Code Generation on Steroids: Enhancing COTS Code Generators via Generative Aspects

Cody Henthorne
Computer Science Dept.
Virginia Tech
codyh@cs.vt.edu

Eli Tilevich
Computer Science Dept.
Virginia Tech
tilevich@cs.vt.edu

Abstract

Commercial of-the-shelf (COTS) code generators have become an integral part of modern commercial software development. Programmers use code generators to facilitate many tedious and error-prone software development tasks including language processing, XML data binding, graphical component creation, and middleware deployment. Despite the convenience offered by code generators, the generated code is not always adequate for the task at hand. This position paper proposes an approach to address this problem. We utilize the power of Aspect Oriented Programming (AOP) to enhance the functionality of generated code. Furthermore, our approach enables the programmer to specify these enhancements through an intuitive graphical interface. Our proof-of-concept software tool provides event-handling AspectJ aspects that enhance the functionality of the XML processing classes automatically generated by a commercial of-the-shelf code generator, Castor.

1. Introduction

One of the most effective approaches to automating menial programming tasks is automatic code generation. A code generator takes a high level description as input and generates lower level code. That is, the input specification for generators is simpler and shorter than the generated code. Hence, code generation not only saves time and effort, but also avoids many programming errors and increases programmer productivity [1, 2].

It is not surprising that software generators constitute an important domain of COTS software that in the future will only become more important. As the complexity of computing systems continues to grow, software generators have the potential to provide elegant solutions that tame the complexity, enabling the design at a higher level of abstraction and the use of declarative approaches [2].

Alas the benefits of automatic code generation can diminish rapidly if the generated code does not fully satisfy the requirements for the task at hand. Examples of generated code deficiencies include not adhering to the in-house coding convention, missing important features, and having incorrect concurrency properties (e.g., not thread-safe).

Of course, automatically generated code can always be further refined by hand to meet the requirements, but this is not a viable solution. Every time a code generator is re-run (in response to changed input or as part of a build process), all the hand-written changes will be lost and re-applying the changes would be a waste of programming effort.

Another approach to customizing the functionality of a code generator is to change the source code of the generator itself. Nevertheless, this is impossible for those COTS code generators that are proprietary. Even the ones that are open-source are often large and intricate. Reverse engineering such a code generator thoroughly enough to be able to change its functionality in a meaningful way could be a prohibitively difficult and time-consuming undertaking. In fact, this process could be on par with the time it would take to develop a custom in-house code generator, completely negating the time-saving benefit of using COTS code generators.

This paper presents a novel approach that provides the programmer with a capability to enhance the functionality of the generated code, without any of the shortcomings of the two approaches outlined above. Specifically, we enable changing the functionality of the generated code externally to the code itself and to the COTS code generator. At the core of the approach is a visual tool that presents generated code to the programmer, who can then use the tool's GUI to express the needed enhancements. The visual tool then encodes the enhancements by generating aspect oriented programs. Finally, an aspects compiler, such as AspectJ [3], weaves these automatically-generated aspects with the generated code, thereby customizing and enhancing the code with the required functionality [4].

The rest of this paper is structured as follows. Section 2 provides concrete examples of inadequacies in generated code. Section 3 summarizes AOP and outlines our approach to enhancing COTS code generators through generative aspects. Section 4 explains our proof-of-concept tool that we used to enhance generated code in an unrelated software project. Section 5 describes related work, and Section 6 contains our future directions and conclusions.

2. Issues with Generated Code

The biggest challenge of using code generators in general, and of COTS ones in particular, is that it is difficult to make the generated code have all the properties required by a given software development scenario. In fact, it is this difficulty that makes general purpose code generators infeasible—instead, code generators proved practical only when applied in a domain-specific fashion [2]. In the introduction, we have outlined three issues that can make generated code inadequate for immediate use, and next we explain each of them in greater detail.

Large software organizations commonly follow established coding conventions (e.g., Java Code Conventions [5]). A coding convention is a policy for naming programming language constructs such as classes, methods, fields, and variables. For example, it might be required to start all the member fields of a class with the underscore character (e.g., `_field`).

A COTS code generator might not provide the means to customize the naming conventions used for automatically generated code. This can be a serious issue—even if it does not stop a programmer from using the generated code, it can cause annoying inconsistencies with the accepted conventions.

Generated code might not completely provide all the desired functionality. To make a code generator useful, it must be general enough to satisfy multiple programming scenarios. Furthermore, customizing code generators for all possible special cases is impossible, and some required functionality cannot be automatically generated. For example, it might be necessary to serialize an object of a generated class to permanent storage [6]. However, the serialization functionality might not be automatically generated.

Generated code might not be equipped for concurrent execution. Concurrency has entered the world of desktop and enterprise applications and is no longer reserved just for high-performance servers and operating systems [7]. Nevertheless, generated code might not be thread-safe, and thus not suitable for safe concurrent execution.

These are just three simple examples that demonstrate issues with generated code that can make it inadequate for immediate use. One could imagine many other cases when automatically generated code misses the mark one way or another. Next we present our approach that can solve this problem.

3. Enhancing Generated Code

One of the new and exciting approaches to the challenge of providing clean and intuitive mechanisms for separating concerns is called Aspect Oriented Programming (AOP) [4]. Specifically, AspectJ is a popular Java language extension that enables programmers to express cross-cutting concerns [3]. For example, the following code snippet demonstrates how the programmer can express a simple logging aspect and apply it to method `foo`[1].

```
void foo(int x, int y) {
        System.out.println(x + y);
}
before():call(* foo(..)) {
        System.out.println("Log call: " +
                                thisJoinPoint);
}
```

The AspectJ compiler then seamlessly weaves this aspect to a given application. In other words, the functionality of method `foo` above is enhanced without modifying its source code. Even if the body of the method should change in the future, it will not affect the functionality added through the logging aspect. Our approach leverages this power of AspectJ to enhance the functionality of generated code. It is applicable in the cases in which automatically generated code does not meet the requirements and changing the behavior of the code generator itself is not feasible.

Figure 1 depicts the main steps of our approach schematically. As the figure shows, our approach starts with the COTS code generator running at will—either in response to changes to the input of the code generator or as part of a build process. In either case the code is re-generated from scratch every time. (Hence, manual changes to the generated code are

[1] `thisJoinPoint` is a language construct of AspectJ similar to `this;` it captures info about the currently advised method.

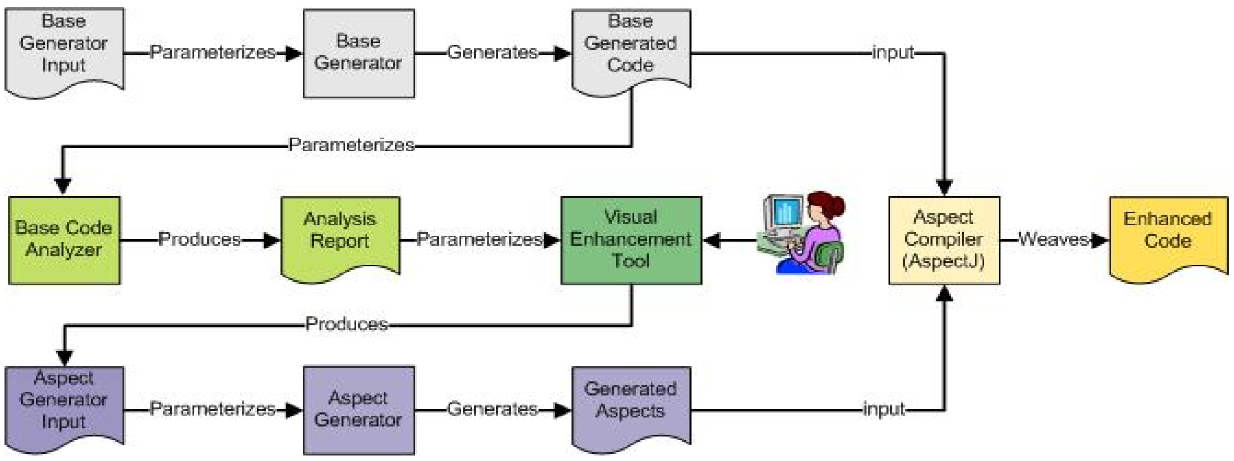

Figure 1: Our Approach-Tools and Control Flow

futile: all of them will be lost during the very next code regeneration.) Thus, the base code generator executes normally and produces a collection of source files. We call these source files *base generated code* because it does not meet the application requirements in some way.

Then a static code analyzer extracts basic structural info from the generated code such as classes, methods, and fields and saves the results for future use. This allows us to compare the results of running the COTS code generator under different inputs and to keep our enhancements up-to-date. The programmer can then resolve the inconsistencies instead of having to complete all the subsequent steps of our approach from scratch.

To make the analysis results useful to the programmer, our Visual Enhancement Tool (VET) displays them in a graphical environment. The analysis considers only classes, methods, and fields rather than method bodies, which facilitates the use of predefined scenarios in VET for expressing the desired enhancements. We call these scenarios *enhancement patterns*. Consider, for example, the case if generated code is not thread safe. The *Make Thread Safe* enhancement pattern can be applied to the selected individual methods, method groups, or entire classes to make them thread-safe. Implementation-wise, it would mean adding some locking capabilities that can be done externally to the base generated code. Furthermore, the locking would be added to the base code through automatically generated aspects.

Since it might not be feasible to generate aspects entirely on visual input, the programmer might add some small code snippets to express the additional functionality that would have to be interposed against the base code. Based on both the VET output and the programmer's supplied code snippets, the aspect generator can produce all the required aspects to enhance the functionality of the base generated code.

The last step uses standard functionality of an aspect compiler such as AspectJ. It can effectively weave together all the automatically generated aspects with the base generated code. For deployment purposes, only the runtime libraries of the aspect compiler would have to be provided.

To summarize, our approach completely avoids having to modify either the code generator itself or the generated code by hand. Instead, we effect the modification externally by employing generative aspects. That is, we automatically generate aspects to enhance base code using input from our visual tool and code analysis.

4. Proof-of-Concept

Our idea for the aforementioned approach to enhancing generated code arose as a practical solution to a real problem that we encountered while working on an unrelated research project. The goal of that project is to create a graphical Integrated Development Environment (IDE) for designing and creating parallel applications. As is often the case, our project is very experimental in nature, and we find ourselves discovering new requirements and needs almost daily.

As it is common for modern GUIs, we implement the Observer pattern, a type of Model-View-Controller (MVC) architecture [8]. As its names suggests, this architecture provides a clean separation of concerns between the model, managing the data; view, providing the graphical representation of the model; and the controller, coordinating the interaction of the two.

Because XML is a de-facto standard for representing, storing, and transporting data, we decided to persist our model to an XML file. However, this model has to be represented as regular Java classes as well. The transformations between XML files and their corresponding Java representations, called *XML data binding*, can be a substantial programming task if done by hand. Fortunately, XML grammar presents an excellent opportunity for automating data binding through code generation.

For this task, we have used a popular commercial code generation suite called Castor [9]. Castor takes a XSD XML schema file as input, and generates code for representing XML content via Java classes and generates a marshalling framework. A marshalling framework provides the ability transform the model from XML to Java and back. Using a COTS code generator such as Castor takes away the necessity of processing XML by hand. The following example presents a Castor input XSD file that generates a class `SampleObject` with getter and setter methods for a string field named `objectname`:

```
<xs:element name="SampleObject">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="objectname"
         type="xs:string"/>
    </xs:sequence>
   </xs:complexType>
</xs:element>
```

Having automated this programming task proved to be a boon for our research project. Since our model changes frequently, we only need to change the Castor XML schema file to re-generate all the XML processing code. It is these kinds of scenarios for which code generation is indispensable.

The careful reader might have noticed that our discussion so far has completely omitted the description of how our model interacts with the controller. In the classical Observer pattern, the `Observable` (model) must have the capability to notify all `Observers` (views) every time data within the `Observable` changes. To accomplish this requires adding some notification-specific logic to the generated model code.

Let us consider the functionality which needs to be added to the Castor generated XML data binding code to make it possible to use this code as `Observable`. The original version of the automatically generated code simply represents XML data and allows accessing and modifying this data through accessor and mutator methods. For example, consider this mutator method that would be generated from the above schema:

```
void setObjectname (String objectname) {
        this.objectname = objectname;
}
```

To implement the desired controller interactions, it is necessary to interpose some logic every time a mutator method is called. To effect the required controller interaction, we would modify this mutator method as follows:

```
void setObjectname (String objectname) {
        this.objectname = objectname;
        for (Listener l : listeners) {
                l.notify(obj);
        }
}
```

Notice that method `setObjectname` is automatically generated. Therefore, every time code re-generation takes place, this method will be created anew and any hand-made changes will be completely lost. This will quickly become a burdensome and time-consuming development task, besides being a huge source of frustration.

Castor, being a commercial quality code generator, has several advanced code generation options that a programmer may specify. One of these options, the `org.exolab.castor.builder.boundproperties` flag, provides an ability to generate all class fields as bound properties. That is, every field of every class will send a `java.beans.PropertyChangeEvent` to all the registered `java.beans.PropertyChangeListeners` when a field in a generated class is modified.

Nevertheless, this approach is too coarse-grained and does not provide enough flexibility to be able to express the advanced requirements of modern MVC interactions. For one, the bounding functionality is provided at a global level. That is, it is impossible to specify that only certain fields raise `PropertyChangeEvents` when modified. In addition, raising such events for all field mutations will generate extra overhead that can be detrimental not only to performance but to comprehensibility of the generated code. In fact, not all of the XML data might be associated with visual representations, and thus would not require any coordination with views.

The automatically generated notification logic is not only too coarse-grained, but it also does not support more complex controller interaction logic that can be required in a modern GUI application. A simple example of such complex logic is to be able to register model objects as `Observers` in addition to their standard roles as `Observables`.

In addition to supplying the controller interaction code to the model by hand and using the coarse-grained notification logic, another possible way is to change the COTS code generator itself. Indeed, Castor follows the Open Source development model and makes all of its ~250,000 lines of Java source code available. Clearly being able to understand such a large code base well enough to be able to introduce changes would require a serious investment of time and effort. Furthermore, even if such intimate understanding has been obtained, customized versions of Castor would need to be created for different application scenarios.

With the exception of the inability to easily interpose controller related logic into the generated code, Castor is a high quality COTS code generator for handling XML processing. Nevertheless, the generated code while well-suited for the task intended was insufficient to satisfy the requirements of our application. Thus, we needed an approach that would allow enhancing the Castor generated code with additional controller-related functionality without changing the base generated code by hand.

Our solution in this case involved using AspectJ aspects and advice to weave the controller related functionality into the Castor generated code. The following code example shows an aspect that adds the controller functionality right before the control flow leaves the `setObjectname` method.

```
private List<Listener> SampleObject.listeners;
after(String name) returning :
  call (* SampleObject.setObjectname(..)) &&
  args(name) {
        for (Listener l : listeners) {
                l.notify(name);
        }
}
```

To summarize, we successfully utilized aspects to express the functionality for adding, removing, and notifying of listeners to all the required Castor generated classes. In this case, however, we had to write the AspectJ code by hand, yet we saw this as a convincing proof-of-concept for a useful development tool. From our implementation experience, we observed that the hand-written AspectJ code was straightforward in structure and repetitive in nature. It is these properties of the AspectJ code that lend it to be a suitable candidate for automatic code generation. Furthermore, following the MVC pattern made the AOP code easier to understand and evolve. Even programmers, who are not familiar with AOP and AspectJ but familiar with MVC, should be able to follow our enhancement code. It also occurred to us that other enhancement patterns could be used in a similar capacity. Applying this approach to an unrelated real software development scenario demonstrated to us that a general tool for enhancing the output of COTS code generators will be useful for any software developer who uses such generators.

5. Related Work and Discussion

This work contributes to a large body of research on enhancing software systems with additional functionality through Aspect Oriented Programming [4]. Our approach presents a compelling case for the utility of using AOP, due to the fact that our approach modifies generated code externally. It would be hard to describe all the extensive body of related works; therefore, we will focus only on closely related research.

In our approach the programmer never has to interact with a lower level transformation tool directly, expressing enhancements through a GUI to have them automatically translated into specific instructions for a lower-level transformation tool. Even though we could use different tools for lower-level transformations, we prefer AspectJ because of its widespread popularity and mature linguistic support for expressing transformations. Nevertheless, one feature that we would find useful is the ability to replace or remove portions of base code. This is not part of AOP, whose raison d'être is weaving in cross-cutting concerns rather than replacing existing code.

GOTECH has proposed the idea of generative aspects to enhance centralized Java programs with distributed functionality as captured by the J2EE standard [10, 11]. Later, MAJ expanded on the GOTECH work and introduced an approach to generating syntactically correct AspectJ programs [12]. It is possible to use MAJ for generating AspectJ code in our enhancement infrastructure. A related but more domain-specific approach proposes using generative aspects and a visual tool for dynamic UML modeling of C++ programs [13].

Our code enhancement approach is closely related to pattern-based design and development. Design Patterns and Architectural Patterns, however, focus on building software systems from scratch [14]. In contrast, our approach starts with an existing code base and then uses patterns to modify it.

In our approach, we take advantage of several known static analysis techniques for discovering various facts about the generated code such as the existing associations between different objects [15]. Also our visual tool will use state-of-the-art software visualization techniques to present the analysis information to the programmer [16].

6. Future Directions and Conclusions

The presented approach and the tool infrastructure is a work in progress. So far we have succeeded in determining the general architecture and the control flow between different tools. Nevertheless, many of the implementation specifics have yet to be properly designed and implemented. While some parts of our approach present a purely engineering challenge, creating some other components will require resolving serious conceptual issues.

In our estimation, the hardest and most important component of our approach and the one which will determine the overall success of this research is our Visual Enhancement Tool (VET). While representing the results of static analysis of the base generated code can be reduced to the problem of visualizing large data sets [16], elegantly and intuitively mapping the enhancement patterns to the analysis results will require a truly innovative solution.

We plan to focus on supporting only a handful of enhancement patterns, rather than supporting as many of them as possible. This will serve to flatten the learning curve for our tools and will provide us with more time to concentrate on improving their usability. Nevertheless, we are determined to make our tools extensible, so that other developers would be able to express and apply their own enhancement patterns using our tools.

To that end, we plan to take advantage of the capabilities of the Eclipse Rich Client Platform (RCP) and expose our tools as plug-ins [17]. Thus, any developer who uses Eclipse will be able to benefit from using our tools. Furthermore, Eclipse RCP has excellent facilities for enabling third-party developers to extend existing plug-ins in a flexible fashion. We do intend to use these facilities to allow developers to supply their own enhancement patterns.

We intend to test the feasibility of our approach on multiple domains that use COTS code generators. While XML data binding is an important domain for code generators, it would be interesting to explore how much practical benefit can be derived from enhancing base generated code in other domains.

Finally, we believe that the concept of Enhancement Patterns has a promise and deserves a careful exploration. Using this concept to modify generated code is only one particular application. The ability to represent higher level enhancements as well known patterns, not easily expressible through the transformations offered by existing AOP tools, may be useful in other application scenarios.

As code generation has entered the mainstream of industrial software development, code generators have become popular third-party commercial products for in-house use. We believe that finding a good approach to enhancing COTS code generators is an important endeavor, and it will make code generation even more successful and widely-used.

7. References

[1] Y. Smaragdakis, S. S. Huang, and D. Zook, "Program Generators and the Tools to Make Them," in *SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*: ACM Press, 2004.

[2] Y. Smaragdakis and D. Batory, "Application generators," *Encyclopedia of Electrical and Electronics Engineering,* 2000.

[3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," in *ECOOP*, 2001.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwing, "Aspect-Oriented Programming," in *ECOOP*: Springer-Verlag, 1997.

[5] Sun Microsystems Inc, "Java Code Conventions," http://java.sun.com/docs/codeconv/CodeConventions.pdf.

[6] Sun Microsystems Inc, "Object Serilization," http://java.sun.com/j2se/1.5.0/docs/guide/serialization/index.html, 2004.

[7] H. Sutter, "A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal,* vol. 30, pp. 16-20, 2005.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*: Addison-Wesley Longman Publishing Co., Inc., 1995.

[9] ExoLab Group, "The Castor Project," http://www.castor.org/index.html.

[10] E. Tilevich, S. Urbanski, Y. Smaragdakis, and M. Fleury, "Aspectizing Server-Side Distribution," in *ASE*, 2003.

[11] Sun Microsystems Inc, "Java EE at a Glance," http://java.sun.com/j2ee/.

[12] D. Zook, S. S. Huang, and Y. Smaragdakis, "Generating AspectJ Programs with Meta-AspectJ," *Generative Programming and Component Engineering Conference,* 2004.

[13] B. A. Malloy and J. F. Power, "Exploiting UML dynamic object modeling for the visualization of C++ programs," in *SoftVis*, 2005.

[14] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-oriented software architecture: a system of patterns*: John Wiley & Sons, Inc. New York, NY, USA, 1996.

[15] D. Jackson and A. Waingold, "Lightweight extraction of object models from bytecode," *IEEE Software Engineering,* vol. 27, pp. 156-169, 2001.

[16] J. Stasko, *Software visualization*: MIT Press Cambridge, Mass, 1998.

[17] Eclipse Foundation, "Eclipse.org," http://www.eclipse.org.