
Review: Code Generation in Action

Aaron Longwell

Table of Contents

1. Overview	1
2. The Case for Code Generation	1
3. Building a Code Generator	2
4. Where To Go From Here	3
5. The Verdict	3

1. Overview

For several decades, computers and their software have been making all areas of our personal and business lives more convenient and more efficient. Hardware advancements have made our computers constantly more capable. We, as software developers, are using these advancements to make our clients' daily operations easier, less time-consuming, and more pleasant. Our software attempts as much as possible to isolate the mundane, repetitive tasks, and automate them. Interestingly, many of the tasks we automate for our clients are nowhere near as repetitive or mundane as software development itself.

That unfortunate irony is the driving force behind a new book from Manning, *Code Generation in Action*, by Jack Herrington. The book deftly balances the theory, techniques and politics of code generation. That's right, I did say "politics of code generation." Surprisingly, Herrington devotes more than a chapter to addressing the personal and staff conflicts that can occur when pursuing a code generation strategy. After making a convincing case that code generation can and should be used in most development projects, the book flows naturally through a quick academic analysis and into a series of code samples and case studies.

2. The Case for Code Generation

In his political analysis mentioned earlier, Herrington identifies six primary benefits of code generation:

Productivity	Through eliminating redundant and repetitive work.
Quality	In generated code, bug fixes are done to the generating system, not the generated system. Hence, a bug fix is immediately global throughout the system. In addition, generated code increases the need for unit testing to verify that the generator is building bug-free code. On the bright side, generated code is easier to unit test because it follows a strict template and structure. At a more basic level, the productivity gains of code generation allow you to either include more features, increase code quality, or both.
Consistency	Generated source code eliminates discrepancies in coding styles from developer to developer. It also solves the problem of classes slipping through the cracks when system-wide changes are made. By definition, generated code is consistent.
Abstraction	Some code generators are built using an abstract model of the system as input. This abstract model is a powerful communication tool that's accessible to engineers and

analysts alike.

These benefits are illustrated in the book's opening case study: a somewhat utopian world in which an EJB accounting application with 150 tables is reduced from a 3+ year timetable to a matter of months. In this case, the hypothetical development team chooses to create a sophisticated code generator which creates EJB classes, user interface code, Swing code, unit tests and an RPC layer in SOAP. Perhaps most unbelievable is that the SOAP project was not part of the initial spec. It is added because there is plenty of time leftover in the timeline after all requirements are fulfilled!

As unbelievable as the example is, it is still extremely persuasive. Most persuasive is the estimate that the data access code alone is a 3 man-year effort. A generator to build the same code can be completed in just over a month. When you consider that virtually all data-access code is the same, and that a rules system to handle special cases is the most complex task in creating the generator, the month-long timeframe seems reasonable.

This example is both stunning and scary. The thought that huge sums of development time are being wasted writing the same code over and over represents an incredible opportunity... but also a giant risk. Chances are, your current development team is much better at writing domain code than writing code for a code generator. Code generators require unique talents: intimate knowledge of language syntax, text templating, regular expressions, XML parsing, and file/directory handling.

3. Building a Code Generator

In addition to requiring different developer talents, the author recommends building your code generator in a language other than the implementation language, primarily because it is extremely difficult to read Java code inside String literals inside Java code. The author wholeheartedly recommends Ruby for code generation tasks, and thankfully provides plenty of simple code samples and a short Ruby introduction. The primary benefits of Ruby as a code generation language are its strong support for regular expressions, XML, files, text processing, and its ERb text template library.

After discussing technique and best practices, the book focuses on six working examples of code generators. Each example gets its own dedicated chapter complete with source code. The topics:

User Interfaces	Demonstrates a code generator which creates JSP and Swing user interfaces from a definition file which describes user interface components and runtime business logic. The chapter includes Ruby source code and some sidebar discussion of user interface design practices.
Documentation	We're all using JavaDoc already (at least we should be) This chapter extends the JavaDoc idea to create an SQLDoc tool which documents SQL commands. It's a great idea, but I haven't found myself needing better documentation for SQL in my applications.
Unit Test Generation	Although source code generation suggests a different mindset than test-driven development, in reality, the two are tightly integrated. First, if you're auto-generating the code... you better be sure it's working as intended... you can't "blame the generator." This chapter explores auto-generating unit tests for your code. All examples in this chapter are C, but the concepts are certainly useful across languages.
Embedded Psuedo-SQL	This chapter focuses on a Perl example, using a code generator to translate a made up language PerlSQL, into 100% functional Perl data access code. I don't use Perl for database projects very often,

and I prefer the next example, which covers creating an entire data access tier.

Database Access Generators

If you're like me, this chapter is the reason you're interested in the book. Data-access code is repetitive, tedious, and prone to bugs. This chapter explores generating an entire data access tier for an application. The generator uses 3 input sources: a schema of the data involved, a business logic definitions file, and a set of sample data (for unit testing). The chapter follows using this strategy called a "tier generator," in EJBs, JDBC code, ASP, ASP.NET, Perl and PHP. Quite an exhaustive list... unfortunately, the chapter does not contain enough sample code and enough detail in any of the languages to really get you started. Instead it provides an architecture for you to create a solution yourself.

Web Services Layers

Covers using a language parser to read Java classes and create XML-RPC web services classes for them. It is perhaps the most difficult generator to write (because of its technique: language parsing), and maybe the least useful in real-life (3rd-party libraries and existing JavaDoc-style code generators already exist. See XDoclet and Axis for examples).

I was hoping that the above example code generators would inspire me to explore code generation in my projects. Unfortunately, the examples provide too little detail to set me on my way. On the bright side, the author provides some in-depth architecture descriptions that do provide a great start to rolling your own generator. There are very few examples in the text that provide an end-to-end walkthrough.

4. Where To Go From Here

Because of its lack of detailed examples, this book will be your map more than your guide on your journey into code generation. It provides a great introduction to the techniques involved, and gives you enough direction to start exploring on your own.

Chances are, you'll need to spend some time honing your Ruby, Perl or Python skills. These languages provide the necessary regular expression, XML, and general text-hacking features. The author highly recommends using Ruby for building your code generator. Ruby's popular 3rd-party template system, ERb, provides an excellent framework for writing Java source files. For me, and those of you who can't part with Python, Cheetah provides the same benefits for that language.

Beyond learning the necessary techniques, you'll need to spend some time examining when and where code generation can benefit your enterprise. The book's highly academic approach will give you a good vocabulary for discussing the type of solution your team can employ. You'll get the biggest return on investment for the strategy the author calls "tier generation" (an approach that generates one entire tier of the application), but you may be able to employ some of the less intensive approaches more effectively.

5. The Verdict

Despite its weakness in depth and detail, I would still recommend this book to virtually any developer or manager. It is an excellent introduction to the techniques and topics of code generation, and includes some solid analysis of the resistance you may experience when selling code generation to your development team. You'd be hard pressed to find a more complete introduction to code generation, and even the limited examples in the text are more complete than you'll likely find on the Internet.

I highly recommend reading the sample chapters available on the book's companion web site, code-generation.net. The two chapters available for download are the best in the book, and make up a ma-

majority of the general code generation discussion. Chapter 1 includes the sales pitch for code generation, and Chapter 4 provides a complete explanation of the six discrete types of code generators. For some, Chapter 4 alone may give you enough background to get started writing a code generator for your project.