance has its own well-known problems [20].

The second problem of OOP is that we still need to write a class for each viable subset of the features — potentially as many as 2^n classes.

The third problem is about expressibility. Because the extension mechanism in OOP is method overriding, expressing *interleaved* or *modifier* impacts requires introducing artificial method calls so that these methods can be overridden appropriately to achieve the behaviour of the impact. Similarly, *local-dependent* impacts require adding artificial methods to make the local variables accessible to the subclass.

In contrast to these problems, OOP can make features reusable via generics. Note that the fields of the View feature in ArrayList and LinkedList classes are the same except their types. By using generics, we can parameterize the View feature on the type of the fields. The feature can then be used for both ArrayList and LinkedList by passing the appropriate type name.

Evaluation of OOP

We base our evaluation on the single inheritance model.

Expressibility: Interleaved, local-dependent and modifier impacts are problematic. Attempting to improve feature reuse via aggregation necessitates additional fields. The number of classes that need to be implemented is exponential in the number of features.

Modularity: Limited. It can be improved by using aggregation instead of inheritance but at the price of sacrificing expressibility. Modifier impacts decrease isolation of the core code because of introduction of artificial method calls as "hook" points.

Reusability: Limited. Again, aggregation can be used for improvement at the price of expressibility loss. Support for generics provides better reusability.

Extensibility: Addition of a new feature may require code duplication because of the exponential growth problem.

Binary operability: Supported.

5. FEATURE-ORIENTED PROGRAMMING

The library specialization problem is a favorite and well-studied case in Feature-Oriented Programming (FOP) [4, 2, 17, 21, 12]. The main idea is to modularize the core functionality, and extend it with features in a step-wise fashion.

An attempt to add the Counter feature to ArrayList and LinkedList using Prehofer's FOP [17] is shown in Figure 4. In this approach, features and the core code are defined in their own modules. How features interact with core functionality and other features is defined by writing *lifters* which specify the impacts of features. It is assumed that features in general interact with each other in a pair-wise fashion. This keeps the number of lifters quadratic, instead of exponential, in terms of the number of features. Prehofer gives two translations based on inheritance and aggregation, to convert a feature-oriented program to plain OOP.

As illustrated by this example, FOP improves feature modularity and reuse over OOP. At its essence, however, FOP's mechanism to extend code with a feature is the same as OOP's method overriding mechanism. As a result, although *prefix/suffix* and *code adder* impacts are expressed straightforwardly, FOP suffers from the same problems related to the *interleaved* or *local-dependent* impacts as discussed in the OOP section. Therefore we omit the implementation of the View feature.

Batory's FOP approach is based on *mixin-layers* where each layer is a *refinement* of the upper layer [3, 2]. A refinement may override existing methods in classes or add new methods and fields. We do not give a proposed solution because of space concerns.

Including both features requires implementation of another lifter or layer, depending on the approach being used.

Traits [20, 19] is another structuring mechanism potentially applicable to library specialization. However, for this application, it is very similar to FOP; modularity and reuse are improved as compared to OOP, but the same expressibility problems exist. Hence we omit a solution attempt.

```
feature Counter {
  int counter = 0;
  void inc() { counter++; }
}

feature Counter
lifts ArrayList{
  void add(Object ob){
    this.inc();
    super.add(ob);
  }
  ...
  void reverse(){
    this.inc();
    super.reverse();
  }
}

feature Counter
lifts LinkedList{
  void add(Object ob){
    this.inc();
    super.add(ob);
  }
  ...
  void reverse(){
    this.inc();
    super.reverse();
  }
}
```

Figure 4: Defining the Counter feature in Prehofer's FOP.

Evaluation of FOP

Expressibility: Interleaved, local-dependent and modifier impacts are problematic due to the extension mechanism being the same as OOP's method overriding; extra method calls may be needed.

Modularity: Better modularity than OOP is provided for both the features and the core code.

Reusability: It is possible to reuse a feature when it is sensible. The Counter feature in Figure 4 is an example.

Extensibility: A new feature can be added by defining new lifters or by implementing a new mixin-layer.

Binary operability: Depends on the system.

6. COLORED IDE (CIDE)

In [12], Kästner et al. take a legacy code (Berkeley DB) with the goal of extracting out the features to make the application customizable. They find that features having fine-granularity — meaning the impacts modify the core code at the expression or statement level — are very common, and they identify the inability of OOP and FOP to express these impacts. To address this challenge, they develop the Colored Integrated Development Environment (CIDE), where feature impacts can be marked manually with a particular color in the source code. The color annotations are used as a mask to include or exclude impacts. A type system to guarantee type-safety of the produced code is also given [11]. A sample coloring for the reverse() method is below, where medium gray denotes the Counter feature, light gray denotes View, and dark gray is their overlap.

```
void reverse() {
  counter++;
  if(underlying != null) underlying.counter++;
  foreach view in views
    check and arrange the positions of views
  Node a = first.next, b = last.prev;
  ...
}
```

Evaluation of CIDE

Expressibility: Support for modifier impacts is not provided. Other impacts can be expressed.

Modularity: All the annotations are made on the original source code. Hence, there is no modularity.

Reusability: Not supported as there is no modularity.

Extensibility: Access to the source code is required.

Binary operability: Not supported. Requires the IDE and the source code.

7. ASPECT-ORIENTED PROGRAMMING

We now consider Aspect-Oriented Programming (AOP) to address the library specialization problem. We use AspectJ [1], the most popular AOP language. An implementation of the Counter feature is below.

```
interface CounterI{}

abstract aspect CounterAspect<T extends List> {
  // introduce the new field
  int CounterI.counter = 0;
  declare parents: T implements CounterI;

  // match method executions of T
  pointcut methodExec(T self) : this(self) &&
    (execution(* T.add(..)) ||
     execution(* T.reverse(..)) || and other methods);

  // increment the counter before a method execution
  before(T s):methodExec(s){((CounterI)s).counter++;}
}
aspect LLCounter extends CounterAspect<LinkedList>{}
aspect ALCounter extends CounterAspect<ArrayList>{}
```

We first implement the feature as an abstract aspect parameterized on type T, a subtype of List. T is then made a subtype of an interface to which the counter field is introduced by an *inter-type member declaration*[1]. Appropriate methods of T are advised to increment the counter before execution. The Counter feature is added to the list classes by extending the abstract aspect with concrete parameters. Using a generic aspect allows for modularization and reuse of the Counter feature. When the same code piece goes into several places that can be expressed with a single pointcut, it is called *homogeneous advising* [6]. The Counter feature is an example of this. AOP is very effective for this case.

The code to enrich LinkedList with View is given in Figure 5. We omit the case for ArrayList since it is similar. Better reuse can be obtained for the common parts using a generic aspect as done for the Counter feature. In contrast to the *homogeneous advising* nature of the Counter, in View, we have *heterogeneous advising*: weaving a piece of code to a single point in the program [6]. This may raise a problem, because now we need to define a *unique* pointcut for the code-to-be-woven. For *prefix/suffix* impacts, this is easy to do with the execution joinpoint and the after or before advices. However, it is more serious for interleaved code such as the call to the mirrorViewSentinels method inside the for-loop in LinkedList. It is not clear how one could specify a unique pointcut for that particular execution point. A finer-grained pointcut mechanism than AspectJ such as in [18] might be needed. Moreover, preserving the uniqueness of the pointcut as the code evolves may be a problem, especially when the number of heterogeneous advices is high.

[1]AspectJ does not allow directly introducing the counter field to T.

```
aspect View {
  // introduce new fields and methods
  LinkedList LinkedList.underlying;
  LinkedList[] LinkedList.views;
  int LinkedList.offset;

  void LinkedList.mirrorViewSentinels(Node a,
         Node b, int i) { ... }

  void LinkedList.fixViewsAfterInsertion(Node newnode)
  { ... }

  // Convert line 39 of Fig. 1 to line 17 of Fig 3
  pointcut sizeInc(LinkedList self) :
    specification of the size field's incrementation
    && this(self);
  void around(LinkedList s): sizeInc(s) {
    (s.underlying!=null?s.underlying:s).size++;
  }

  // Insert the line 18 of Figure 3
  after(LinkedList s):
  execution(void LinkedList.add(Object)) && this(s)
  { s.fixViewsAfterInsertion(How to access newnode ?); }

  // Insert the line 35 of Figure 3
  pointcut insideReverse(LinkedList self) :
    uniquely capture the point to do mirroring inside the for-loop
    && this(self);
  before(LinkedList s) : insideReverse(s)
  { s.mirrorViewSentinels(How to access a,b and i ?); }

  ... //and other impacts...
}
```

Figure 5: Adding the View feature to LinkedList using an aspect. Problematic places are underlined.

Another problem is with *local-dependent* impacts. Although advised code can access the fields of the class it is being woven into — both existing fields and fields that are introduced by the advice's aspect are visible — it cannot access the local variables that become accessible after weaving. Ad-hoc solutions need to be invented for each case to overcome this problem.

Let us now look at how to combine the two features. We consider only the LinkedList class. The case for ArrayList is similar. The aspect is below.

```
aspect CounterAndView {
  // increment the underlying list's counter
  // using the same pointcut from the Counter aspect
  before(LinkedList self):
  Counter<LinkedList>.methodExec(self){
    if(self.underlying != null)
      ((CounterI)self.underlying).counter++;
  }
  // enforce the order because two aspects compete
  declare precedence: Counter, CounterAndView;
}
```

Since CounterAspect and CounterAndView both weave code to the same pointcut, the pointcut definition from CounterAspect is reused. This requires declaring an order to control which aspect is woven first. In our example, inclusion of both features adds a new piece of code to the core without changing the features' existing impacts. If the combination required a change in one of the impacts of the individual features, a more complicated solution would be needed.

We finally note that AOP requires the programmer to learn a new language beyond the basic language used to implement the library. By contrast, other techniques involve extensions that are relatively modest.

Evaluation of AOP

Expressibility: Interleaved and local-dependent impacts are problematic. Specifying pointcuts uniquely can be hard. Ad-hoc and tricky solutions or introducing artificial method calls may be needed to declare the desired pointcuts.

AspectJ handles pointcuts and advices by introducing several auxiliary methods to the woven class. This may bring a runtime overhead. Yiihaw [9] is an aspect-weaver that does not add any extra computation other than the woven advice. It may be used to avoid the runtime overhead.

Modularity: Supported in general. The core code is mostly oblivious of the features. Problems with expressibility may force modularity loss, though.

Reusability: Use of generics improves reusability, such as the case with the Counter aspect.

Extensibility: Extra care needed to not break uniqueness of existing pointcuts when introducing a new feature.

Binary operability: Depends on the system. AspectJ supports weaving into bytecode.

8. C-STYLE PREPROCESSOR DIRECTIVES

We next discuss the preprocessor (PP) directives as found in C. A snippet of a possible solution is given in Figure 6. Pieces of code can be conditionally included using #ifdef directives. Since this is a text-based approach at very fine-granularity, any impact can be expressed. The #else directive allows for expressing modifier impacts. Defining macros improves reuse of impacts. Macros can be parameterized for better reuse, as done with VW_FIELDS in Figure 6.

Preprocessor directives, however, provide no semantic or syntactic guarantees for the produced code. They also make the code very hard to read.

Evaluation of PP Directives

Expressibility: Any impact can be expressed. Exact desired code can be produced.

Modularity: The core is not oblivious of the features. Impacts of a feature are not modularized, but grouped in the same module.

Reusability: Impacts can be reused whenever sensible. Macros allow for increased reuse.

Extensibility: New features can be added by introducing new preprocessor directives and by referring to them in the core code.

Binary operability: Not supported. Source code needed for preprocessing.

```
// remove/keep these two defs to exclude/include features
#define COUNTER
#define VIEW
// Counter feature
#ifdef COUNTER
#define CNT_FIELD int counter = 0;
#define CNT_INC   counter++;
#else
#define CNT_FIELD // empty
#define CNT_INC   // empty
#endif
// View feature
#ifdef VIEW
#define VW_FIELDS(TYPE) TYPE underlying;\
                        TYPE[] views;   \
                        int offset;
... // and other View macros
#else
#define VW_FIELDS(TYPE)// empty
... // and other View macros
#endif
// the common impact
#define UNDERLYING_CNT_INC // empty
#ifdef VIEW
#ifdef COUNTER
// redefine the macro
#define UNDERLYING_CNT_INC \
    if(underlying != null) underlying.counter++;
#endif
#endif

class LinkedList implements List {
  Node first,last;
  int size = 0;
  CNT_FIELD
  VW_FIELDS(LinkedList)

  void add(Object item) {
    CNT_INC
    UNDERLYING_CNT_INC
    Node newnode = new Node(item);
    ...
```

Figure 6: Feature impacts defined as macros.

9. FRAGMENT-ORIENTED PROGRAM GENERATION

Another technique one may consider using to solve the library specialization problem is Fragment-Oriented Program Generation (FOPG), which is a program generation methodology where pieces of code are combined and composed to form a complete program. Several FOPG systems exist in the literature [10, 25, 15, 16, 22, 7]. Although each of these differs by the generality, flexibility, and safety that it offers to the programmer, they all have a *quotation* syntax to define the program fragments, and an *anti-quotation* syntax to mark the holes inside those fragments. Quotations have been inspired from the quasiquotations in Lisp [5]. In this paper we will use Jumbo [10] as the FOPG language because it provides general-purpose program generation using a real-world language: Java. In Jumbo, $< and >$ are used to mark the start and end of a program fragment (i.e. a "quoted fragment"). Backquote (`) is the syntax for declaring holes. Holes are filled in with other code fragments.

To address the library specialization problem with FOPG, we first write a function, shown in Figure 7, that takes two features as parameters and uses them to generate a LinkedList class where locations of impacts are marked with holes that are filled in with impact codes. This generator can be compared to the original LinkedList class from Figure 3. We omit the generator for ArrayList.

Figure 8 shows the implementations of the features. In this approach, a feature is a set of code fragments (or, more generally, a set of fragment-producing functions) to be inserted into holes in the LinkedList generator. Each feature has at least two implementations: one for when the feature is included, one for when it is not. The desired class can be generated by passing the generator method appropriate instances of the feature classes. A possible invocation that generates a LinkedList class with the View feature but not the Counter is genLinkedList(new NoCounter(), new View()). Variations of a feature can be created by subclassing and

```
Code genLinkedList(Counter counter, View view) {
  return $<
    class LinkedList implements List {
      Node first,last;
      int size = 0;
      `(counter.fields())
      `(view.fields($<LinkedList>$))

      void add(Object item) {
        `(counter.inc())
        `(counter.underlyingCounter())
        Node newnode = new Node(item);
        newnode.next = last;
        newnode.prev = last.prev;
        last.prev.next = newnode;
        last.prev = newnode;
        `(view.targetList()).size++;
        `(view.fixViews())
      }
      ...
      void reverse() {
        `(counter.inc())
        `(counter.underlyingCounter())

        `(view.forEachView())
        Node a = first.next, b = last.prev;
        for(int i=0; i<size/2; i++) {
          Object swap = a.item;
          a.item = b.item; b.item = swap;
          `(view.mirror1())
          a = a.next; b = b.prev;
        }
        `(view.mirror2())
      }
    } >$;
}
```

Figure 7: Writing a library generator in FOPG. A LinkedList class enriched with features according to the given arguments is generated.

overriding certain methods, such as in the CounterWithView class, whose instance should be used as the Counter feature if the View feature is enabled.

The fields() method of the View feature uses its parameter as the types of the fields that will be returned. Thus, it is possible to use the method in both ArrayList and LinkedList generators by passing the appropriate type as the argument. The Counter feature also is reusable by both generators.

With the help of the flexibility that FOPG provides in definition of fragments, which are first-class citizens, the most problematic impacts (i.e. interleaving local-dependent) can be expressed without difficulty. Similar to the preprocessor approach, FOPG can produce the exact desired code.

On the other hand, the core functionality is not oblivious of the features. To extend the library with a new feature, the generator has to be annotated with necessary anti-quotations to specify the locations of feature impacts. Furthermore, although the FOPG system we assumed here, Jumbo [10], guarantees parsability of the produced code, it does not provide type-safety. There are several FOPG type systems in the literature [15, 22, 7, 8, 13]. However, none of them, to our knowledge, is general and strong enough to type-check the library generator presented here. Type systems that require fragments to be closed, such as [23, 24], can type-check the generators if the features and impacts are implemented as higher-order functions, but this may be inconvenient for library developers. We do not go into more details due to space constraints.

```
class Counter {
  Code fields() { return $< int counter = 0; >$; }
  Code inc() { return $< counter++; >$; }
  Code underlyingCounter() { return $< >$; }
}
class NoCounter extends Counter {
  Code fields() { return $< >$; }
  Code inc() { return $< >$; }
}
class CounterWithView extends Counter {
  Code underlyingCounter() {
    return $< if(underlying != null)
                underlying.counter++; >$;
  }
}

class View {
  Code fields(Code type) {
    return $< `(type) underlying;
              `(type)[] views;
              int offset; >$;
  }
  Code targetList() {
    return $<(underlying!=null?underlying:this)>$;
  }
  Code mirror1() {
    return $< mirrorViewSentinels(a,b,i); >$;
  }
  ... // and other impacts
}
class NoView extends View {
  Code fields() { return $< >$; }
  Code targetList() { return $< this >$; }
  Code mirror1() { return $< >$; }
  ... // and other impacts
}
```

Figure 8: Defining the features in FOPG.

Evaluation of FOPG

Expressibility: Any impact can be expressed. Exact desired code can be generated.

Modularity: Features are modularized, but the core code is not isolated; it contains hook points for features.

Reusability: Fragments are first-class citizens; they can be freely passed around. Fragment-producing methods of features can be parameterized over other fragments. These allow for reusability.

Extensibility: Core source is needed to introduce new holes at the locations of the new feature's impacts.

Binary operability: Supported by Jumbo [10].

10. CONCLUSION AND FUTURE DIRECTIONS

Library specialization is the problem of customizing a library according to the needs of the user. A feature of the library is to be excluded if it is not going to be used, so that a more efficient version of the library can be produced. We have started our study of the problem from a real-world implementation of collections library, C5 [14]. We have seen that features add new or modify existing pieces of code, as well as introduce new fields to the class they are enriching. The piece of code that is added or modified can be anywhere: at the beginning, middle, or end of a method. It may refer to the fields introduced by the feature, the fields of the class, or the local variables that are visible at the impact location. This fine-granulated character of impacts

	OOP	FOP	AOP (5)	CIDE	PP (6)	FOPG (7)
Expressibility	◑	◑	◑	●(1)	●	●
Modularity	◑	●	●	○	◑	●
Reusability	◑	●	●	○	●	●
Extensibility	◑	●	●(2)	◑(3)	◑(3)	◑(3)
Binary operability	●	◑(4)	●	○	○	●

(1) Modifier impacts are not supported, but this is a technical issue rather than fundamental.
(2) Preserving the uniqueness of pointcuts requires extra care.
(3) Source code has to be available to add annotations for the new feature.
(4) Depends on the system.
(5) Requires learning a new language.
(6) No syntactic or semantic safety provided for the produced code.
(7) Guaranteeing type-safety of the generated code has problems.

Table 1: A summary of the evaluation of techniques. ○ denotes no support, ◑ denotes partial support, and ● denotes good support.

was previously identified in [12]. All these properties affect how a feature is handled by a particular technique.

We have evaluated six techniques. A summary of the evaluation is given in Table 1. None of the techniques offers a completely satisfactory solution to the library specialization problem. It appears that an ideal methodology would be a mixture of these techniques. CIDE, FOPG and Preprocessor directives have better expressibility than OOP, FOP, and AOP. However, they cannot make the core code oblivious of the features as well as AOP, which also is very successful for homogeneous advicing. AOP, on the other hand, achieves obliviousness at the price of the unique pointcut problem. FOPG provides better modularity and reusability than CIDE or PP, but it has problems in type-checking. A middle-ground approach between FOPG and AOP that supports modularity and reuse of impacts via first-class fragments, but also supports homogeneous advising via (implicitly) defined locations could provide a satisfactory solution. The feasibility and effectiveness of the combination of these strengths, however, is a future research problem.

Acknowledgments

We thank Peter Sestoft for introducing us to the library specialization problem in the context of C5.

11. REFERENCES

[1] Aspectj web site. `http://www.aspectj.org`.
[2] D. Batory. A tutorial on feature oriented programming and the ahead tool suite. In *GTTSE'05*, volume 4143 of *LNCS*, pages 3–35. Springer, 2006.
[3] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE TSE*, 30(6):355–371, 2004.
[4] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. In *FSE'93*, pages 191–199. ACM Press, 1993.
[5] A. Bawden. Quasiquotation in lisp. In *PEPM*, pages 4–12, 1999.
[6] A. Colyer, A. Rashid, and G. Blair. The separation of concerns in program families. Technical report, Computing Department, Lancaster University, January 2004.
[7] S. S. Huang, D. Zook, and Y. Smaragdakis. Statically safe program generation with safegen. In *GPCE'05*, volume 3676 of *LNCS*, pages 309-326. Springer, 2005.
[8] S. S. Huang, D. Zook, and Y. Smaragdakis. cj: Enhancing java with safe type conditions. In *AOSD'07*, pages 185–198. ACM Press, 2007.
[9] R. Johansen, P. Sestoft, and S. Spangenberg. Zero-overhead composable aspects for .net. In *Proceedings of Lipari 2007 Summer School*, Italy, 2007.
[10] S. Kamin, L. Clausen, and A. Jarvis. Jumbo: run-time code generation for java and its applications. In *CGO '03*, pages 48–56. IEEE Computer Society, 2003.
[11] C. Kästner and S. Apel. Type-checking software product lines - a formal approach. In *ASE'08*. IEEE Computer Society, Sept. 2008.
[12] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE '08*, pages 311–320. ACM, 2008.
[13] I.-S. Kim, K. Yi, and C. Calcagno. A polymorphic modal type system for lisp-like multi-staged languages. In *POPL'06*, pages 257–268. ACM, 2006.
[14] N. Kokholm and P. Sestoft. The c5 generic collection library. `http://www.itu.dk/research/c5/`.
[15] Y. Oiwa, H. Masuhara, and A. Yonezawa. Dynjava: Type safe dynamic code generation in java. In *The 3rd JSSST Workshop on Programming and Programming Languages (PPL2001)*, March 2001.
[16] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: a language and compiler for dynamic code generation. *ACM TOPLAS*, 21(2):324–369, 1999.
[17] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP'97*, volume 1241 of *LNCS*, Finland, 1997. Springer.
[18] T. Rho, G. Kniesel, and M. Appeltauer. Fine-grained generic aspects. In *FOAL'06*, Bonn, Germany, 2006.
[19] Scala language. `http://www.scala-lang.org`.
[20] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *ECOOP'03*, volume 2743 of *LNCS*, pages 248–274. Springer, 2003.
[21] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *ECOOP'98*, volume 1445 of *LNCS*, pages 550–570, London, UK, 1998. Springer.
[22] F. Smith, D. Grossman, G. Morrisett, L. Hornof, and T. Jim. Compiling for template-based run-time code generation. *JFP*, 13(3):677–708, 2003.
[23] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *TCS*, 248(1–2):211–242, 2000.
[24] Y. Yuse and A. Igarashi. A modal type system for multi-level generating extensions with persistent code. In *PPDP'06*, pages 201–212, New York, 2006. ACM.
[25] D. Zook, S. S. Huang, and Y. Smaragdakis. Generating aspectj programs with meta-aspectj. In *GPCE'04*, volume 3286 of *LNCS*, pages 1–18, Vancouver, Canada, October 2004. Springer.