A Comparison of Generative Approaches: XVCL and GenVoca

James Blair and Don Batory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712 U.S.A.
{batory, jblair}@cs.utexas.edu

Abstract

We report one of the first comparative studies on two mature approaches to generative programming: XVCL and GenVoca. XVCL is the latest incarnation of Bassett's frames; GenVoca melds feature modularity with compositional programming. Both approaches explicitly rest on a pair of ideas: (1) parameterized functions return source code and (2) composing such functions synthesizes applications. Although their foundations are identical, XVCL and GenVoca have very different design philosophies. These differences raise an interesting debate on what design methodologies and programming constructs should be used in writing generative programs.

1 Introduction

The history of software design closely parallels the history of modularity in programming languages. Structured programming was the dominant software design technique in the 1970s and 80s; Pascal and C were common languages used to express such designs. System architects conceived an application's structure in terms of functions and procedures. Today, *object-oriented (OO)* designs and programming languages dominate. System architects express an application's structure using a richer vocabulary of objects, classes, and relationships among classes (such as *inheritance* and *part-of*) and use OO languages to implement their designs.

A characteristic that both structured languages (like C and Pascal) and OO languages (like C++, Smalltalk, and Java) share is their generality. It is hard to conceive of application domains where these design techniques and programming languages could not be used. However, there is a glaring exception: applications that *generate* programs are not well supported by either conventional structured programming languages or by object-oriented languages. In fact, understanding what generative concepts should be supported in programming languages remains the subject of current research and debate. Meta-ML [12], AspectJ pointcut-advice pairs [10], and C++ template meta-programming [6] are just a few of the attempts to advance conventional programming languages with generative programming constructs.

Concomitantly, *generative design methodologies (GDMs)* — or how to structure or design generative applications — are also in their infancy. It is not obvious whether existing GDMs relate to structured designs or OO design methodologies, or if they are something quite different.

In this paper, we compare two mature GDMs and the structure of an application that follows each one. XVCL is the most recent incarnation of Bassett's frames [2], a preprocessor technology that has been used successfully in synthesizing large COBOL (and now Java) business applications. GenVoca is a general model of generative programming based on feature modularity [5]. Both approaches explicitly rest on a pair of ideas: (1) parameterized functions return source code and (2) composing such functions synthesizes applications. Obvious questions are: if both are based on the same ideas, do both use the same design methodology? If not, how are they different? And are these differences important? What are their relationships to known design methodologies, like structured programming and OO?

The contribution of this paper is one of the first detailed comparative studies of two mature approaches to generative programming. We eschew methodology comparisons in the abstract (e.g., [11]); we believe that the best way to understand the relationships and distinctions among GDMs is to compare implementations of the same application. To this end, we use a notepad application, originally published by Jarzabek as an exemplar of XVCL design [8], as a concrete basis for comparison. We begin with an overview of XVCL and a review of Jarzabek's notepad.

2 XVCL

Bassett [2] observed that similar programs often differ by small amounts of code; new programs are created manually through an error-prone process of copying, pasting, and editing. Forgetting to change a variable name everywhere it appears, or not realizing that a data structure needs to be resized leads to inconsistent and, ultimately, "buggy" applications.

Bassett realized that this process of customization could be mechanized by text preprocessors. The core of his design paradigm and language was the *frame*, a parameterized function that returns a text string which is interpreted as source code. Functions (frames) are themselves implemented by the usual panoply of imperative programming constructs such as variable declarations, assignment statements, if statements, while-loops, and the ability to invoke other frames/functions. To synthesize a customized program — a text string — is a matter of assigning values to variables and invoking one or more functions (frames).

Jarzabek refreshed these ideas in *XML-based Variant Configuration Language (XVCL)* [14], which expresses frame programming concepts as a markup language similar to XML. Each frame definition, called an *x-frame*, is a separate XML file. An x-frame can invoke other x-frames through a process called *adaptation*, which is how program customization is realized. One consequence of the ability of x-frames to adapt other x-frames is that the frame hierarchy (or call graph) of an XVCL program can be viewed as a tree. Each frame adapts any or all of its descendants, and each frame can be adapted by its parent frame.

An important concept in adaptation is a breakpoint.[1] A *breakpoint* is a named fragment of code that is defined within a frame. An ancestor frame can assign (or more accurately, override) the code fragment of a descendant frame. Code can be deleted, for example, by simply specifying an empty code fragment. More importantly, the code of a breakpoint cannot be partially modified — replacement is “all or none.”

The top-level x-frame (the root of the frame hierarchy) is called a *specification* or `.s` file — and serves the purpose of the “`main`” function in a C program. Invoking the XVCL preprocessor on a `.s` file executes the sequence of XVCL statements in the order they are listed; the resulting text string is the file that is to be produced. This file can then be fed into a compiler (such as `javac`) and eventually executed. Since the XVCL preprocessor removes all XVCL commands as it processes them, down-stream tools never see XVCL statements.

Details. The specific details of XVCL and its syntax are not relevant to our paper; what is important are its language concepts. Although our following observations focus on XVCL, they hold for prior frame languages as well.

XVCL is a primitive single-assignment imperative programming language. It provides no support for user-defined types (e.g., C “`struct`” or Java “`class`” constructs) other than arrays and lists[2]. For example, to implement a C-language `struct` with two fields `X` and `Y`, XVCL programmers must create arrays `X[]` and `Y[]`, where the contents of `struct` instance `i` is represented by `X[i]` and `Y[i]`. The lack of `struct` and user-defined types tend to obscure XVCL programs.

Second, variables have unusual scoping rules in XVCL. Any frame can create and set variables to be used by its *children* (the frames that it calls) and its *descendants*. Once a parent frame has set the value of a variable, no child frame can change that value. (This is the single-assignment aspect to frame languages.) Consequently, a common design technique is for a frame to declare default values for all of its variables. These values can be overridden by ancestor frames, and such overriding is how customizations (or adaptations) are accomplished.

Third, function names can be passed as parameters. A function (frame) can thus invoke another function (frame) that is known to it only at XVCL processor run-time. This allows frames to be “first-class” citizens and provides powerful program synthesis capabilities. We will see an example shortly.

Design Philosophy. Conceptually, XVCL and other frame languages approximate the C programming language. The mindset that frame designers bring to (or impose on) the structure of generative programs is the same program design or structured design techniques used in the 1970s and 80s. (Because of the inability for users to define their own data types, frame programs are more primitive.) Although frame designs and languages are unquestionably successful, frames do not express standard programming abstractions (objects, classes) in modern languages and software designs. This is an important point which we will explore.

2.1 Jarzabek’s Notepad

Jarzabek wrote a generator for notepad applications in XVCL. Figure 1 shows a GUI of a synthesized notepad. We will restructure this generator in later sections.

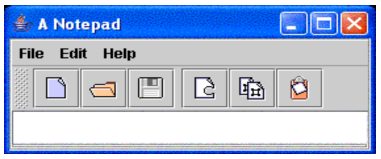

Figure 1. Notepad GUI

Figure 2 shows the frame call graph in Jarzabek’s design. The root is the `Notepad.s` file which initializes lists and variables whose values are key to notepad customization.

1. Actually, the name “breakpoint” is a bit of a misnomer in XVCL, as each `break` can be comprised of many lines of code. It may be more clear to think of them as “break sections.”

2. Lists are multi-valued variables, and arrays are emulated using structured variable names such as A1, A2, A3. So to access element ‘4’ in array ‘A’, the name for variable A4 is produced by concatenating ‘A’ with ‘4’.

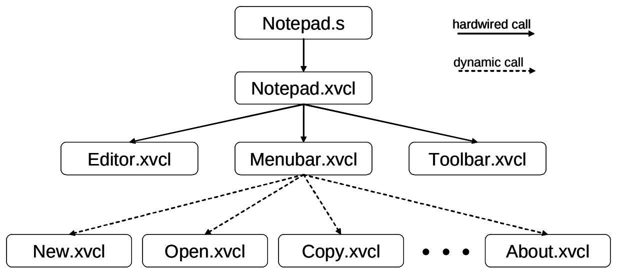

Figure 2. Notepad Call Graph

The primary initializations are the two lists depicted in Figure 3. **`Menubar`** is a list of named lists — **`ItemsFile`**, **`ItemsEdit`**, and **`ItemsHelp`**. These lists contain the names of actions that are to appear on the **`File`**, **`Edit`**, and **`Help`** menus. The second list, **`Toolbar`**, is a list of names of the actions (**`New`**, **`Open`**, etc.) that are to appear on the toolbar.

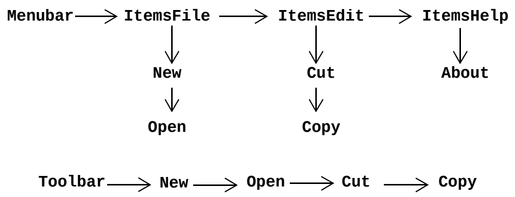

Figure 3. List Initializations in Notepad.s

The design of this generative program follows the principles of top-down, step-wise program construction. The **`Notepad.xvcl`** frame creates the shell of the notepad program, which is simply a window with a title. **`Notepad.xvcl`** then makes three hard-wired calls to frames that elaborate the window: a text editor is added (**`Editor.xvcl`**), a menu bar is added (**`Menubar.xvcl`**), and a toolbar is added (**`Toolbar.xvcl`**).

The lists that were initialized in the **`.s`** file are traversed twice in **`Menubar.xvcl`**. In the first pass, code is produced to populate items on the **`File`**, **`Edit`**, and **`Help`** menus. A second pass is used to invoke the XVCL frames for each action (e.g., **`New.xvcl`**, **`Open.xvcl`**).[3] Each called frame adds a method to the **`Notepad`** class to process an action. For example, **`New.xvcl`** adds a method to create files, **`Open.xvcl`** adds a method to open a file, **`Save.xvcl`** adds a method to save a file, etc.[4]

`Toolbar.xvcl` traverses the **`Toolbar`** list, to include a button for each action listed, to display a particular image (**`.gif`**) in the button, and to add code to invoke the action method when the button is pressed.

3. Note: this design illustrates the use of frame names being passed as a parameter and then later dynamically invoked.
4. We renamed Jarzabek's frames (**`New.xvcl`** instead of **`NewFile.xvcl`**) to make our discussions clearer.

The notepad generator is easy to understand and the code to customize the variables and lists is straightforward to write. However, a more challenging task is to add a new action that is not yet part of the notepad generator. The first thing a designer of this generator (and more generally any generative program written in XVCL) will notice is that the changes to add a new action are *not* localized. There indeed will need to be a **`NewAction.xvcl`** frame to add an action method, but there may need to be additional specifications, such as a tool-tip string, an image, and possibly customized code fragments for menubar and toolbar items. Further, a more general design would allow a **`NewAction`** to be placed on the toolbar, but not on the menubar. As currently written, such a change would force a programmer to understand the details of the affected x-frames and to make multiple modifications to those frames, which is exactly the type of error-prone program maintenance we wish to avoid.

In principle, grouping related frames (methods) and variables (i.e., members such as tool-tip string and image file name) of an action into a single class would provide a better program organization. Our next step is to recast Jarzabek's program into an OO design with these improvements in mind.

3 An Object-Oriented Design

While XVCL is not object-oriented, we can emulate class modularity with directories. Each directory corresponds to a class. Frames within a directory are methods of that class, and files (e.g., **`.gif`**) correspond to data members. (Frames can also serve as variable accessors.) Inheritance relationships, such as child classes inheriting the methods of their parent, are enforced manually.

A straightforward restructuring of Jarzabek's program is to create a pair of inheritance hierarchies that implement subtype polymorphism. One hierarchy roots all notepad actions; another roots notepad components (see Figure 4).

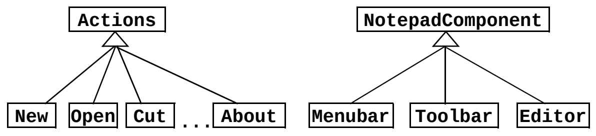

Figure 4. Inheritance Hierarchies in Notepad

The **`Actions`** class (or interface) defines a common set of methods (frames) and variables that all actions should have. Each action has three methods — **`addMenuBar()`**, **`addToolBar()`**, and **`install()`** — and three variables — **`ToolTip`**, **`Gif`**, and **`ActionName`**. The methods use **`ActionName`** to provide the name of the method that executes the given action.

addMenuBar() adds an item to one of the (already constructed) **File**, **Edit**, or **Help** menus. **addToolBar()** builds a toolbar button, using the **Gif** and **ToolTip** variables, and adds the button to the toolbar. Of course, if an action adds nothing to the menubar or toolbar, the corresponding method/frame is a no-op. **install()** introduces the method that executes the action, where the method name is stored in **ActionName**.

NotepadComponents also share an interface, albeit with fewer members. There is only a single method, **install()**, and no variables. **install()** introduces the code that constructs the component. In the case of Menubar and Toolbar, this method calls the associated **add***Bar()** methods of all desired **Actions**.

The **.s** file simplifies to the creation of a pair of lists, **ComponentList** and **ActionList**. **ComponentList** is a list of notepad components to install. **ActionList** is a list of actions to install, in order of their appearance on a menu or toolbar.

Given this input, our OO XVCL notepad generator works in the following way. A **notepad** object creates the shell of a notepad program. It iterates through the objects of the **ActionList** and invokes the **install()** method. Further it iterates through each object on the **ComponentList**, and invokes its **install()** method. Each component, in turn, iterates through the **ActionList** and invokes its **add***Bar()** method.

Fundamentally, the logic by which a notepad application is generated is the same as Jarzabek's design. In our view, the only significant change that an OO structure provides is that the generator itself is easier to maintain. To extend the notepad generator by adding a new action (or component) requires the addition of a single class, whose methods and variables are defined by **Actions** (or **NotepadComponent**). Thus, *the structure of the program defines more precisely what needs to be changed when a new action is added or removed*. For this reason, we assert that an OO design leads to more maintainable and more extensible organization of a generator program.

The ability to maintain generators and retain intellectual control over their design is important. Generators are a technological statement that the development of software in a domain is understood well enough to be automated. However, we must make the same claim for generators themselves: the complexity of generators must also be controlled and must remain low as application complexity scales, otherwise generator technology will be unlikely to have widespread adoption. OO designs of generator programs are a step forward in this regard. A further improvement in maintainability and extensibility is to construct generators in a compositional manner, our next step in evolving the notepad generator.

4 A Compositional Design

Compositional designs of programs, whether or not they are generative, is a paradigm of step-wise construction. The idea is to start with a simple program and to progressively elaborate it with incremental units of program functionality — called *features* — to construct a more complex, feature-rich program.

GenVoca is a composition model of program development that is based on two ideas: feature modularity and program objectification. First, feature modularity is an outgrowth of inheritance and a programming style called *programming-by-difference* [9]. A programmer defines a new class by picking a closely related class as its superclass and describes the differences between the old and new classes. *Features generalize this concept by modularizing changes to multiple classes.*

Java programmers are familiar with elementary forms of compositional programming. For example, an object that reads a file line by line is created by dynamically composing a series of J2SDK decorators or adaptors that provide the needed functionality:

```
LineNumberReader r =
     new LineNumberReader(
         new InputStreamReader(
             new FileInputStream("my.txt")));
```

Feature modules scale this idea beyond a single class to the static "adaptation" or "extension" of multiple classes.

The second idea is that programs are objects (i.e., program objectification). Feature modules are functions that map programs from simple programs to more complex programs. In effect, features implement *program deltas* or *program extensions*. That is, a feature encapsulates the changes to an input program for adding the functionality of that particular feature. Compositions of feature modules is intuitively a *summation*: the desired application is the sum of the changes made by its feature modules. Let • denote function composition (or equivalently a summation operation). If program **P** is defined by the composition of features **A**, **B**, **C**, and **D**, written as:

```
P = A • B • C • D
```

Then **P** is synthesized by starting with feature **D**, applying the changes by feature **C**, then the changes by **B**, then those by **A**. That is, the accumulation of all these changes is **P**. This is the essence of GenVoca.

Subclassing is a standard implementation technique for implementing program deltas and summations. If program **P** was defined by a single Java class, we could approximate the result of expression **A • B • C • D** as a subclassing hierarchy, where each feature is a "class" that encapsulates changes it makes to its parent:

```
class D             { /* base program    */ }
class C extends D   { /* add C's changes */ }
class B extends C   { /* add B's changes */ }
class A extends B   { /* add A's changes */ }
class P extends A   {}
```

In reality, subclassing hierarchies only approximate the effects of program deltas. There is no way in Java to fully emulate and encapsulate the effects of a program delta or feature. Features can more closely be approximated as *mixins* — classes whose superclass is specified as a parameter [13]. But even mixins aren't sufficient, as one would like to be able to inherit constructors and have the mixin assume the name of its parent class, neither of which even advanced forms of mixins support [1][5]. Specialized programming language support for features is needed [5]. Furthermore, the inheritance or subclassing chains that are produced by compositional designs are unnatural to OO designers. In fact, OO experts explicitly discourage the use of "extending extensions" and the use of class inheritance as is done above [7].

The point here is that the GenVoca design paradigm, which conceptualizes program construction as a summation of program deltas, is really neither an object-oriented design paradigm nor a structured programming design paradigm. As discussed above, it requires special language constructs that are not supported by mainstream OO languages, and imposes a way of thinking (e.g., the extensive use of subclassing) that is definitely not mainstream OO design.

In the next section, we describe a compositional design for the notepad generator.

4.1 Composing Notepad Generators

Jarzabek's notepad application is a single Java class. This means that each feature in a notepad generator can be represented by a single class, or a "subclass" or "mixin" as described above. (We'll show how features scale to larger programs later). Rather than composing features statically using an inheritance chain, we will emulate this concept by composing features dynamically as a chain of features.

The organization we use is actually the equivalent of a *generator* of notepad generators. All features implement the **NotepadFeature** abstract class or interface, which has the methods **install()**, **addToolBar()**, **addMenuBar()**, and variables **ActionName**, **ToolTip**, and **Gif**, whose values are subclass specific (Figure 5).

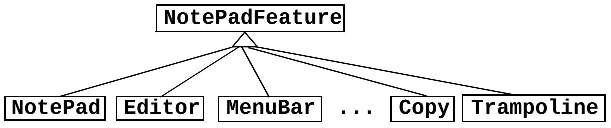

Figure 5. Inheritance Hierarchies in a Compositional Design

The order in which features are composed is defined by a GenVoca grammar [6], where tokens are features and sentences are compositions:

```
App  : NotePad Editor
        OptComponents+ Actions+ Trampoline;
OptComponents : MenuBar | ToolBar ;
Actions : New | Open | Copy | ... | About ;
```

For example, a notepad application that supports a menubar and the **New**, **Open**, and **Copy** features is the sentence:

```
NotePad Editor MenuBar New Open Copy Trampoline
```

All of the above features are obvious except **Trampoline**, which serves as a marker or object that terminates a chain of features. Its existence is an artifact of composing features dynamically, which we now elaborate.

A **.s** file specifies the equivalent of a sentence in this grammar as a composition of feature objects. The expression for the above sentence in Java would be:

```
Application = new NotePad( new Editor(
     new MenuBar( new New( new Open(
         new Copy( new Trampoline() )))))); (1)
```

which creates a chain of features headed by a **NotePad** object (Figure 6). We call this the *Application List*. This restructuring collapses the two lists (**ActionList** and **ComponentList**) in our OO design of Section 3 to a single list. The reason for this follows the program delta semantics of features. Each feature object (in Figure 6) encapsulates a set of changes to a notepad program. As this chain is traversed, each feature contributes its changes to a generated notepad program.

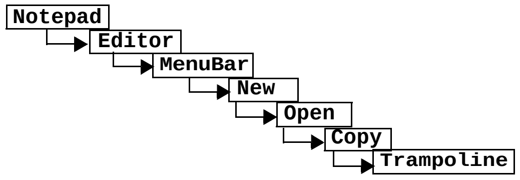

Figure 6. A Chain of Feature Objects

A notepad application is synthesized in the following way. The **Notepad.s** file defines the Application List. It invokes the **`install()`** method on the first object of this list, which is a **`Notepad`** object. This object generates the code for a window, and then invokes the **`install()`** method on the next object in the chain, an **`Editor`** object. This object adds in the code for a text editor, and invokes the **`install()`** method on the next object, a **`Menubar`**. The **`Menubar`** adds the code to produce a menubar, and invokes the **`addMenuBar()`** method on all features "downstream" to it. That is, **`addMenuBar()`** is invoked on **`New`**, which adds its item to a menu and invokes **`addMenuBar()`** on **`Open`**, etc. When the **`Trampoline`** object is reached, it returns immediately (hence **`Trampoline`**'s name). Only after all downstream features have added their items to the menubar will **`Menubar`** invoke the **`install()`** method on **`New`**, which adds its method. And so on.

If a **`ToolBar`** object were added to a feature list, it would first add the code for a toolbar, and then it would invoke the **`addToolBar()`** method on all objects downstream from it to add their toolbar functionality, if any. Once completed, **`Toolbar`** would invoke **`install()`** on the next downstream object.

In general, if an object has nothing to contribute to a menubar or toolbar, its **`add***Bar()`** method does nothing except propagate the method call to the next object in the chain. In all, an object on the Application List may be visited three times. The main traversal calls the **`install()`** method, a second to call **`addMenuBar()`**, and a third for **`addToolBar()`**. Figure 7 illustrates this control flow.

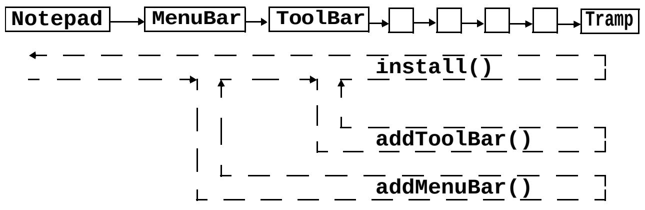

Figure 7. List Traversals During Synthesis

Compositional designs have the benefits of OO designs in that changes made by features are encapsulated, but go further in that they simplify the specification and maintenance of customized generators. That is, a generator specification is just a single expression, e.g., **`(1)`**. Thus, reconfiguring the generator (and hence the notepad application that it produces) is usually straightforward.

5 Compositional Design versus Frame Design

Let's examine the benefits and drawbacks of each design to learn more about GDMs.

First, it is clear that both frame and compositional designs break programs into smaller pieces. These pieces tend to correspond to application features, though this is not always the case. What constitutes a feature is not completely obvious, admittedly, and the decision can sometimes turn into a subjective preference. In any case, a compositional GDM modularizes features into classes, while frame designs need not embrace such modularity.

One benefit of XVCL that disappears in our compositional design is that pure XVCL only defines the changes that must be made. With the creation of a standard interface, compositional designs require extra work to implement that interface on all objects, regardless of how they behave.[5] In that sense, the compositional GDM has taken us a step backward.

Another question is the use of design patterns. Traditional OO programmers consider it poor style to use a long chain of class adaptations to reach a final, specialized type. But in compositional designs, this is just the pattern that seems necessary and logical.

We speculate that the benefits of compositional designs outweigh the drawbacks. In particular, maintenance of a compositional GDM seems clearer than the maintenance of a frame-based design. Since features are modularized, and each feature is self-sufficient, changes are localized. A programmer who may normally look through thousands of lines of code to maintain an application can focus on a small subset of that code which is pertinent to the feature(s) he or she wishes to modify. And since only a small slice of the application must be maintained, the programmer will be more confident that the changes were made consistently and also will not affect other aspects of the application.[6]

6 Inversion Problem

Readers may have noticed an inconvenient difference in the way the composition chains are depicted in the previous sections. At the beginning of Section 4, when we first discussed composing program features, the specification was:

```
P = A•B•C•D
```

5. A slight modification to the XVCL specification could eliminate this problem, and is proposed below in section 9.

6. Examples of compositional design and development are all around us in the world, so programmers are used to working with them. When one purchases a new car, there is a base car to which features and packages of features can be added (for a price, of course). Something as simple as ordering a hamburger is done compositionally: "I'll have a number four plus tomatoes, mustard instead of ketchup, with fries, and a lemonade." Compositional designs are common in our culture; just as OO sought to model the world around us in a way that made sense, compositional designs model the generation of something new intuitively.

In the case above, `D` represents the basic application and is listed on the right, while `A`, `B`, and `C` are extensions written on the left.

Conversely, in Figure 6, the base application is listed at the top of the chain and on the left, while extensions are to the right. This inversion is not simply a difference in the way the diagrams are drawn, but rather it highlights an important difference between feature implementations using inheritance chains and XVCL frame trees.

Consider class `A` and a delta represented by subclass `B`:

```
class A {
   void foo() { /*1*/ }
}
class B extends A {
   void foo() { super.foo(); /*2*/ }
   void bar() { ... }
}
```

The class that is produced by "squashing" this inheritance hierarchy (emulating the idea of applying a delta to `A`) is:

```
class A {
   void foo() { /*1*/ /*2*/ }
   void bar() { ... }
}                                  (2)
```

Now let's see how this design would be implemented in a compositional XVCL setting. The Application List that we would use to produce the above is `new A( new B( new Trampoline()))`. That is, the base feature defines the "shell" of class `A`. Within `A`, there are "holes" that must be filled with code to be supplied by "downstream" features:

```
class A {
   void foo() { /*1*/  moreFoo  }
   moreMembers
}
```

Figure 8. A Frame

In particular, the `A` object has a single method/frame called `install()` which generates the code in Figure 8. The "holes" labeled `moreFoo` and `moreMembers` denote points where the `moreFoo()` and `moreMembers()` methods/frames are invoked on the object downstream from `A`. These frames "fill in" the holes in Figure 8 with code.

Continuing, the `B` object would have two methods/frames. The `moreFoo()` frame would produce "`/*2*/`", and the `moreMembers()` frame would return "`void bar {...}`". The way we described these methods is that they have no "holes" to be filled by downstream objects (e.g., `Trampoline`), but in general they would. So after the `install()` method of object `A` is executed, the class definition `(2)` is produced. Thus by using inheritance or XVCL, we have sketched how two functionally identical definitions for class `A` are produced.

Embodied in these two approaches is a fundamental distinction between inheritance-based implementations representing program deltas and generative or XVCL-like implementations. In an inheritance-based approach, the base program can be thought of as a "constant" — it has no explicit holes that need to be filled. A delta, represented by a subclass, can be recognized as a function — it takes a program or class as input, and at well-designated but *implicit* points in this program or class, it injects its changes. So in our example, program/class `A` is a constant and the program/class delta `B` is a function. Their composition is `B(A)` or `B•A`.

Now consider this same problem from the perspective of XVCL. The base program is definitely *not* a constant. There are *explicit* parameters (e.g., `moreFoo` and `moreMembers`) that must be supplied to `A` for it to produce its results. We have simplified this parameterization so that a single object, `B`, has methods whose return values can fill these parameters. So in an XVCL design, "holes" or parameters in a base program (and its deltas) are explicit. Base program/class `A` is a function, program delta `B` is a function, and `Trampoline` is a constant. Their composition is `A(B(Trampoline))` or `A•B•Trampoline`, which gives the inverted appearance to an inheritance-based approach.

Although we did not encounter difficulties in Figure 8 and notepad, to emulate inheritance chains in XVCL is not always easy. Consider Figure 9a which shows a simple chain, where program deltas wrap previously defined bodies of method `m()`. The body of `m()` once the chain is "squashed" is:

```
m() { /*4*/ /*2*/ /*1*/ /*3*/ /*5*/ }
```

To synthesize this body using XVCL requires considerable work because code fragments must be filled in by downstream objects before and after a code fragment is inserted. Figure 9b sketches an XVCL design that does this. Fortunately for notepad, method deltas required code to be appended to a method body, rather than around. If around or wrapping methods arose, XVCL encoding of inheritance chains would be harder.

7 Scalability

An important test of a software design is how it adapts to incrementally larger problems. Let's consider our XVCL design first.

The change is to add a Help capability to the notepad, so that users can see documentation on its various features. We must create a new list in the `Notepad.S` file that specifies

```
class Base {
  m() { /*1*/ }
}
class A extends Base {
  m(){ /*2*/ super.m()
       /*3*/ }
}
class B extends A {
  m(){ /*4*/ super.m()
       /*5*/ }
}
```

(a)

```
class Base {
  m() { before
        /*1*/ after }
}

class A {
 before(){ before /*2*/}
 after(){ /*3*/ after }
}

class B {
 before(){ /*4*/ }
 after(){ /*5*/ }
}
```

(b)

Figure 9. Before and After Encodings

which actions will have Help information. Additionally, we must add the text for each action somewhere, and but where is not clear. It is not logical to place all of the Help text in the **Help.XVCL** frame, and the structure of the application prevents us from adding the information to each action's individual x-frame.

Let's make the change even larger. Suppose we want to combine older applications into one new application — perhaps we are adding existing French and German spell-checkers to the program. To include these new applications, we must specify a completely new set of lists to customize the new pieces of the application. These low-level details will quickly accumulate, which makes a specification harder as a program's complexity grows.

Let's make the same changes to our compositional design. To add a **Help** feature, we must add "Help" to our list of **NotepadFeatures**, and then add the **Help** information for each of the features. In the case of our compositional design, it is obvious where the new text should go — in a method of each class named **addHelp()**. And if we grow the application to include the foreign language spell-checker, we simply add those features to our list. The beauty of compositional designs (and their feature building blocks) is that they are highly structured. This means that they are generally easier to modify or it is generally easier to understand what needs to be modified. As a consequence, compositional program specifications tend to be simpler than in XVCL.

8 Conclusions

We have examined the benefits and drawbacks of two important GDMs. We began with a frame-based design and noticed that it was similar in programming style to the structured programming languages of the 1970s and 80s. Using XVCL encourages programmers to think in terms of functions and actions but does not require those actions to be organized into cohesive, consistent units. Following the evolution of programming languages, we morphed the frame-based, XVCL design of a notepad generator into an object-oriented style similar to the C++ or Java programming language. This change did not provide any benefit in the logical operation of the generator, but perceptible gains in the area of software maintenance were evident. Just as modern OO languages offer better organization of code, our OO design reorganized the notepad code into a more manageable structure.

Taking our restructuring one step further, we modified the OO design to create a compositional design based on the idea that programs can be treated as objects and are compositions of modularized features. This provided us with additional advantages over both the frame-based and OO designs with regard to long-term maintenance of the application, and our compositional design created a structure that made it clearer how the application could be extended. By implementing a standard interface across features, we simplified three aspects of the maintenance process: the time required to make revisions, the level of understanding needed to make those revisions, and the number of errors introduced by those revisions. Since an application spends a large percentage of its life-cycle in the "maintenance" phase, streamlining that process decreases the overall cost of producing software.[7]

It has become clear to us that the main advantage of a compositional approach to generative programming is the ease of program modification and extension that comes with a modularized, well-structured solution. The benefits of object-orientation are not lost in the realm of generative programming. Furthermore, just as the object-oriented programming languages improved upon structured languages and structured designs, compositional programming extends object-orientation to further simplify software maintenance.

Acknowledgements. We appreciate the insights on OO programming from Dr. Mark Johnstone and Dr. Greg Lavender.

9 References

[1] E.E. Allen, "A First-Class Approach to Genericity", Ph.D. Rice University, 2002.

[2] P. Bassett, *Framing Software Reuse: Lessons from the Real World*, Prentice-Hall, 1996.

[3] D. Batory, J. Thomas, and M. Sirkin. "Reengineering a Complex Application Using a Scalable Data Structure Compiler". *ACM SIGSOFT* (New Orleans), December 1994.

7. Additional ideas for reducing maintenance costs are discussed in the Appendix.

[4] D. Batory, G. Chen, E. Robertson, and T. Wang, “Design Wizards and Visual Programming Environments for GenVoca Generators”, *IEEE Transactions on Software Engineering*, May 2000, 441-452.

[5] D. Batory, J.N. Sarvela, A. Rauschmayer, “Scaling Step-Wise Refinement”, appear in *IEEE Transactions on Software Engineering*, June 2004.

[6] K. Czarnecki and U. Eisenecker. *Generative Programming Methods, Tools, and Applications*. Addison-Wesley, Boston, MA, 2000.

[7] G. Heineman, “An Instance-Oriented Approach to Constructing Product Lines from Layers”, Worster Polytech, 2004.

[8] S. Jarzabek, “The Notepad example,” XVCL - Technology for Reuse. February 2004, <http://fxvcl.sourceforge.net/XVCL-Examples_files/Notepad.zip>.

[9] R.E. Johnson and B. Foote, “Designing Reusable Classes”, *Journal of Object-Oriented Programming*, June/July 1988.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-oriented programming”. *ECOOP*, 1997.

[11] M. Matinlassi, “Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA, and QADA”, *ICSE 2004*, 127-136.

[12] W. Taha, “A Gentle Introduction to Multi-stage Programming”. *Domain-Specific Program Generation*, 2004.

[13] Y. Smaragdakis and D. Batory, “Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs”, *ACM TOSEM*, April 2002.

[14] S.M. Swe, H. Zhang, and S. Jarzabek “XVCL: a tutorial”, Software Engineering and Knowledge Engineering 2002, 341-349. Also available online at <http://fxvcl.sourceforge.net>.

10 Appendix

We briefly discuss some ideas that we were unable to completely explore.

In some industries, a high demand is placed on applications that operate within strict performance boundaries. Can generative programming work in these sectors, where customizability is paramount? Indeed, both of the GDMs we have discussed can handle the challenges of high-performance applications. The key to maintaining high performance lies in choosing appropriately-sized features, so that no “extra baggage” exists in any feature [3][4]. If the situation arises in which no existing frames or features provide an acceptable solution, one always has the ability to override sections of code from a higher-level frame.

There are interesting extensions to our program designs. The first two involve adding more flexibility to our OO and compositional designs, and the third is a proposed addition to the XVCL language.

One problem we see in our design is that features must hard-wire which menu they would like to be placed on, based only on the menu’s location in the menubar, rather than the menu’s name. If we parameterized this choice, the author of the generator could customize menus, rather than being forced to live with the choices of the x-frame author. As an example: instead of hard-wiring “**`Close`**” to be on the left-most menu — which happens to be named “**`File`**” — we could parameterize that value and place it on the “**`Help`**” menu. While this particular change doesn’t make much logical sense, it illustrates the concept. One could imagine a menu option such as “Preferences” that could easily be placed on one of many different menus.

Another option that we chose not to implement was that of separators on a menubar or toolbar. These are the small, gray lines that “group” menu items or toolbar buttons that have similar functions. While they do not enhance functionality, they can enhance usability. Adding these separators to our compositional design might be painless; one could create another instance of a **`NotepadFeature`** whose **`install()`** method was empty, and whose **`addMenubar()`** and **`addToolbar()`** methods inserted the appropriate separators. The only difficulty that we know is that the order in which menu items appear in a menu shouldn’t be dictated by the order in which features are composed.

The final extension does not affect our designs at all, but could impact future, large-scale generators that change over time. Imagine a programmer wishes to add another “functional bar” to an application, perhaps a “FooBar.” Currently, the XVCL processor requires the programmer to add another method to every **`NotepadFeature`** contained in any application with a **`FooBar`**. But perhaps only one or two features modify the **`FooBar`** — the programmer must spend time creating lots of extra, *blank* **`addFooBar()`** methods. What would be ideal is for the XVCL processor to be able to skip x-frames if they did not exist, perhaps issuing a warning when such a skip was made. If that were the case, no time would be wasted creating unnecessary blank x-frames, and the programmer could concentrate on creating only the x-frames that actually implement a modification to the application.